

VisualAge COBOL



# Programming Guide

*Version 3.0.2*



VisualAge COBOL



# Programming Guide

*Version 3.0.2*

**Note!**

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 575.

**Second Edition (December 2000)**

This edition applies to VisualAge COBOL for Windows NT, Version 3.0.2 (program number 5639-I44), and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Order publications by phone or fax. IBM Software Manufacturing Solutions takes publication orders between 8:30 a.m. and 7:00 p.m. Eastern Standard Time (EST). The phone number is (800) 879-2755. The fax number is (800) 445-9269.

You can also order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

A form for readers' comments appears at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation, Department HHX/H3  
P.O. Box 49023  
San Jose, CA 95161-9023  
USA

or fax it to this U.S. number: 800-426-7773

or use the form on the Web at:

<http://www.software.ibm.com/ad/rcf/>

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

© Copyright International Business Machines Corporation 1996, 2000. All rights reserved.

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

---

# Contents

## About this book . . . . . xi

Who should use this book . . . . .	xi
Terminology used in this book . . . . .	xi
How to read syntax diagrams . . . . .	xii
Related information . . . . .	xiii
How to send us your comments . . . . .	xiii

---

## 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 . . . . .	9
Identifying files to the operating system . . . . .	9
Describing the data . . . . .	11
Using data in input and output operations . . . . .	11
Comparison of WORKING-STORAGE and LOCAL-STORAGE . . . . .	12
Using data from another program . . . . .	13
Processing the data . . . . .	14
How logic is divided in the PROCEDURE DIVISION . . . . .	15
Declaratives . . . . .	19

### Chapter 2. Using data . . . . . 21

Using variables, structures, literals, and constants . . . . .	21
Variables . . . . .	21
Data structure: data items and group items . . . . .	21
Literals . . . . .	22
Constants . . . . .	22
Figurative constants . . . . .	22
Assigning values to data items . . . . .	23
Examples: initializing variables . . . . .	23
Initializing a structure (INITIALIZE) . . . . .	24
Assigning values to variables or structures (MOVE) . . . . .	25
Assigning arithmetic results (MOVE or COMPUTE) . . . . .	26
Assigning input from a screen or file (ACCEPT) . . . . .	26
Displaying values on a screen or in a file (DISPLAY). . . . .	27
Using intrinsic functions (built-in functions) . . . . .	27
Types of intrinsic functions . . . . .	28
Nesting functions . . . . .	28
Using tables (arrays) and pointers . . . . .	28

### Chapter 3. Working with numbers and arithmetic . . . . . 31

Defining numeric data. . . . .	31
Displaying numeric data . . . . .	32
Controlling how numeric data is stored . . . . .	33
Formats for numeric data. . . . .	34
External decimal (DISPLAY) items. . . . .	34
External floating-point (DISPLAY) items. . . . .	34
Binary (COMP) items . . . . .	35
Native binary (COMP-5) items . . . . .	35
Byte reversal of binary data . . . . .	36
Packed-decimal (COMP-3) items . . . . .	36
Floating-point (COMP-1 and COMP-2) items . . . . .	36
Examples: numeric data and internal representation . . . . .	36
Data format conversions . . . . .	38
Conversions and precision . . . . .	39
Sign representation and processing . . . . .	40
Checking for incompatible data (numeric class test) . . . . .	40
Performing arithmetic . . . . .	41
COMPUTE and other arithmetic statements . . . . .	41
Arithmetic expressions . . . . .	41
Numeric intrinsic functions . . . . .	42
Nesting functions and arithmetic expressions . . . . .	43
ALL subscripting and special registers . . . . .	43
Examples: numeric intrinsic functions . . . . .	43
General number handling . . . . .	43
Date and time . . . . .	43
Finance. . . . .	44
Mathematics . . . . .	44
Statistics . . . . .	45
Fixed-point versus floating-point arithmetic . . . . .	45
Floating-point evaluations . . . . .	45
Fixed-point evaluations . . . . .	46
Arithmetic comparisons (relation conditions) . . . . .	46
Examples: fixed-point and floating-point evaluations . . . . .	46
Using currency signs . . . . .	47
Example: multiple currency signs . . . . .	48

### Chapter 4. Handling tables . . . . . 51

Defining a table (OCCURS) . . . . .	51
Nesting tables . . . . .	52
Subscripting . . . . .	52
Indexing . . . . .	53
Referring to an item in a table . . . . .	53
Subscripting . . . . .	54
Indexing . . . . .	55
Putting values into a table . . . . .	56
Loading a table dynamically. . . . .	56
Initializing a table (INITIALIZE) . . . . .	56
Assigning values when you define a table (VALUE) . . . . .	57
Example: PERFORM and subscripting . . . . .	58
Example: PERFORM and indexing. . . . .	59
Creating variable-length tables (DEPENDING ON) . . . . .	60

Loading a variable-length table . . . . .	61
Assigning values to a variable-length table . . . . .	62
Searching a table . . . . .	62
Doing a serial search (SEARCH) . . . . .	63
Doing a binary search (SEARCH ALL) . . . . .	64
Processing table items using intrinsic functions . . . . .	65
Example: intrinsic functions . . . . .	65

## Chapter 5. Selecting and repeating program actions . . . . . 67

Selecting program actions . . . . .	67
Coding a choice of actions . . . . .	67
Coding conditional expressions . . . . .	71
Repeating program actions . . . . .	74
Choosing inline or out-of-line PERFORM . . . . .	75
Coding a loop . . . . .	76
Coding a loop through a table . . . . .	76
Executing multiple paragraphs or sections . . . . .	77

## Chapter 6. Handling strings . . . . . 79

Joining data items (STRING) . . . . .	79
Example: STRING statement . . . . .	79
Splitting data items (UNSTRING) . . . . .	81
Example: UNSTRING statement . . . . .	81
Manipulating null-terminated strings . . . . .	83
Example: null-terminated strings . . . . .	84
Referring to substrings of data items . . . . .	84
Reference modifiers . . . . .	85
Example: arithmetic expressions as reference modifiers . . . . .	86
Example: intrinsic functions as reference modifiers . . . . .	86
Tallying and replacing data items (INSPECT) . . . . .	87
Examples: INSPECT statement . . . . .	87
Converting data items (intrinsic functions) . . . . .	88
Converting to uppercase or lowercase (UPPER-CASE, LOWER-CASE) . . . . .	88
Converting to reverse order (REVERSE) . . . . .	89
Converting to numbers (NUMVAL, NUMVAL-C) . . . . .	89
Evaluating data items (intrinsic functions) . . . . .	90
Evaluating single characters for collating sequence . . . . .	90
Finding the largest or smallest data item . . . . .	90
Finding the length of data items . . . . .	92
Finding the date of compilation . . . . .	93

## Chapter 7. Processing files . . . . . 95

Identifying files . . . . .	95
Identifying Btrieve files . . . . .	96
Identifying STL files . . . . .	96
Identifying remote files . . . . .	96
File system . . . . .	97
STL file system . . . . .	97
Protecting against errors when opening files . . . . .	100
Specifying a file organization and access mode . . . . .	100
File organization and access mode . . . . .	100
Setting up a field for file status . . . . .	104
Describing the structure of a file in detail . . . . .	104
Coding input and output statements for files . . . . .	104
Example: COBOL coding for files . . . . .	105

File position indicator . . . . .	106
Opening a file . . . . .	107
Reading records from a file . . . . .	109
Adding records to a file . . . . .	110
Replacing records in a file . . . . .	111
Deleting records from a file . . . . .	112
PROCEDURE DIVISION statements used to update files . . . . .	112

## Chapter 8. Sorting and merging files 115

Sort and merge process . . . . .	115
Describing the sort or merge file . . . . .	116
Describing the input to sorting or merging . . . . .	116
Example: describing sort and input files for SORT . . . . .	117
Coding the input procedure . . . . .	118
Describing the output from sorting or merging . . . . .	118
Coding the output procedure . . . . .	119
Restrictions on input and output procedures . . . . .	119
Requesting the sort or merge . . . . .	120
Setting sort or merge criteria . . . . .	121
Choosing alternate collating sequences . . . . .	121
Example: sorting with input and output procedures . . . . .	121
Determining whether the sort or merge was successful . . . . .	122
Stopping a sort or merge operation prematurely . . . . .	123

## Chapter 9. Handling errors. . . . . 125

Handling errors in joining and splitting strings . . . . .	125
Handling errors in arithmetic operations . . . . .	126
Example: checking for division by zero . . . . .	126
Handling errors in input and output operations . . . . .	127
Using the end-of-file condition (AT END) . . . . .	128
Coding ERROR declaratives . . . . .	129
Using file status keys . . . . .	129
Using file system return codes . . . . .	131
Coding INVALID KEY phrases . . . . .	132
Handling errors when calling programs . . . . .	133

## Part 2. Compiling, linking, running and debugging your program . . . 135

### Chapter 10. Compiling, linking, and running programs . . . . . 137

Setting environment variables . . . . .	137
Setting environment variables temporarily . . . . .	137
Setting environment variables persistently . . . . .	138
Precedence of environment variables . . . . .	138
Compiler environment variables . . . . .	138
Run-time environment variables . . . . .	140
Compiling programs . . . . .	144
Compiling from the command line . . . . .	144
Compiling using batch files or command files . . . . .	148
Specifying compiler options with the PROCESS (CBL) statement . . . . .	148
Correcting errors in your source program . . . . .	149
Severity codes for compiler error messages . . . . .	149
Generating a list of compiler error messages . . . . .	150

Linking programs . . . . .	152
Specifying linker options . . . . .	153
Linking within a project environment . . . . .	153
Linking through the compiler . . . . .	153
Linking from a make file . . . . .	154
Linking from the command line . . . . .	154
Linker input and output files . . . . .	155
File name defaults . . . . .	156
Correcting errors in linking . . . . .	156
Linker return codes . . . . .	157
Linker errors in program names . . . . .	157
Running programs . . . . .	158

## Chapter 11. Compiler options . . . . . 159

ADATA . . . . .	160
ANALYZE . . . . .	161
BINARY . . . . .	161
CALLINT . . . . .	162
CHAR. . . . .	163
COLLSEQ . . . . .	165
COMPILE . . . . .	166
CURRENCY. . . . .	166
DATEPROC . . . . .	167
DYNAM . . . . .	168
ENTRYINT . . . . .	168
EXIT . . . . .	169
Character string formats . . . . .	170
User-exit work area . . . . .	170
Linkage conventions . . . . .	171
Parameter list for exit modules . . . . .	171
Using INEXIT . . . . .	171
Using LIBEXIT . . . . .	172
Using PRTEXT. . . . .	173
Using ADEXIT . . . . .	173
FLAG . . . . .	174
FLAGSTD . . . . .	175
FLOAT . . . . .	176
IDLGEN . . . . .	177
LIB . . . . .	178
LINECOUNT . . . . .	179
LIST . . . . .	179
MAP . . . . .	180
NUMBER . . . . .	181
OPTIMIZE . . . . .	181
PGMNAME . . . . .	182
PGMNAME(UPPER) . . . . .	183
PGMNAME(MIXED) . . . . .	183
PROBE . . . . .	183
PROFILE . . . . .	184
QUOTE/APOST . . . . .	184
SEPOBJ . . . . .	185
Batch compilation . . . . .	185
SEQUENCE . . . . .	186
SIZE . . . . .	187
SOSI . . . . .	187
SOURCE . . . . .	188
SPACE . . . . .	189
SQL . . . . .	189
SSRANGE . . . . .	189
TERMINAL . . . . .	190
TEST . . . . .	190

THREAD. . . . .	191
TRUNC . . . . .	192
TRUNC example 1 . . . . .	193
TRUNC example 2 . . . . .	193
TYPECHK . . . . .	194
VBREF . . . . .	195
WSCLEAR . . . . .	195
XREF . . . . .	196
YEARWINDOW . . . . .	197
ZWB . . . . .	197
Compiler-directing statements . . . . .	198

## Chapter 12. Linker options. . . . . 203

/? . . . . .	204
/ALIGNADDR. . . . .	204
/ALIGNFILE . . . . .	204
/BASE . . . . .	205
/CODE . . . . .	205
/DATA . . . . .	206
/DBGPACK, /NODBGPACK . . . . .	206
/DEBUG, /NODEBUG . . . . .	206
/DEFAULTLIBRARYSEARCH, /NODEFAULTLIBRARYSEARCH . . . . .	207
/DLL . . . . .	207
/ENTRY . . . . .	208
/EXECUTABLE . . . . .	208
/EXTDICTIONARY, /NOEXTDICTIONARY . . . . .	208
/FIXED, /NOFIXED . . . . .	209
/FORCE . . . . .	209
/HEAP . . . . .	209
/HELP . . . . .	210
/INCLUDE . . . . .	210
/INFORMATION, /NOINFORMATION . . . . .	210
/LINENUMBERS, /NOLINENUMBERS . . . . .	210
/LOGO, /NOLOGO . . . . .	211
/MAP, /NOMAP . . . . .	211
/OUT . . . . .	212
/PMTYPE . . . . .	212
/SECTION . . . . .	212
/SEGMENTS . . . . .	213
/STACK . . . . .	214
/STUB . . . . .	214
/SUBSYSTEM . . . . .	214
/VERBOSE . . . . .	215
/VERSION . . . . .	215

## Chapter 13. Run-time options . . . . . 217

CHECK . . . . .	217
DEBUG . . . . .	218
ERRCOUNT. . . . .	218
FILESYS . . . . .	218
TRAP . . . . .	219
UPSI . . . . .	219

## Chapter 14. Debugging . . . . . 221

Debugging with source language . . . . .	221
Tracing program logic . . . . .	222
Finding and handling input-output errors . . . . .	223
Validating data . . . . .	223
Finding uninitialized data . . . . .	223

Generating information about procedures . . . . .	223
Debugging using compiler options . . . . .	225
Finding coding errors . . . . .	226
Finding line sequence problems . . . . .	227
Checking for valid ranges . . . . .	227
Selecting the level of error to be diagnosed . . . . .	228
Finding program entity definitions and references . . . . .	230
Listing data items . . . . .	230
Preparing to use the debugger. . . . .	231
Getting listings . . . . .	231
Example: short listing . . . . .	232
Example: SOURCE and NUMBER output . . . . .	234
Example: MAP output . . . . .	235
Example: XREF output - data-name cross-references. . . . .	237
Example: VBREF compiler output . . . . .	240
Debugging user exits . . . . .	240
Debugging assembler routines. . . . .	241

## Part 3. Accessing databases . . . . . 243

### Chapter 15. Programming for a DB2 environment . . . . . 245

DB2 coprocessor . . . . .	245
Coding SQL statements . . . . .	245
Using SQL INCLUDE . . . . .	246
Using binary items . . . . .	246
Determining the success of SQL statements . . . . .	246
Starting DB2 before compiling. . . . .	246
Compiling with the SQL option . . . . .	247
Separating SQL suboptions. . . . .	247
Using package and bind file names . . . . .	247
Ignored options . . . . .	248

### Chapter 16. Developing COBOL programs for CICS . . . . . 249

Coding COBOL applications to run under CICS . . . . .	250
Coding for ASCII-EBCDIC differences . . . . .	251
Getting the system date under CICS. . . . .	251
Making dynamic calls under CICS . . . . .	251
Calling between COBOL and C/C++ under CICS . . . . .	253
Compiling and running CICS programs . . . . .	253
EBCDIC-enabled COBOL programs under CICS . . . . .	253
Selecting run-time options . . . . .	254
Debugging CICS programs . . . . .	254

### Chapter 17. Open Database Connectivity (ODBC) . . . . . 255

Comparison of ODBC and embedded SQL . . . . .	255
Background . . . . .	256
Installing and configuring software for ODBC . . . . .	256
Coding ODBC calls from COBOL: overview . . . . .	256
Using data types appropriate for ODBC . . . . .	256
Passing pointers as arguments in ODBC calls . . . . .	257
Accessing function return values in ODBC calls . . . . .	259
Testing bits in ODBC calls . . . . .	259
Using COBOL copybooks for ODBC APIs . . . . .	260

Example: sample program using ODBC copybooks . . . . .	261
Example: copybook for ODBC procedures. . . . .	262
Example: copybook for ODBC data definitions . . . . .	265
ODBC names truncated or abbreviated for COBOL . . . . .	266
Compiling and linking programs that make ODBC calls . . . . .	267
Understanding ODBC error messages . . . . .	267
Errors from an ODBC driver . . . . .	267
Errors from the data source. . . . .	267
Errors from the driver manager . . . . .	268

## Part 4. Developing object-oriented programs . . . . . 269

### Chapter 18. Writing object-oriented programs . . . . . 271

Example: mail-order catalog . . . . .	271
Subclasses . . . . .	273
Defining a class . . . . .	274
CLASS-ID paragraph for defining a class . . . . .	274
REPOSITORY paragraph for defining a class . . . . .	275
WORKING-STORAGE SECTION for defining a class . . . . .	275
Example: defining a class . . . . .	276
Defining a class method. . . . .	277
METHOD-ID paragraph for defining a class method . . . . .	277
Overriding a method. . . . .	277
INPUT-OUTPUT SECTION for defining a method . . . . .	278
DATA DIVISION for defining a method . . . . .	278
PROCEDURE DIVISION for defining a method . . . . .	279
Coding special methods. . . . .	279
Example: defining a method . . . . .	280
Defining a client program . . . . .	285
REPOSITORY paragraph for defining a client . . . . .	285
WORKING-STORAGE SECTION for defining a client . . . . .	285
Creating and freeing instances of classes . . . . .	286
Manipulating object references . . . . .	286
Invoking methods. . . . .	287
Example: defining a client . . . . .	288
Defining a subclass . . . . .	289
CLASS-ID paragraph for defining a subclass . . . . .	290
REPOSITORY paragraph for defining a subclass . . . . .	290
WORKING-STORAGE SECTION for defining a subclass . . . . .	290
Defining a subclass method . . . . .	291
Example: defining a subclass (with methods) . . . . .	292
Defining a metaclass . . . . .	301
CLASS-ID paragraph for defining a metaclass . . . . .	301
REPOSITORY paragraph for defining a metaclass. . . . .	302
WORKING-STORAGE SECTION for defining a metaclass. . . . .	302
Defining a metaclass method . . . . .	302
Changing the definition of a class or subclass . . . . .	303
Changing a client program . . . . .	303

Example: defining a metaclass (with methods)	304
<b>Chapter 19. System Object Model.</b>	<b>311</b>
SOM Interface Repository	311
Accessing the SOM Interface Repository	311
Populating the SOM Interface Repository	312
SOM environment variables	312
SOM services	313
SOM methods and functions	313
Class initialization.	314
Changing SOM class interfaces	315
<b>Chapter 20. Using SOM IDL-based class libraries</b>	<b>317</b>
SOM objects.	317
SOM IDL.	318
Mapping IDL to COBOL	319
Using IDL operations.	319
Expressing IDL attributes	320
Common IDL types	321
Complex IDL types	324
Passing COBOL arguments and return values	327
Example: using a SOM IDL-based class library	332
Handling errors and exceptions	334
Passing environment variables.	335
Checking the exception type field	335
Handling exceptions	335
Example: checking SOM exceptions	335
Creating and initializing object instances	337
Looking at the IDL file	338
Avoiding memory leaks	339
Example: COBOL variable-length string class	340
Source code for helper routines	342
<b>Chapter 21. Wrapping or converting procedure-oriented programs</b>	<b>343</b>
OO view of COBOL programs.	343
Wrapping procedure-oriented programs	344
Coordinating procedural code with interface actions	344
Integrating procedural code into OO systems	344
Changing procedural code	345
Converting from procedure-oriented to OO programs.	345
Identifying objects.	346
Analyzing data flow and usage	346
Reallocating code to objects	347
Writing the object-oriented code	347
<b>Part 5. Working with more complex applications</b>	<b>349</b>
<b>Chapter 22. Porting applications between platforms</b>	<b>351</b>
Getting mainframe applications to compile	351
Choosing the right compiler options.	351
Allowing for language features of mainframe COBOL	351

Using the COPY statement to help port programs.	352
Getting mainframe applications to run: overview	353
Fixing differences caused by data representations	353
Fixing environment differences affecting portability	355
Fixing differences caused by language elements	356
Writing code to run on the mainframe	356
Language features supported only by the workstation compiler.	356
Compiler options supported only on the workstation	356
Names supported only on the workstation	357
Differences with THREAD	357
Writing applications that are portable between the workstation and AIX	357
<b>Chapter 23. Using subprograms</b>	<b>359</b>
Main programs, subprograms, and calls	359
Ending and reentering main programs or subprograms	360
Calling nested COBOL programs.	360
Nested programs	361
Example: structure of nested programs	362
Scope of names.	363
Calling nonnested COBOL programs	364
CALL identifier and CALL literal.	364
Calling between COBOL and C/C/C++ programs	364
Initializing environments	365
Passing data.	365
Setting linkage conventions.	365
Collapsing stack frames and terminating run units or processes	366
Handling exceptions	366
COBOL and C/C/C++ data types	367
Example: COBOL program calling C/C/C++ functions	367
Example: C programs that are called by and call COBOL programs	368
Example: COBOL program called by a C program	370
Example: results of compiling and running examples.	370
Making recursive calls	370
<b>Chapter 24. Sharing data</b>	<b>373</b>
Passing data.	373
Describing arguments in the calling program	374
Describing parameters in the called program	375
Coding the LINKAGE SECTION	375
Coding the PROCEDURE DIVISION for passing arguments	375
Grouping data to be passed	376
Handling null-terminated strings.	376
Using pointers to process a chained list	377
Using procedure pointers to pass data	380
Dealing with a Windows restriction	380
Coding multiple entry points	381
Passing return code information	382

Understanding the RETURN-CODE special register . . . . .	382
Using PROCEDURE DIVISION RETURNING . . . . .	382
Specifying CALL . . . RETURNING . . . . .	382
Sharing data by using the EXTERNAL clause. . . . .	383
Sharing files between programs (external files) . . . . .	383
Example: using external files . . . . .	383
Using command-line arguments . . . . .	386
Example: command-line arguments with -host option. . . . .	387

## Chapter 25. Building dynamic link libraries. . . . . 389

Static linking and dynamic linking . . . . .	389
How the linker resolves references to DLLs . . . . .	390
Creating DLLs . . . . .	390
Example: DLL source file and related files. . . . .	391
Creating object-oriented DLLs . . . . .	393
Creating module definition files . . . . .	394
Reserved words for module statements. . . . .	395
Summary of module statements . . . . .	395
BASE . . . . .	396
DESCRIPTION . . . . .	396
EXPORTS . . . . .	397
HEAPSIZE . . . . .	398
LIBRARY. . . . .	398
NAME . . . . .	399
STACKSIZE . . . . .	400
STUB . . . . .	401
VERSION . . . . .	401

## Chapter 26. Preparing COBOL programs for multithreading . . . . . 403

Multithreading . . . . .	403
Working with language elements with multithreading . . . . .	404
Working with elements that have run-unit scope	405
Working with elements that have program invocation instance scope . . . . .	405
Scope of COBOL language elements with multithreading . . . . .	405
Choosing THREAD to support multithreading . . . . .	406
Transferring control with multithreading . . . . .	406
Controlling the state . . . . .	406
Ending a program. . . . .	406
Preinitializing the COBOL environment . . . . .	407
Handling COBOL limitations with multithreading	407
Example: using COBOL in a multithreaded environment. . . . .	408
Source code for thr cob.c. . . . .	408
Source code for subd.cbl. . . . .	410
Source code for sube.cbl. . . . .	410

## Chapter 27. National language support 411

Setting the locale . . . . .	411
Code page . . . . .	411
Messages. . . . .	412
Collating sequence . . . . .	412
Locales and code sets supported . . . . .	413

Using DBCS user-defined words and comments	414
Restrictions on certain user-defined words. . . . .	415
Declaring DBCS data. . . . .	415
Specifying DBCS literals. . . . .	416
Using ALL . . . . .	416
Comparing literals. . . . .	417
Controlling the collating sequence . . . . .	417
DBCS collating sequence . . . . .	417
ASCII collating sequence . . . . .	417
Intrinsic functions that are sensitive to collating sequence . . . . .	418
Testing for valid DBCS characters . . . . .	418

## Chapter 28. Preinitializing the COBOL run-time environment . . . . . 419

Initializing persistent COBOL environment . . . . .	419
Terminating preinitialized COBOL environment	420
Example: preinitializing the COBOL environment	421

## Chapter 29. Processing two-digit-year dates . . . . . 425

Millennium language extensions (MLE) . . . . .	426
Principles and objectives of these extensions . . . . .	426
Resolving date-related logic problems . . . . .	427
Using a century window . . . . .	428
Using internal bridging . . . . .	429
Moving to full field expansion. . . . .	430
Using year-first, year-only, and year-last date fields	432
Compatible dates . . . . .	433
Example: comparing year-first date fields . . . . .	434
Using other date formats . . . . .	434
Example: isolating the year. . . . .	434
Manipulating literals as dates . . . . .	435
Assumed century window . . . . .	436
Treatment of nondates . . . . .	437
Setting triggers and limits . . . . .	437
Example: using limits . . . . .	438
Using sign conditions . . . . .	439
Sorting and merging by date . . . . .	439
Example: sorting by date and time . . . . .	440
Performing arithmetic on date fields. . . . .	441
Allowing for overflow from windowed date fields . . . . .	441
Specifying the order of evaluation . . . . .	442
Controlling date processing explicitly . . . . .	443
Using DATEVAL . . . . .	443
Using UNDATE . . . . .	443
Analyzing and avoiding date-related diagnostic messages . . . . .	444
Avoiding problems in processing dates. . . . .	446
Avoiding problems with packed-decimal fields	446
Moving from expanded to windowed date fields	446

## Part 6. Improving performance and productivity . . . . . 449

### Chapter 30. Tuning your program. . . . . 451

Using an optimal programming style . . . . .	451
Using structured programming . . . . .	452

Factoring expressions . . . . .	452
Using symbolic constants . . . . .	452
Grouping constant computations . . . . .	452
Grouping duplicate computations . . . . .	453
Choosing efficient data types . . . . .	453
Computational data items . . . . .	453
Consistent data types . . . . .	454
Arithmetic expressions . . . . .	454
Exponentiations . . . . .	454
Handling tables efficiently . . . . .	455
Optimization of table references . . . . .	456
Optimizing your code . . . . .	458
Optimization . . . . .	458
Choosing compiler features to enhance performance . . . . .	459
Performance-related compiler options . . . . .	460
Evaluating performance . . . . .	461

## Chapter 31. Simplifying coding . . . . . 463

Eliminating repetitive coding . . . . .	463
Example: using the COPY statement . . . . .	464
Manipulating dates and times . . . . .	465
Getting feedback . . . . .	465
Handling conditions . . . . .	465
Example: manipulating dates . . . . .	466
Example: formatting dates for output . . . . .	466
Feedback token . . . . .	467
Picture character terms and strings . . . . .	468
Example: date-and-time picture strings . . . . .	470
Century window . . . . .	471

## Part 7. Appendixes . . . . . 473

### Appendix A. Summary of differences with host COBOL . . . . . 475

Compiler options . . . . .	475
Data representation . . . . .	475
Environment variables . . . . .	476
File specification . . . . .	476
Interlanguage communication (ILC) . . . . .	476
Input and output . . . . .	477
Run-time options . . . . .	477
Source code line . . . . .	477

### Appendix B. System/390 host data type considerations . . . . . 479

CICS access . . . . .	479
Date and time callable services . . . . .	479
Floating-point overflow exceptions . . . . .	479
DB2 . . . . .	480
MQSeries . . . . .	480
Remote file access . . . . .	480
Local file access . . . . .	480
SORT . . . . .	480

### Appendix C. Intermediate results and arithmetic precision . . . . . 481

Terminology used for intermediate results . . . . .	481
Example: calculation of intermediate results . . . . .	482

Fixed-point data and intermediate results . . . . .	482
Addition, subtraction, multiplication, and division . . . . .	482
Exponentiation . . . . .	483
Example: exponentiation in fixed-point arithmetic . . . . .	484
Truncated intermediate results . . . . .	485
Binary data and intermediate results . . . . .	485
Intrinsic functions evaluated in fixed-point arithmetic . . . . .	485
Integer functions . . . . .	485
Mixed functions . . . . .	486
Floating-point data and intermediate results . . . . .	487
Exponentiations evaluated in floating-point arithmetic . . . . .	487
Intrinsic functions evaluated in floating-point arithmetic . . . . .	488
Arithmetic expressions in nonarithmetic statements . . . . .	488

### Appendix D. Complex OCCURS DEPENDING ON . . . . . 491

Example: complex ODO . . . . .	491
How length is calculated . . . . .	492
Setting values of ODO objects . . . . .	492
Effects of change in ODO object value . . . . .	492
Preventing index errors when changing ODO object value . . . . .	493
Preventing overlay when adding elements to a variable table . . . . .	493

### Appendix E. Date and time callable services . . . . . 497

CEECBLDY—convert date to COBOL integer format . . . . .	498
CEEDATE—convert Lilian date to character format . . . . .	502
CEEDATM—convert seconds to character timestamp . . . . .	505
CEEDAYS—convert date to Lilian format . . . . .	509
CEEDYWK—calculate day of week from Lilian date . . . . .	513
CEEGMT—get current Greenwich Mean Time . . . . .	515
CEEGMTO—get offset from Greenwich Mean Time to local time . . . . .	517
CEEISEC—convert integers to seconds . . . . .	519
CEELOCT—get current local date or time . . . . .	521
CEEQCEN—query the century window . . . . .	523
CEESCEN—set the century window . . . . .	525
CEESECI—convert seconds to integers . . . . .	526
CEESECS—convert timestamp to seconds . . . . .	529
CEEUTC—get coordinated universal time . . . . .	533
IGZEDT4—get current date . . . . .	533

### Appendix F. Run-time messages . . . . . 535

Notices . . . . .	575
Trademarks . . . . .	576

### Glossary . . . . . 577

<b>List of resources . . . . .</b>	<b>599</b>
VisualAge COBOL . . . . .	599
Related publications . . . . .	599

<b>Index . . . . .</b>	<b>601</b>
------------------------	------------

---

## About this book

Welcome to IBM VisualAge COBOL, IBM's COBOL development environment for Windows NT! VisualAge COBOL gives you a comprehensive development environment designed specifically for mission-critical applications.

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

There are some differences between host and workstation COBOL. For details on language and system differences between VisualAge COBOL and IBM COBOL for OS/390 & VM, see "Appendix A. Summary of differences with host COBOL" on page 475.

---

## Who should use this book

This book assumes that you have 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, refer to *IBM COBOL Language Reference*.

This book also assumes familiarity with Windows and the VisualAge COBOL development environment. For information on Windows, see your operating system documentation. To learn about the VisualAge COBOL development environment, see the *Getting Started* guide.

---

## Terminology used in this book

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

Term used	Long form
CICS	CICS for Windows NT or VisualAge CICS Enterprise Application Development
DB2	Database 2
OS/2	Operating System/2
SOM	System Object Model
STL	Standard Language file system
VisualAge CICS	VisualAge CICS Enterprise Application Development
VisualAge COBOL	IBM VisualAge COBOL

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

- ISO 1989:1985, Programming languages - COBOL

- The two ISO standards are identical to the American National Standards.

The following rules apply to syntax diagrams:

- Diagrams of syntactical units other than complete statements start with the  $\blacktriangleright$ — symbol and end with the  $\longrightarrow$  symbol.

- ▶▶ required\_item ◀◀

- ▶▶ required\_item [ optional\_item ] ▶▶

```

>> required_item [ required_choice1
                   | required_choice2 ] <<

```

```

>>—required_item—————>>
      [ optional_choice1
      [ optional_choice2

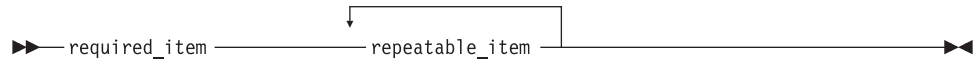
```

```

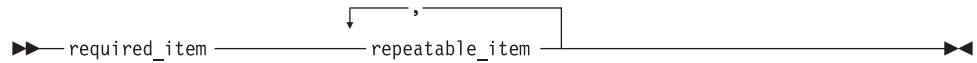
graph LR
    A[required_item] --- B[default_choice]
    A --- C[optional_choice]
    A --- D[optional_choice]
    B --- E[ ]
    C --- E
    D --- E
    E --- F[ ]
    style E width:0px,height:0px
    style F width:0px,height:0px

```

xii Programming Guide



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



A repeat arrow above a stack indicates that you can repeat the items in the stack.

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

---

## Related information

The information in this *Programming Guide* is available online in the Information Center both as topics integrated with other COBOL topics and as a PDF for viewing or printing. The Information Center also has all the COBOL language reference information and the information for using the COBOL components such as the editor and debugger, and has links to information for related products such as CICS and DB2. You might also want to check the list of resources online or in this book.

---

## How to send us your comments

Your feedback is important in helping to provide the most accurate and highest quality information. If you have any comments about this book or any other VisualAge COBOL documentation:

- Go to the VisualAge COBOL home page at: <http://www.ibm.com/software/ad/cobol>. There you will find the feedback page where you can enter and submit comments.
- Use the form at <http://www.ibm.com/software/ad/rcf/>. Be sure to include the name of the book, the part number of the book, the version of VisualAge COBOL, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).
- Fill out one of the forms at the back of this book and return it by mail, by fax, or by giving it to an IBM representative.



---

## Part 1. Coding your program

<b>Chapter 1. Structuring your program</b> . . . . .	5	<b>Chapter 3. Working with numbers and arithmetic</b>	31
Identifying a program . . . . .	5	Defining numeric data. . . . .	31
Identifying a program as recursive . . . . .	6	Displaying numeric data . . . . .	32
Marking a program as callable by containing programs . . . . .	6	Controlling how numeric data is stored . . . . .	33
Setting a program to an initial state. . . . .	6	Formats for numeric data. . . . .	34
Changing the header of a source listing . . . . .	6	External decimal (DISPLAY) items. . . . .	34
Describing the computing environment . . . . .	7	External floating-point (DISPLAY) items. . . . .	34
Example: FILE-CONTROL paragraph . . . . .	7	Binary (COMP) items . . . . .	35
Specifying the collating sequence . . . . .	8	Native binary (COMP-5) items . . . . .	35
Example: specifying the collating sequence . . . . .	8	Byte reversal of binary data . . . . .	36
Defining symbolic characters . . . . .	9	Packed-decimal (COMP-3) items . . . . .	36
Defining a user-defined class . . . . .	9	Floating-point (COMP-1 and COMP-2) items . . . . .	36
Identifying files to the operating system . . . . .	9	Examples: numeric data and internal representation	36
Varying the input or output file at run time	10	Data format conversions . . . . .	38
Describing the data. . . . .	11	Conversions and precision . . . . .	39
Using data in input and output operations . . . . .	11	Conversions that preserve precision . . . . .	39
FILE SECTION entries. . . . .	12	Conversions that result in rounding . . . . .	39
Comparison of WORKING-STORAGE and LOCAL-STORAGE . . . . .	12	Sign representation and processing . . . . .	40
Example: storage sections. . . . .	13	Checking for incompatible data (numeric class test)	40
Using data from another program . . . . .	13	Performing arithmetic . . . . .	41
Sharing data in separately compiled programs	14	COMPUTE and other arithmetic statements . . . . .	41
Sharing data in nested programs . . . . .	14	Arithmetic expressions . . . . .	41
Sharing data in recursive or multithreaded programs . . . . .	14	Numeric intrinsic functions . . . . .	42
Processing the data. . . . .	14	Nesting functions and arithmetic expressions . . . . .	43
How logic is divided in the PROCEDURE DIVISION . . . . .	15	ALL subscripting and special registers . . . . .	43
Imperative statements . . . . .	16	Examples: numeric intrinsic functions . . . . .	43
Conditional statements . . . . .	16	General number handling . . . . .	43
Compiler-directing statements . . . . .	17	Date and time . . . . .	43
Scope terminators . . . . .	17	Finance. . . . .	44
Declaratives . . . . .	19	Mathematics . . . . .	44
<b>Chapter 2. Using data</b> . . . . .	21	Statistics . . . . .	45
Using variables, structures, literals, and constants	21	Fixed-point versus floating-point arithmetic . . . . .	45
Variables . . . . .	21	Floating-point evaluations . . . . .	45
Data structure: data items and group items. . . . .	21	Fixed-point evaluations . . . . .	46
Literals . . . . .	22	Arithmetic comparisons (relation conditions) . . . . .	46
Constants . . . . .	22	Examples: fixed-point and floating-point evaluations . . . . .	46
Figurative constants . . . . .	22	Using currency signs . . . . .	47
Assigning values to data items . . . . .	23	Example: multiple currency signs . . . . .	48
Examples: initializing variables. . . . .	23	<b>Chapter 4. Handling tables.</b> . . . . .	51
Initializing a structure (INITIALIZE) . . . . .	24	Defining a table (OCCURS) . . . . .	51
Assigning values to variables or structures (MOVE) . . . . .	25	Nesting tables . . . . .	52
Assigning arithmetic results (MOVE or COMPUTE) . . . . .	26	Subscripting . . . . .	52
Assigning input from a screen or file (ACCEPT)	26	Indexing . . . . .	53
Displaying values on a screen or in a file (DISPLAY). . . . .	27	Referring to an item in a table . . . . .	53
Using intrinsic functions (built-in functions) . . . . .	27	Subscripting . . . . .	54
Types of intrinsic functions . . . . .	28	Indexing . . . . .	55
Nesting functions . . . . .	28	Putting values into a table . . . . .	56
Using tables (arrays) and pointers . . . . .	28	Loading a table dynamically. . . . .	56
		Initializing a table (INITIALIZE) . . . . .	56
		Assigning values when you define a table (VALUE) . . . . .	57
		Initializing each table item individually . . . . .	57
		Initializing a table at the 01 level . . . . .	57
		Initializing all occurrences of a table element	57

Example: PERFORM and subscripting . . . . .	58
Example: PERFORM and indexing. . . . .	59
Creating variable-length tables (DEPENDING ON) . . . . .	60
Loading a variable-length table. . . . .	61
Assigning values to a variable-length table . . . . .	62
Searching a table . . . . .	62
Doing a serial search (SEARCH) . . . . .	63
Example: serial search . . . . .	63
Doing a binary search (SEARCH ALL) . . . . .	64
Example: binary search . . . . .	64
Processing table items using intrinsic functions . . . . .	65
Example: intrinsic functions . . . . .	65

## Chapter 5. Selecting and repeating program actions . . . . .

actions . . . . .	67
Selecting program actions . . . . .	67
Coding a choice of actions . . . . .	67
Using nested IF statements . . . . .	68
Using the EVALUATE statement . . . . .	69
Coding conditional expressions. . . . .	71
Switches and flags . . . . .	72
Defining switches and flags . . . . .	72
Example: switches . . . . .	72
Example: flags . . . . .	73
Resetting switches and flags. . . . .	73
Example: set switch on . . . . .	73
Example: set switch off . . . . .	74
Repeating program actions . . . . .	74
Choosing inline or out-of-line PERFORM . . . . .	75
Example: inline PERFORM statement. . . . .	75
Coding a loop . . . . .	76
Coding a loop through a table . . . . .	76
Executing multiple paragraphs or sections . . . . .	77

## Chapter 6. Handling strings . . . . .

Joining data items (STRING) . . . . .	79
Example: STRING statement. . . . .	79
STRING program results . . . . .	80
Splitting data items (UNSTRING) . . . . .	81
Example: UNSTRING statement . . . . .	81
UNSTRING program results. . . . .	82
Manipulating null-terminated strings. . . . .	83
Example: null-terminated strings . . . . .	84
Referring to substrings of data items . . . . .	84
Reference modifiers. . . . .	85
Example: arithmetic expressions as reference modifiers . . . . .	86
Example: intrinsic functions as reference modifiers . . . . .	86
Tallying and replacing data items (INSPECT) . . . . .	87
Examples: INSPECT statement . . . . .	87
Converting data items (intrinsic functions) . . . . .	88
Converting to uppercase or lowercase (UPPER-CASE, LOWER-CASE). . . . .	88
Converting to reverse order (REVERSE) . . . . .	89
Converting to numbers (NUMVAL, NUMVAL-C) . . . . .	89
Evaluating data items (intrinsic functions) . . . . .	90
Evaluating single characters for collating sequence . . . . .	90
Finding the largest or smallest data item . . . . .	90
MAX and MIN . . . . .	91

ORD-MAX and ORD-MIN . . . . .	91
Returning variable-length results with alphanumeric functions . . . . .	91
Finding the length of data items . . . . .	92
Finding the date of compilation . . . . .	93

## Chapter 7. Processing files . . . . .

Identifying files . . . . .	95
Identifying Btrieve files . . . . .	96
Identifying STL files . . . . .	96
Identifying remote files . . . . .	96
File system . . . . .	97
STL file system . . . . .	97
STL file system return codes. . . . .	98
Protecting against errors when opening files . . . . .	100
Specifying a file organization and access mode . . . . .	100
File organization and access mode . . . . .	100
Sequential file organization. . . . .	101
Line-sequential file organization . . . . .	101
Indexed file organization . . . . .	101
Relative file organization . . . . .	102
Sequential access . . . . .	102
Random access . . . . .	102
Dynamic access. . . . .	103
File input-output limitations . . . . .	103
Setting up a field for file status . . . . .	104
Describing the structure of a file in detail . . . . .	104
Coding input and output statements for files. . . . .	104
Example: COBOL coding for files. . . . .	105
File position indicator . . . . .	106
Opening a file . . . . .	107
Valid COBOL statements for sequential files . . . . .	107
Valid COBOL statements for line-sequential files . . . . .	108
Valid COBOL statements for indexed and relative files . . . . .	108
Reading records from a file. . . . .	109
Statements used when writing records to a file . . . . .	110
Adding records to a file . . . . .	110
Adding records sequentially . . . . .	111
Adding records randomly or dynamically . . . . .	111
Replacing records in a file . . . . .	111
Deleting records from a file. . . . .	112
PROCEDURE DIVISION statements used to update files . . . . .	112

## Chapter 8. Sorting and merging files. . . . .

Sort and merge process . . . . .	115
Describing the sort or merge file . . . . .	116
Describing the input to sorting or merging . . . . .	116
Example: describing sort and input files for SORT . . . . .	117
Coding the input procedure . . . . .	118
Describing the output from sorting or merging . . . . .	118
Coding the output procedure . . . . .	119
Restrictions on input and output procedures . . . . .	119
Requesting the sort or merge . . . . .	120
Setting sort or merge criteria . . . . .	121
Choosing alternate collating sequences . . . . .	121

Example: sorting with input and output procedures . . . . .	121
Determining whether the sort or merge was successful . . . . .	122
Stopping a sort or merge operation prematurely	123
 <b>Chapter 9. Handling errors . . . . .</b>	<b>125</b>
Handling errors in joining and splitting strings . .	125
Handling errors in arithmetic operations . . . .	126
Example: checking for division by zero. . . .	126
Handling errors in input and output operations	127
Using the end-of-file condition (AT END) . . .	128
Coding ERROR declaratives . . . . .	129
Using file status keys. . . . .	129
Example: file status key . . . . .	130
Using file system return codes. . . . .	131
STL and Btrieve file systems . . . . .	131
VSAM file system . . . . .	131
Example: checking file system return codes	131
Coding INVALID KEY phrases . . . . .	132
INVALID KEY and ERROR declaratives . .	132
NOT INVALID KEY . . . . .	133
Example: FILE STATUS and INVALID KEY	133
Handling errors when calling programs . . . .	133



---

## Chapter 1. Structuring your program

A COBOL program consists of four divisions, each with a specific logical function:

- IDENTIFICATION DIVISION
- ENVIRONMENT DIVISION
- DATA DIVISION
- PROCEDURE DIVISION

Only the IDENTIFICATION DIVISION is required.

To define a COBOL class or method, you need to define some divisions differently than you would 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 14

"Defining a class" on page 274

"Defining a class method" on page 277

---

## Identifying a program

Use the IDENTIFICATION DIVISION to name your program and, if you want, give other identifying information.

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

IDENTIFICATION DIVISION.

Program-ID. Helloprog.

Author. A. Programmer.

Installation. Computing Laboratories.

Date-Written. 08/18/1997.

Date-Compiled. 09/30/2000.

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

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

**Tip:** When the 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 used.

### 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 (*IBM COBOL Language Reference*)

Conventions for program names (*IBM COBOL Language Reference*)

## Identifying a program as recursive

Code the `RECURSIVE` attribute on the `PROGRAM-ID` clause to specify that your 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 containing nested subprograms can be recursive.

#### RELATED TASKS

“Sharing data in recursive or multithreaded programs” on page 14

“Making recursive calls” on page 370

## Marking a program as callable by containing programs

Use the `COMMON` attribute on the `PROGRAM-ID` clause to specify that your 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 361

## Setting a program to an initial state

Use the `INITIAL` attribute to specify that whenever a program is called, it is placed in its initial state. If the program contains programs, these are also placed in their initial states.

A program is in its initial state when the following has occurred:

- Data items having `VALUE` clauses are set to the specified value.
- Changed `GO TO` statements and `PERFORM` statements are set to their initial states.
- Non-`EXTERNAL` files are closed.

## Changing the header of a source listing

The header on the first page of your source statement listing contains the identification of the compiler and the current release level, plus the date and time of compilation and the page number. Example:

```
PP 5639-B92 IBM VisualAge COBOL (Windows) 3.0.2      Date 09/30/2000  Time 15:05:19  Page    1
```

The header indicates the compilation platform used.

You can customize the header on succeeding pages of the listing with the compiler-directing `TITLE` statement.

#### RELATED REFERENCES

`TITLE` statement (*IBM COBOL Language Reference*)

---

## Describing the computing environment

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

Use the CONFIGURATION SECTION to specify the following items:

- Computer for compiling your program (in the SOURCE-COMPUTER paragraph)
- Computer for running your 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 do the following:

- Identify and describe the characteristics of your program files.
- Associate your files with the corresponding system file name, directly or indirectly.
- Optionally identify the file system (for example, VSAM or STL file system) associated with the file. You can also do this at run time.
- Provide information on how the file is 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 9

“Identifying files to the operating system” on page 9

### RELATED REFERENCES

Sections and paragraphs (*IBM COBOL 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 your file system. This example shows a FILE-CONTROL paragraph for a VSAM 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-VSAM-STATUS.
```

- (1) The SELECT clause chooses a file in the COBOL program to be associated with a corresponding system file.
- (2) The ASSIGN clause associates the program’s name for the file 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) Use the ORGANIZATION clause to describe the file’s organization. If omitted, the default is ORGANIZATION IS SEQUENTIAL.

- (4) Use the ACCESS MODE clause to define the manner in which the records in the file will be 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

Use the PROGRAM COLLATING SEQUENCE clause and the ALPHABET clause of the SPECIAL-NAMES paragraph to establish the collating sequence used in the following operations:

- 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 on the SORT or MERGE statement

“Example: specifying the collating sequence”

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

- NATIVE  
NATIVE is 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.
- EBCDIC
- ASCII (use NATIVE if the native character set is ASCII, STANDARD-1 if it is not; STANDARD-1 refers to *American National Standard X3.4, Code for Information Interchange*)
- ISO 7-bit code (as defined in *International 646, 7-Bit Coded Character Set for Information Processing Interchange, International Reference*, International Reference Version (use STANDARD-2))
- An alteration of the ASCII sequence that you define in the SPECIAL-NAMES paragraph

You can also specify a collating sequence of your own definition.

**Restriction:** If the code page is DBCS, you cannot use the ALPHABET-NAME clause.

#### RELATED TASKS

“Choosing alternate collating sequences” on page 121

### Example: specifying the collating sequence

The following example shows the ENVIRONMENT DIVISION coding used to specify a collating sequence where uppercase and lowercase letters are similarly handled for comparisons and for sorting or 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 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 CHARACTER clause to give symbolic names to any character of the specified alphabet. Use ordinal position to identify the character. Position 1 corresponds to character X'00'. Example: To give a name to the plus character (X'2B' in the ASCII alphabet), code:

```
SYMBOLIC CHARACTERS PLUS IS 44
```

## Defining a user-defined class

Use the CLASS clause to give a name to a set of characters listed in the clause. For example, name the set of digits by using this code:

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

The class name can be referenced only in a class condition. This user-defined class is not the same as an object-oriented class.

## Identifying files to the operating system

The ASSIGN clause associates the program's name for a file 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, its value is evaluated at run time and is used as the system file name with optional directory and path names.

If you plan to use a file system other than the default file system, you need to select the file system explicitly, for example, by specifying the file system identifier before the system file name. For example, if the file MYFILE is a Btrieve file and you use F1 as the file's name in your program, the ASSIGN clause would be

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

This code assumes that MYFILE is a system file name and not an environment variable. If MYFILE is an environment variable, then its value will be used. For example, if it is set to MYFILE=VSAM-YOURFILE, the system file name in the ASSIGN clause becomes YOURFILE at run time, and the file is treated as a VSAM file, overriding the file system ID used in the ASSIGN clause in the program.

#### RELATED TASKS

"Varying the input or output file at run time"

### Varying the input or output file at run time

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

Changing a *file-name* in your COBOL program requires changing the input statements and output statements and recompiling the program. Alternatively, you can change the *assignment-name* in your SET command.

Environment variable values in effect at the time of the program invocation 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:** Consider a COBOL program that is used in exactly the same way for several different master files. It contains this SELECT clause:

```
SELECT MASTER  
  ASSIGN TO MASTERA
```

Suppose you want the program to access either the checking or savings file using the same MASTER file. To do so, set the MASTERA environment variable (before the program runs) by using one of the following two statements as appropriate, assuming 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 by way of the COBOL MASTER file 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:

- Define data used in input-output operations (FILE SECTION).
- Define 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 called and deallocated when the program ends (LOCAL-STORAGE SECTION).
- Describe data from another program (LINKAGE SECTION).

The VisualAge COBOL compiler limits the maximum size of DATA DIVISION elements.

### RELATED CONCEPTS

“Comparison of WORKING-STORAGE and LOCAL-STORAGE” on page 12

### RELATED TASKS

“Using data in input and output operations”

“Using data from another program” on page 13

### RELATED REFERENCES

Compiler limits (*IBM COBOL Language Reference*)

## Using data in input and output operations

Define the data you use in input and output operations in the FILE SECTION:

- Name the input and output files your program will use. Use the FD entry to give names to your 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 following the FD entry, describe the records and their fields in the file. The record-name established 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 361

#### RELATED TASKS

“Sharing files between programs (external files)” on page 383

#### RELATED REFERENCES

“FILE SECTION entries”

### 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).
RECORD IS VARYING	Size of logical records (variable length).
RECORD CONTAINS <i>n</i> TO <i>m</i>	Size of logical records (variable length).
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.

## Comparison of WORKING-STORAGE and LOCAL-STORAGE

When a program is invoked, the WORKING-STORAGE associated with the program is allocated. Any data items with 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 on the PROGRAM-ID  
In this case, WORKING-STORAGE data items are reinitialized each time the program is entered.
- A subprogram that is 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.

LOCAL-STORAGE is allocated each time a program is called, and is deallocated when the program returns by means of an EXIT PROGRAM, GOBACK, or STOP RUN. Note however that for nested programs, LOCAL-STORAGE is allocated upon entry to, and deallocated upon exit from, the containing outermost program. Any LOCAL-STORAGE data items with VALUE clauses in a nested program are initialized to the appropriate value each time the nested program is called.

#### RELATED CONCEPTS

“Example: storage sections” on page 13

#### RELATED TASKS

“Ending and reentering main programs or subprograms” on page 360

#### RELATED REFERENCES

Working-Storage section (*IBM COBOL Language Reference*)

Local-Storage section (*IBM COBOL 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(1u)
*****
* 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 following tables show the changing values of the data items in LOCAL-STORAGE (L-S) and WORKING-STORAGE (W-S) 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:		Main	1	2	3	4	5
L-S	num	5	4	3	2	1	0
<hr/>							
W-S	numb	5	4	3	2	1	0
	fact	0	0	0	0	0	0
<hr/>							
Recursive							
GOBACKs:		5	4	3	2	1	Main
L-S	num	0	1	2	3	4	5
<hr/>							
W-S	numb	0	0	0	0	0	0
	fact	1	1	2	6	24	120
<hr/>							

#### RELATED CONCEPTS

“Comparison of WORKING-STORAGE and LOCAL-STORAGE” on page 12

## Using data from another program

How you share data depends on whether the programs are separately compiled or are nested, as discussed in the topics referenced below.

#### RELATED TASKS

“Sharing data in separately compiled programs”

“Sharing data in nested programs”

“Sharing data in recursive or multithreaded programs”

### 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 373

### Sharing data in nested programs

Some applications consist of nested programs—programs that are contained in other programs. Level-01 LINKAGE SECTION data items can include the GLOBAL attribute. This attribute allows any nested program that includes the declarations to access these LINKAGE SECTION 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 361

### Sharing data in recursive or multithreaded programs

If you compile your program as RECURSIVE or with the THREAD option, data defined in the LINKAGE SECTION might not be accessible between entries.

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 you compile your program as RECURSIVE or with the THREAD option, the address to that record is valid for a particular instance of the program invocation. The address to the record in another execution instance of the same program must be reestablished for that execution instance. Unpredictable results will occur if you make reference to a data item whose address has not been established.

#### RELATED CONCEPTS

“Multithreading” on page 403

#### RELATED TASKS

“Making recursive calls” on page 370

#### RELATED REFERENCES

SET statement (*IBM COBOL Language Reference*)

---

## Processing the data

In the PROCEDURE DIVISION of a program, you code the executable statements that process the data you have 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 your division header to receive parameters with the USING phrase or to return a value with 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 the *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 of the DATA DIVISION.

#### RELATED CONCEPTS

“How logic is divided in the PROCEDURE DIVISION”

#### RELATED TASKS

“Eliminating repetitive coding” on page 463

## How logic is divided in the PROCEDURE DIVISION

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

### 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 ending with a period.

Many structured programs do not have separate sentences. Each paragraph can contain one sentence.

**Statement**

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

A statement is a valid combination of words, beginning 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 17

“Scope terminators” on page 17

“Imperative statements”

“Conditional statements”

“Declaratives” on page 19

**RELATED REFERENCES**

PROCEDURE DIVISION structure (*IBM COBOL Language Reference*)

**Imperative statements**

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

An imperative statement can be ended 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 17

**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 statement 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 16

“Scope terminators”

#### RELATED TASKS

“Selecting program actions” on page 67

#### RELATED REFERENCES

Conditional statements (*IBM COBOL Language Reference*)

## Compiler-directing statements

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

#### RELATED REFERENCES

“Compiler-directing statements” on page 198

Compiler-directing statements (*IBM COBOL Language Reference*)

## Scope terminators

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 16

“Imperative statements” on page 16

## 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 223

#### RELATED REFERENCES

Declaratives (*IBM COBOL Language Reference*)



---

## Chapter 2. Using data

This section is intended to help the non-COBOL programmer relate terms used in other programming languages to COBOL terms for data. It introduces COBOL fundamentals for:

- Variables, structures, literals, and constants
- Assigning and displaying values
- Intrinsic (built-in) functions
- Tables (arrays) and pointers

### RELATED TASKS

“Using variables, structures, literals, and constants”

“Assigning values to data items” on page 23

“Using intrinsic functions (built-in functions)” on page 27

“Using tables (arrays) and pointers” on page 28

---

## Using variables, structures, literals, and constants

Most high-level programming languages share the concept of data being represented as variables, structures, literals, and constants. You place all data definitions in the DATA DIVISION of your program. The data in a COBOL program can be alphabetic, alphanumeric, or numeric.

### Variables

A variable is a data item whose value can change during a program. The values are restricted, however, to the data type that you define when you give the variable a name and a length. For example, if a customer name is a variable in your program, you could code:

Data Division.

```
. . .  
01 Customer-Name           Pic X(20).  
01 Original-Customer-Name  Pic X(20).  
. . .
```

Procedure Division.

```
. . .  
    Move Customer-Name to Original-Customer-Name  
. . .
```

### Data structure: data items and group items

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

In this example, Customer-Record is a group item composed of two group items (Customer-Name and Part-Order), each of which contains elementary data items. You can refer to the entire group item or to parts of the group item as shown in the MOVE statements in the PROCEDURE DIVISION.

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
. . .

```

## Literals

A *literal* is a character string whose value is given by the characters themselves. When you know the value you want to use for a data item, use a literal representation of the data value in the PROCEDURE DIVISION. You do not need to define it or refer to a data-name. For example, you can prepare an error message for an output file this way:

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

You can compare a data item to a certain number this way:

```

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

```

In these examples, "Name is not valid" is a nonnumeric literal, and 03519 is a numeric literal.

## Constants

A *constant* is a data item that has only one value. COBOL does not define a construct specifically for constants. However, most COBOL programmers define a data item with an initial VALUE (as opposed to initializing a variable using the INITIALIZE statement). For example:

```

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

```

## Figurative constants

A *variable* is a data item whose value can change during a program. Certain commonly used constants and literals are provided as reserved words called *figurative constants*: ZERO, SPACE, HIGH-VALUE, LOW-VALUE, QUOTE, NULL, and ALL. Because they represent fixed values, figurative constants do not require a data definition. For example:

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

### RELATED TASKS

"Setting environment variables" on page 137

#### RELATED REFERENCES

PICTURE clause (*IBM COBOL Language Reference*)

Literals (*IBM COBOL Language Reference*)

Figurative constants (*IBM COBOL 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.

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"><li>• INITIALIZE statement</li><li>• MOVE statement</li><li>• STRING or UNSTRING statement</li><li>• VALUE clause (to set data items to the values you want them to have when the program is in its initial state)</li></ul>
Assign the results of arithmetic	Use the COMPUTE statement.
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 screen or a file	Use the ACCEPT statement.
Display values on a screen or in a file	Use the DISPLAY statement.

“Examples: initializing variables”

#### RELATED TASKS

“Initializing a structure (INITIALIZE)” on page 24

“Assigning values to variables or structures (MOVE)” on page 25

“Assigning arithmetic results (MOVE or COMPUTE)” on page 26

“Assigning input from a screen or file (ACCEPT)” on page 26

“Displaying values on a screen or in a file (DISPLAY)” on page 27

## Examples: initializing variables

**Initializing a variable 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	00000 <sup>1</sup>
99XX9	12AB3	00000 <sup>1</sup>
XXBX/XX	ABbC/DE	0000/00 <sup>1</sup>
**99.9CR	1234.5CR	**00.000 <sup>1</sup>
A(5)	ABCDE	00000 <sup>1</sup>
+99.99E+99	+12.34E+02	+00.00E+00

1. The symbol 0 represents a blank space.

**Initializing a right-justified field:**

```

01 ANJUST          PIC X(8)  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	00000000 <sup>1</sup>	0000ABCD <sup>1</sup>
1. The symbol 0 represents a blank space.		

#### Initializing an alphanumeric variable:

```

01 ALPHANUMERIC-1  PIC X.
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 a numeric variable:

```

01 NUMERIC-1       PIC 9(8).
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 an edited alphanumeric variable:

```

01 ALPHANUM-EDIT-1 PIC XXBX/XXX.
01 ALPHANUM-EDIT-3 PIC X/BB VALUE "M/000".
. . .
  INITIALIZE ALPHANUM-EDIT-1
    REPLACING ALPHANUMERIC-EDITED DATA BY ALPHANUM-EDIT-3

```

ALPHANUM-EDIT-3	ALPHANUM-EDIT-1 before	ALPHANUM-EDIT-1 after
M/00 <sup>1</sup>	AB0C/DEF <sup>1</sup>	M/00/000 <sup>1</sup>
1. The symbol 0 represents a blank space.		

#### RELATED TASKS

“Initializing a structure (INITIALIZE)”

## Initializing a structure (INITIALIZE)

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

The following example shows how you can reset fields in a transaction record produced by a program to spaces and zeros. The fields are not identical in each record produced.

```

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	R001383000240000000000000000000	900000000000000000000000000000 <sup>1</sup>
2	R001390000480000000000000000000	900000000000000000000000000000 <sup>1</sup>
3	S001410000120000000000000000000	900000000000000000000000000000 <sup>1</sup>
4	C00138300000000004250000000000	900000000000000000000000000000 <sup>1</sup>
5	C00201000000000000001000000000	900000000000000000000000000000 <sup>1</sup>

1. The symbol 9 represents a blank space.

## Assigning values to variables or structures (MOVE)

Use the MOVE statement to assign values to variables or structures.

For example, the following statement assigns the contents of the variable Customer-Name to the variable Orig-Customer-Name:

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

If Customer-Name were longer than Orig-Customer-Name, truncation would occur on the right. If it were shorter, the extra character positions on the right would be filled with spaces.

When you move a group item to another group item, be sure the subordinate data descriptions are compatible. The compiler performs all MOVE statements regardless of whether the items fit, even if a destructive overlap could occur at run time.

For variables containing numbers, moves can be more complicated because there are several ways numbers are represented. In general, the algebraic values of numbers are moved if possible (as opposed to the digit-by-digit move performed with character data):

```

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

```

This move would result in Item-x containing the value 3.0, represented by 0030.

```
CBL CODEPAGE(00875)
```

```

. . .
01 Data-in-Unicode Pic N(100) usage national.
01 Data-in-Greek   Pic N(100).

```

```

      Read Greek-file into Data-in-Greek
      Move data-in-Greek to Data-in-Unicode

```

### RELATED REFERENCES

MOVE statement (*IBM COBOL Language Reference*)

## Assigning arithmetic results (MOVE or COMPUTE)

When assigning a number to a variable, consider using the COMPUTE statement instead of the MOVE statement. For example, the following two statements accomplish the same thing in most cases:

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

The MOVE statement carries out the assignment with truncation.

When significant left-order digits would be lost in execution, the COMPUTE statement can detect the condition and allow you to handle it.

When 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 the ON SIZE ERROR phrase is not specified, 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 variable. For example:

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

### RELATED REFERENCES

Intrinsic functions (*IBM COBOL Language Reference*)

## Assigning input from a screen or file (ACCEPT)

One way to assign a value to a variable 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 variable.

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 using the SET command. For example:  
SET CONSOLE=\myfiles\myinput.rpt

The environment variable must be the same as the system device used. In the above example, 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 SYSIN also.

#### RELATED TASKS

“Setting environment variables” on page 137

#### RELATED REFERENCES

SPECIAL-NAMES paragraph (*IBM COBOL Language Reference*)

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

You can display the value of a variable on a screen or write it to a file using the DISPLAY statement. For example:

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

If the content of the variable *Customer-Name* is JOHNSON, then the statement above 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 system logical output device, use the UPON clause. For example, the following statement writes to the file specified in the SYSOUT environment variable if defined:

```
Display "Hello" UPON SYSOUT.
```

#### RELATED REFERENCES

DISPLAY statement (*IBM COBOL Language Reference*)

---

## Using intrinsic functions (built-in functions)

Some high-level programming languages have built-in functions that you can reference in your program as if they were variables having defined attributes and a predetermined value. In COBOL, these 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).

The groups of highlighted words in the following examples are referred to as *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 having 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 variable. For example, you could use Sqrt to invoke an intrinsic function and to name a variable 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)
. . .

```

## Types of intrinsic functions

A function-identifier represents a value that is either a character string (alphanumeric data class) or a number (numeric data class) depending on the type of function. You can include a substring specification (reference modifier) in a function-identifier for alphanumeric 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.

## Nesting functions

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:

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

In this case, Function Sqrt(5) returns a numeric value. Thus, the three arguments to the MAX function are all numeric, which is an allowable argument type for this function.

### RELATED TASKS

“Processing table items using intrinsic functions” on page 65

“Converting data items (intrinsic functions)” on page 88

“Evaluating data items (intrinsic functions)” on page 90

---

## 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 explicitly with the USAGE IS POINTER clause in the DATA DIVISION or implicitly as ADDRESS OF special registers.

You can perform the following operations on 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 address that is not valid 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.

A procedure pointer is a pointer to an entry point of a procedure. Define the entry address for a procedure with the USAGE IS PROCEDURE-POINTER clause in the DATA DIVISION.

#### RELATED TASKS

“Defining a table (OCCURS)” on page 51



---

## 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.

This section describes how 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 32

"Controlling how numeric data is stored" on page 33

"Checking for incompatible data (numeric class test)" on page 40

"Performing arithmetic" on page 41

"Using currency signs" on page 47

---

## 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 items:

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

You can code up to 18 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 it 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 as a separate position in storage. In this 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 data items that will be printed or displayed.

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 a floating-point literal:

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

“Examples: numeric data and internal representation” on page 36

#### RELATED CONCEPTS

“Appendix C. Intermediate results and arithmetic precision” on page 481

#### RELATED TASKS

“Displaying numeric data”

“Controlling how numeric data is stored” on page 33

“Performing arithmetic” on page 41

#### RELATED REFERENCES

“Sign representation and processing” on page 40

---

## 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 data easier to read and understand when you display it or print it. For example, in the code below, Edited-price is a numeric-edited item:

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

If the contents of Price are 0150099 (representing the value 1,500.99), then \$ 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 cannot use numeric-edited items as operands in arithmetic expressions or in ADD, SUBTRACT, MULTIPLY, DIVIDE, or COMPUTE statements. You use numeric-edited items primarily for displaying or printing numeric data. (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.)

You can move numeric-edited items to numeric or numeric-edited items. In the following example, the value of the numeric-edited item is moved to the numeric item:

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

If these two statements immediately followed the statements in the previous example, then Price would be displayed as 0150099, representing the value 1,500.99.

“Examples: numeric data and internal representation” on page 36

#### RELATED TASKS

“Controlling how numeric data is stored”

“Defining numeric data” on page 31

“Performing arithmetic” on page 41

#### RELATED REFERENCES

MOVE statement (*IBM COBOL 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 on computational data types is more efficient than on USAGE DISPLAY data types.
- Packed-decimal format requires less storage per digit than USAGE DISPLAY data types.
- Packed-decimal format converts to and from DISPLAY 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)
- External floating point (USAGE DISPLAY)
- Internal decimal (USAGE PACKED-DECIMAL)
- Binary (USAGE BINARY)
- Native binary (USAGE COMP-5)
- Internal floating point (USAGE COMP-1, 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 rather than as DISPLAY. 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 floating-point data, for which you cannot use a PICTURE clause).

“Examples: numeric data and internal representation” on page 36

#### RELATED CONCEPTS

“Formats for numeric data”

“Data format conversions” on page 38

“Appendix C. Intermediate results and arithmetic precision” on page 481

#### RELATED TASKS

“Defining numeric data” on page 31

“Displaying numeric data” on page 32

“Performing arithmetic” on page 41

#### RELATED REFERENCES

“Conversions and precision” on page 39

“Sign representation and processing” on page 40

---

## Formats for numeric data

The following are the available formats for numeric data.

### External decimal (DISPLAY) items

When USAGE DISPLAY is in effect for a data item (either because you have coded it, or by default), each position (byte) of storage contains one decimal digit. This means the items are stored in displayable form.

External decimal (also known as *zoned decimal*) 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) items

Displayable numbers coded in a floating-point format are called *external floating-point items*. As with external decimal items, you define external floating-point items explicitly by coding USAGE DISPLAY or implicitly by omitting the USAGE clause. You cannot use the VALUE clause for external floating-point items.

In the following example, Compute-Result is implicitly defined as an external floating-point item. Each byte of storage contains one of the characters (except for the v).

```
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.

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.

## Binary (COMP) items

BINARY, COMP, and COMP-4 are synonyms on all platforms.

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

COMP-5 is a USAGE type based on the X/OPEN COBOL specification.

Data items that you declare as USAGE COMP-5 are represented in storage as binary data. 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 originating in non-COBOL programs where the data might not conform to a COBOL PICTURE clause.

The table below shows the ranges of values possible for 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.

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 the 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). System/390 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 or 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 on all platforms.

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.

## 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 is in effect.

### RELATED CONCEPTS

"Appendix C. Intermediate results and arithmetic precision" on page 481

### RELATED REFERENCE

"`TRUNC`" on page 192

"`BINARY`" on page 161

"`FLOAT`" on page 176

---

## Examples: numeric data and internal representation

This table shows the internal representation of numeric items for native data types. Assume that the `BINARY(NATIVE)`, `CHAR(NATIVE)`, and `FLOAT(NATIVE)` compiler options are in effect.

Numeric type	PICTURE and USAGE and optional SIGN clause	Value	Internal representation
External decimal	PIC S9999 DISPLAY	+ 1234 - 1234 1234	31 32 33 34 71 32 33 34 31 32 33 34
	PIC 9999 DISPLAY	1234	31 32 33 34
	PIC S9999 DISPLAY SIGN LEADING	+ 1234 - 1234	31 32 33 34 71 32 33 34
	PIC S9999 DISPLAY SIGN LEADING SEPARATE PIC S9999 DISPLAY SIGN TRAILING SEPARATE	+ 1234 - 1234 + 1234 - 1234	2B 31 32 33 34 2D 31 32 33 34 31 32 33 34 2B 31 32 33 34 2D
Binary	PIC S9999 BINARY COMP COMP-4	+ 1234 - 1234	D2 04 2E FB
	COMP-5	+ 1234 - 1234	D2 04 2E FB
	PIC 9999 BINARY COMP COMP-4	1234	D2 04
	COMP-5	1234	D2 04
Internal decimal	PIC S9999 PACKED-DECIMAL COMP-3	+ 1234 - 1234	01 23 4C 01 23 4D
	PIC 9999 PACKED-DECIMAL COMP-3	1234	01 23 4F
Internal floating point	COMP-1	+ 1234 - 1234	00 40 9A 44 00 40 9A C4
Internal floating point	COMP-2	+ 1234 - 1234	00 00 00 00 00 48 93 40 00 00 00 00 00 48 93 C0
External floating point	PIC +9(2).9(2)E+99 DISPLAY	+ 1234	2B 31 32 2E 33 34 45 2B 30 32
		- 1234	2D 31 32 2E 33 34 45 2B 30 32

This table shows the internal representation of numeric items for System/390 data types. Assume that the BINARY(S390), CHAR(EBCDIC), and FLOAT(HEX) compiler options are in effect.

Numeric type	PICTURE and USAGE and optional SIGN clause	Value	Internal representation
External decimal	PIC S9999 DISPLAY	+ 1234 - 1234 1234	F1 F2 F3 C4 F1 F2 F3 D4 F1 F2 F3 C4
	PIC 9999 DISPLAY	1234	F1 F2 F3 F4
	PIC S9999 DISPLAY SIGN LEADING	+ 1234 - 1234	C1 F2 F3 F4 D1 F2 F3 F4
	PIC S9999 DISPLAY SIGN LEADING SEPARATE PIC S9999 DISPLAY SIGN TRAILING SEPARATE	+ 1234 - 1234 + 1234 - 1234	4E F1 F2 F3 F4 60 F1 F2 F3 F4 F1 F2 F3 F4 4E F1 F2 F3 F4 60
Binary	PIC S9999 BINARY COMP COMP-4	+ 1234 - 1234	04 D2 FB 2E
	COMP-5	+ 1234 - 1234	D2 04 2E FB
	PIC 9999 BINARY COMP COMP-4	1234	04 D2
	COMP-5	1234	D2 04
Internal decimal	PIC S9999 PACKED-DECIMAL COMP-3	+ 1234 - 1234	01 23 4C 01 23 4D
	PIC 9999 PACKED-DECIMAL COMP-3	1234	01 23 4F
Internal floating point	COMP-1	+ 1234 - 1234	43 4D 20 00 C3 4D 20 00
Internal floating point	COMP-2	+ 1234 - 1234	43 4D 20 00 00 00 00 00 C3 4D 20 00 00 00 00 00
External floating point	PIC +9(2).9(2)E+99 DISPLAY	+ 1234	4E F1 F2 4B F3 F4 C5 4E F0 F2
		- 1234	60 F1 F2 4B F3 F4 C5 4E F0 F2

## Data format conversions

When the code in your program involves the interaction of items with different data formats, the compiler converts these items as follows:

- Temporarily, for comparisons and arithmetic operations
- 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 as opposed to 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 and processing” on page 40

## Conversions and precision

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 also 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.

When a USAGE COMP-1 data item is moved to a fixed-point data item with 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 with 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 with 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.

---

## Sign representation and processing

Sign representation affects the processing and interaction of your numeric data.

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

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) COBOL data are:

**Positive:**

A, C, E, and F

**Negative:**

B and D

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

---

## Checking for incompatible data (numeric class test)

The compiler assumes that the values you supply for a data item are valid for the item's PICTURE and USAGE clauses, and assigns the values without checking for validity. 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 will be unpredictable.

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 both cases, the receiving fields contain invalid data. Ensure that the contents of a data item conform to its PICTURE and USAGE clauses before using the data item in any further processing steps.

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 particular PICTURE and USAGE of the data item.

## Performing arithmetic

You can use any of several COBOL language features to perform arithmetic:

- COMPUTE, ADD, SUBTRACT, MULTIPLY, and DIVIDE statements
- Arithmetic expressions
- Intrinsic functions

### 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 one COMPUTE statement instead of several individual 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 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.

### 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:

Operator	Meaning	Order of evaluation
Unary + or -	Algebraic sign	First

Operator	Meaning	Order of evaluation
**	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, necessary or not, make your program easier to read.

## 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. They are treated as temporary numeric data items.

Numeric functions are classified into these categories:

### Integer

Those that return an integer

### Floating point

Those that return a long (64-bit) floating-point value

**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.

Number handling	Date and time	Finance	Mathematics	Statistics
LENGTH MAX MIN NUMVAL NUMVAL-C ORD-MAX ORD-MIN	CURRENT-DATE DATE-OF-INTEGER DATE-TO-YYYYMMDD DATEVAL DAY-OF-INTEGER 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” on page 43

## Nesting functions and arithmetic expressions

You can reference one function as the argument of another. A nested function is evaluated independently of the outer function, except when determining 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:

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

In this example, there are only three function arguments: a, b, and the arithmetic expression (c / d).

## ALL subscripting and special registers

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

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

### RELATED CONCEPTS

“Fixed-point versus floating-point arithmetic” on page 45

“Appendix C. Intermediate results and arithmetic precision” on page 481

---

## Examples: numeric intrinsic functions

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

### General number handling

Suppose you want to find the maximum value of two prices (represented 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 string) and the MAX and LENGTH functions to do this:

```
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)
```

---

## Fixed-point versus floating-point arithmetic

Many statements in your 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)
```

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

## 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.

A data item is floating point if you code it as a floating-point literal or if you define it as USAGE COMP-1, USAGE COMP-2, or external floating point (USAGE DISPLAY 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 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 containing decimal places, or use an arithmetic expression or function whose result has decimal places.

An arithmetic expression or numeric function yields a result with 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 will cause 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 488

## Examples: fixed-point and floating-point evaluations

Assume 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 one or more of the following signs in a program:

- Symbols such as the dollar sign (\$)
- Currency signs of more than one character (such as USD, DEM, 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 the 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. This 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 hex 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.

Code page	Euro sign
IBM-1250 (Latin 2)	X'80'
IBM-1251 (Cyrillic)	X'88'
IBM-1252 (Latin 1)	X'80'
IBM-1253 (Greek)	X'80'
IBM-1254 (Turkish)	X'80'
IBM-1255 (Hebrew)	X'80'
IBM-1256 (Arabic)	X'80'
IBM-1257 (Baltic)	X'80'
IBM-1258 (Vietnamese)	X'80'
IBM-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 French francs (as FRF):

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EuroExample.
Environment Division.
Configuration Section.
Special-Names.
    Currency Sign is "FRF " 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-FRF      Pic S99999V99.
01 Deposit-Report.
    02 Report-in-Franc  Pic -FFFFFF9.99.
    02 Report-in-Euro   Pic -UUUUU9.99.
. . .
01 EUR-to-FRF-Conv-Rate Pic 9V99999 Value 6.78901.
. . .
PROCEDURE DIVISION.
Report-Deposit-in-FRF-and-EUR.
    Move Deposit-in-Euro to Report-in-Euro
    . . .
    Compute Deposit-in-FRF Rounded
        = Deposit-in-Euro * EUR-to-FRF-Conv-Rate
    On Size Error
        Perform Conversion-Error
    Not On Size Error
        Move Deposit-in-FRF 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 FRF"
    Display "Euro value: " Report-in-Euro.
```

The above example produces the following display output:

Deposit in Euro = EUR 8000.00  
Deposit in Franc = FRF 54312.08

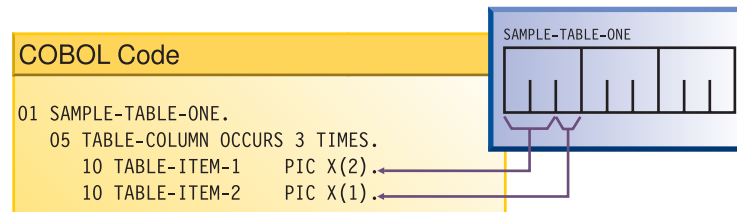
The exchange rate used in this example is for illustrative purposes only.



---

## Chapter 4. Handling tables

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



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 define repetitious items as separate, consecutive entries in the DATA DIVISION, you can 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

"Defining a table (`OCCURS`)"

"Referring to an item in a table" on page 53

"Putting values into a table" on page 56

"Nesting tables" on page 52

"Creating variable-length tables (`DEPENDING ON`)" on page 60

"Searching a table" on page 62

"Processing table items using intrinsic functions" on page 65

"Handling tables efficiently" on page 455

---

## Defining a table (`OCCURS`)

To code a table, give the table a group name and define a subordinate item (the *table element*) that is to be repeated *n* times. *table-name* is the group name in the following example:

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

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.

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

#### RELATED TASKS

“Creating variable-length tables (DEPENDENT ON)” on page 60

“Nesting tables”

“Putting values into a table” on page 56

“Referring to an item in a table” on page 53

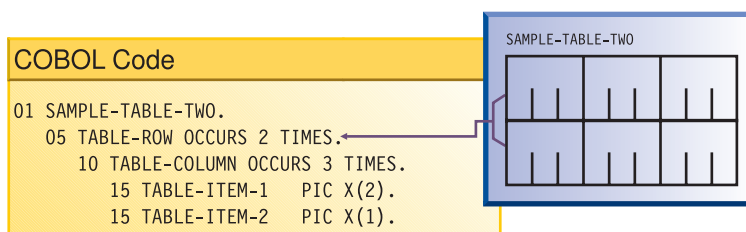
“Searching a table” on page 62

#### RELATED REFERENCES

OCCURS clause (*IBM COBOL Language Reference*)

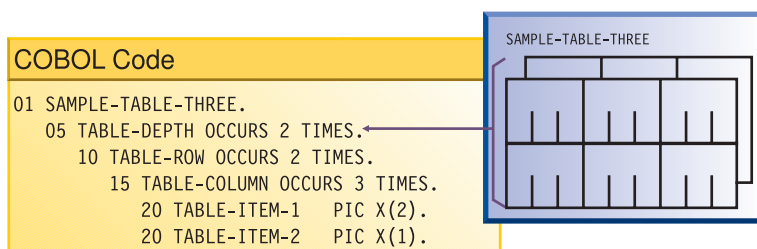
## Nesting tables

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, 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.

## Subscripting

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.

The following valid references to SAMPLE-TABLE-THREE 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)
```

## Indexing

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

“Defining a table (OCCURS)” on page 51

“Referring to an item in a table”

“Putting values into a table” on page 56

“Creating variable-length tables (DEPENDING ON)” on page 60

“Searching a table” on page 62

“Processing table items using intrinsic functions” on page 65

“Handling tables efficiently” on page 455

### RELATED REFERENCES

OCCURS clause (*IBM COBOL Language Reference*)

---

## 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 (the 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

“Indexing” on page 55

“Subscripting”

## Subscripting

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

You can use a data name or a literal for a subscript.

If a data item with 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 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. Simply refer to the character position and length of the substring to be changed within the subscripted element. 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 moves the value ?? into table element number 1, beginning at character position 3, for a length of 2:

ANY-TABLE

before the change:

A	B	C	D	E	F	G	H	I	J
A	B	C	D	E	F	G	H	I	J
A	B	C	D	E	F	G	H	I	J

ANY-TABLE
after the change:
AB??EFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ

#### RELATED TASKS

“Indexing” on page 55

“Putting values into a table” on page 56  
“Searching a table” on page 62  
“Handling tables efficiently” on page 455

## Indexing

You can create an index either with a particular table (using OCCURS INDEXED BY) or separately (using USAGE IS INDEX).

For example:

```
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 this index to index another table only if both table descriptions have the same number of table elements, and the table elements are the same length.

If you use USAGE IS INDEX to create an index, you can use the index data item with any table. For example:

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

INX-B is used to traverse table TABLE-ITEM above, but could be used to traverse other tables also.

You can increment or decrement an index name by an unsigned numeric literal. The literal is considered to be an occurrence number. It is converted to an index value before being added to or subtracted from the index name.

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

Use the SET statement to assign to an index the value that you stored in the index data item defined by USAGE IS INDEX. For example, when you load records into a variable-length table, you can store the index value of the last record read in 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 the table.

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

### RELATED TASKS

“Subscripting” on page 54  
“Putting values into a table” on page 56

“Searching a table” on page 62  
“Processing table items using intrinsic functions” on page 65  
“Handling tables efficiently” on page 455

#### RELATED REFERENCES

INDEXED BY phrase (*IBM COBOL Language Reference*)  
INDEX phrase (*IBM COBOL Language Reference*)

---

## Putting values into a table

Use one of these methods to put values into a table:

- Load the table dynamically.
- Initialize the table (INITIALIZE statement).
- Assign values when you define the table (VALUE clause).

#### RELATED TASKS

“Loading a table dynamically”  
“Loading a variable-length table” on page 61  
“Initializing a table (INITIALIZE)”  
“Assigning values when you define a table (VALUE)” on page 57  
“Assigning values to a variable-length table” on page 62

## Loading a table dynamically

If the initial values of your table are different with each execution of your program, you can define the table without initial values. You can then read the changed values into the table before your 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 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 58  
“Example: PERFORM and indexing” on page 59

#### RELATED REFERENCES

PERFORM with VARYING phrase (*IBM COBOL Language Reference*)

## Initializing a table (INITIALIZE)

You can load your table with a value during execution with the INITIALIZE statement. For example, to fill a table with 3s, you can use this code:

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

The INITIALIZE statement cannot load a variable-length table (one that was defined using OCCURS DEPENDING ON).

#### RELATED REFERENCES

INITIALIZE statement (*IBM COBOL Language Reference*)

## Assigning values when you define a table (VALUE)

If your table contains stable values (such as days and months), set the specific values your table holds when you define it.

Define static values in WORKING-STORAGE in one of these ways:

- Initialize each table item individually.
- Initialize an entire table at the 01 level.
- Initialize all occurrences of a given table element to the same value.

### Initializing each table item individually

If your table is small, you can use this technique:

1. Declare a record that contains the same items as are in your 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.

For example:

```
*****
***      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 this example, the items could all be initialized with one VALUE clause at the 01 level, because each item was being initialized to the same value.)

To initialize larger tables, use MOVE, PERFORM, or INITIALIZE statements.

### Initializing a table at the 01 level

Code a level-01 record and assign to it, through the VALUE clause, the contents of the whole table. Then, in a subordinate level data item, use an OCCURS clause to define the individual table items.

For example:

```
01  TABLE-ONE                                   VALUE "1234".
   05 TABLE-TWO OCCURS 4 TIMES                 PIC X.
```

### Initializing all occurrences of a table element

You can use the VALUE clause on a table element to initialize the element to the indicated value. For example:

```
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".
```

The above code 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-0BJ is then set to 3.

#### RELATED REFERENCES

REDEFINES clause (*IBM COBOL Language Reference*)  
 PERFORM statement (*IBM COBOL Language Reference*)  
 INITIALIZE statement (*IBM COBOL Language Reference*)  
 OCCURS clause (*IBM COBOL 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
77 ERROR-ON                                         Pic X Value "E".
   Indexed By Flag-Index                           Pic X.
*****
***      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.
*****
***          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-0n
      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, you need to set up a variable-length table definition. To do this, use the OCCURS DEPENDING ON (ODO) clause. For example:

```
X OCCURS 1 TO 10 TIMES DEPENDING ON Y
```

In this example, *X* is called the *ODO subject*, and *Y* is the *ODO object*.

Two factors affect the successful manipulation of variable-length records:

- Correct calculation of records lengths.

The length of the variable portions of a group item is the product of the object of the DEPENDING ON option 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 abnormally terminate. 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 (that is, if FIELD-1 contains an external 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, that would be five occurrences of FIELD-2, plus FIELD-1, for a length of 26 bytes.

In this case, you need not 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. 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).

```

When you do a MOVE to REC-1 in this case, the actual length of REC-1 is calculated immediately before the move using the current value of the ODO object (FIELD-1). The compiler issues a message letting 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 (here LOCATION-TABLE-LENGTH) 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 491

#### RELATED TASKS

“Assigning values to a variable-length table” on page 62

“Loading a variable-length table”

#### RELATED REFERENCES

OCCURS DEPENDING ON clause (*IBM COBOL 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 use a VALUE clause on a group item that contains an OCCURS clause with the DEPENDING ON option. Each subordinate structure that contains the DEPENDING ON option is initialized using the maximum number of occurrences. If you define the entire table with the DEPENDING ON option, all the elements are initialized using the maximum defined value of the DEPENDING ON object.

If the ODO *object* has a VALUE clause, it is logically initialized after the ODO *subject* has been initialized. For example, in the following code

```

01 TABLE-THREE              VALUE "3ABCDE".
05 X                          PIC 9.
05 Y OCCURS 5 TIMES
   DEPENDING ON X             PIC X.

```

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 the first three elements, Y(1) through Y(3).

### RELATED REFERENCES

OCCURS DEPENDING ON clause (*IBM COBOL 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 only of the order of 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 64

## Doing a serial search (SEARCH)

Use the SEARCH statement to do a serial search beginning at the current index setting. To modify the index setting, use the SET statement.

The conditions in the WHEN option 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 do not use AT END, control passes to the next statement in your 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.

If the found condition comes after some intermediate point in the table, you can speed up the search. Use 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 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 (*IBM COBOL Language Reference*)

### Example: serial search

Suppose you define a three-dimensional table, each with its own index (set to 1, 4, and 1, respectively). The innermost table (TABLE-ENTRY3) has an ascending key. The object of the search is to find a particular string in the innermost table.

```
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)

When you use SEARCH ALL to do a binary search, you do not need to set the index before you begin. The index used is always the one 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, your table must already be ordered on the key or keys coded in the OCCURS clause. You can use any key in the WHEN condition, but you must test all preceding data names in the KEY option, if any. The test must be an equal-to condition, and the KEY *data-name* must be either the subject of the condition or the name of a conditional variable with which the tested condition-name is associated. The WHEN condition can also be a compound condition, formed from simple conditions with AND as the only logical connective. The key and its object of comparison must be compatible.

“Example: binary search”

#### RELATED REFERENCES

SEARCH statement (*IBM COBOL Language Reference*)

### Example: binary search

Suppose you define a table that contains 90 elements of 40 bytes each, with 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 using the following instructions:

```

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 the three keys are equal to the given values (VALUE-1, VALUE-2, and VALUE-3), PART-1 of that entry will be moved to OUTPUT-AREA. If matching keys are not found in any of the entries in TABLE-A, the NOENTRY routine is performed.

---

## Processing table items using intrinsic functions

You can use an intrinsic function to process an alphanumeric or numeric table item. However, the data description of the table item must be compatible with the argument requirements for the function.

Use a subscript or index to reference an individual data item as a function argument. For example, assuming Table-One is a 3x3 array of numeric items, you can find the square root of the middle element with this statement:

```
Compute X = Function Sqrt(Table-One(2,2))
```

You might often need to process the data in tables iteratively. For intrinsic functions that accept multiple arguments, you can use the ALL subscript to reference all the items in the table or a single dimension of the table. The iteration is handled automatically, making 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: intrinsic functions”

### RELATED TASKS

“Using intrinsic functions (built-in functions)” on page 27

### RELATED REFERENCES

Intrinsic functions (*IBM COBOL Language Reference*)

## Example: intrinsic functions

This example sums a cross-section of Table-Two:

```
Compute Table-Sum = FUNCTION SUM (Table-Two(ALL, 3, ALL))
```

Assuming that Table-Two is a 2x3x2 array, the statement above causes the following elements to be summed:

```

Table-Two(1,3,1)
Table-Two(1,3,2)
Table-Two(2,3,1)
Table-Two(2,3,2)

```

This example computes values for all employees.

```

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:

- IF statement
- EVALUATE statement
- Switches and flags
- PERFORM statement

### RELATED TASKS

“Selecting program actions”

“Repeating program actions” on page 74

---

### 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 71

### RELATED REFERENCES

IF statement (*IBM COBOL Language Reference*)

EVALUATE statement (*IBM COBOL Language Reference*)

### Coding a choice of actions

Use IF . . . ELSE to code a choice between two processing actions. (The word THEN is optional in a COBOL program.)

```
IF condition-p
    statement-1
ELSE
    statement-2
END-IF
```

When one of the processing choices is no action, code the IF statement with or without ELSE. Because the ELSE clause is optional, you can code the following:

```
IF condition-q
    statement-1
END-IF
```

This coding is suitable for simple programming cases. For complex logic, you probably need to use the ELSE clause. For example, suppose you have nested IF statements with 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

```

Use the EVALUATE statement to code a choice among three or more possible conditions instead of just two. The EVALUATE statement is an expanded form of the IF statement that allows you to avoid nesting IF statements for such coding, a common source of logic errors and debugging problems.

With the EVALUATE statement, you can test any number of conditions in a single statement and have separate actions for each. In structured programming terms, this is a case structure. It can also be thought of as a decision table.

“Example: EVALUATE using THRU phrase” on page 70

“Example: EVALUATE using multiple WHEN statements” on page 70

“Example: EVALUATE testing several conditions” on page 70

#### RELATED TASKS

“Coding conditional expressions” on page 71

“Using the EVALUATE statement” on page 69

“Using nested IF statements”

### Using nested IF statements

When an IF statement has another IF statement as one of its possible processing branches, these IF statements are said to be nested. Theoretically, there is no limit to the depth of nested IF statements. However, when the program has to test a variable for more than two values, EVALUATE is the better choice.

Use nested IF statements sparingly. The logic can be difficult to follow, although explicit scope terminators and proper indentation help.

The following pseudocode depicts a nested IF statement:

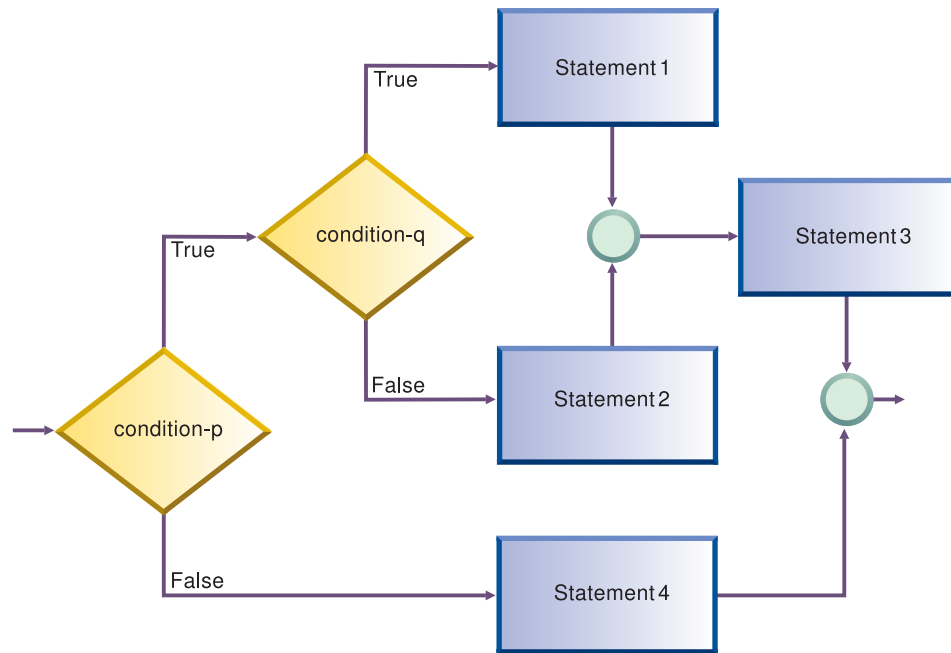
```

IF condition-p
    IF condition-q
        statement-1
    ELSE
        statement-2
    END-IF
    statement-3
ELSE
    statement-4
END-IF

```

Here an IF is nested, along with a sequential structure, in one branch of another IF. In a structure like this, the END-IF closing the inner nested IF is very important. Use END-IF instead of a period, because a period would end the outer IF structure as well.

The following figure shows the logic structure for nested IF statements.



#### RELATED TASKS

“Coding a choice of actions” on page 67

#### RELATED REFERENCES

Explicit scope terminators (*IBM COBOL Language Reference*)

### Using the EVALUATE statement

Use the EVALUATE statement to test several conditions and design a different action for each, a construct often known as a *case structure*. The expressions to be tested are called selection subjects; the answer selected is called a selection object. You can code multiple subjects and multiple objects in the same structure.

You can code the EVALUATE statement to handle the case where multiple conditions lead to the same processing by using the THRU phrase and by using multiple WHEN statements.

When evaluated, each pair of selection subjects and selection objects must belong to the same class (numeric, character, CONDITION TRUE or FALSE).

The execution of the EVALUATE statement ends when:

- 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 they are coded. Therefore, you should order these phrases for the best performance: code first the WHEN phrase containing selection objects 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 67

**Example: EVALUATE using THRU phrase:** This example shows how you can use the THRU phrase to easily code several conditions in a range of values that lead to the same processing action. 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 statements:** You can use multiple WHEN statements when several conditions lead to the same processing action. This gives you more flexibility than using the THRU phrase, because the conditions do not have to evaluate to values that fall in a range or evaluate to alphanumeric values.

```
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:** In this example both selection subjects in a WHEN phrase must satisfy the TRUE, TRUE condition before the phrase is performed. If both subjects do not evaluate to TRUE, the next WHEN phrase is processed.

```
Identification Division.
  Program-ID. MiniEval.
Environment Division.
```

```

Configuration Section.
Data Division.
Working-Storage Section.
01  Age                Pic  999.
01  Sex                Pic  X.
01  Description        Pic  X(15).
01  A                  Pic  999.
01  B                  Pic  9999.
01  C                  Pic  9999.
01  D                  Pic  9999.
01  E                  Pic  99999.
01  F                  Pic  999999.
Procedure Division.
PN01.
    Evaluate True Also True
        When Age < 13 Also Sex = "M"
            Move "Young Boy" To Description
        When Age < 13 Also Sex = "F"
            Move "Young Girl" To Description
        When Age > 12 And Age < 20 Also Sex = "M"
            Move "Teenage Boy" To Description
        When Age > 12 And Age < 20 Also Sex = "F"
            Move "Teenage Girl" To Description
        When Age > 19 Also Sex = "M"
            Move "Adult Man" To Description
        When Age > 19 Also Sex = "F"
            Move "Adult Woman" To Description
        When Other
            Move "Invalid Data" To Description
    End-Evaluate
    Evaluate True Also True
        When A + B < 10 Also C = 10
            Move "Case 1" To Description
        When A + B > 50 Also C = ( D + E ) / F
            Move "Case 2" To Description
        When Other
            Move "Case Other" To Description
    End-Evaluate
Stop Run.

```

## Coding conditional expressions

Using the IF and EVALUATE statements, you can code program actions that will be performed depending on the truth value of a conditional expression. You can specify any of these conditions:

- Numeric condition
- Nonnumeric condition
- Class of a field
- Switches and flags that you define
- Sign condition
- Status of UPSI switch

### RELATED CONCEPTS

“Switches and flags” on page 72

### RELATED TASKS

“Defining switches and flags” on page 72

“Resetting switches and flags” on page 73

“Checking for incompatible data (numeric class test)” on page 40

#### RELATED REFERENCES

Rules for condition-name values (*IBM COBOL Language Reference*)  
Switch-status condition (*IBM COBOL Language Reference*)  
Sign condition (*IBM COBOL Language Reference*)  
Comparing numeric and nonnumeric operands (*IBM COBOL Language Reference*)  
Combined conditions (*IBM COBOL Language Reference*)  
Class condition (*IBM COBOL Language Reference*)  
“UPSI” on page 219

### 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 with 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 73

### Defining switches and flags

In the DATA DIVISION, define level-88 items to give meaningful names (condition names) to values that will act as switches or flags.

To test for more than two values, as 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 have some association with logical values.

“Example: switches”  
“Example: flags” on page 73

### Example: switches

To test for an end-of-file condition for an input file named Transaction-File, you could 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 a condition named Transaction-EOF is turned on when Transaction-EOF-Switch has value “y”. Referencing Transaction-EOF in your PROCEDURE DIVISION expresses the same condition as testing for Transaction-EOF-Switch = "y". For example, the following statement causes the

report to be printed only if your program has read to the end of the Transaction-File and if the Transaction-EOF-Switch has been set to "y":

```
If Transaction-EOF Then
    Perform Print-Report-Summary-Lines
```

### Example: flags

Consider a program that updates a master file. The updates are read from a transaction file. The transaction file's records contain a field for the function 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 might look like this:

```
Evaluate True
    When Add-Transaction
        Perform Add-Master-Record-Paragraph
    When Change-Transaction
        Perform Update-Exisitng-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 change flags back to the original values they have in their data descriptions. To do so, use either a SET statement or define your own data item to use.

When you use the SET *condition-name* TO TRUE statement, the switch or flag is set back to the original value that was assigned in its data description.

For a level-88 item with multiple values, SET *condition-name* TO TRUE assigns the first value (here, A):

```
88 Record-is-Active Value "A" "0" "S"
```

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

"Example: set switch on"

"Example: set switch off" on page 74

### Example: set switch on

The SET statement in the following example does the same thing as 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 for 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

You could use a data item called SWITCH-OFF throughout your program to set on/off switches to off, as in the following code:

```

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

```

This code resets the switch to indicate that the end of the file has not been reached.

---

## Repeating program actions

Use the PERFORM statement to run a paragraph and then implicitly return control to the next executable statement. In effect, the PERFORM statement is a way of coding a closed subroutine that you can enter from many different parts of the program.

Use the PERFORM statement to loop (repeat the same code) a set number of times or to loop based on the outcome of a decision.

PERFORM statements can be inline or out-of-line.

#### RELATED TASKS

“Choosing inline or out-of-line PERFORM” on page 75

“Coding a loop” on page 76

“Coding a loop through a table” on page 76

“Executing multiple paragraphs or sections” on page 77

#### RELATED REFERENCES

PERFORM statement (*IBM COBOL Language Reference*)

## Choosing inline or out-of-line PERFORM

The inline PERFORM statement has the same general rules as the out-of-line PERFORM statement except for one area: statements within the inline PERFORM statement are executed rather than those within the range of the procedure named in the out-of-line PERFORM statement.

To determine whether to code an inline or out-of-line PERFORM statement, consider the following questions:

- Is the PERFORM statement used from several places?

Use out-of-line PERFORM when you use the same piece of code from several places in your program.

- Which placement of the statement will be easier to read?

Use an out-of-line PERFORM if the logical flow of the program will be less clear because the PERFORM extends over several screens. If, however, the PERFORM paragraph is short, an inline PERFORM can save the trouble of skipping around in the code.

- What are the efficiency tradeoffs?

Avoid the overhead of branching around an out-of-line PERFORM if performance is an issue. But remember, 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 an explicit branch to a separate paragraph and has an implicit return. If the performed paragraph is in the subsequent sequential flow of your program, it is also executed in that flow of the logic. To avoid this additional execution, you must 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. Each paragraph performs one logical function.

“Example: inline PERFORM statement”

### Example: inline PERFORM statement

This example shows the structure of an inline PERFORM statement with the required scope terminators and the required END-PERFORM statement.

```
Perform 100-Initialize-Paragraph
* The following 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 certain number of times:

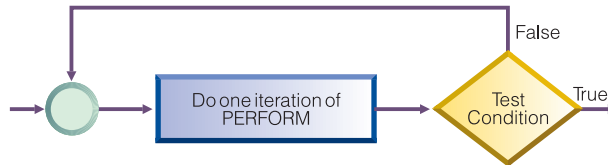
```
PERFORM 010-PROCESS-ONE-MONTH 12 TIMES  
INSPECT . . .
```

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 . . .
```

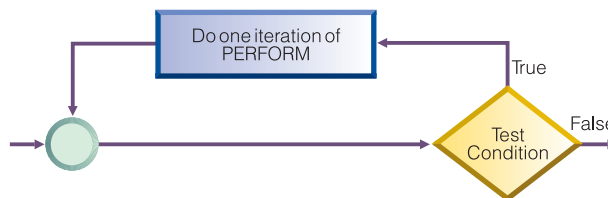
Use the `PERFORM . . . WITH TEST AFTER . . . UNTIL` if you want to execute the paragraph at least once and then test before any subsequent execution. This statement is equivalent to the do-until structure:



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 EQUAL DECEMBER`) 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.)



## Coding a loop through a table

You can use `PERFORM . . . VARYING` 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 a loop through a table. You can use either of the following forms:

```
PERFORM . . . WITH TEST AFTER . . . VARYING . . . UNTIL . . .  
PERFORM . . . [WITH TEST BEFORE] . . . VARYING . . . UNTIL .
```

The following 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 . . .
```

In the code above, when control reaches the PERFORM statement, 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 it 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.

In terms of the application, this loop controls input-checking for the 12 fields of item WS-DATA. Empty fields are not allowed, and this section of code loops through and issues error messages as appropriate.

## Executing multiple paragraphs or sections

In structured programming, the paragraph you execute is usually a single paragraph. However, you can execute a group of paragraphs, a single section, or a group of sections using the PERFORM . . . THRU statement.

When you use PERFORM . . . THRU use a paragraph-EXIT statement to clearly indicate the end point for the series of paragraphs.

Intrinsic functions can make the coding of the iterative processing of tables simpler and easier.

### RELATED TASKS

“Processing table items using intrinsic functions” on page 65



---

## Chapter 6. Handling strings

COBOL provides language constructs for performing the following operations associated with string data items:

- Joining and splitting data items
- Manipulating null-terminated strings, such as counting or moving characters
- Referring to substrings by their ordinal position and, if needed, length
- Tallying and replacing data items, such as counting the number of times a specific character occurs in a data item
- Converting data items, such as changing to uppercase or lowercase
- Evaluating data items, such as determining the length of a data item

### RELATED TASKS

“Joining data items (STRING)”

“Splitting data items (UNSTRING)” on page 81

“Manipulating null-terminated strings” on page 83

“Referring to substrings of data items” on page 84

“Tallying and replacing data items (INSPECT)” on page 87

“Converting data items (intrinsic functions)” on page 88

“Evaluating data items (intrinsic functions)” on page 90

---

### Joining data items (STRING)

Use the STRING statement to join all or parts of several data items into one data item. One STRING statement can save you several MOVE statements.

The STRING statement transfers data items into the receiving item in the order you indicate. In the STRING statement you can also specify the following:

- Delimiters that cause a sending field to be ended and another to be started
- Actions to be taken when the single receiving field is filled before all of the sending characters have been processed (ON OVERFLOW condition)

You can specify a national item for any operand except the POINTER identifier. However, if you do, you must specify all of them as class national.

“Example: STRING statement”

### RELATED TASKS

“Handling errors in joining and splitting strings” on page 125

### RELATED REFERENCES

STRING statement (*IBM COBOL Language Reference*)

### Example: STRING statement

In the following example, the STRING statement selects and formats information from record RCD-01 as an output line: line number, customer name and address, invoice number, next billing date, and balance due. The balance is truncated to the dollar figure shown.

In the FILE SECTION, the following record is defined:

```

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).

```

In the WORKING-STORAGE SECTION, the following fields are defined:

```

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 9 indicates a blank space):

```

J.B.9SMITH99999
4449SPRING9ST.,9CHICAGO,9ILL.999999
A14275
$4,736.85
$2,400.00
09/22/76
$2,336.85
10/22/76

```

In the PROCEDURE DIVISION, the programmer initializes RPT-LINE to SPACES and sets LINE-POS, the data item to be used as the POINTER field, to 4. (By coding the POINTER phrase of the STRING statement, you can use the explicit pointer field to control placement of data in the receiving field.) Then, the programmer codes this 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.

```

## STRING program 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 appears as 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 one sending field into several receiving fields. One UNSTRING statement can save you several MOVE statements.

In the UNSTRING statement you can specify the following:

- Delimiters that, when encountered in the sending field, cause the current receiving field to stop receiving and the next to begin receiving
- Fields that store the number of characters placed in receiving fields
- A field that stores a count of the total number of characters transferred
- Special actions to take if all the receiving fields are filled before the end of the sending item is reached

You can specify national items as the sending field, receiving fields, and delimiters. However, if you do, you must specify all of them as class national.

“Example: UNSTRING statement”

### RELATED TASKS

“Handling errors in joining and splitting strings” on page 125

### RELATED REFERENCES

UNSTRING statement (*IBM COBOL Language Reference*)

## Example: UNSTRING statement

In the following example, selected information is taken from the input record. Some is organized for printing and some for further processing.

In the FILE SECTION, the following records are defined:

\* 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, the programmer writes the following 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.

```

Before issuing the UNSTRING statement, the programmer places the value 3 in CHAR-CT (the POINTER field) to avoid working with the two control characters in INV-RCD. A period (.) is placed in DBY-1 for use as a delimiter, and the value 0 (zero) is placed in FLDS-FILLED (the TALLYING field). The data is then read into INV-RCD, as shown below.

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

## UNSTRING program 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, /, is placed in DLTR-1, and the value 6 is placed in CTR-2.

4. Positions 31 through 33 are placed in INV-CLASS. The delimiter is a 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 examined and placed in M-UNITS. The value 6 is placed in CTR-3. The delimiter is a 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 0 (zero). The delimiter is a SPACE, but because no field has been 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 character 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 of data in INV-RCD have not been examined, the ON OVERFLOW exit is taken, and execution of the UNSTRING statement is completed.

After the UNSTRING statement is performed, the fields contain the values shown below.

Field	Value
DISPLAY-REC	707890 FOUR-PENNY-NAILS 000379
WORK-REC	475120000122BBA
CHAR-CT (the POINTER field)	55
FLDS-FILLED (the TALLYING field)	6

## Manipulating null-terminated strings

You can construct and manipulate null-terminated strings (passed to or from a C program, for example), by various mechanisms:

- 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 the field by using a reference modifier such as source-field (I:1).

“Example: null-terminated strings”

#### RELATED REFERENCES

Nonnumeric literals (*IBM COBOL Language Reference*)

## Example: null-terminated strings

The following example shows several ways you can manipulate 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 character-string data item (including EBCDIC data items) with reference modifiers. Intrinsic functions that return character-string values are also considered alphanumeric data items, and can include a reference modifier.

The following example shows how to use a reference modifier to refer to a substring of a data item:

```
Move Customer-Record(1:20) to Orig-Customer-Name
```

As this example shows, you code two values separated by a colon, in parentheses, immediately following the data item:

- Ordinal position (from the left) of the character that you want the substring to start with
- Length of the desired substring

The length is optional. If you omit the length, the substring extends to the end of the item. Omit the length when possible as a simpler and less error-prone coding technique.

You can code either of the two values as a variable or as an arithmetic expression.

Because numeric function identifiers can be used anywhere arithmetic expressions are allowed, you can use them in the reference modifier as the leftmost character position or as the length.

You can also refer to substrings of table entries, including variable-length entries.

Both numbers in the reference modifier must have a value of at least 1. Their sum should not exceed the total length of the data item by more than 1 so that you do not reference beyond the end of the desired 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 run-time message:

- SSRANGE compiler option
- CHECK run-time option

#### RELATED CONCEPTS

“Reference modifiers”

#### RELATED TASKS

“Referring to an item in a table” on page 53

#### RELATED REFERENCES

“SSRANGE” on page 189

Reference modification (*IBM COBOL Language Reference*)

Function definitions (*IBM COBOL Language Reference*)

## Reference modifiers

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”

“Example: intrinsic functions as reference modifiers”

#### RELATED TASKS

“Referring to substrings of data items” on page 84

#### RELATED REFERENCES

Reference modification (*IBM COBOL Language Reference*)

## Example: arithmetic expressions as reference modifiers

Suppose that a field contains some right-justified characters, and you want to move the characters to another field where they will be left-justified. You can do that using reference modifiers and an INSPECT statement.

Suppose the 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

The following code fragment causes a substring of Customer-Record to be moved into the variable 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 requiring 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-PART` (*IBM COBOL Language Reference*)

`INTEGER` (*IBM COBOL Language Reference*)

---

## Tallying and replacing data items (`INSPECT`)

Use the `INSPECT` statement to do the following:

- Fill selective portions of a data item with a value. If you specify a national data item, you must also specify a national value.
- Replace portions of a data item with a corresponding portion of another data item. To specify any national items or literals in the statement, you must specify all items as national.
- Count the number of times a specific character (zero, space, or asterisk, for example) occurs in a data item. For a national data item, the count is in national character encoding units.

“Examples: `INSPECT` statement”

#### RELATED REFERENCES

`INSPECT` statement (*IBM COBOL Language Reference*)

## Examples: `INSPECT` statement

The following examples show some uses of the `INSPECT` statement.

In the following example, the `INSPECT` statement is used to examine and replace characters in data item `DATA-2`. The number of times a leading 0 occurs in the data item is accumulated in `COUNTR`. Every instance of the character A following the first instance of the character C is replaced by the character 2.

```
77 COUNTR          PIC 9   VALUE ZERO.
01 DATA-2        PIC X(11).
. . .
  INSPECT DATA-2
    TALLYING COUNTR FOR LEADING "0"
    REPLACING FIRST "A" BY "2" AFTER INITIAL "C"
```

DATA-2 before	COUNTR after	DATA-2 after
00ACADEMY00	2	00AC2DEMY00
0000ALABAMA	4	0000ALABAMA
CHATHAM0000	0	CH2THAM0000

In the following example, the `INSPECT` statement is used to examine and replace characters in data item `DATA-3`. Every character in the data item preceding the first instance of a quote (") 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 in the data item following the first instance of the character / but preceding 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:

- Uppercase or lowercase
- Reverse order
- Numbers

You can also use the INSPECT statement to convert characters.

“Examples: INSPECT statement” on page 87

## Converting to uppercase or lowercase (UPPER-CASE, LOWER-CASE)

```
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 be moved to the actual contents of Item-2.

When you convert the case of a national item, each character is mapped to a single character; there is no context-dependent mapping.

## Converting to reverse order (REVERSE)

The following code reverses the order of the characters in Orig-cust-name.

```
Move Function Reverse(Orig-cust-name) To Orig-cust-name
```

For example, if the starting value is JOHNSON<sup>000</sup>, the value after the statement is performed is <sup>000</sup>NOSNH0J, where <sup>0</sup> represents a blank space.

When you reverse the order of a national string, the result is a national string.

## Converting to numbers (NUMVAL, NUMVAL-C)

The NUMVAL and NUMVAL-C functions convert character strings to numbers. Use these functions to convert alphanumeric data items that contain free-format character-representation numbers to numeric form, and process them numerically. For example:

```
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. 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, not including the editing symbols.

NUMVAL and NUMVAL-C return long (64-bit) floating-point values. A reference to either of these functions, therefore, represents a reference to a numeric data item.

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 34

#### RELATED TASKS

“Assigning input from a screen or file (ACCEPT)” on page 26

“Displaying values on a screen or in a file (DISPLAY)” on page 27

#### RELATED REFERENCES

NUMVAL (*IBM COBOL Language Reference*)

NUMVAL-C (*IBM COBOL Language Reference*)

---

## Evaluating data items (intrinsic functions)

You can use several intrinsic functions in evaluating data items:

- CHAR and ORD for evaluating integers and single alphanumeric characters with respect to the collating sequence used in your program
- MAX, MIN, ORD-MAX, and ORD-MIN for finding the largest and smallest items in a series of data items
- LENGTH for finding the length of data items
- WHEN-COMPILED for finding the date and time the program was compiled

#### RELATED TASKS

“Evaluating single characters for collating sequence”

“Finding the largest or smallest data item”

“Finding the length of data items” on page 92

“Finding the date of compilation” on page 93

## Evaluating single characters for collating sequence

To find out the ordinal position of a given character in the collating sequence, use the ORD function with the character as the argument. ORD returns an integer representing that ordinal position. One convenient way to find a character’s ordinal position is to 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 know the character that it corresponds to, use the CHAR function with the integer ordinal position as the argument. CHAR returns the desired character:

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

#### RELATED REFERENCES

CHAR (*IBM COBOL Language Reference*)

ORD (*IBM COBOL Language Reference*)

## Finding the largest or smallest data item

If you want to know which of two or more alphanumeric data items has the largest value, use the MAX or ORD-MAX function. Supply the data items in question as arguments. If you want to know which item contains the smallest value, use the MIN or ORD-MIN function. These functions evaluate the values according to the collating sequence. You can also use MAX, ORD-MAX, MIN, or ORD-MIN for numbers. In that case, the algebraic values of the arguments are compared.

## MAX and MIN

The MAX and MIN functions return the contents of one of the variables you supply.

For example, suppose you have these data definitions:

```
05 Arg1      Pic x(10) Value "THOMASSON ".
05 Arg2      Pic x(10) Value "THOMAS   ".
05 Arg3      Pic x(10) Value "VALLEJO   ".
```

The following statement assigns VALLEJO to the first 10 character positions of Customer-record, where represents a blank space:

```
Move Function Max(Arg1 Arg2 Arg3) To Customer-record(1:10)
```

If you use MIN instead, then THOMAS is assigned.

If you specify a national item for any argument, you must specify all arguments as national.

## ORD-MAX and ORD-MIN

The functions ORD-MAX and ORD-MIN return an integer that represents the ordinal position of the argument with the largest or smallest value in the list of arguments you supply (counting from the left).

If you used the ORD-MAX function in the example above, you would receive a syntax error message at compile time; the reference to a numeric function is in an invalid place. The following is a valid use of the ORD-MAX function:

```
Compute x = Function Ord-max(Arg1 Arg2 Arg3)
```

This code assigns the integer 3 to x if the same arguments are used as in the previous example. If you use ORD-MIN instead, the integer 2 is returned. The above examples would probably be more realistic if Arg1, Arg2, and Arg3 were instead successive elements of an array (table).

If you specify a national item for any argument, you must specify all arguments as national.

## Returning variable-length results with alphanumeric functions

The results of alphanumeric 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))
```

Here, R2 is evaluated to be larger than R1. Therefore:

- The string f is moved to R3, where 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" then R1 would evaluate as larger than R2, and:

- The string g000000000 would be moved to R3. (The unfilled character positions in R3 would be padded with spaces.)
- The value 10 would be assigned to L.

You might be dealing with variable-length output from alphanumeric functions. Plan your program accordingly. For example, you might need to think about using variable-length files when the records 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

“Performing arithmetic” on page 41

“Processing table items using intrinsic functions” on page 65

#### RELATED REFERENCES

ORD-MAX (*IBM COBOL Language Reference*)

ORD-MIN (*IBM COBOL Language Reference*)

## Finding the length of data items

You can use the LENGTH function in many contexts (including numeric data and tables) to determine the length of string items.

The following COBOL statement demonstrates moving a data item into that field in a record that holds customer names:

```
Move Customer-name To Customer-record(1:Function Length(Customer-name))
```

You could also use the LENGTH OF special register. Coding either Function Length(Customer-Name) or LENGTH OF Customer-Name returns the same result: 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 41

“Processing table items using intrinsic functions” on page 65

#### RELATED REFERENCES

LENGTH (*IBM COBOL Language Reference*)

## Finding the date of compilation

If you want to know the date and time when a program was compiled, you can use the `WHEN-COMPILED` function. The result returned has 21 character positions, with the first 16 positions in the following format:

`YYYYMMDDhhmmsshh`

These characters show the four-digit year, month, day, and time (in hours, minutes, seconds, and hundredths of seconds) of compilation.

The `WHEN-COMPILED` special register is another means you can use to find the date and time of compilation. It has 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. This special register be used only as the sending field in a `MOVE` statement.

### RELATED REFERENCES

`WHEN-COMPILED` (*IBM COBOL 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. The major tasks you can perform in processing your files are first opening a file and then reading it and (depending on the type of file organization and access) adding, replacing, or deleting records.

Before the actual processing, however, you need to identify the files and describe their physical structure, including whether they are organized as sequential, relative, indexed, or line sequential. Identifying them 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.

### RELATED TASKS

“Identifying files”

“Specifying a file organization and access mode” on page 100

“Protecting against errors when opening files” on page 100

“Setting up a field for file status” on page 104

“Describing the structure of a file in detail” on page 104

“Opening a file” on page 107

“Reading records from a file” on page 109

“Adding records to a file” on page 110

“Replacing records in a file” on page 111

“Deleting records from a file” on page 112

---

## Identifying files

In the FILE-CONTROL clause of the INPUT-OUTPUT section of the ENVIRONMENT DIVISION, you must specify a name for the file you want to use and a name for the file system:

```
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 the file system as one of the following:

**STL** STL file system.

### VSAM

VSAM file system. You can abbreviate VSAM as VSA.

*Filename* must start with \\, indicating remote file access. Only remote files are supported using VSAM.

**BTR** Btrieve file system.

If you do not provide the file-system specification, the run-time option FILESYS is used to select the file system. If a file system is not specified using FILESYS, the default is STL.

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

The following file status indicators are not set for Btrieve:

- 02
- 21
- 39

#### RELATED TASKS

“Identifying Btrieve files”

“Identifying STL files”

“Identifying remote files”

#### RELATED REFERENCES

“STL file system” on page 97

## Identifying Btrieve files

To use the Btrieve file system, the following assignment would be valid:

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

If the run-time option FILESYS(BTRIEVE) were in effect, the following assignment would be valid:

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

Given that you have defined the environment variable MYFILE (for example, SET MYFILE=BTR-MYFILE), the following assignment would be valid:

```
SELECT file1 ASSIGN TO MYFILE
```

## Identifying STL files

To use the STL file system, the following assignment would be valid:

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

If the run-time option FILESYS(STL) were in effect, the following assignment would be valid:

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

Given that you have defined the environment variable MYFILE (for example, SET MYFILE=STL-MYFILE), the following assignment would be valid:

```
SELECT file1 ASSIGN TO MYFILE
```

## Identifying remote files

Using the Distributed File feature of SMARTdata Utilities, you can access a remote file (such as OS/390 VSAM, SAM, or PDS) without any source program change.

Suppose that your program contains the following SELECT clause:

```
SELECT myfile ASSIGN TO TARGETFILE
```

You can associate the file *myfile* in your workstation program to an OS/390 VSAM file called MVSMAS by setting the TARGETFILE environment variable like this:

```
set TARGETFILE=VSA-\\mvssystem\user.MVSMAS
```

Or if the run-time option FILESYS is set to VSA, you can associate the file this way:

```
set TARGETFILE=\\mvssystem\user.MVSMAS
```

Here VSA is the VSAM file-system identifier, *mvssystem* is the specific OS/390 system, *user* is the user ID, and MVSMAST is the data set name on the OS/390 system.

#### RELATED REFERENCES

“FILESYS” on page 218

*VSAM in a Distributed Environment*

## File system

Record-oriented files that are organized as sequential, relative, indexed, or line sequential (byte stream) files 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.

VisualAge COBOL supports the following file systems:

- The STL file system, which provides the basic facilities for local files. It is provided with VisualAge COBOL, and supports sequential, relative, and indexed files.
- The VSAM file system, which allows you to read and write files on remote systems such as OS/390. It is provided with VisualAge COBOL, and supports sequential, relative, and indexed files.
- The Btrieve file system, which allows you to access Btrieve files. Btrieve is a separate product available from Pervasive Software.

**Tip:** By using the Btrieve file system, you can access files created by VisualAge CICS Enterprise Application Development.

Most programs will have the same results 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 run-time option. All the file systems allow you to use COBOL statements to read or write COBOL files.

You can use any of these file systems to access local EBCDIC files.

#### RELATED TASKS

“Identifying files” on page 95

#### RELATED REFERENCES

“STL file system”

System/390 host data type considerations (“Local file access” on page 480)

“FILESYS” on page 218

## STL file system

The STL file system (Standard Language file system) supports sequential, indexed, and relative files on the local system. It provides the basic file facilities that you need for accessing local files. It conforms to ANSI standards, and gives good performance and the ability to port easily between AIX and Windows systems.

Line sequential files are the only files not supported.

The file system is safe for use with threads. However, you must ensure that multiple threads do not access COBOL buffers 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 100

#### RELATED REFERENCES

“STL file system return codes”

### STL file system return codes

If you code the following statement for an STL file in the FILE-CONTROL paragraph:

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

then after an input-output operation on the file, *data-name-1* will contain a status code that is independent of the file system used, and *data-name-8* will contain one of the STL file system return codes shown in the tables below.

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	An end of file was detected. This is not an error.
10	I/O operation must be preceded by I/O GET op	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.

Code	Meaning	Notes
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)	
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 arg for create	This return code should not occur; it indicates an error in the file system.
23	Index or rel record > 64K	Index and relative records are limited to a length of 64K.
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 seq 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.

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 129

#### RELATED REFERENCES

“STL file system” on page 97

FILE STATUS clause (*IBM COBOL Language Reference*)

---

## Protecting against errors when opening files

If your 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:

```
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, but 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. You should decide on the file organization and access modes when you design your program. The access mode defines how COBOL will read and write files, but not how files are organized.

Your file management system handles the input and output requests and record retrieval from the input-output devices.

The following table summarizes file organization and access modes for COBOL files.

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

#### RELATED CONCEPTS

“Sequential file organization” on page 101

“Line-sequential file organization”  
“Indexed file organization”  
“Relative file organization” on page 102  
“Sequential access” on page 102  
“Random access” on page 102  
“Dynamic access” on page 103

#### RELATED TASKS

“Specifying a file organization and access mode” on page 100

#### RELATED REFERENCES

“File input-output limitations” on page 103

### 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 sequentially only.

After you have placed a record into a sequential file, you cannot shorten, lengthen, or delete it. 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 102

#### RELATED REFERENCES

“Valid COBOL statements for sequential files” on page 107

### Line-sequential file organization

Line-sequential files are like sequential files, except that the records can only contain 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 as well as to a disk.

#### RELATED CONCEPTS

“Sequential file organization”

#### RELATED REFERENCES

“Valid COBOL statements for line-sequential files” on page 108

### Indexed file organization

An indexed file contains records ordered by a record key. Each record contains a field that contains the record key. The record key uniquely identifies the record and determines the sequence in which it is accessed with respect to other records. 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*—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 108

### Relative file organization

A relative record file contains records ordered by their *relative key*—the record number that represents record location relative to where the file begins. For example, the first record in the 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 108

### 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” on page 103

#### RELATED REFERENCES

“File position indicator” on page 106

### Random access

Code `ACCESS IS RANDOM` in the `FILE-CONTROL` paragraph.

For indexed files, records are accessed according to the value you place in a key field (primary, alternate, or relative). There can be one or more alternate indexes.

For relative files, records are accessed according to the value you place in the relative key.

#### RELATED CONCEPTS

"Sequential access" on page 102

"Dynamic access"

### Dynamic access

Code `ACCESS IS DYNAMIC` in the `FILE-CONTROL` paragraph.

Dynamic access is 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 earning between \$12.00 and \$18.00 per hour and those earning \$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 102

"Random access" on page 102

### File input-output limitations

- For line sequential files:
  - Maximum record size: 64K
  - Maximum number of bytes allocated for a file: no limit
- For VSAM files:
  - Minimum record size: 1 byte
  - Maximum record size: 64,000 bytes
  - Maximum record key length: 255 bytes
  - Maximum relative key value:  $2^{32} - 1$
  - Maximum number of bytes allocated for a file:  $2^{31} - 1$
- 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 bytes
  - 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)

Additional or more restrictive limits might be applicable depending on the platform where the target file is located. See the appropriate books for the file system of the target platform for these limits.

---

## Setting up a field for file status

Establish a FILE STATUS key 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
```

*file-status* specifies the two-character COBOL file status key that you define in the WORKING-STORAGE section:

```
WORKING-STORAGE SECTION.  
01 file-status PIC 99.
```

**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 129

### RELATED REFERENCES

FILE STATUS clause (*IBM COBOL Language Reference*)

---

## Describing the structure of a file in detail

In the FILE SECTION of the DATA DIVISION, start a file description using the keyword FD and the same file name you used in the corresponding SELECT statement in the FILE-CONTROL paragraph:

```
DATA DIVISION.  
FILE SECTION.  
FD filename  
01 recordname  
   nn . . . fieldlength & type  
   nn . . . fieldlength & type  
   . . .
```

*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 (*IBM COBOL Language Reference*)

Level-numbers (*IBM COBOL Language Reference*)

PICTURE clause (*IBM COBOL Language Reference*)

USAGE clause (*IBM COBOL Language Reference*)

---

## Coding input and output statements for files

After you have identified and described 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 have decided to use, whether sequential, line sequential, indexed, or relative. The general format for coding input and output (as shown in the example) involves opening the file, reading it, writing information into it, and then closing it.

“Example: COBOL coding for files”

#### RELATED TASKS

- “Identifying files” on page 95
- “Specifying a file organization and access mode” on page 100
- “Protecting against errors when opening files” on page 100
- “Setting up a field for file status” on page 104
- “Describing the structure of a file in detail” on page 104
- “Opening a file” on page 107
- “Reading records from a file” on page 109
- “Adding records to a file” on page 110
- “Replacing records in a file” on page 111
- “Deleting records from a file” on page 112

## Example: COBOL coding for files

The general format of input-output coding is shown below. 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 on the SELECT and the FD statements, and on 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, FD, and input-output statements. For WRITE and REWRITE, you specify a record defined for the file.

(2) *assignment-name*

You can specify 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, you can 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. The system file identification can optionally be preceded by the file-system type identification.

The following example illustrates how inventory-file is dynamically (by way of a MOVE statement) associated with a 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 this clause is omitted, the default is ORGANIZATION SEQUENTIAL.

(4) *access*

Indicates the access mode, SEQUENTIAL, RANDOM, or DYNAMIC. If this clause is omitted, the default is ACCESS SEQUENTIAL.

(5) *file-status*

The two-character COBOL FILE STATUS key.

(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 file's record format. If you break the record description entry beyond the level-01 description, each element should map 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 any WRITE, START, READ, REWRITE, or DELETE statement to process records in a file, it must first open the file with an OPEN statement:

PROCEDURE DIVISION.

```
      . . .  
      OPEN iomode filename
```

*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 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 you add must have a record key higher than the highest record in the file.

### RELATED CONCEPTS

“File organization and access mode” on page 100

### RELATED TASKS

“Protecting against errors when opening files” on page 100

### RELATED REFERENCES

“Valid COBOL statements for sequential files”

“Valid COBOL statements for line-sequential files” on page 108

“Valid COBOL statements for indexed and relative files” on page 108

OPEN statement (*IBM COBOL Language Reference*)

## Valid COBOL statements for sequential files

The following table shows the possible combinations of input-output statements for sequential files. The ‘X’ indicates that the statement can be used with the open mode given at the top of the column.

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 101

“Sequential access” on page 102

### Valid COBOL statements for line-sequential files

The following table shows the possible combinations of input-output statements for line-sequential files. The ‘X’ indicates that the statement can be used with the open mode given at the top of the column.

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 101

“Sequential access” on page 102

### Valid COBOL statements for indexed and relative files

The following table shows the possible combinations of input-output statements for indexed and relative files. The ‘X’ indicates that the statement can be used with the open mode given at the top of the column.

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	

Access mode	COBOL statement	OPEN INPUT	OPEN OUTPUT	OPEN I-O	OPEN EXTEND
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 101

“Relative file organization” on page 102

“Sequential access” on page 102

“Random access” on page 102

“Dynamic access” on page 103

## Reading records from a file

Use the READ statement to retrieve (read) 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 statements 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 102  
“Random access” on page 102  
“Dynamic access” on page 103  
“File organization and access mode” on page 100

#### RELATED TASKS

“Opening a file” on page 107  
“Using file status keys” on page 129

#### RELATED REFERENCES

“File position indicator” on page 106  
FILE STATUS clause (*IBM COBOL 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.

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 100

#### RELATED TASKS

“Specifying a file organization and access mode” on page 100  
“Opening a file” on page 107  
“Setting up a field for file status” on page 104  
“Adding records to a file”

#### RELATED REFERENCES

“PROCEDURE DIVISION statements used to update files” on page 112

## 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

Use `ACCESS IS SEQUENTIAL` and code the `WRITE` statement to add records sequentially to the end of a file that has been opened with either `OUTPUT` or `EXTEND`.

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`, the records can be written in any order.

### RELATED CONCEPTS

“File organization and access mode” on page 100

### RELATED TASKS

“Specifying a file organization and access mode” on page 100

“Using file status keys” on page 129

### RELATED REFERENCES

“Statements used when writing records to a file” on page 110

“PROCEDURE DIVISION statements used to update files” on page 112

`FILE STATUS` clause (*IBM COBOL Language Reference*)

## Replacing records in a file

To replace a record in a file, use `REWRITE` on a file that you have opened `I-0`. If you try to use `REWRITE` on a file that is not opened `I-0`, the record is not replaced and the status key is set to 49. Check the file status key after each `REWRITE` statement. Note the following about the length of the record:

- For sequential files, the length of the record you replace must be the same as the length of the original record.
- For indexed files, you can change the length of the record you replace.
- For 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`.

### RELATED CONCEPTS

“File organization and access mode” on page 100

### RELATED TASKS

“Opening a file” on page 107

“Using file status keys” on page 129

#### RELATED REFERENCES

FILE STATUS clause (*IBM COBOL Language Reference*)

## Deleting records from a file

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

When ACCESS IS SEQUENTIAL, the record to be deleted must first be read by the COBOL program. The DELETE statement then 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 100

#### RELATED TASKS

“Opening a file” on page 107

“Reading records from a file” on page 109

“Using file status keys” on page 129

#### RELATED REFERENCES

FILE STATUS clause (*IBM COBOL 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.

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 method	Sequential	Line sequential	Indexed	Relative
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 100

#### RELATED TASKS

“Opening a file” on page 107

“Reading records from a file” on page 109

“Adding records to a file” on page 110

“Replacing records in a file” on page 111

“Deleting records from a file” on page 112

#### RELATED REFERENCES

“Statements used when writing records to a file” on page 110



---

## Chapter 8. Sorting and merging files

You can arrange records in a particular sequence by using the SORT or MERGE statements:

### **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.

You can mix SORT and MERGE statements in the same COBOL program. 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 general procedure for sorting or merging is 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 116

“Describing the input to sorting or merging” on page 116

“Describing the output from sorting or merging” on page 118

“Requesting the sort or merge” on page 120

“Determining whether the sort or merge was successful” on page 122

### **RELATED REFERENCES**

SORT statement (*IBM COBOL Language Reference*)

MERGE statement (*IBM COBOL Language Reference*)

---

## Sort and merge process

During the sorting of a file, all of its records are ordered according to the contents of one or more fields (*keys*) in each record. 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. You can sort the records in either ascending or descending order of each key.

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. 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. You can order the records in either ascending or descending order of each key.

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 REFERENCES

`SORT` statement (*IBM COBOL Language Reference*)

`MERGE` statement (*IBM COBOL Language Reference*)

---

## Describing the sort or merge file

Describe the sort file to be used for sorting or merging:

1. Write one or more `SELECT` statements 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).
```

You need `SELECT` statements and `SD` entries for sorting or merging, even if you are sorting or merging data items from `WORKING-STORAGE` only.

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 this procedure:

1. Write one or more `SELECT` statements 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 118

“Requesting the sort or merge” on page 120

#### 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 files and an input file.

```
ID Division.
Program-ID. Smp1Sort.
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 120

## Coding the input procedure

If you want 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 do the following:

- Release data items to the sort file from WORKING-STORAGE.
- Release records that have already been read in 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 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 119  
RELEASE statement (*IBM COBOL Language Reference*)

---

## Describing the output from sorting or merging

If the output from sorting or merging is a file, describe the file by following this procedure:

1. Write a SELECT statement 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 120

#### RELATED REFERENCES

“FILE SECTION entries” on page 12

## Coding the output procedure

If you want 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 or RETURN INTO statement
- 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 your 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 on 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 the RETURN INTO statement instead of RETURN, your records will be returned to WORKING-STORAGE or to an output area.

#### RELATED REFERENCES

“Restrictions on input and output procedures”

RETURN statement (*IBM COBOL Language Reference*)

---

## Restrictions on input and output procedures

The following restrictions 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.
- The procedure must not contain any STOP RUN, EXIT PROGRAM, or GOBACK 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 118

“Coding the output procedure” on page 119

---

## Requesting the sort or merge

To read records directly from an input file (files for MERGE) without any preliminary processing, use SORT . . . USING or MERGE . . . USING and the name of the input file (files) that you have declared in a SELECT statement. The compiler generates an input procedure to open the input file (files), read the records, release the records to the sort or merge program, and close the input file (files). The input file or files must not be open when the SORT or MERGE statement begins execution.

To transfer sorted or merged records from the sort or merge program directly to another file without any further processing, use SORT . . . GIVING or MERGE . . . GIVING and the name of the output file that you have declared in a SELECT statement. The compiler generates an output procedure to open the output file, return the records, write the records, and close the file. The output file must not be open when the SORT or MERGE statement begins execution. For example:

```
SORT Sort-Work-1
  ON ASCENDING KEY Sort-Key-1
  USING Input-File
  GIVING Output-File.
```

“Example: describing sort and input files for SORT” on page 117

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 121

**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.
```

You must define *Merge-Work* in an SD statement in the FILE SECTION of the DATA DIVISION, and the input files in FD statements in the FILE SECTION.

## Setting sort or merge criteria

To set sort or merge criteria, follow these steps:

1. In the record description of the files to be sorted or merged, define the key or keys on which the operation is to be performed.  
**Restriction:** A key cannot be variably located.
2. In the SORT or MERGE statement, specify the key fields to be used for sequencing. You can code keys as ascending or descending. When you code more than one key, some can be ascending, and some descending.  
The leftmost key is the primary key. The next key is the secondary key, and so on.

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 CONCEPTS

“Appendix D. Complex OCCURS DEPENDING ON” on page 491 (variably located items)

### RELATED REFERENCES

SORT statement (*IBM COBOL Language Reference*)

MERGE statement (*IBM COBOL Language Reference*)

## Choosing alternate collating sequences

You can sort or merge records in 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. To override the PROGRAM COLLATING SEQUENCE specified either explicitly (in the OBJECT-COMPUTER paragraph) or by default, 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.

For DBCS keys, the collating sequence is specified by the locale setting in effect at run time.

### RELATED TASKS

“Specifying the collating sequence” on page 8

### RELATED REFERENCES

SORT statement (*IBM COBOL 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 primary key SORT-GRID-LOCATION and secondary key SORT-SHIFT in the DATA DIVISION 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 120

---

## Determining whether the sort or merge was successful

The SORT or MERGE statement returns one of the following completion codes after a sort or merge has finished:

- 0        Successful completion of the sort or merge
- 16      Unsuccessful completion of the sort or merge

The completion code is stored in the SORT-RETURN special register. The contents of this register change after each SORT or MERGE statement is performed.

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 return code. If the return code is 16, COBOL issues a run-time diagnostic message and terminates the run unit (or the thread, in a multithread environment).

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 return code.

---

## Stopping a sort or merge operation prematurely

To stop a sort or merge operation, use the SORT-RETURN special register. Move the integer 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

Anticipate possible coding or system problems by putting code into your program to handle them. Such code is like built-in distress flares or lifeboats. With this code, output data and files should not be corrupted, and the user will know when there is a problem.

Your error-handling code can take actions such as handling the situation, issuing a message, or halting the program. In any event, coding a warning message is a good idea.

You might create error-detection routines for data-entry errors or for errors as your installation defines them.

COBOL contains special elements to help you anticipate and correct error conditions:

- ON OVERFLOW in STRING and UNSTRING operations
- ON SIZE ERROR in arithmetic operations
- Technique handling for 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 126

"Handling errors in input and output operations" on page 127

"Handling errors when calling programs" on page 133

---

## Handling errors in joining and splitting strings

During the joining string or splitting of strings, the pointer, used by STRING or UNSTRING, might fall outside the range of the receiving field. Here a potential overflow condition exists, 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. However, you do not have an ON OVERFLOW clause on the STRING or UNSTRING statement, and 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

Data item	PICTURE	Value before	Value after
Item-4	X(8)	00000000 <sup>1</sup>	00000000 <sup>1</sup>
String-ptr	9(2)	0	0
1. The symbol 0 represents a blank space.			

Because String-ptr has a value of zero that falls short of the receiving field, an overflow condition occurs and the STRING operation is not completed. (A String-ptr greater than nine would have the same result.) If ON OVERFLOW had not been specified, you would not be notified that the contents of Item-4 remain 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) run-time 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

Code your ON SIZE ERROR imperative statement so that it issues an informative message. For example:

```
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.
```

In this example, if division by zero occurs, the program writes a message identifying the trouble 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 occurs.

You can use any of the following techniques for intercepting and handling certain input or output errors:

- End-of-file condition (AT END)
- ERROR declarative
- File status key
- File system return code
- INVALID KEY phrase

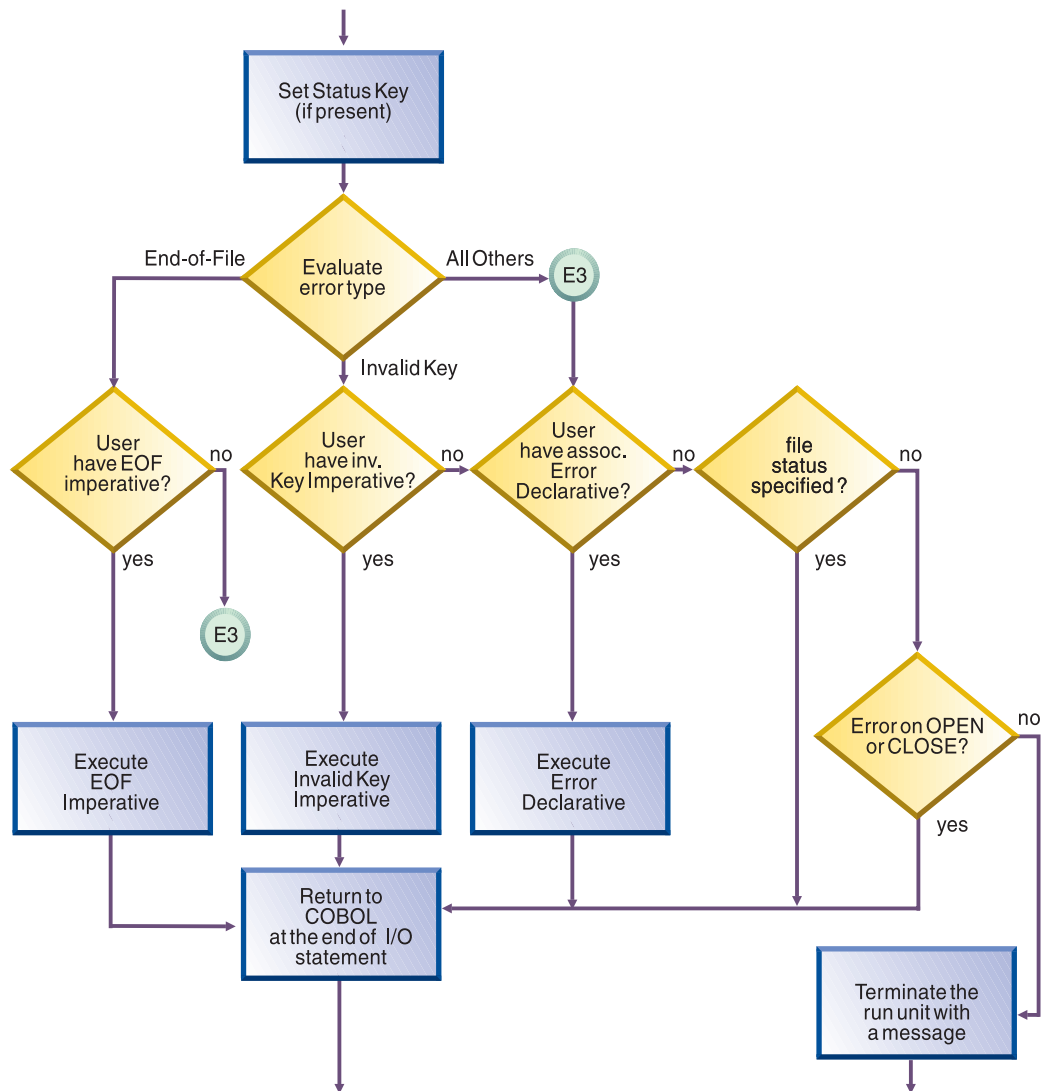
To have your program continue, you must code the appropriate error-recovery procedure: you might code, for example, a procedure to check the value of the file status key.

If you do not handle an input or output error in any of these ways, a COBOL run-time message is written and the run unit ends.

The following figure shows the flow of logic after a file system input or output error occurs:

- VSAM input or output error
- QSAM and line-sequential input or output error

The following figure shows the flow of logic after a VSAM input or output error:



#### RELATED TASKS

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

"Coding ERROR declaratives" on page 129

"Using file status keys" on page 129

"Using file system return codes" on page 131

"Coding INVALID KEY phrases" on page 132

## 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. If you code an AT END phrase, on end-of-file the 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 containing 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 you code 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 performing any associated declarative procedure.

You might choose to code neither an AT END phrase nor an EXCEPTION declarative procedure, but 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 conditions. Here presumably you have some code to take appropriate action.

#### RELATED REFERENCES

AT END phrases (*IBM COBOL Language Reference*)

## Coding ERROR declaratives

You can code one or more ERROR declarative procedures in your COBOL program that will be given control if an input or output error occurs. You can have:

- 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 particular file

Place each such procedure in the declaratives section of your PROCEDURE DIVISION.

In your procedure, you can choose to try corrective action, retry the operation, continue, or end execution. You can use the ERROR declaratives procedure in combination with the file status key if you want a further analysis of the error.

If you continue processing a blocked file, you might lose the remaining records in a block after the record that caused the error.

Write an ERROR declarative procedure if you want the system to return control to your program after an error occurs. If you do not write an ERROR declarative procedure, your job could be canceled or abnormally terminated after an error occurs.

#### RELATED REFERENCES

EXCEPTION/ERROR declarative (*IBM COBOL Language Reference*)

## Using file status keys

After each input or output statement is performed on a file, the system updates values in the two digits of the file status key. In general, a zero in the first digit indicates a successful operation, and a zero in both digits means there is nothing abnormal. Establish a file status key by using the FILE STATUS clause in the FILE-CONTROL paragraph and data definitions in the DATA DIVISION.

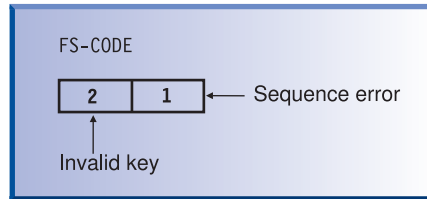
```
FILE STATUS IS data-name-1
```

The variable *data-name-1* specifies the two-character COBOL file status key that should be defined in the WORKING-STORAGE SECTION. This *data-name* cannot be variably located.

Your program can check the COBOL file status key to discover whether an error has been made and, if so, what type of error it is. For example, suppose 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.  
You can then 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 every input or output request.  
If it contains a value other than 0, your program can issue an error message or can act based on the value.  
You do not have to reset the status key code, because it is set after each input or output attempt.

For VSAM, STL, and Btrieve files, in addition to the file status key, you can code a second identifier in the FILE STATUS clause to get more detailed information on file system input or output requests.

You can use the status key alone, or in conjunction with the INVALID KEY option, or to supplement the EXCEPTION or ERROR declarative. Using the 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 return codes” on page 131

#### RELATED TASKS

“Using file system return codes” on page 131

### Example: file status key

This COBOL code performs a simple check on the 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 return codes

Often the two-character FILE STATUS code is too general to pinpoint the disposition of a request. You can get more detailed information about VSAM file system input and output requests by coding a second status area:

```
FILE STATUS IS data-name-1 data-name-2
```

The variable *data-name-1* specifies the two-character COBOL file status key. The variable *data-name-2* specifies a data item that contains the file system return code when the COBOL file status key is not zero. *data-name-2* is at least 6 bytes long.

### STL and Btrieve file systems

If *data-name-2* is 6 bytes long, it contains the return code. If it is longer than 6 bytes, it also contains a message with further information. For example:

```

01 my-file-status-2.
   02 exception-return-value PIC 9(6).
   02 additional-info PIC X(100).

```

Suppose 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 telling what keys you need to perform the open would be returned in additional-info.

### VSAM file system

You must define *data-name-2* as PICTURE  $X(n)$  and USAGE DISPLAY attributes, where  $n$  is 6 or greater. The PICTURE string value represents the first  $n$  bytes of the VSAM reply message structure. If the size of the reply message structure,  $m$ , is less than  $n$ , only the first  $m$  bytes contain useful information.

“Example: checking file system return codes”

#### RELATED REFERENCES

“STL file system” on page 97

### Example: checking file system return codes

This COBOL code does the following actions:

- Reads an indexed file (starting at the fifth record)
- Checks the file status key after each input or output request
- Displays the VSAM codes when the file status key is not zero

This example also illustrates how output from this program might look if the file being processed contains six records.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FILESYSFILE ASSIGN TO FILESYSFILE
    ORGANIZATION IS INDEXED
    ACCESS DYNAMIC
    RECORD KEY IS FILESYSFILE-KEY
    FILE STATUS IS FS-CODE, FILESYS-CODE.
DATA DIVISION.
FILE SECTION.

```

```

FD  FILESYSFILE
   RECORD 30.
01  FILESYSFILE-REC.
    10 FILESYSFILE-KEY          PIC X(6).
    10 FILLER                   PIC X(24).
WORKING-STORAGE SECTION.
01  RETURN-STATUS.
    05 FS-CODE                  PIC XX.
    05 FILESYS-CODE             PIC X(6).
PROCEDURE DIVISION.
    OPEN INPUT FILESYSFILE.
    DISPLAY "OPEN INPUT FILESYSFILE FS-CODE: " FS-CODE.

    IF FS-CODE NOT = "00"
        PERFORM FILESYS-CODE-DISPLAY
        STOP RUN
    END-IF.

    MOVE "000005" TO FILESYSFILE-KEY.
    START FILESYSFILE KEY IS EQUAL TO FILESYSFILE-KEY.
    DISPLAY "START FILESYSFILE KEY=" FILESYSFILE-KEY
           " FS-CODE: " FS-CODE.

    IF FS-CODE NOT = "00"
        PERFORM FILESYS-CODE-DISPLAY
    END-IF.

    IF FS-CODE = "00"
        PERFORM READ-NEXT UNTIL FS-CODE NOT = "00"
    END-IF.

    CLOSE FILESYSFILE.
    STOP RUN.

READ-NEXT.
    READ FILESYSFILE NEXT.
    DISPLAY "READ NEXT FILESYSFILE FS-CODE: " FS-CODE.
    IF FS-CODE NOT = "00"
        PERFORM FILESYS-CODE-DISPLAY
    END-IF.
    DISPLAY FILESYSFILE-REC.

FILESYS-CODE-DISPLAY.
    DISPLAY "FILESYS-CODE ==>", FILESYS-CODE.

```

## Coding INVALID KEY phrases

You can include INVALID KEY phrases on READ, START, WRITE, REWRITE, and DELETE requests for VSAM indexed and relative files. The INVALID KEY phrase is given control if an input or output error occurs because of a faulty index key.

Use the FILE STATUS clause with INVALID KEY to evaluate the status key and determine the specific INVALID KEY condition.

### INVALID KEY and ERROR declaratives

INVALID KEY phrases differ from ERROR declaratives in these ways:

- INVALID KEY phrases operate for only limited types of errors, whereas the ERROR declarative encompasses all forms.
- INVALID KEY phrases are coded directly onto the input or output verb, whereas ERROR declaratives are coded separately.
- INVALID KEY phrases are specific for a single input or output operation, whereas ERROR declaratives are more general.

If you code INVALID KEY in a statement that causes an INVALID KEY condition, control is transferred to the INVALID KEY imperative statement. Here, any ERROR declaratives you have coded are not performed.

### **NOT INVALID KEY**

A NOT INVALID KEY phrase that you code is performed only if the statement completes successfully. If the operation fails because of a condition other than INVALID KEY, neither the INVALID KEY nor the NOT INVALID KEY phrase is performed. Instead control passes to the end of the statement after the program performs any associated ERROR declaratives.

“Example: FILE STATUS and INVALID KEY”

### **Example: FILE STATUS and INVALID KEY**

Assume you have a file containing master customer records and need to update some of these records with information in 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 defining indexed organization, random access, MASTER-CUSTOMER-NUMBER as the prime record key, and CUSTOMER-FILE-STATUS as the file status key. The following example shows how you can use FILE STATUS with the INVALID KEY to more specifically determine why an input or output statement failed.

```
.  
  (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 to the system. For example, the system could run out of storage or it could be unable to locate the load module. If you do not have an ON EXCEPTION or ON OVERFLOW clause on the CALL statement, your application might abend.

Use the ON EXCEPTION clause to perform a series of statements and to perform your own error handling. For example:

```
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
```

If program REPORTA is unavailable, control will continue with the ON EXCEPTION clause.

The ON EXCEPTION clause applies only to the availability of the called program. If an error occurs while the called program is running, the ON EXCEPTION clause will not be performed.

---

## Part 2. Compiling, linking, running and debugging your program

### Chapter 10. Compiling, linking, and running

<b>programs</b>	137
Setting environment variables	137
Setting environment variables temporarily	137
Setting environment variables persistently	138
Precedence of environment variables	138
Compiler environment variables	138
Run-time environment variables	140
TZ environment parameter variables	143
Compiling programs	144
Compiling from the command line	144
Examples: using cob2 for compiling	145
cob2 options	145
File names and extensions supported by cob2	147
Compiling using batch files or command files	148
Specifying compiler options with the PROCESS (CBL) statement	148
Correcting errors in your source program	149
Severity codes for compiler error messages	149
Generating a list of compiler error messages	150
Messages and listings for compiler-detected errors	150
Format of compiler error messages	151
Linking programs	152
Specifying linker options	153
Linking within a project environment	153
Linking through the compiler	153
Linking from a make file	154
Linking from the command line	154
Examples: using cob2 for linking	154
Example: overriding linker options	155
Linker input and output files	155
Linker search rules	155
Example: linker search rules	156
File name defaults	156
Correcting errors in linking	156
Linker return codes	157
Linker errors in program names	157
Running programs	158

### Chapter 11. Compiler options

ADATA	160
ANALYZE	161
BINARY	161
CALLINT	162
CHAR.	163
COLLSEQ	165
COMPILE	166
CURRENCY	166
DATEPROC	167
DYNAM	168
ENTRYINT	168
EXIT	169
Character string formats	170

User-exit work area	170
Linkage conventions	171
Parameter list for exit modules	171
Using INEXIT	171
Using LIBEXIT	172
Nested COPY statements	172
Using PRTEXT	173
Using ADEXIT	173
FLAG	174
FLAGSTD	175
FLOAT	176
IDLGEN	177
LIB	178
LINECOUNT	179
LIST	179
MAP	180
NUMBER	181
OPTIMIZE	181
PGMNAME	182
PGMNAME(UPPER)	183
PGMNAME(MIXED)	183
PROBE	183
PROFILE	184
QUOTE/APOST	184
SEPOBJ	185
Batch compilation	185
SEQUENCE	186
SIZE	187
SOSI	187
SOURCE	188
SPACE	189
SQL	189
SSRANGE	189
TERMINAL	190
TEST	190
THREAD	191
TRUNC	192
TRUNC example 1	193
TRUNC example 2	193
TYPECHK	194
VBREF	195
WSCLEAR	195
XREF	196
YEARWINDOW	197
ZWB	197
Compiler-directing statements	198

### Chapter 12. Linker options

/?	204
/ALIGNADDR	204
/ALIGNFILE	204
/BASE	205
/CODE	205
/DATA	206

/DBGPACK, /NODBGPACK . . . . .	206
/DEBUG, /NODEBUG . . . . .	206
/DEFAULTLIBRARYSEARCH, /NODEFAULTLIBRARYSEARCH . . . . .	207
/DLL . . . . .	207
/ENTRY . . . . .	208
/EXECUTABLE . . . . .	208
/EXTDICTIONARY, /NOEXTDICTIONARY . . . . .	208
/FIXED, /NOFIXED . . . . .	209
/FORCE . . . . .	209
/HEAP . . . . .	209
/HELP . . . . .	210
/INCLUDE . . . . .	210
/INFORMATION, /NOINFORMATION . . . . .	210
/LINENUMBERS, /NOLINENUMBERS . . . . .	210
/LOGO, /NOLOGO . . . . .	211
/MAP, /NOMAP . . . . .	211
/OUT . . . . .	212
/PMTYPE . . . . .	212
/SECTION . . . . .	212
/SEGMENTS . . . . .	213
/STACK . . . . .	214
/STUB . . . . .	214
/SUBSYSTEM . . . . .	214
/VERBOSE . . . . .	215
/VERSION . . . . .	215

<b>Chapter 13. Run-time options . . . . .</b>	<b>217</b>
CHECK . . . . .	217
DEBUG . . . . .	218
ERRCOUNT. . . . .	218
FILESYS . . . . .	218
TRAP . . . . .	219
UPSI . . . . .	219

<b>Chapter 14. Debugging . . . . .</b>	<b>221</b>
Debugging with source language. . . . .	221
Tracing program logic . . . . .	222
Finding and handling input-output errors. . . . .	223
Validating data . . . . .	223
Finding uninitialized data . . . . .	223
Generating information about procedures . . . . .	223
Debugging lines . . . . .	224
Debugging statements . . . . .	224
Example: USE FOR DEBUGGING . . . . .	224
Debugging using compiler options . . . . .	225
Finding coding errors . . . . .	226
Checking syntax only. . . . .	226
Compiling conditionally. . . . .	226
Finding line sequence problems . . . . .	227
Checking for valid ranges . . . . .	227
Selecting the level of error to be diagnosed . . . . .	228
Example: embedded messages. . . . .	228
Finding program entity definitions and references . . . . .	230
Listing data items . . . . .	230
Preparing to use the debugger. . . . .	231
Getting listings . . . . .	231
Example: short listing . . . . .	232
Example: SOURCE and NUMBER output . . . . .	234
Example: MAP output . . . . .	235

Example: embedded map summary . . . . .	235
Terms and symbols used in MAP output . . . . .	236
Example: nested program map . . . . .	237
Example: XREF output - data-name cross-references. . . . .	237
Example: XREF output - program-name cross-references. . . . .	238
Example: embedded cross-reference . . . . .	239
Example: VBREF compiler output . . . . .	240
Debugging user exits. . . . .	240
Debugging assembler routines. . . . .	241

---

## Chapter 10. Compiling, linking, and running programs

You can compile and run your applications on supported platforms, whether they are created on a mainframe, an AIX workstation, or a personal computer with Windows. The IBM VisualAge COBOL compiler and run-time environment supports the high subset of ANSI 85 COBOL functions, as do other IBM COBOL products. Although the IBM COBOL language is practically the same across platforms, some minor differences exist between IBM COBOL for OS/390 & VM and IBM VisualAge COBOL.

This section explains how to:

- Set environment variables for the compiler and run time
- Compile and link programs
- Respond to compiler and linker errors
- Run programs that have been compiled and linked

### RELATED TASKS

“Setting environment variables”

“Compiling programs” on page 144

“Correcting errors in your source program” on page 149

“Running programs” on page 158

### RELATED REFERENCES

“Appendix A. Summary of differences with host COBOL” on page 475

---

## Setting environment variables

An environment variable is a variable that describes the way an operating system will run and the devices it will recognize. Environment variables are used by both the compiler and run-time library.

When you installed IBM VisualAge COBOL, the installation process set environment variables to access the IBM VisualAge COBOL compiler and run-time libraries. These variables are listed in the Registry for Windows NT.

You use environment variables to set values that programs need. For example, you use the COBPATH environment variable to define the location where the COBOL run time can find a program when another program dynamically calls it.

To specify environment variables, use the SET command. If you do not specify environment variables, either a default value is applied or the variable is not defined.

When you create object-oriented programs, System Object Model (SOM) requires you to set SOM-specific environment variables.

You can set environment variables temporarily or persistently.

### Setting environment variables temporarily

When you set environment variables temporarily, the values of these variables apply only to programs that you run from the window where you issue the SET

command. You can set environment variables temporarily by using the SET command at the command prompt or as part of a command (.cmd) file.

**Example:** The following command sets the COBPATH environment variable to include two directories when you run programs from this window:

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

## Setting environment variables persistently

When you set environment variables persistently, the values of these variables are defined automatically whenever you boot your computer and apply to all windows and graphical applications. To set environment variables persistently add the environment variable in the System window. Changes made to user environment variables in the System window are stored, but you must restart your computer to make the values available to processes, including the command prompt.

The value that you assign to an environment variable can include other environment variables or the variable itself.

**Example:** To add a directory to the value of COBPATH, which has already been set, issue the following command:

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

## Precedence of environment variables

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.

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

### RELATED REFERENCES

“Compiler environment variables”

“Run-time environment variables” on page 140

“SOM environment variables” on page 312

## Compiler environment variables

The COBOL compiler uses the following environment variables:

### COBCPYEXT

Specifies the file extensions to search for COPY files 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.

### 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. Individual compiler option default values apply.

#### **Example:**

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

### **COBPATH**

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

### **SYSLIB**

Specifies paths to be used for COBOL COPY statements with 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.

### **Library-name**

A user-defined word that specifies the path for the library text.

#### **Example:**

```
SET MYLIB=D:\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 -Ixxx option, if set
3. Paths specified by the SYSLIB environment variable

The search ends when the file is found.

### **Text-name**

A user-defined word that specifies the path for the copybook text.

If you do not set text-name as an environment variable, the compiler uses the default search described with the COPY statement (a compiler-directing statement).

### **DB2DBDFT**

Specifies the database for compiling your programs with embedded SQL statements.

### **RELATED CONCEPTS**

"DB2 coprocessor" on page 245

### **RELATED TASKS**

"Coding SQL statements" on page 245

#### RELATED REFERENCES

“Compiler-directing statements” on page 198

“Chapter 11. Compiler options” on page 159

“cob2 options” on page 145

## Run-time environment variables

The COBOL run-time library uses the following environment variables (the case does not matter on Windows):

### 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.

#### Example:

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

You must set all assignment-names.

If you make an assignment to a user-defined word that was not set as an environment variable, the assignment is made to a file with 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 the *assignment-name*, you can use the environment variable as a COBOL user-defined word in an ASSIGN clause.

Based on the previous SET statement, your COBOL source program could include the following:

```
SELECT CARPOOL ASSIGN TO OUTPUTFILE
```

Because OUTPUTFILE was defined in the environment, this statement would result 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), VSAM, or Btrieve.

### COBMSGs

Specifies the name of a file to which run-time error messages will be written.

To capture run-time error messages into a file, use the SET command to set COBMSGs to a file name. If your program has a run-time error that terminates the application, the file that COBMSGs is set to will contain the error message indicating the reason for termination.

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

### COBPATH

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.

#### Example:

```
SET COBPATH=D:\pgmpath\pgmd11
```

### COBRTOPT

Specifies the COBOL run-time options.

Separate run-time options by a comma or a colon. Use parentheses or equal signs (=) as the delimiters for suboptions. Options are not case sensitive.

The defaults for individual run-time option apply.

**Example:**

The following commands are equivalent:

```
SET COBRTOPT=TRAP=ON:errcount
```

```
SET COBRTOPT=trap(on):ERRCOUNT
```

**EBCDIC\_CODEPAGE**

Specifies an EBCDIC code set applicable to the EBCDIC data being processed by programs compiled with the CHAR(EBCDIC) or CHAR(S390) compiler option.

To set the EBCDIC code set, issue the following command, where *codepage* is the name of the code set to be used:

```
SET EBCDIC_CODEPAGE=codepage
```

If EBCDIC\_CODEPAGE is not set, the default is the EBCDIC code page of the current locale. If multiple code pages are available for the current locale, the CHAR(EBCDIC) compiler option must be set.

**LANG**

Specifies the national language locale name in effect for message catalogs and help files.

LANG must always be set and is given an initial value during installation. The default is EN\_US.

The run-time library uses LANG to access the message catalog. If LANG is not set correctly, run-time messages appear in an abbreviated form.

**Example:**

The following command sets the language locale name to U.S. English:

```
SET LANG=En_US
```

**LC\_COLLATE**

Determines the locale to be used to define the behavior of ranges, equivalence classes, and multicharacter collating elements.

The default is the locale specified by the LANG environment variable.

**LC\_MESSAGES**

Determines the locale which defines the language in which messages are written.

The default is the locale specified by the LANG environment variable.

**LC\_TIME**

Determines the locale for date and time formatting information.

The default is the locale specified by the LANG environment variable.

**LOCPATH**

Specifies the search path where the locale information database exists. It is a colon-separated list of directory names.

LOCPATH is used when setting up locale for a process. It is also used to locate conversion tables for EBCDIC data support.

## 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. The defaults vary.

When you set NLSPATH, be sure to add to the NLSPATH, not replace it. Other programs might use this environment variable. Also, %L and %N must be uppercase.

The run-time library uses NLSPATH to access the message catalog. If NLSPATH is not set correctly, run-time messages appear in an abbreviated form.

### Example:

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

## SYSIN, SYSIPT, SYSOUT, SYSLIST, SYSLST, CONSOLE, SYSPUNCH, SYSPCH

These COBOL environment names 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.

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 value by default and are not valid unless you explicitly define them.

### Example:

The following command defines CONSOLE:

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

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

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

**Tip:** If you set the environment variables SYSIN and SYSOUT to files that have write permission, Visual Builder applications can use ACCEPT and DISPLAY statements to communicate with the user.

## TEMP

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.

### Example:

```
SET TEMP=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 give values for any of *sm*, *sw*, *sd*, *st*, *em*, *ew*, *ed*, *et*, or *shift*, you must give values for all of them. If any of these values is not valid, the entire statement is considered not valid, and the time zone information is not changed.

**Example:**

```
SET TZ=CST6CDT
```

The preceding statement sets the standard time zone to CST, the daylight saving time to CDT, and sets a difference of six hours between CST and UTC. It does not set any values for the start and end of daylight saving time.

Other possible values are PST8PDT for Pacific United States and MST7MDT for Mountain United States.

**RELATED TASKS**

“Identifying files” on page 95

“Setting the locale” on page 411

**RELATED REFERENCES**

“TZ environment parameter variables”

“Chapter 13. Run-time options” on page 217

“CHAR” on page 163

## **TZ environment parameter variables**

The values for the TZ variable are defined below.

Variable	Description	Default value
SSS	Standard time zone identifier. This must be three characters, must begin with a letter, and can contain spaces.	EST
<i>n</i>	Difference (in hours) between the standard time zone and coordinated universal time (UTC), formerly Greenwich mean time (GMT). A positive number denotes time zones west of the Greenwich meridian, a negative number denotes time zones east of the Greenwich meridian.	5
DDD	Daylight saving time (DST) zone identifier. This must be three characters, must begin with a letter, and can contain spaces.	EDT
<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 to select the options used to compile your programs:

- Use the compiler options dialog in a project environment.
- Set the COBOPT environment variable from the command line.
- Specify compiler environment variables and cob2 options in a batch file or on a command line.
- Use PROCESS (CBL) or \*control statements.

An option that you specify using PROCESS overrides every other option specification, except for nonoverridable options that are selected during product installation.

When you work in a project environment, you can compile and link your application using the build action. Alternatively, you can start the IBM VisualAge COBOL compiler separately to compile an individual .CBL file or a group of .CBL files. You can also check the syntax of remote host COBOL files using the IBM VisualAge COBOL compiler and submit JCL to compile on the host from an MVS project.

You can also compile IBM VisualAge COBOL programs from the command line.

### RELATED TASKS

Compiling host applications (Remote applications online help)

Building a non-GUI application (Project environment online help)

Building a Visual Builder application (Project environment online help)

"Compiling from the command line"

Porting applications between platforms

("Chapter 22. Porting applications between platforms" on page 351) "Specifying compiler options with the PROCESS (CBL) statement" on page 148

### RELATED REFERENCES

"Appendix A. Summary of differences with host COBOL" on page 475

"cob2 options" on page 145

"Chapter 11. Compiler options" on page 159

## Compiling from the command line

The command cob2 is the command-line utility that starts the COBOL compiler and linker. cob2 accepts options to control the compilation and link-edit in any order on the command line. To compile multiple files, specify the file names at any position in the command syntax, using spaces to separate options and file names.

```
»cob2 [options] filenames«
```

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 all files with certain extensions to the linker and passes all other files to the compiler. The default location for compiler input and output is the current directory.

"Examples: using cob2 for compiling" on page 145

#### RELATED TASKS

Compiling in a project environment (Project environment online help)

#### RELATED REFERENCES

"File names and extensions supported by cob2" on page 147

"cob2 options"

"Chapter 11. Compiler options" on page 159

### Examples: using cob2 for compiling

The following two commands are equivalent:

```
cob2 -g filea.cbl fileb.cbl -v -qflag(w)
cob2 filea.cbl -qflag(w) -g -v fileb.cbl
```

The following examples show the output produced by the compiler for specific cob2 specifications.

To compile:	Enter:	The compiler produces:
alpha.cbl	cob2 -c alpha.cbl	alpha.obj, alpha.lst, and alpha.adt
alpha.cbl and beta.cbl	cob2 -c alpha.cbl c:\mydir\beta.cbl	alpha.obj, beta.obj; alpha.lst, beta.lst; alpha.adt, beta.adt in the current directory.
alpha.cbl with the LIST and NOADATA options	cob2 -qlist,noadata alpha.cbl	alpha.asm, alpha.obj, alpha.lst, and alpha.exe

#### RELATED TASKS

"Compiling from the command line" on page 144

### cob2 options

] To specify a cob2 option, precede it by a dash (-). Do not use a slash (/) for cob2 options unless you want to pass linker options using cob2.

#### -b"xxx"

Passes the xxx string 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 between -b and xxx.

Alternatively, you can specify linker options directly as individual cob2 options.

**Example:** To pass the /DE option to the linker, use:

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

**-c** Compiles programs but does not link them.

#### -cmain

Makes a C or PL/I object file containing 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 containing a C or PL/I main. Unpredictable execution behavior will occur if this is attempted and no diagnostics are issued.

**Example:** The following commands are equivalent:

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

Both generate the executable file myCmain.exe with the main entry point being the C main() function contained in the myCmain.obj object file.

**-comprc\_ok=*n***

Controls the cob2 behavior on the return code from the compiler. If the return code returned by the compiler 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 returned 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 or .PPR) file specified in the cob2 command is the name of the DLL (and .LIB and .EXP files).

**-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.

**-host** Sets all host data compiler options:

- BINARY(S390)
- CHAR(EBCDIC)
- COLLSEQ(EBCDIC)
- FLOAT(S390)

This option will present run-time command-line arguments in host data format, that is, EBCDIC for character data, and big-endian for binary data.

**-l*xxx*** Adds a path *xxx* to the directories to be searched for COPY files if a *library-name* is not specified (This is uppercase “eye,” not lowercase “el.”)

Only a single path is allowed per -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.

**-main:*xxx***

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

**Example:** The following command results in abc being 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 will, in the absence of a response file, be 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.

**-p** Includes the profile hooks that allow the Performance Analyzer to monitor the application execution and create a trace file.

This option is equivalent to compiling with the `PROFILE` compiler option and linking in the Performance Analyzer module `IWZPAN40.OBJ`.

**-qxxx** Calls the compiler, where `xxx` is any compiler option.

If a parenthesis is part of the compiler option or suboption, or if a series of options are specified, include them in quotes. For multiple options, delimit each option by a blank or comma. Do not insert spaces between `-q` and `xxx`.

**Example:** The following two commands are equivalent:

`-qoptiona,optionb`  
`-q"optiona optionb"`

**-v** Displays compile and link steps, and executes them.

**-#** Displays compile and link steps, but does not execute them.

#### RELATED REFERENCES

"File names and extensions supported by `cob2`"

"Compiler environment variables" on page 138

"Run-time environment variables" on page 140

### File names and extensions supported by `cob2`

Files with `@` as the first character and files with the following extensions are assumed to be linker parameters and 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 syntax with an extension of `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 with the file 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 not specified, no map file is generated.

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

All other files are processed by the compiler. The file extension `CBL` is most commonly used for COBOL source.

#### RELATED TASKS

"Creating module definition files" on page 394

## Compiling using batch files or command files

To use a batch file or command file to automate your cob2 tasks, use the following syntax to prevent the command shell from passing invalid syntax to cob2:

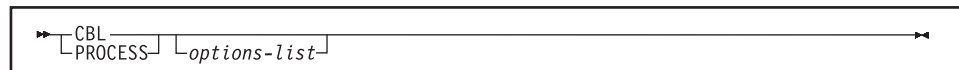
- Use an equal sign and colon rather than parentheses to specify compiler suboptions.
- Use an underscore rather than apostrophe where a compiler option requires a suboption to be delimited by apostrophes.
- Do not use any blanks in the option string.

Examples:

Use with batch or command file	Use with command line
-qBINARY=NATIVE:,ENTRYINT=OPTLINK:	-qBINARY(NATIVE),ENTRYINT(OPTLINK)
-qEXIT=INEXIT=_String_,MYMODULE::	-qEXIT(INEXIT('String',MYMODULE))

## Specifying compiler options with the PROCESS (CBL) statement

You can code compiler options on the PROCESS statement in your COBOL source .CBL programs.



Place the PROCESS statement before the IDENTIFICATION DIVISION header and before any comment lines or compiler-directing statements.

You can start PROCESS in columns 1 through 66. A sequence field is allowed in columns 1 through 6. When used with a sequence field, PROCESS can start in columns 8 through 66. If used, the sequence field must contain six characters, and the first character must be numeric.

You can use CBL as a synonym for PROCESS. CBL can start in columns 1 through 70. When used with a sequence field, CBL can start in columns 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 multiple PROCESS statements are used, they must follow one another with no intervening statement of any other type. You cannot continue options across multiple PROCESS statements.

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

---

## Correcting errors in your source program

Messages about source code errors indicate where the error happened (LINEID), and the text of the message tells you what the problem is. With this information, you can correct the source program and recompile.

Although you should try to correct errors, it is not necessary to fix all of them. A warning-level or informational-level message can be left in a program without much risk, and you might decide that the recoding and compilation 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.

Unrecoverable-level errors are in a class by themselves. In contrast with the four lower levels of errors, a 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 and listing. If the message is received for a program with many S-level syntax errors, correct those errors and compile the program again. You can also resolve job set-up cases yourself (problems such as missing data set definitions or insufficient storage for compiler processing) by making changes to the compile job. If your compile job set-up is correct and you have corrected the S-level syntax errors, you need to call IBM to investigate other U-level errors.

After correcting the errors in your source program, recompile the program. If this second compilation is successful, go on 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” on page 150

### RELATED REFERENCES

“Messages and listings for compiler-detected errors” on page 150

## Severity codes for compiler error messages

Errors the compiler can detect fall into the following five categories of severity:

Level of message	Purpose
Informational (I) (return code=0)	To inform you. No action is required and the program executes correctly.
Warning (W) (return code=4)	To indicate a possible error. The program probably executes correctly as written.
Error (E) (return code=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) (return code=12)	To indicate a condition that is a serious error. The compiler was unable to correct the error. The program does not execute correctly, and execution should not be attempted. Object code might not be created.
Unrecoverable (U) (return code=16)	To indicate an error condition of such magnitude that the compilation was terminated.

In the following example, the part of the statement that caused the message to be issued is enclosed in quotes.

Line ID	Message code	Message text
2	IGYDS0009-E	"PROGRAM" should not begin in area "A." It was processed as if found in area "B."
2	IGYDS1089-S	"PROGRAM" was invalid. Scanning was resumed at the next area "A" item, level-number, or the start of the next clause.
2	IGYDS0017-E	"ID" should begin in area "A." It was processed as if found in area "A."
2	IGYDS1003-E	A "PROGRAM-ID" paragraph was not found. Program name "CBLPGM01" was assumed.
2	IGYSC1082-E	A period was required. A period was assumed before "ID."
2	IGYDS1102-E	Expected "DIVISION," but found "ALONGPRO." "DIVISION" was assumed before "ALONGPRO."
2	IGYDS1082-E	A period was required. A period was assumed before "ALONGPRO."
2	IGYDS1089-S	"ALONGPRO" was not valid. Scanning was resumed at the next area "A" item, level-number, or the start of the next clause.
2	IGYDS1003-E	A "PROGRAM-ID" paragraph was not found. Program name "CBLPGM02" was assumed.
3	IGYPS0017-E	"PROCEDURE" should begin in area "A." It was processed as if found in area "A."
34	IGYSC0137-E	Program-name "ALONGPRO" did not match the name of any open program. The "END PROGRAM" statement was assumed to have ended program "CBLPGM02."
34	IGYSC0136-E	Program "CBLPGM01" required an "END PROGRAM" statement at this point in the program. An "END PROGRAM" statement was assumed.

## Generating a list of compiler error messages

You can generate a complete listing of compiler diagnostic messages, with their explanations, by compiling a program with a program name of ERRMSG specified in the PROGRAM-ID paragraph, like this:

```
Identification Division.
Program-ID. ErrMsg.
```

You can omit the rest of the program.

### Messages and listings for compiler-detected errors

As the compiler processes your source program, it checks for COBOL language errors that you might have made. 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, with the file extension LST. For example, the compiler listing for myfile.cbl would be myfile.lst. The listing file is written to the directory from which cob2 was run.

Each message in the listing gives 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, or BASIS, COPY, and REPLACE statements are displayed near the top of your 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 your listing.

#### RELATED TASKS

- “Correcting errors in your source program” on page 149
- “Generating a list of compiler error messages” on page 150

#### RELATED REFERENCES

- “Format of compiler error messages”
- “Severity codes for compiler error messages” on page 149
- “FLAG” on page 174

### Format of compiler error messages

Each message issued by the compiler has the following form:

*nnnnnn IGYppxxx-l message-text*

*nnnnnn*

The number of the source statement of the last line 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 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.

**xxx** 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 itself 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 149
- “FLAG” on page 174

---

## Linking programs

After the compiler has created object modules from your source files, use the linker to link them together with the IBM VisualAge COBOL run-time libraries to create an .EXE file or a .DLL file. The linker produces .EXE files by default.

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

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

- In a project environment by using the options dialogs. The options that you select while creating or changing a project are saved with that project.
- On the command line. You can specify options anywhere on the command line, as in `ilink /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 (until you reboot your computer).
- In the ILINK environment variable in the CONFIG.SYS file. The options are set when you boot your computer, and are in effect every time you use the linker unless you override them by using a .CMD file or by specifying options on the command line.

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:

- In a project environment, from the menu for an object file or from the menu for a project as part of the make or build process
- Through the compiler using COB2, which automatically starts the linker
- Through a make file, which starts both the compiler and the linker
- From the command line

### RELATED TASKS

- “Specifying linker options” on page 153
- “Linking within a project environment” on page 153
- “Linking through the compiler” on page 153
- “Linking from a make file” on page 154
- “Linking from the command line” on page 154
- “Creating module definition files” on page 394

### RELATED REFERENCES

- “Chapter 12. Linker options” on page 203
- “Linker input and output files” on page 155
- “Linker search rules” on page 155

## Specifying linker options

You can specify linker options in lowercase, uppercase, or mixed case. You can also substitute a dash (-) for the slash (/) preceding the option. For example, -DEBUG is equivalent to /DEBUG. You can specify options in either a short or long form. For example, /DE, /DEB, and /DEBU are all equivalent to /DEBUG. The summary of linker options has the shortest acceptable form for each option.

You can use lowercase and uppercase, short and long forms, dashes, and slashes on one command line, as in:

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

Separate options with a space or tab character.

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 is a decimal number. For example, 1234 is a decimal number.

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

### Hexadecimal

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

#### RELATED TASKS

"Linking programs" on page 152

#### RELATED REFERENCES

"Chapter 12. Linker options" on page 203

## Linking within a project environment

To use the linker in a project environment, do the following steps:

1. Set the linker options in the Link Options Dialog from the **Options -> Link** menu. If you want to compile and link in one step, set the linker options on the Link page in the Compile Options Dialog from the **Options -> Compile** menu. Options that you have set in either dialog are unique to that dialog. Therefore, options that you have set in the Link Options Dialog are not reflected in the Compiler Options Dialog. The same is true for options that you have set in the Compiler Options Dialog.
2. Select **Build** from either the **Actions** menu or the project toolbar. Your project is built using the linker as required.

## Linking through the compiler

When you start the IBM VisualAge COBOL compiler, it compiles the object files from your source code and then automatically 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 cob2 option to pass options to the linker.

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

“Examples: using cob2 for linking”

#### RELATED REFERENCES

“Chapter 12. Linker options” on page 203

## Linking from a make file

Use a make file to organize the sequence of actions (such as compiling and linking) required to build your project. You can then invoke all the actions in one step. The `NMAKE` utility saves you time by performing actions on only the files that have changed, and on the files that incorporate or depend on the changed files.

You can write the make file yourself, or you can manage the make file from within a project. When you build in a project environment, a make file is created and maintained automatically.

## Linking from the command line

When you start the linker from the command line, the linker assumes that any input it cannot recognize as other files, options, or directories must be an object file. 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.

Examples: using cob2 for linking

“Example: overriding linker options” on page 155

#### RELATED REFERENCES

“Linker search rules” on page 155

### Examples: using cob2 for linking

The following examples illustrate the use of `cob2`:

- 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
```

This command 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 alpha.cb1 alpha.def
```

#### RELATED TASKS

“Linking from the command line” on page 154

### Example: overriding linker options

In the following example, options on the command line override options in the environment variable. Suppose you enter the following commands:

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

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

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

The last command links the file prog.obj to produce prog.exe, using the option /NODEFAULTLIBRARYSEARCH, in addition to the options /NOIGNORECASE and /ALIGNMENT:256. The /NODEBUG option on the command line overrides the /DEBUG option in the environment variable, and the linker links without the /DEBUG option.

#### RELATED TASKS

“Linking from the command line” on page 154

## 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. The executable output can be either an executable program file (extension EXE) or a dynamic link library (extension 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 compiled or assembled:

- In 32-bit OMF format
- For Windows NT Version 3.5.1 (or higher) or Windows 95
- For the 80386, 80486, and Pentium microprocessors

### Linker search rules

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

1. The directory you specified for the file, or the current directory if you did not give 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 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

“File name defaults”

### Example: linker search rules

In this example, you want the linker to link 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 following 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 155

## File name defaults

If you do not enter a file name, the linker assumes the defaults shown below.

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

If you use the PGMNAME(UPPER) compiler option, then the names of subprograms referenced in CALL statements are translated to uppercase. So, for example, the compiler translates Call "RexxStart" to Call "REXXSTART".

This change affects the linker, which recognizes case-sensitive names. If the real name of the called program is REXXStart, the linker will not find it, and will produce an error message saying that REXXSTART is an unresolved external reference.

This type of error typically happens when you are calling API routines 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 your CALL statements specify the correct names, with the correct mix of uppercase and lowercase, of the API routines.

#### RELATED REFERENCES

“Linker errors in program names”

## Linker return codes

The linker has the following return codes:

Code	Meaning
0	The link completed successfully. The linker detected no errors, and issued no warnings.
4	The linker issued warnings. There may 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).

If you start the linker through a makefile, you can force NMAKE to ignore warnings by putting -7 before the ILINK command. If you start the linker through the compiler, then a return code of zero is issued for warnings.

## 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 these two ways:

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

This convention is known as “name decoration.” 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:

```
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 an error in the linker because the linker will not be able to resolve the call to `_SubProg@8`.

---

## Running programs

To run a COBOL program, do the following:

1. Make sure that any needed environment variables are set. For example, if your program uses an environment variable name to assign a value to a system file name, set the environment variable.
2. Run the program.
  - From a project environment, click the **Run** icon.
  - From the command line, type the name of the executable module on the command line or execute a command file that invokes the module. For example, if `cob2 alpha.cbl beta.cbl` is successful, you can run the program by typing `alpha`.
3. Correct run-time errors.

**Tip:** If run-time messages are abbreviated or incomplete, the environment variables `LANG` and `NLSPATH` might be incorrectly set.

### RELATED TASKS

“Setting environment variables” on page 137

### RELATED REFERENCES

“Run-time environment variables” on page 140

“Appendix F. Run-time messages” on page 535

## Chapter 11. Compiler options

You can direct and control your compilation in two ways:

- By using compiler options
- By using compiler-directing statements (compile 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.

Aspect of your program	Compiler option	Default	Abbreviations
Source language	"LIB" on page 178	LIB	None
	"NUMBER" on page 181	NONUMBER	NUM NONUM
	"SEQUENCE" on page 186	SEQUENCE	SEQ NOSEQ
	"QUOTE/APOST" on page 184	QUOTE	Q APOST
	"SQL" on page 189	SQL("")	None
Date processing	"DATEPROC" on page 167	NODATEPROC, or DATEPROC(FLAG) if only DATEPROC is specified	DP
	"YEARWINDOW" on page 197	YEARWINDOW(1900)	YW
Maps and listings	"LINECOUNT" on page 179	LINECOUNT(60)	LC
	"LIST" on page 179	NOLIST	None
	"MAP" on page 180	NOMAP	None
	"SOURCE" on page 188	SOURCE	S NOS
	"SPACE" on page 189	SPACE(1)	None
	"TERMINAL" on page 190	NOTERMINAL	TERM NOTERM
	"VBREF" on page 195	NOVBREF	None
	"XREF" on page 196	NOXREF	X NOX
Object module generation	"COMPILE" on page 166	NOCOMPILE(S)	C NOC
	"OPTIMIZE" on page 181	NOOPTIMIZE	OPT NOOPT
	"PGMNAME" on page 182	PGMNAME(UPPER) or PGMNAME(MIXED) for Visual Builder GUI applications	PGMN(LU LM)
	"SEPOBJ" on page 185	SEPOBJ, or NOSEPOBJ for Visual Builder GUI applications	None
Object code control	"BINARY" on page 161	NATIVE	None
	"CHAR" on page 163	CHAR(NATIVE)	None
	"FLOAT" on page 176	FLOAT(NATIVE)	None
	"TRUNC" on page 192	TRUNC(STD)	None
	"ZWB" on page 197	ZWB	None
CALL statement behavior	"DYNAM" on page 168	NODYNAM	DYN NODYN

Aspect of your program	Compiler option	Default	Abbreviations
Debugging and diagnostics	"FLAG" on page 174	FLAG(I)	F NOF
	"FLAGSTD" on page 175	NOFLAGSTD	None
	"TEST" on page 190	NOTEST	None
	"SSRANGE" on page 189	NOSSRANGE	SSR NOSSR
Other	"ADATA" on page 174	ADATA	None
	"ANALYZE" on page 161	NOANALYZE	None
	"CALLINT" on page 162	CALLINT(SYSTEM,NODESC)	None
	"COLLSEQ" on page 165	COLLSEQ(BIN)	None
	"ENTRYINT" on page 168	ENTRYINT(SYSTEM)	None
	"EXIT" on page 169	EXIT(ADEXIT(IWZRMGUX))	EX(INX,LIBX,PRTX,ADX)
	"IDLGEN" on page 177	NOIDLGEN	IDL NOIDL
	"PROBE" on page 183	PROBE	None
	"PROFILE" on page 184	PROFILE	None
	"SIZE" on page 187	2097152 bytes (approximately 2 MB)	SZ
	"THREAD" on page 191	NOTHRED	None
	"TYPECHK" on page 194	NOTYPECHK	TC NOTC
	"WSCLEAR" on page 195	NOWSCLEAR	None

**Installation defaults:** The defaults listed with the options above are the defaults shipped with the product. They might have been changed by your installation. The default options that were set up when your compiler was installed are in effect for your program unless you override them with other options.

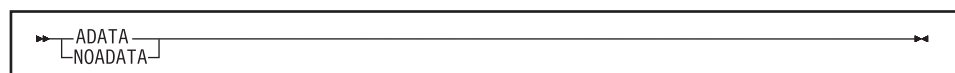
To find out the default compiler options in effect, run a test compilation without specifying any options; the output listing lists the default options specified by your installation. In some installations, certain compiler options are set up so that you cannot override them. If you have problems, see your system administrator.

**Performance considerations:** The BINARY, CHAR, DYNAM, FLOAT, OPTIMIZE, SSRANGE, TEST, and TRUNC compiler options can all affect run-time performance.

#### RELATED REFERENCES

"Compiler-directing statements" on page 198

## ADATA



Default is: ADATA

Abbreviations are: None

Use ADATA when you want the compiler to create a SYSADATA file, which contains records of additional compilation information. This information is used by other tools, which will set ADATA ON for their use. The size of this file generally grows with the size of the associated program.

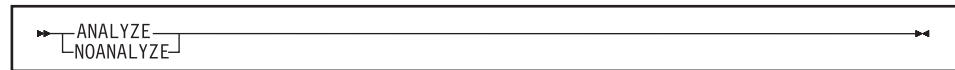
You cannot specify ADATA in a PROCESS (CBL) statement; it can be specified only in one of the following ways:

- On invocation of the compiler using an option list
- As a command option
- As an installation default

RELATED REFERENCES  
"EXIT" on page 169

---

## ANALYZE



Default is: NOANALYZE

Abbreviations are: None

Use ANALYZE when you want the compiler to check the syntax of embedded SQL and CICS statements in addition to native COBOL statements.

No executable code is generated when this compiler option is specified, regardless of the COMPILE|NOCOMPILE setting.

When you specify the ADATA option with this option, you can create a SYSADATA file for later analysis by program understanding tools.

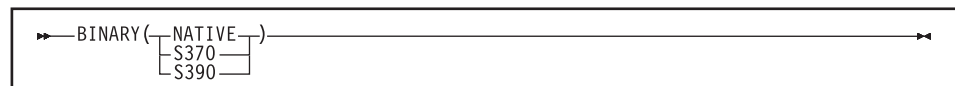
This option can be set as the installation default option or as a compiler invocation option, but cannot be set on a CBL or PROCESS statement.

The specification of the ANALYZE option forces the handling of the following character strings as reserved words:

- CICS
- EXEC
- END-EXEC
- SQL

---

## BINARY



Default is: NATIVE

Abbreviations are: None

Specifying **NATIVE** means that **BINARY**, **COMP**, and **COMP-4** data items are represented in the native format of the platform or product. For example, binary data on a workstation would be stored in *little-endian* format (most significant digit at the highest address). Binary data on AIX would be stored in *big-endian* format (least significant digit at the highest address).

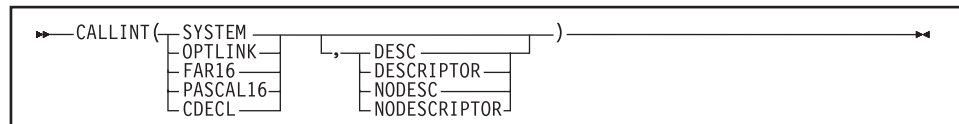
Specifying **S370** or **S390** means that binary data is represented in the big-endian format. However, **COMP-5** binary data and data items defined with the **NATIVE** keyword on the **USAGE** clause are not impacted by the **BINARY(S390)** option. They are always stored in the native format of the platform.

**Visual Builder:** Visual Builder applications require **BINARY(NATIVE)**. Do not change this default setting.

**Object-oriented programs:** Do not specify **BINARY(S370)** or **BINARY(S390)** in object-oriented programs.

---

## CALLINT



Default is: **CALLINT(SYSTEM,NODESC)**

Abbreviations are: None

Use **CALLINT** to indicate the call interface convention applicable to **CALLs**.

This option may be overridden for specific call statements via the compiler directive **>>CALLINT**.

**ENTRYINT** is used for the selection of the call interface convention for the program entry point or points.

- 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 system Windows APIs.

**Alert:** This convention cannot be used in all cases when the called program has multiple entry points.

### **OPTLINK**

The **OPTLINK** suboption specifies that the call convention is that of the **\_OPTLINK** convention of VisualAge for C++ for OS/2 and VisualAge for C++ for Windows.

**FAR16** The **FAR16** suboption specifies that the call convention is that of the **\_FAR16\_Cdecl** convention.

### **PASCAL16**

The **PASCAL16** suboption specifies that the call convention is that of the **\_FAR16\_Pascal** convention.

**CDECL** The CDECL suboption specifies that the call interface convention is that of the CDECL calling convention as defined by Microsoft Visual C/C++ for Windows.

- Specifying if the argument descriptors are to be generated or not:

**DESC** The DESC suboption specifies that an argument descriptor is passed for each argument on a CALL statement.

**Note:** 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 on a CALL statement.

**NODESCRIPTOR**

The NODESCRIPTOR suboption is synonymous with the NODESC suboption.

**Visual Builder:** Visual Builder applications require CALLINT(SYSTEM,NODESCRIPTOR), which is the default specification in the GUI compile options notebook. Do not change this default setting.

**RELATED TASKS**

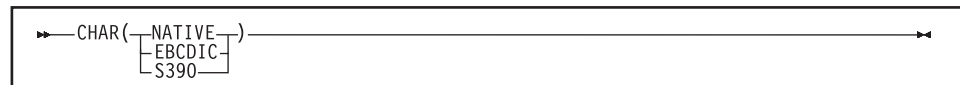
“Coding multiple entry points” on page 381

**RELATED REFERENCES**

“Compiler-directing statements” on page 198

---

## CHAR



Default is: CHAR(NATIVE)

Abbreviations are: None

Specify CHAR(NATIVE) to use the native character representation format of the platform. For VisualAge COBOL, this is ASCII.

CHAR(EBCDIC) and CHAR(S390) are synonymous and indicate that DISPLAY data items are in the data representation of System/390 (EBCDIC).

The following are affected by the CHAR(EBCDIC) compiler option:

- **USAGE DISPLAY items**

- Single-byte characters with USAGE DISPLAY and double-byte characters with USAGE DISPLAY-1 are treated as EBCDIC:
  - ASCII data is converted to EBCDIC on ACCEPT from the terminal.
  - EBCDIC data is converted to ASCII on DISPLAY to the terminal.
  - The EBCDIC equivalent of an ASCII literal is used for assignment to EBCDIC character data. See the table below for the rules on the comparison of character data with the CHAR(EBCDIC) option in effect.
  - Editing is also done with EBCDIC characters.

- Any padding is done using EBCDIC spaces. This includes alphanumeric operations (for example, assignments and compares) on group items regardless of the definition of the elementary items in the group items.
- Figurative constant SPACE or SPACES used in a VALUE clause for an assignment to or in a relational condition with a DISPLAY item is treated as single-byte EBCDIC spaces (that is, X'40').
- CLASS tests are performed based on EBCDIC value ranges.
- The program name in CALL *identifier*, CANCEL *identifier*, or in the format-6 SET statement is converted to ASCII characters if the identifier is EBCDIC.
- The file name in the data name in ASSIGN USING *data-name* is converted to ASCII characters if the data name is EBCDIC.
- The file name in SORT-CONTROL is converted to ASCII characters before being passed to the sort or merge function.  
Note that the SORT-CONTROL special register has the implicit USAGE DISPLAY definition.
- Zoned decimal data (numeric picture with USAGE DISPLAY) and external floating-point data. For example, zoned decimal PIC S9 value "1" is treated as X'C1' instead of X'31'.

- **Group items**

Group items are treated similarly to USAGE DISPLAY items. Note that any USAGE clause on a 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' will compare equal to an alphanumeric item with the value "A".

Figurative constants HIGH-VALUE, LOW-VALUE, SPACE or SPACES, ZERO or ZEROS, and QUOTE or QUOTES are treated logically as their EBCDIC character representations for assignments or comparisons with EBCDIC characters.

In comparisons between nonnumeric DISPLAY items, the collating sequence is the ordinal sequence of the characters based on their binary (hexadecimal) values (as modified by the alternate collating sequence for the single byte characters, if specified). The collating sequence for EBCDIC characters is not affected by the locale setting or the COLLSEQ compiler option.

The table below summarizes the conversion and the collating sequence applicable based on the types of data (ASCII, EBCDIC) and the COLLSEQ option in effect when PROGRAM COLLATING SEQUENCE is not specified. If it is specified, the source specification has precedence over the compiler option specification.

Comparands	COLLSEQ(BIN)	COLLSEQ(NATIVE)	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).

Comparands	COLLSEQ(BIN)	COLLSEQ(NATIVE)	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).

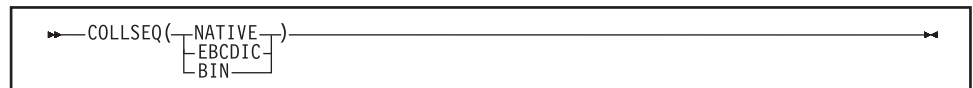
**Visual Builder:** Visual Builder applications require CHAR(NATIVE), which is the default specification in the GUI compile options notebook. Do not change this default setting.

**Object-oriented programs:** Do not specify CHAR(EBCDIC) in object-oriented programs.

#### RELATED REFERENCES

“Appendix B. System/390 host data type considerations” on page 479

## COLLSEQ



Default is: COLLSEQ(BIN)

Abbreviations are: None

Specify COLLSEQ(EBCDIC) to use the EBCDIC collating sequence rather than the ASCII collating sequence.

Specify COLLSEQ(BIN) to use the hex values of the characters; the locale setting has no effect. This setting will give better execution-time 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 will be ignored. If you specify PROGRAM COLLATING SEQUENCE is NATIVE, the value of NATIVE is taken from the COLLSEQ option.

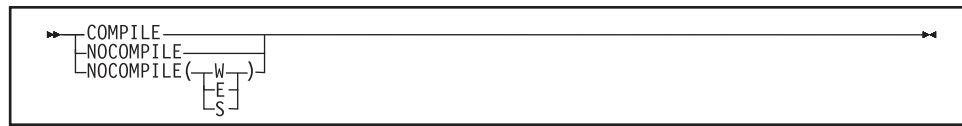
Otherwise, when the alphabet-name specified on the PROGRAM-COLLATING-SEQUENCE clause is defined with literals, the collating sequence used is that given by the COLLSEQ option, modified by the user-defined sequence given by alphabet-name.

The PROGRAM-COLLATING-SEQUENCE clause has no effect on DBCS data.

**Visual Builder:** Visual Builder applications require COLLSEQ(NATIVE), which is the default specification in the GUI compile options notebook. Do not change this default setting.

---

## COMPILE



Default is: NOCOMPILE(S)

Abbreviations are: C|NOC

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 generated if the compilation resulted in serious errors—the results could be unpredictable or an abnormal termination could occur.

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:

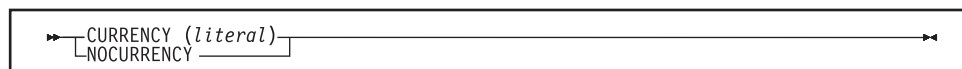
- LIST
- SSRANGE
- OPTIMIZE
- TEST

### RELATED REFERENCES

“Messages and listings for compiler-detected errors” on page 150

---

## CURRENCY



Default is: NOCURRENCY

The default currency symbol is the dollar sign (\$). You can use the CURRENCY option to provide an alternate default currency symbol to be used for the COBOL program.

NOCURRENCY specifies that no alternate default currency symbol will be used.

To change the default currency symbol, use the CURRENCY(*literal*) option where *literal* is a valid COBOL nonnumeric literal (including a hex literal) representing a single character that must not be any of the following:

- 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 selecting 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.

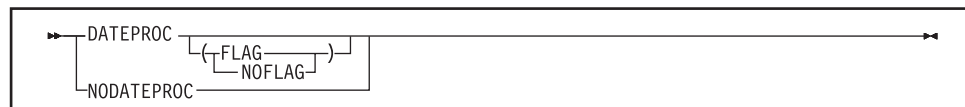
If you use both the CURRENCY option and the CURRENCY SIGN clause in a program, the CURRENCY option is ignored. Currency symbols specified in the CURRENCY SIGN clause or clauses can be used in PICTURE clauses.

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 note:** The CURRENCY option literal can be delimited by either the quote or the apostrophe, regardless of the QUOTE|APOST compiler option setting.

---

## DATEPROC



Default is: NODATEPROC, or DATEPROC(FLAG) if only DATEPROC is specified

Abbreviations are: DP|NODP

Use the DATEPROC option to enable the millennium language extensions of the COBOL compiler.

### 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 affects date-related program constructs as follows:



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.

CALLINT is used for the selection of 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 APIs.

**Alert:** This convention cannot be used in all cases when the called program has multiple entry points.

#### OPTLINK

The OPTLINK suboption specifies that the call convention is that of the \_OPTLINK convention of VisualAge for C++ for OS/2 and VisualAge for C++ for Windows.

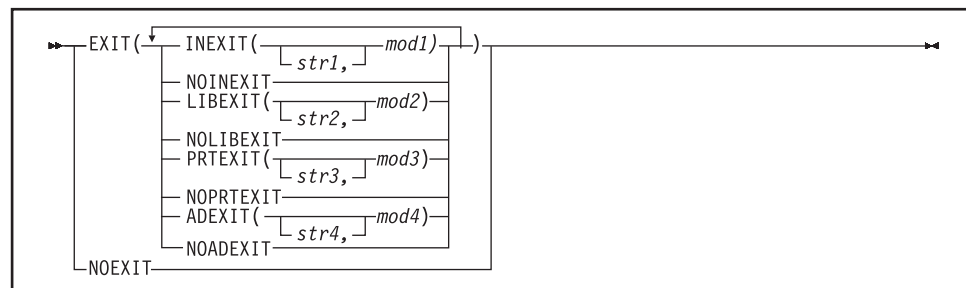
**CDECL** The CDECL suboption specifies that the call interface convention is that of the CDECL calling convention as defined by Microsoft Visual C/C++ for Windows.

**Visual Builder:** Visual Builder applications require ENTRYINT(SYSTEM), which is the default specification in the GUI compile options notebook. Do not change this default setting.

#### RELATED TASKS

“Coding multiple entry points” on page 381

## EXIT



Default is: EXIT(ADEXIT(IWZRMGUX))

Abbreviations are: EX(INX|NOINX,LIBX|NOLIBX,PRTX|NOPRTX,ADX|NOADX)

If you specify the EXIT option without providing at least one suboption, NOEXIT will be in effect. The suboptions can be specified 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 one time, the one specified last takes effect.

Use the EXIT option to allow the compiler to 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.

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; it can be specified only via the environment variable COBOPT, via the cob2 command option, or at installation time.

**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 copy code from a user-supplied load module (where *mod2* is the module name), instead of *library-name* or SYSLIB. For use with either COPY or BASIS statements.

**PRTEXIT(['str3'],mod3)**

The compiler passes printer-destined output to the user-supplied load module (where *mod3* is the module name), instead of SYSPRINT.

**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 apostrophes. Any character is allowed, but included apostrophes 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:

LL	string
----	--------

where LL is a halfword (on a halfword boundary) containing the length of the string. The table shows the location of the character string in the parameter list.

## 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 PRTEXIT 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.

### LIBEXIT

Offset	Contains address of	Description of item
00	User-exit type	Halfword identifying which user exit is to perform the operation.  1=INEXIT; 2=LIBEXIT; 3=PRTEXTIT; 4=ADEXIT
02	Operation code	Halfword indicating the type of operation.  0=OPEN; 1=CLOSE; 2=GET; 4=FINDD
04	Return code	Fullword, placed by the exit module, indicating status of the requested operation.  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	Fullword, placed by the exit module, containing the address of the record in a user-owned buffer, for the GET operation.
	or 'str2'	'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 containing the address of a null-terminated string containing the fully qualified text-name. Applies only to FIND.
36	User exit parameter string	Fullword containing 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 copy text 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 copy source.” 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 copy source. 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 copy source 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 based on the following:

- 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 COPY member can be named only once in a set of nested COPY statements until the end-of-data for that copy member is reached.

When the exit module receives the OPEN or FIND request, it should push its control information concerning the active copy source onto a stack and then complete the requested action (OPEN or FIND). The newly requested text-name (or basis-name) now becomes the active copy source.

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 copy source, 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 copy source.

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 copy source. 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

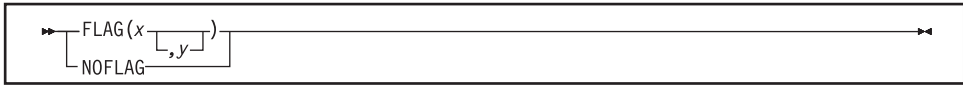
When ADEXIT is specified, the compiler loads the exit module (*mod4*) during initialization. The exit module is called for each record written to the SYSADATA data set.

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



Default is: FLAG(I)

Abbreviations are: F|NOF

*x* and *y* can be either I, W, E, S, or U.

Use FLAG(*x*) to produce diagnostic messages for errors of a severity level *x* or above at the end of the source listing.

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 embedding the statement number in an arrow that points to the message code. The message code is then 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. . . .
```

With FLAG(*x,y*) selected, messages of severity *y* and above will be embedded in the listing following the line that caused the message. (Refer to the notes below for exceptions.)

Use NOFLAG to suppress error flagging. NOFLAG will not suppress error messages for compiler options.

## Embedded messages:

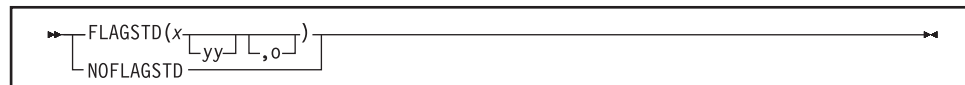
1. Specifying embedded level-U messages is accepted, but will not produce any messages in the source. Embedding a level-U message is not recommended.
2. The FLAG option does not affect diagnostic messages produced before the compiler options are processed.
3. Diagnostic messages produced during processing of compiler options, CBL and PROCESS statements, or BASIS, COPY, and REPLACE statements, are never embedded in the source listing. All such messages appear at the beginning of the compiler output.
4. Messages produced during processing of the \*CONTROL (\*CBL) statement are not embedded in the source listing.

## RELATED REFERENCES

"Messages and listings for compiler-detected errors" on page 150

---

## FLAGSTD



Default is: NOFLAGSTD

*x* specifies the level or subset of COBOL 85 Standard to be regarded as conforming:

- M** Language elements that are *not* from the minimum subset are to be flagged as “nonconforming standard.”
- I** Language elements that are *not* from the minimum or the intermediate subset are to be flagged as “nonconforming standard.”
- H** The high subset is being used and elements will not be flagged by subset. And, elements in the IBM extension category 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*).

*o* specifies that obsolete language elements are flagged as “obsolete.”

Use FLAGSTD to get informational messages about the COBOL 85 Standard elements 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 FLAGSTD is specified and are identified as “nonconforming nonstandard”)

This includes the new language syntax for object-oriented COBOL and for improved interoperability, the PGMNAME(MIXED) compiler option, and the Millennium Language Extensions.

The informational messages appear in the source program listing and contain the following information:

- Identify the element as “obsolete,” “nonconforming standard,” or “nonconforming nonstandard” (a language element that is both obsolete and nonconforming is flagged as obsolete only).
- Identify the clause, statement, or header that contains the element.

- Identify the source program line and beginning location of the clause, statement, or header that contains the element.
- Identify 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			<div>STANDARD</div> <div>NONSTANDARD</div> <div>OBSOLETE</div>
		3	<div>1</div> <div>1</div> <div>1</div>

# FLOAT



Default is: `FLOAT(NATIVE)`

Abbreviations are: None

Specify `FLOAT(NATIVE)` to use the native floating-point data representation format of the platform. For VisualAge COBOL, this is the IEEE format.

FLOAT(HEX) and FLOAT(S390) are synonymous and indicate that COMP-1 and COMP-2 data items are represented consistently with System/390 (that is, in the hex floating-point format):

- Hex floating-point values are converted to IEEE format prior to any arithmetic operations (computations or comparisons).
- IEEE floating-point values are converted to hex format prior to 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 hex floating point as necessary.

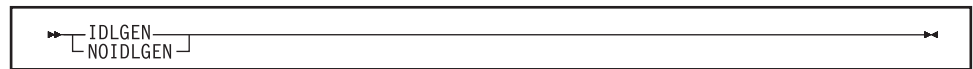
**Object-oriented programs:** Do not specify `FL0AT(S390)` in object-oriented programs.

## RELATED REFERENCES

“Appendix B. System/390 host data type considerations” on page 479

---

## IDLGEN



Default is: NOIDLGEN

Abbreviations are: IDL | NOIDL

Use the IDLGEN option to indicate whether SOM Interface Definition Language (IDL) should be generated for COBOL class definitions contained in the COBOL source file.

Use IDLGEN to request that in addition to the compile of the COBOL source file, IDL definitions for classes defined in the file be generated.

Use NOIDLGEN to request that no IDL definitions be generated.

The IDL file has the same name as the compiler source file, with the file extension IDL. For example, the IDL file generated for myfile.cbl is myfile.idl. The IDL file is written to the directory from which cob2 was run.

When a class definition includes references to other classes (such as on the INHERITS or METAClass IS phrases, or typed object references as method parameters) defined in separate source files, the generated IDL contains include statements for the IDL files of the referenced classes. The COBOL compiler attempts to obtain the file name (referred to as the *filestem* in the SOM documentation) for a referenced class from the SOM interface repository (IR). If the referenced class does not have an IR entry, the external class-name of the referenced class is assumed as the *filestem*. An include is then generated of the form: `#include <filestem..idl>`. This might be adequate for classes where external class-names are the same as the original source file name. However, in many cases this include statement needs to either be updated to reflect the correct *filestem*, or preferably the entire IDL file should be regenerated after the missing definition has been added to the IR.

When a COBOL source file contains more than one class definition (batch compile) and you use the IDLGEN option, the COBOL class definitions must be sequenced in an appropriate order within the source file. The generated IDL for such a batch compile contains multiple class interfaces with the IDL interfaces in the same order as the COBOL classes were defined in the COBOL source file. The SOM IDL compiler requires that interfaces be defined before they are referenced. So if there are references between the classes in the COBOL batch compile, the referenced classes must precede the referencing classes in the COBOL source file.

The mapping of COBOL to IDL is designed to balance two conflicting objectives, namely enablement of object-oriented COBOL type checking and enabling COBOL classes to operate with other SOM-based programming languages. At a high level:

- COBOL classes map to IDL interfaces.
- COBOL methods map to IDL operation declarations.
- Where possible, the data types of COBOL method parameters are mapped to corresponding native IDL types. These cases include binary integer, floating point, pointer, object reference, and character types.

All elementary USAGE DISPLAY types and fixed-length COBOL groups are mapped to IDL as array of character.

Remaining COBOL types that do not naturally map to any native IDL data type are mapped to COBOL-specific “foreign” IDL types. These cases include packed-decimal, scaled binary, DBCS, and variable-length groups.

- Method formal parameters that specify BY REFERENCE on the method PROCEDURE DIVISION header are given the IDL parameter attribute *inout* and parameters that specify BY VALUE are given the IDL parameter attribute *in*.

The IDL generated for the same class by the IBM COBOL compiler on Windows and AIX might differ; therefore you should regenerate the IDL for the target platform rather than port it between platforms. For example, on AIX, a PIC S9(8) BINARY data item maps naturally to an IDL long type, whereas on Windows, the same data item could map either to an IDL long or to a COBOL-specific data type that emulates System/390 binary format, depending on the compilation options used.

**Restriction:** You cannot specify the IDLGEN options on the PROCESS(CBL) statement.

#### RELATED TASKS

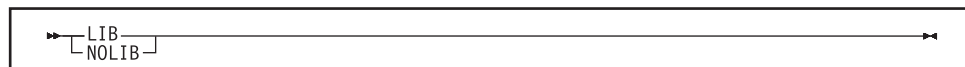
“Chapter 18. Writing object-oriented programs” on page 271

“Chapter 20. Using SOM IDL-based class libraries” on page 317

*SOMobjects Programmer’s Guide*

---

## LIB



Default is: LIB

Abbreviations are: None

If your program uses COPY, BASIS, or REPLACE statements, the LIB compiler option must be in effect.

In addition, for COPY and BASIS statements, you need 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/path for the copy file.
- If the library-name is omitted for a COPY statement, the path to be searched can be specified via the -Ixxx option on the cob2 command.
- If the library-name is specified with a literal, the literal value is treated as the actual path name.

**Visual Builder:** Visual Builder applications require LIB, which is the default specification in the GUI compile options notebook. Do not change this default setting.

LIB conforms to the COBOL 85 Standard.

#### RELATED REFERENCES

“Compiler-directing statements” on page 198

---

## LINECOUNT

»—LINECOUNT(*nnn*)—«

Default is: LINECOUNT(60)

Abbreviations are: LC

*nnn* must be an integer between 10 and 255, or 0.

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.

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

»LIST  
└─NOLIST┐«

Default is: NOLIST

Abbreviations are: None

Use the LIST compiler option to produce a listing of the assembler-language expansion of your source code.

You will also get these in your output listing:

- Global tables
- Literal pools
- Information about WORKING-STORAGE
- Size of the program's WORKING-STORAGE

If you want to limit the assembler listing output, use \*CONTROL LIST or NOLIST statements in your PROCEDURE DIVISION. Your source statements following a \*CONTROL NOLIST are not included in the listing until a \*CONTROL LIST statement switches the output back to normal LIST format.

**Batch compiles:** The number of and names of the resulting .asm files depend on the SEPOBJ option:

**SEPOBJ** 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.

**NOSEPOBJ**

The one file for all programs in the source file has the name of the source file.

#### RELATED TASKS

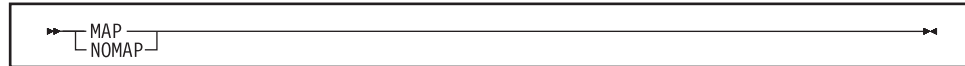
“Getting listings” on page 231

#### RELATED REFERENCES

\*CONTROL (\*CBL) statement (*IBM COBOL Language Reference*)

---

## MAP



Default is: NOMAP

Abbreviations are: None

Use the MAP compiler option to produce a listing of the items you defined in the DATA DIVISION. The output includes the following:

- DATA DIVISION map
- Global tables
- Literal pools
- Nested program structure map, and program attributes
- Size of the program’s WORKING-STORAGE

If you want to limit the MAP output, use \*CONTROL MAP or NOMAP statements in the DATA DIVISION. Source statements following a \*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, WORKING-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 235

#### RELATED CONCEPTS

“Chapter 14. Debugging” on page 221

#### RELATED TASKS

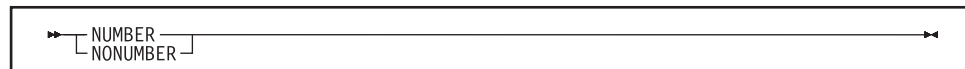
“Getting listings” on page 231

#### RELATED REFERENCES

\*CONTROL (\*CBL) statement (*IBM COBOL Language Reference*)

---

## NUMBER



Default is: NONUMBER

Abbreviations are: NUM | NONUM

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.

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.

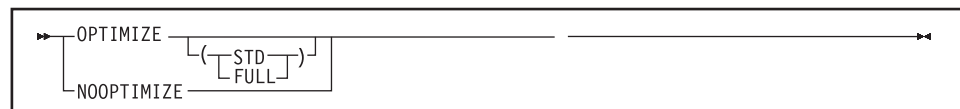
If you use COPY statements and NUMBER is in effect, be sure that your source program line numbers and the COPY member line numbers are coordinated.

Use NONUMBER if you do not have line numbers in your source code, or if you want the compiler to ignore the line numbers you do have in your source code. With NONUMBER in effect, the compiler generates line numbers for your source statements and uses those numbers as references in listings.

NONUMBER conforms to the COBOL 85 Standard.

---

## OPTIMIZE



Default is: NOOPTIMIZE

Abbreviations are: OPT | NOOPT

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, it should not be used when debugging.

If OPTIMIZE is specified without any suboptions, OPTIMIZE(STD) will be in effect.

The FULL suboption requests that, in addition to the optimizations performed under OPT(STD), the compiler discard unreferenced data items from the DATA

DIVISION and suppress generation of code to initialize these data items to their VALUE clauses. When OPT(FULL) is in effect, all unreferenced 77-level items and elementary 01-level items will be discarded. In addition, 01-level group items will be discarded, if none of the subordinate items are referenced. The deleted items are shown in the listing. If the MAP option is in effect, a BL number of XXXX 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 will be eliminated.

**However:**

Do not use OPT(FULL) if your programs depend on making use of unused data items. Two common ways this has been done in the past are:

1. A technique sometimes used in OS/VS COBOL programs is 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 have this problem, use the SSRANGE compiler option with the CHECK(ON) run-time option. To work around this problem, use the ability of COBOL to code large tables and use just one table.
2. The second technique utilizing unused data items is to place eyecatcher 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 a 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 as used, and will not delete them.

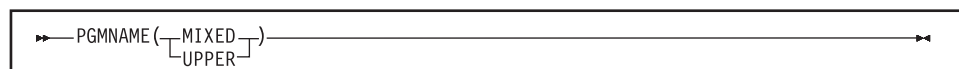
The OPTIMIZE option is turned off in the case of a severe-level error or higher. The OPTIMIZE and TEST options are mutually exclusive; if you use both, OPTIMIZE will be ignored.

RELATED CONCEPTS

“Optimization” on page 458

---

## PGMNAME



Default is: PGMNAME(UPPER), or  
PGMNAME(MIXED) for Visual Builder GUI applications

Abbreviations are: PGMN(LU | LM)

For compatibility with IBM 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.

The PGMNAME option controls the handling of names used in the following contexts:

- Program names defined in the PROGRAM-ID paragraph.
- Program entry point names on the ENTRY statement.
- Program name references in:
  - CALL statements
  - CANCEL statements
  - SET *procedure-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 or method 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)

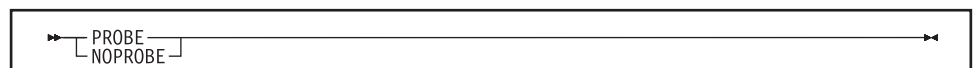
With PGMNAME(MIXED), program names are processed as is, without truncation, translation, or folding to uppercase.

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.

**Visual Builder:** Visual Builder applications require PGMNAME(MIXED), which is the default specification in the GUI compile options notebook. Do not change this default setting.

---

## PROBE



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.

Use PROBE if the program might be executed in a multithreading environment.

NOPROBE produces more efficient code and is appropriate for nonthreading environments.

If you compile a program with NOPROBE, you must increase the size of the stack by using the /STACK linker option. The stack must be large enough for the program to receive exceptions and for the Distributed Debugger to work properly.

#### RELATED TASKS

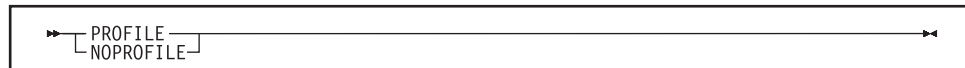
“Chapter 26. Preparing COBOL programs for multithreading” on page 403

#### RELATED REFERENCES

“/STACK” on page 214

---

## PROFILE



Default is: PROFILE

Abbreviations are: None

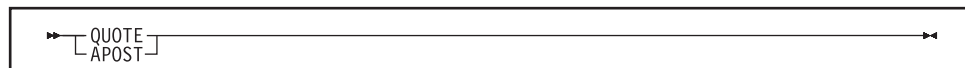
PROFILE instructs the compiler to generate the profile hooks that allow the Performance Analyzer to monitor application execution and generate a trace file. Use this option with the -p option of the cob2 command.

#### RELATED TASKS

“Compiling from the command line” on page 144

---

## QUOTE/APOST



Default is: QUOTE

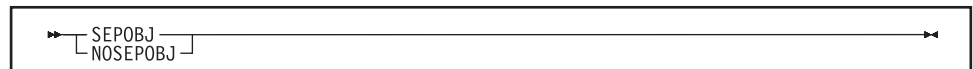
Abbreviations are: Q | APOST

Use QUOTE if you want the figurative constant [ALL] QUOTE or [ALL] QUOTES to represent one or more quotation mark (") characters. QUOTE conforms to the COBOL 85 Standard.

Use APOST if you want the figurative constant [ALL] QUOTE or [ALL] QUOTES to represent one or more apostrophe (') characters.

**Delimiters:** Either quotes or apostrophes can be used 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



Default is: SEPOBJ, or NOSEPOBJ for Visual Builder GUI applications

Abbreviations are: None

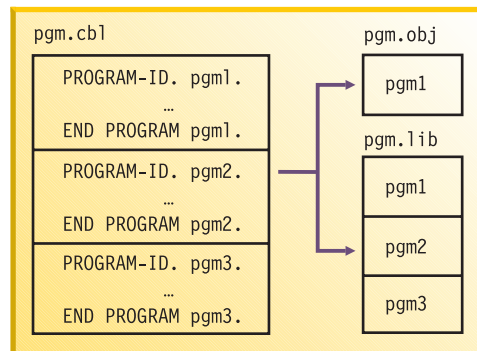
The option specifies whether or not each of the outermost COBOL programs in a batch compilation is to be generated as a separate object file rather than a single object file.

### Batch compilation

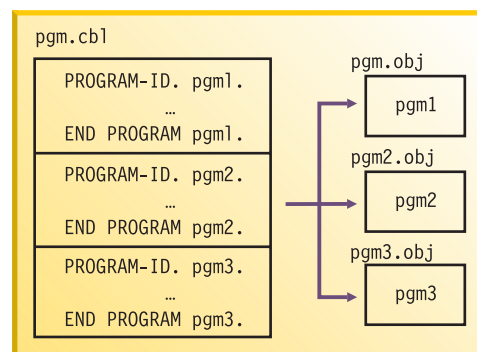
When multiple outermost programs (nonnested programs) are compiled with a single batch invocation of the compiler, the number of separate files produced for the object program output of the batch compilation depends on the compiler option SEPOBJ.

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 (with NOSEPOBJ) or three (with SEPOBJ) files.

#### Batch compilation with NOSEPOBJ



#### Batch compilation with SEPOBJ



#### Considerations:

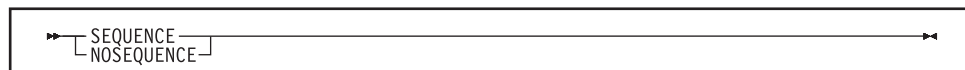
1. The SEPOBJ option is required to conform to the ANSI COBOL standard where pgm2 or pgm3 in the above example is called via CALL *identifier* from another program.
2. If the NOSEPOBJ option is in effect, the object module files are named with the name of the source file with extension o, OBJ, or LIB. If the SEPOBJ option is in effect, the names of the object files (except for the first one) are based on the PROGRAM-ID name with extension o or OBJ.
3. The programs called via 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 are responsible for giving the object file a valid file name for the platform and the file system. For example, if the FAT file system is used for Windows, the length of the PROGRAM-ID name must be eight characters or fewer *except* when the object file names are created from the source file name (as in the case with NOSEPOBJ option) as described above.

**Visual Builder:** Visual Builder applications require NOSEPOBJ, which is the default specification in the GUI compile options notebook. Do not change this default setting.

---

## SEQUENCE



Default is: SEQUENCE

Abbreviations are: SEQ | NOSEQ

When you use SEQUENCE, the compiler examines columns 1 through 6 of your source statements to check that the 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 sequence (source statements with blanks in columns 1 through 6 do not participate in this sequence check and do not result in messages).

If you use COPY statements and SEQUENCE is in effect, be sure that your source program sequence fields and the copy member sequence fields are coordinated.

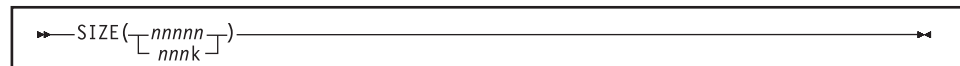
If you use NUMBER and SEQUENCE, the sequence is checked according to numeric, rather than EBCDIC, collating sequence.

Use NOSEQUENCE to suppress this checking and the diagnostic messages.

NOSEQUENCE conforms to the COBOL 85 Standard.

---

## SIZE



Default is: 2097152 bytes (approximately 2 M)

Abbreviations are: SZ

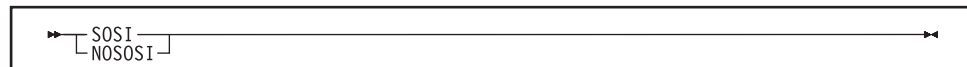
*nnnnn* specifies a decimal number that must be at least 778240.

*nnnK* specifies a decimal number in 1K increments. The minimum acceptable value is 782K.

Use SIZE to indicate the amount of main storage available for compilation (where 1K = 1024 bytes decimal).

---

## SOSI



Default is: NOSOSI

Abbreviations are: None

**NOSOSI** With NOSOSI, character positions with X'1E' and X'1F' values are treated as data characters, as in previous releases of VisualAge COBOL.

NOSOSI conforms to the COBOL 85 Standard.

**SOSI** With SOSI, workstation SO/SI control characters delimit ASCII DBCS character strings in COBOL source programs. The workstation SO and SI characters have the encoded values of X'1E' and X'1F' respectively.

Workstation SO/SI characters have no effect on workstation COBOL source code, except to act as place holders for host DBCS SO/SI characters to ensure proper data handling when converting remote files from EBCDIC to ASCII.

When the SOSI option is in effect, in addition to existing rules for workstation COBOL, the following rules apply:

- All double-byte character strings (in user-defined words, DBCS literals, nonnumeric literals and in comments) must be delimited by the workstation SO/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

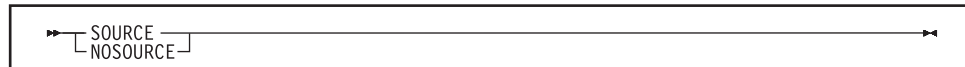
double-byte representation. For example, the hyphen (-) cannot appear as the first or the last character in a user-defined word.

- For nonnumeric literals that contain X'1E' or X'1F' values, the following apply:
  - Character positions with X'1E' and X'1F' are treated as workstation SO/SI characters and are not included as part of the character string value of the literal when included in an N-literal or a G-literal. They are treated as part of the literal data when included in a nonnumeric literal, which is not part of a G, N, or X literal.
  - The workstation SO/SI characters must be paired (with each pair starting with the workstation SO character) with zero or an even number of intervening bytes.
  - The workstation SO/SI character pairs cannot be nested.
- If you embedded DBCS quotes within an N-literal delimited by quotes, use two consecutive DBCS quotes to represent a single DBCS quote. Do not include a single DBCS quote in an N-literal if the literal is delimited by quotes. The same rule applies to apostrophes.
- The SHIFT-OUT and SHIFT-IN special registers are defined with X'0E' and X'0F' regardless of the SOSI option 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 workstation.

---

## SOURCE



Default is: SOURCE

Abbreviations are: S | NOS

Use SOURCE to get a listing of your source program. This listing will include any statements embedded by PROCESS or COPY statements.

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. Your source statements following a \*CONTROL NOSOURCE are not included in the listing at all, unless a \*CONTROL SOURCE statement switches the output back to normal SOURCE format.

“Example: MAP output” on page 235

### RELATED REFERENCES

\*CONTROL (\*CBL) statement (*IBM COBOL Language Reference*)

---

## SPACE

```
»—SPACE(1)—«
      [2]
      [3]
```

Default is: SPACE(1)

Abbreviations are: None

Use SPACE to select single-, double-, or triple-spacing in your source code listing.

SPACE has meaning only when the SOURCE compiler option is in effect.

### RELATED REFERENCES

“SOURCE” on page 188

---

## SQL

```
»—SQL(,suboptions for DB2 SQL,)—«
```

Default is: SQL(“”)

Abbreviations are: None

Use this option when you have SQL statements embedded in your COBOL source. It allows you to specify options to be used in handling the SQL statements in your program and is required if the suboption string, which gives SQL options, is to be specified explicitly to DB2.

The syntax shown can be used on either the CBL or PROCESS statements. If the SQL option is given on the cob2 command, only ' is allowed for the string delimiter: -q"SQL('options')".

### RELATED TASKS

“Chapter 15. Programming for a DB2 environment” on page 245

---

## SSRANGE

```
»—[SSRANGE]
   [NOSSRANGE]—«
```

Default is: NOSSRANGE

Abbreviations are: SSR|NOSSR

Use SSRANGE to generate code that checks if 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 will also be checked to ensure that the reference is within their maximum defined length.

Reference modification expressions will be checked to ensure that:

- The reference modification starting position is greater than or equal to 1.
- The reference modification starting position is not greater than the current length of the subject data item.
- The reference modification length value (if specified) is greater than or equal to 1.
- The reference modification 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, the range-checking code is generated. You can inhibit range checking by specifying CHECK(OFF) as a run-time option. This leaves range-checking code dormant in the object code. The range-checking code can then be optionally used to aid in resolving any unexpected errors without recompilation.

If an out-of-range condition is detected, an error message is displayed and the program is terminated.

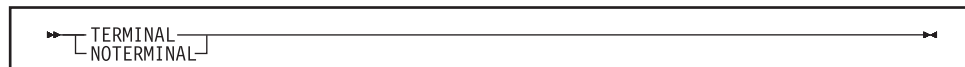
**Remember:** You will get range checking only if you compile your program with the SSRANGE option and run it with the CHECK(ON) run-time option.

#### RELATED CONCEPTS

“Reference modifiers” on page 85

---

## TERMINAL



Default is: NOTERMINAL

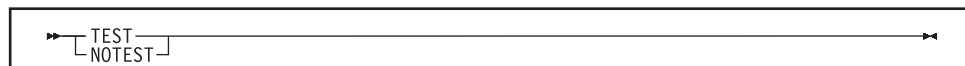
Abbreviations are: TERM|NOTERM

Use TERMINAL to send progress and diagnostic messages to the terminal.

Use NOTERMINAL if this additional output is not desired.

---

## TEST



Default is: NOTEST

Abbreviations are: None

Use TEST to produce object code that contains symbol and statement information that enables the debugger to perform symbolic source-level debugging.

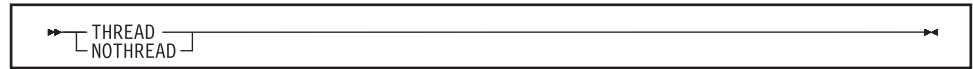
Use NOTEST if you do not want to generate object code with debugging information.

Programs compiled with NOTEST execute with the debugger, but there is limited debugging support.

The TEST option will be turned off if you use the WITH DEBUGGING MODE clause. The TEST option will appear in the list of options, but a diagnostic message will be issued to advise you that because of the conflict, TEST will not be in effect.

---

## THREAD



Default is: NOTHREAD

Abbreviations are: None

THREAD indicates that the COBOL program is to be enabled for execution in a Language Environment enclave with multiple POSIX threads or PL/I tasks. A program compiled with the THREAD option can also be used in a nonthreaded application, but all COBOL programs within a Language Environment enclave must be compiled with the same setting: either the THREAD or the NOTHREAD option.

When the THREAD option is in effect, the following language elements are not supported. If encountered, they are diagnosed as errors:

- STOP RUN
- ALTER statement
- DEBUG-ITEM special register
- GO TO statement without procedure-name
- RERUN
- STOP literal statement
- Segmentation module
- USE FOR DEBUGGING statement
- WITH DEBUGGING MODE clause
- INITIAL phrase in PROGRAM-ID clause

RERUN is flagged as an error with THREAD, but is accepted as a comment with NOTHREAD.

**Visual Builder:** Visual Builder applications require NOTHREAD, which is the default specification in the GUI compile options notebook. Do not change this default setting.

**CICS TXSeries:** The THREAD option is required.

### RELATED TASKS

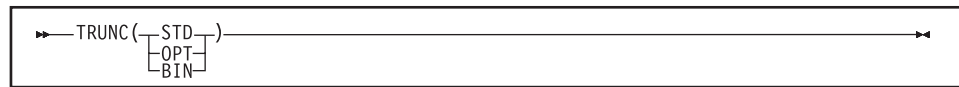
“Compiling and running CICS programs” on page 253

“Compiling from the command line” on page 144

“Chapter 26. Preparing COBOL programs for multithreading” on page 403

---

# TRUNC



Default is: TRUNC(STD)

Abbreviations are: None

TRUNC(STD) conforms to the COBOL 85 Standard, whereas TRUNC(OPT) and TRUNC(BIN) are IBM extensions.

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)

Use TRUNC(STD) to control the way arithmetic fields are truncated during MOVE and arithmetic operations. 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 will be 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.

**Recommendation:** 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 avoid the performance overhead of using TRUNC(BIN) as your installation default by using COMP-5 for individual binary data items passed to non-COBOL programs or other products and subsystems. The use of COMP-5 is not affected by the TRUNC suboption in effect.

## TRUNC example 1

```
01 BIN-VAR      PIC 99 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	3B1E2101100	123451
Receiver TRUNC(STD)	51	33100	51
Receiver TRUNC(OPT)	-7621	3B1E2	2J
Receiver TRUNC(BIN)	-7621	3B1E2	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 the 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	D3102196149	1234567891

Data item	Decimal	Hex <sup>1</sup>	Display
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.

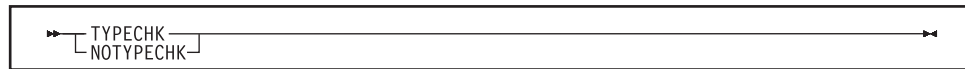
#### RELATED CONCEPTS

“Formats for numeric data” on page 34

#### RELATED TASKS

“Compiling and running CICS programs” on page 253

## TYPECHK



Default is: NOTYPECHK

Abbreviations are: TC|NOTC

Use TYPECHK to have the compiler enforce the rules for object-oriented type checking, and generate diagnostics for any violations.

Use NOTYPECHK to turn off the checking for typing violations.

The type conformance requirements are covered in the *IBM COBOL Language Reference* under the appropriate language elements. Type-checking requirements include:

- The method being invoked on an INVOKE statement must be supported by the class of the referenced object.
- Method parameters on an INVOKE and the corresponding method PROCEDURE DIVISION USING must conform.
- The SET *object-reference-1* TO *object-reference-2* statement requires that the classes of the objects be of appropriate derivation relationships.
- A method override must have a conforming interface to the corresponding method in the parent class.

When you specify TYPECHK, there must be entries in the SOM Interface Repository (IR) for each class that is referenced in the COBOL source being compiled.

For COBOL classes, you can create the IR entries by using the COBOL IDLGEN option when compiling the class definitions. Doing so creates an IDL file that describes the interface of the COBOL class. Compile the IDL using the SOM Compiler with its “ir” emitter.

Note that if the COBOL program references classes that are provided by the SOM product itself (such as the SOMObject class), then you can use the pregenerated IR for these classes (provided as part of the OS/390 SOMObjects product) to verify that the COBOL usage conforms to the class interfaces.

#### RELATED CONCEPTS

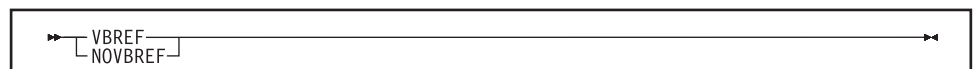
“Chapter 19. System Object Model” on page 311

#### RELATED REFERENCES

“IDLGEN” on page 177

---

## VBREF



Default is: NOVBREF

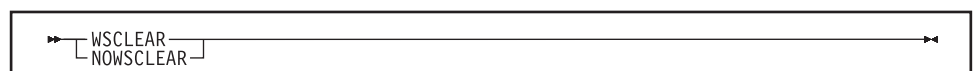
Abbreviations are: None

Use VBREF to get a cross-reference among all verb types 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.

Use NOVBREF for more efficient compilation.

---

## WSCLEAR



Default is: NOWSCLEAR

Abbreviations are: None

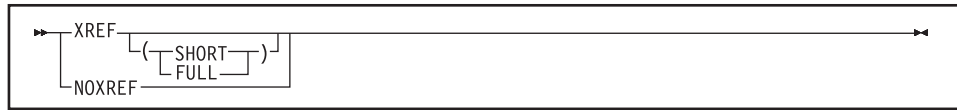
Use WSCLEAR to clear your program’s WORKING-STORAGE to binary zeros when the program is initialized. The storage is cleared before any VALUE clauses are applied.

Use NOWSCLEAR to bypass the storage clearing process.

If you use WSCLEAR and you are concerned about the size or performance of the object program, then you should also use OPTIMIZE(FULL). This instructs the compiler to eliminate all unreferenced data items from the DATA DIVISION, which will speed up the initialization process.

---

## XREF



Default is: NOXREF

Abbreviations are: X | NOX

You can choose XREF, XREF(FULL), or XREF(SHORT).

Use the XREF compiler option to get a sorted cross-reference listing. Names are listed in the order of the collating sequence indicated by the locale setting. This applies whether the names are in single-byte characters or contain multibyte characters (such as DBCS).

Also included is a section listing all the program names that are referenced in your program, and the line number where they are defined. External program names are identified as such.

If you use XREF and SOURCE, cross-reference information will also be printed on the same line as the original source in the listing. Line number references or other information, will appear on the right-hand side of the listing page. On the right of source lines that reference intrinsic functions, the letters 'IFN' appear with the line numbers of the location where the function's arguments are defined. Information included in the embedded references lets you know if an identifier is undefined or defined more than once (UND or DUP will be printed); if an item is implicitly defined (IMP), such as special registers or figurative constants; and if a program name is external (EXT).

If you use XREF and NOSOURCE, you'll get only the sorted cross-reference listing.

XREF(SHORT) will print only the explicitly referenced variables in the cross-reference listing. XREF(SHORT) applies to MBCS data names and procedure-names as well as ASCII names.

NOXREF suppresses this listing.

### Observe:

1. Group names used in a MOVE CORRESPONDING statement are in the XREF listing. In addition, the elementary names in those groups are also listed.
2. In the data name XREF listing, line numbers preceded by the letter "M" indicate that the data item is explicitly modified by a statement on that line.
3. XREF listings take additional storage.

### RELATED CONCEPTS

"Chapter 14. Debugging" on page 221

### RELATED TASKS

"Getting listings" on page 231

---

## YEARWINDOW

»— YEARWINDOW —(*base-year*)—«

Default is: YEARWINDOW(1900)

Abbreviation is: YW

Use the YEARWINDOW option 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.

*base-year* represents the first year of the 100-year window, and must be specified as one of the following:

- 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, where the first year of the window is calculated from the current run-time date. The number is subtracted from the current year to give the starting year of the century window. For example, YEARWINDOW(-80) indicates that the first year of the century window is 80 years before the current year at the time the program is run.

### Notes:

1. The YEARWINDOW option has no effect unless the DATEPROC option is also in effect.
2. At run time, two conditions must be true:
  - The century window must have its beginning year in the 1900s.
  - The current year must lie within the century window for the compilation unit.

For example, if the current year is 2001, the DATEPROC option is in effect, and you use the YEARWINDOW(1900) option, the program will terminate with an error message.

---

## ZWB

»— ZWB —  
NOZWB—«

Default is: ZWB

Abbreviations are: None

With ZWB, the compiler removes the sign from a signed external decimal (DISPLAY) field when comparing this field to an alphanumeric elementary field during execution.

If the external decimal item is a scaled item (contains the symbol P in its PICTURE character string), its use in comparisons is not affected by ZWB. Such items always have their sign removed before the comparison is made to the alphanumeric field.

ZWB affects how the program runs; the same COBOL source program can give different results, depending on the option setting.

ZWB conforms to the COBOL 85 Standard.

Use NOZWB if you want to test input numeric fields for SPACES.

---

## Compiler-directing statements

Several statements help you to direct the compilation of your program.

### **BASIS statement**

This extended source program library statement provides a complete COBOL program as the source for a compilation. For processing rules, see the description under text-name of the COPY statement.

### **\*CONTROL (\*CBL) statement**

This compiler-directing statement selectively suppresses or allows output to be produced. The names \*CONTROL and \*CBL are synonymous.

### **>>CALLINTERFACE statement**

This compiler-directing statement specifies the interface convention for calls, including whether argument descriptors are to be generated. The convention specified using >>CALLINT is in effect until another >>CALLINT specification is made. >>CALLINT can be used only in the PROCEDURE DIVISION.

The syntax and usage of the >>CALLINT statement is similar to the CALLINT compiler option. Exceptions are:

- CALLINT is a valid abbreviation in the statement syntax.
- The statement syntax does not include parentheses.
- The statement form can be used to apply to selective calls as described below.
- The statement 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 an IBM C program whose default call interface convention is \_OPTLINK, or it is a COBOL program compiled with the ENTRYINT(OPTLINK) option, use the >>CALLINT directive to change the interface for this call only:

```
>>CALLINT OPTLINK DESC
CALL "PROG1" USING PARM1 PARM2.
>>CALLINT
CALL "PROG2" USING PARM1.
```

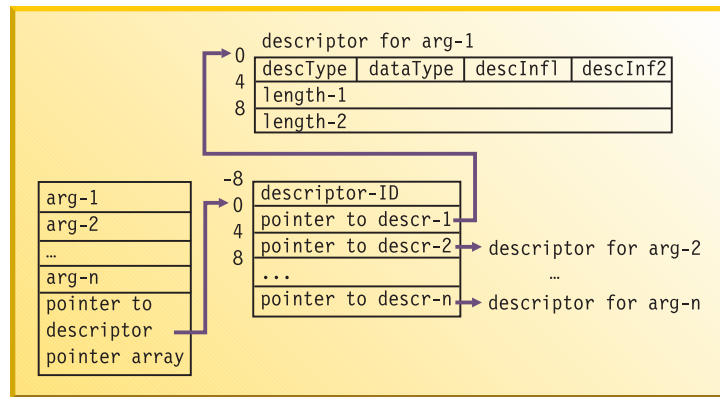
The >>CALLINT statement can be specified anywhere that a COBOL procedure statement can be specified. For example, the following is valid COBOL syntax:

```
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, not 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'

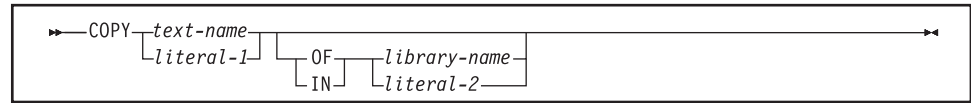
### length-1

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 library statement places prewritten text into a COBOL program. The uniqueness of text-name and library-name is determined after the formation and conversion rules for a system-dependent name have been applied. A user-defined word can be the same as a text-name or a library-name. If more than one COBOL library is available during compilation, text-name need not be qualified. 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 (*literal-2*), it is treated as the actual path. If you specify library-name with 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 COPY text. To specify multiple path names, delimit them by using a semicolon (;).

If you do not specify library-name, the path used is as described under text-name.

#### text-name

The processing of text-name as a user-defined word depends on whether the environment variable corresponding to the 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:

- library-name (or literal-2) is not given, *and*
- text-name is a literal (literal-1) or an environment variable, *and*
- The first character is '\' or the second character is ':

For example,

```
COPY "\mycpylib\ . ." or COPY "d:\mycpylib\ . ."
```

If the environment variable corresponding to the text-name is *not* set, the copy text is searched for as the following names:

1. The text-name with the extension of cpy
2. The text-name with the extension of cbl
3. The text-name with the extension of cob
4. The text-name without an 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 path searched 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, while COPY "X\Y" will be searched in the current directory.

COPY A OF SYSLIB is equivalent to COPY A. The -I option does not impact 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 will insert 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)

### **DELETE statement**

This extended source library statement removes COBOL statements from the BASIS source program.

### **EJECT statement**

This compiler-directing statement specifies that the next source statement is to be printed at the top of the next page.

### **ENTER statement**

The compiler handles this statement as a comment.

### **INSERT statement**

This library statement adds COBOL statements to the BASIS source program.

### **PROCESS (CBL) statement**

This statement, which is placed before the IDENTIFICATION DIVISION header of an outermost program, indicates which compiler options are to be used during compilation of the program.

### **REPLACE statement**

This statement is used to replace source program text.

### **SKIP1/2/3 statement**

These statements indicate lines to be skipped in the source listing.

### **TITLE statement**

This statement specifies that a title (header) be printed at the top of each page of the source listing.

### **USE statement**

The USE statement provides *declaratives* to specify the following:

- Error-handling procedures—EXCEPTION/ERROR

- Debugging lines and sections—DEBUGGING

#### RELATED TASKS

“Changing the header of a source listing” on page 6

“Compiling from the command line” on page 144

#### RELATED REFERENCES

“Specifying compiler options with the PROCESS (CBL) statement” on page 148

“cob2 options” on page 145

CONTROL (CBL) statement (*IBM COBOL Language Reference*)

CALLINTERFACE statement (*IBM COBOL Language Reference*)

COPY statement (*IBM COBOL Language Reference*)

---

## Chapter 12. Linker options

To control the linking process and the files that it produces, use the linker options. The linked-to information for each option provides the syntax for specifying the option and accepted abbreviations. It also describes the option, its parameters, and its interaction with other parameters.

Option	Description	Default	Abbreviation
/?	Display help	None	None
/ALIGNADDR	Set address alignment	/A:0x00010000	/ALIGN
/ALIGNFILE	Set file alignment	/A:512	/A
/BASE	Set preferred loading address	/BAS:0x00400000	/BAS
/CODE	Set section attributes for executable	/CODE:RX	None
/DATA	Set section attributes for data	/DATA:RW	None
/DBGPACK, /NODBGPACK	Pack debugging information	/NODB	/DB   /NODB
/DEBUG, /NODEBUG	Include debugging information	/NODEB	/D   /NODEB
/DEFAULTLIBRARYSEARCH, /NODEFAULTLIBRARYSEARCH	Search default libraries	/DEF	/DEF   /NOD
/DLL	Generate DLL	/EXEC	/EXEC
/DLL	Specify an entry point in an executable file	None	/EXEC
/EXECUTABLE	Generate .EXE file	/EXEC	/EXEC
/EXTDICTIONARY, /NOEXTDICTIONARY	Use extended dictionary to search libraries	/EXT	/EXT   /NOE
/FIXED, /NOFIXED	Do not relocate the file in memory	/NOFI	/FI   /NOFI
/FORCE	Create executable output file even if errors are detected	/NOFO	/FO   /NOFO
/HEAP	Set the size of the program heap	/HEAP: 0x100000,0x1000	/HEA
/HELP	Display help	None	/H
/INCLUDE	Forces a reference to a symbol	None	/INC
/INFORMATION, /NOINFORMATION	Display status of linking process	/NOIN	/I   /NOIN
/LINENUMBERS, /NOLINENUMBERS	Include line numbers in map file	/NOLI	/L   /NOLI
/LOGO, /NOLOGO	Display logo, echo response file	/LO	/LO   /NOL
/MAP, /NOMAP	Generate map file	/NOM	/M   /NOM
/OUT	Name output file	Name of first .obj file	/O
/PMTYPE	Specify application type	/PMTYPE:VIO	/PM
/SECTION	Set attributes for section	Set by /CODE and /DATA	/SEC
/SEGMENTS	Set maximum number of segments	/SE:256	/SE

Option	Description	Default	Abbreviation
/STACK	Set stack size of application	/STACK: 0x100000,0x1000	/ST
/STUB	Specify the name of the DOS stub file	None	/STU
/SUBSYSTEM	Specify the required subsystem and version	/SUBSYSTEM: WINDOWS,4.0	/SU
/VERBOSE	Display status of linking process	/NOV	/VERB /NOV
/VERSION	Write a version number in the run file	/VERSION:0.0	/VER

#### RELATED TASKS

“Linking programs” on page 152

---

## /?

» /? «

Use /? to display a list of valid linker options. This option is equivalent to /HELP.

#### RELATED REFERENCES

“/HELP” on page 210

---

## /ALIGNADDR

» /ALIGNADDR:*factor* «

Default is: /ALIGNADDR:0x00010000

Abbreviation is: /ALIGN

Use /ALIGNADDR to set the address alignment for segments.

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 256M.

---

## /ALIGNFILE

» /ALIGNFILE:*factor* «

Default is: /ALIGNFILE:512

Abbreviation is: /A

Use /ALIGNFILE to set the file alignment for segments.

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 64K.

---

## /BASE

» /BASE: address  
          └─@filename, key┘

Default is: /BASE:0x00400000

Abbreviations are: /BAS

Use /BASE to specify the preferred load address for the first load segment of a .DLL file.

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 beginning with the specified key. Each line in the memory-map file has the syntax:

*key address maxsize*

Separate the elements with one or more spaces or tabs. The key is a unique name in the file. The address is the location of the memory image in the virtual address space. The maxsize is an amount of memory within which the image must fit. The linker will issue a warning when the memory image of the program exceeds the specified size. A comment in the memory-map file begins with a semicolon (;) and runs to the end of the line.

---

## /CODE

» /CODE: attribute

Default is: /CODE:RX

Abbreviations are: None

Use /CODE to specify the default attributes for all code sections. Letters can be specified in any order.

Letter	Attribute
E or X	EXECUTE
R	READ
S	SHARED
W	WRITE

---

## /DATA



Default is: /DATA:RW

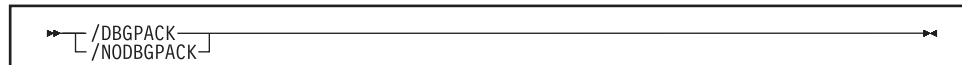
Abbreviations are: None

Use /DATA to specify the default attributes for all data sections. Letters can be specified in any order.

Letter	Attribute
E or X	EXECUTE
R	READ
S	SHARED
W	WRITE

---

## /DBGPACK, /NODBGPACK



Default is: /NODBGPACK

Abbreviations are: /DB | /NODB


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.

**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 actually speed up your linking, because there is less information to write to file.

When you specify /DBGPACK, /DEBUG is turned on by default.

---

## /DEBUG, /NODEBUG



Default is: /NODEBUG

Abbreviations are: /D | /NODEB

Use /DEBUG to include debug information in the output file, so you can debug the file with the debugger, or analyze its performance with Performance Analyzer. The linker embeds symbolic data and line number information in the output file.

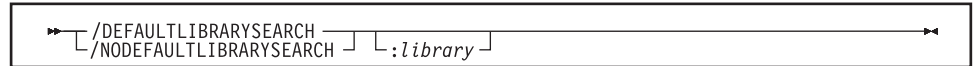
For debugging, specify the cob2 option -g.

For the Performance Analyzer, compile the object files with the cob2 option -p.

Linking with /DEBUG increases the size of the executable output file.

---

## /DEFAULTLIBRARYSEARCH, /NODEFAULTLIBRARYSEARCH



Default is: /DEFAULTLIBRARYSEARCH

Abbreviations are: /DEF | /NOD

Use /DEFAULTLIBRARYSEARCH to have the linker search the default libraries of object files when resolving references.

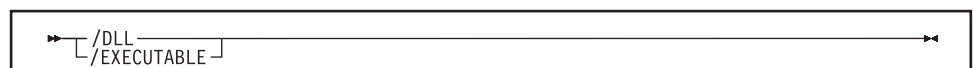
If you specify a *library* with the option, the linker adds the library name to the list of default libraries. The default libraries for an object file are defined at compile time, and embedded in the object file. The linker searches the default libraries by default.

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).

If you specify /NODEFAULTLIBRARYSEARCH without specifying *library*, then you must explicitly specify all the libraries you want to use, including IBM VisualAge COBOL run-time libraries.

---

## /DLL



Default is: /EXECUTABLE

Abbreviation is: /EXEC

Use /DLL to identify the output file as a dynamic link library (.DLL file). Compile the object files with the cob2 option -d11.

If you specify /DLL with /EXEC, only the last specified of the options takes effect.

If you do not specify /DLL, or any of the other options above, by default the linker produces an .EXE file (/EXEC).

---

## /ENTRY

» — /ENTRY:*name* — «

Default is: None

Abbreviation is: /EN

Use /ENTRY to specify an entry point (name of a routine or function) in an executable.

---

## /EXECUTABLE

» [ /DLL  
/EXECUTABLE ] «

Default is: /EXECUTABLE

Abbreviation is: /EXEC

Use /EXEC to identify the output file as an executable program (.EXE file). The linker generates .EXE files by default.

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

» [ /EXTDICTIONARY  
/NOEXTDICTIONARY ] «

Default is: /EXTDICTIONARY

Abbreviations are: /EXT | /NOE

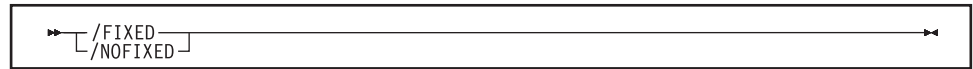
Use /EXTDICTIONARY to have the linker search the extended dictionaries of libraries when it resolves external references. The extended dictionary is a list of module relationships within a library. When the linker pulls in a module from the library, it checks the extended dictionary to see if that module requires other modules in the library, and then pulls in the additional modules automatically.

The linker searches the extended dictionary by default, to speed up the linking process.

Use /NOEXTDICTIONARY if you are defining a symbol in your object code that is also defined in one of the libraries you are linking to. Otherwise the linker issues an error because you have defined the same symbol in two different places. 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



Default is: /NOFIXED

Abbreviations are: /FI | /NOFI

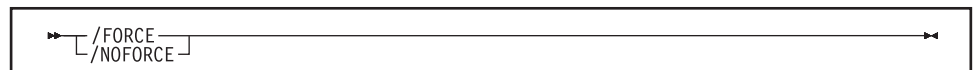
Use /FIXED to tell the loader not to relocate a file in memory when the specified base address is not available.

### RELATED REFERENCES

“/BASE” on page 205

---

## /FORCE



Default is: /NOFORCE

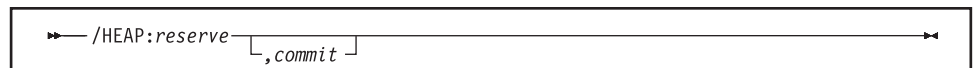
Abbreviations are: /FO | /NOFO

Use /FORCE to produce an executable output file even if there are errors during the linking process.

By default, the linker does not produce an executable output file if it encounters an error.

---

## /HEAP



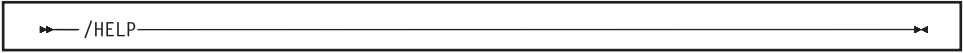
Default is: /HEAP:0x100000,0x1000

Abbreviation is: /HEA

Use /HEAP to set the size of the program heap in bytes. The *reserve* argument sets the total virtual address space reserved. The *commit* sets the amount of physical memory to allocate initially. When *commit* is less than *reserve*, memory demands are reduced, but execution time can be slower.

---

## /HELP



» /HELP «

Default is: None

Abbreviation is: /H

Use /HELP to display a list of valid linker options. This option is equivalent to /?.

---

## /INCLUDE



» /INCLUDE:symbol «

Default is: None

Abbreviation is: /INC

Use /INCLUDE to force a reference to a symbol. The linker searches for an object module that defines the symbol.

---

## /INFORMATION, /NOINFORMATION



» /INFORMATION  
/NOINFORMATION «

Default is: /NOINFORMATION

Abbreviations are: /I | /NOIN

### RELATED REFERENCES

“/VERBOSE” on page 215

---

## /LINENUMBERS, /NOLINENUMBERS



» /LINENUMBERS  
/NOLINENUMBERS «

Default is: /NOLINENUMBERS

Abbreviations are: /L | /NOLI

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 you are linking.

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).

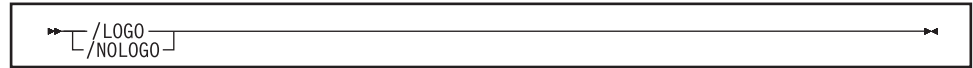
If you give the linker an object file without line number information, the /LINENUMBERS option has no effect.

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, plus the extension `.map`. You can override the default name by specifying a map file name.

---

## **/LOGO, /NOLOGO**



Default is: `/LOGO`

Abbreviations are: `/LO` | `/NOL`

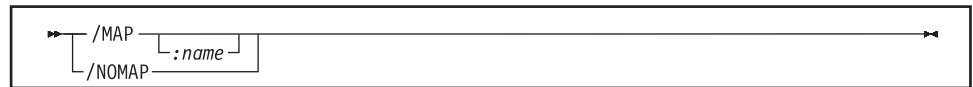
Use `/NOLOGO` to suppress the product information that appears when the linker starts.

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.

---

## **/MAP, /NOMAP**



Default is: `/NOMAP`

Abbreviations are: `/M` | `/NOM`

Use `/MAP` to generate a map file called *name*. The 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.

If you do not specify a directory, the map file is generated into the current working directory. If you do not specify *name*, the map file has the same name as the executable output file, with the extension `map`.

By default, the linker does not produce a map file.

---

## /OUT

» /OUT:*name* «

Default is: Name of first .OBJ file with appropriate extension.

Abbreviation is: /O

Use /OUT to specify a name for the executable output file.

If you do not provide an extension with *name*, then the linker provides an extension based on the type of file you are producing:

File produced	Default extension
Executable program	EXE
Dynamic link library	DLL

If you do not use the /OUT option, then the linker uses the file name of the first object file you specified, with the appropriate extension.

---

## /PMTYPE

» /PMTYPE:*type* «

Default is: /PMTYPE:VIO

Abbreviation is: /PM

Use /PMTYPE to specify the type of EXE file that the linker generates. Do not use this option when generating dynamic link libraries (DLLs).

You must 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.

---

## /SECTION

» /SECTION:*name*, *attribute* «

Default is: Depends on segment type

Abbreviation is: /SEC

Use /SECTION to specify memory-protection attributes for the *name* section. *name* is case sensitive. You can specify the following attributes:

Letter	Sets attribute
E or X	EXECUTE
R	READ
S	SHARED
W	WRITE

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:

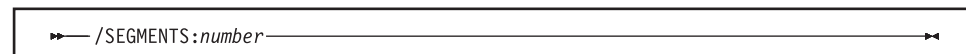
```
/SEC:dseg1,RS
```

### Defaults

Sections are assigned attributes by default, as follows:

Segment	Default attributes
Code sections	EXECUTE, READ (ER)
Data sections (in .EXE file)	READ, WRITE (RW), not shared
Data sections (in .DLL file)	READ, WRITE, not shared
CONST32_RO section	READ, SHARED (RS)

## /SEGMENTS



Default is: /SEGMENTS:256

Abbreviation is: /SE

Use /SEGMENTS to set the number of logical segments a program can have. You can set *number* to any value in the range 1 to 16375.

For each logical segment, the linker must allocate space to keep track of segment information. By using a relatively low segment limit as a default (256), the linker is able to link faster and allocate less storage space.

When you set the segment limit higher than 256, the linker allocates more space for segment information. This results in slower linking, but allows you to link programs with a large number of segments.

For programs with fewer than 256 segments, you can improve link time and reduce linker storage requirements by setting *number* to the actual number of segments in the program.

#### RELATED TASKS

“Specifying linker options” on page 153

---

## /STACK

» /STACK:reserve└┐,commit«

Default is: /STACK:0x100000,0x1000

Abbreviation is: /ST

Use /STACK to set the stack size (in bytes) of your program. The size must be an even number from 0 to 0xFfffffe. If you specify an odd number, it is rounded up to the next even number.

*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 may be slower.

---

## /STUB

» /STUB:filename«

Default is: None

Abbreviation is: /STU

Use /STUB to specify the name of the DOS executable at the beginning of the output file created.

By default, the linker defines its own stub.

---

## /SUBSYSTEM

» /SUBSYSTEM:subsystem└┐,major└┐.minor«

Default is: /SUBSYSTEM:WINDOWS,4.0

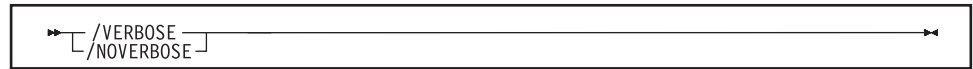
Abbreviation is: /SU

Use /SUBSYSTEM to specify the subsystem and version required to run the program. 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



Default is: /NOVERBOSE

Abbreviations are: /VERB | /NOV

Use /VERBOSE to have the linker display information about the linking process as it occurs, including the phase of linking and the names and paths of the object files being linked.

If you are having 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 being linked 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.

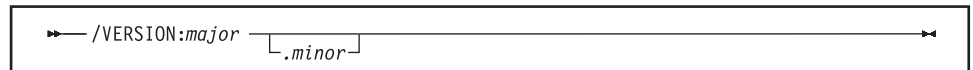
/VERBOSE is the same as /INFORMATION.

### RELATED REFERENCES

“/INFORMATION, /NOINFORMATION” on page 210

---

## /VERSION



Default is: /VERSION:0.0

Abbreviation is: /VER

Use /VERSION to write a version number in the header of the run file. The *major* and *minor* arguments are integers in the range 0 to 65535.



---

## Chapter 13. Run-time options

The following run-time options are supported:

Option	Description	Default	Abbreviation
"CHECK"	Flags checking errors.	CHECK(ON)	CH
"DEBUG" on page 218	Specifies whether the COBOL debugging sections specified by the USE FOR DEBUGGING declarative are active.	NODEBUG	None
"ERRCOUNT" on page 218	Specifies how many conditions of severity 1 (W-level) can occur before the run unit terminates abnormally.	ERRCOUNT(20)	None
"FILESYS" on page 218	Specifies the file system used for files for which no explicit file system selections are made, either through an ASSIGN or an environment variable.	FILESYS(STL) for Windows, FILESYS(VSA) for AIX	None
"TRAP" on page 219	Indicates whether COBOL intercepts exceptions.	TRAP(ON)	None
"UPSI" on page 219	Sets the eight UPSI switches on or off for applications that use COBOL routines.	UPSI(00000000)	None

---

### CHECK

CHECK flags checking errors in an application. In COBOL, index, subscript, and reference modification ranges are checking errors.

»— CHECK — ( <table border="1"><tr><td>ON</td></tr><tr><td>OFF</td></tr></table> ) —————>>	ON	OFF
ON		
OFF		

Default is: CHECK(ON).

Abbreviation is: CH

**ON** Specifies that run-time checking is performed.

**OFF** Specifies that run-time checking is not performed.

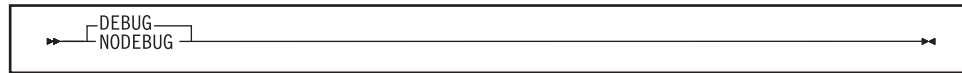
**Usage note:** CHECK(ON) has no effect if NOSSRANGE was in effect at compile time.

**Performance consideration:** If you compiled your 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.

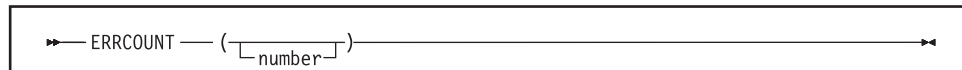
**NODEBUG**  
Suppresses the debugging sections.

**Performance consideration:** To improve performance, use this option only while debugging.

---

## ERRCOUNT

ERRCOUNT specifies how many conditions of severity 1 (W-level) can occur before the run unit terminates abnormally. Any severity 2 (E-level) or higher condition will result in termination of the run unit regardless of the ERRCOUNT option.



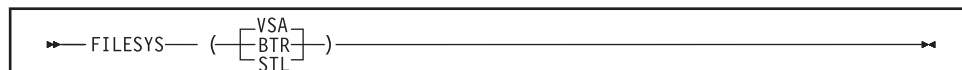
Default: ERRCOUNT(20).

**number**  
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.

---

## 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.



Default is: FILESYS(STL).

**VSA** The file system is VSAM.

**BTR** The file system is Btrieve.

**STL** The file system is STL.

Only the first three characters of the file-system identifier are used and the identifier is case insensitive. For example, the following examples are all valid specifications for VSAM:

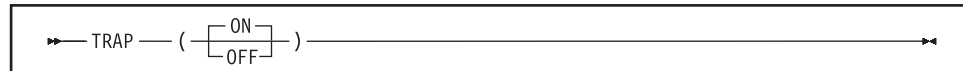
- FILESYS(VSA)

- FILESYS(vSAM)
- FILESYS(vsa)

---

## TRAP

TRAP indicates whether COBOL intercepts exceptions.



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:

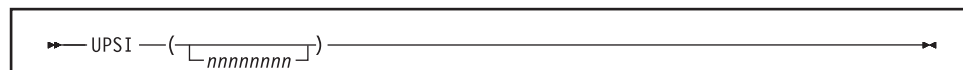
- Use TRAP(OFF) only when you need to analyze a program exception before COBOL handles it.
- When you specify TRAP(OFF) in a non-CICS environment, no exception handlers are established.
- Running with TRAP(OFF) (for exception diagnosis purposes) can cause many side effects because COBOL requires TRAP(ON). 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).

### nnnnnnnn

*n* represents one UPSI switch between 0 and 7, the leftmost *n* representing the first switch. Each *n* can either be 0 (off) or 1 (on).



---

## Chapter 14. Debugging

You can choose from two approaches to determine the cause of problems in program behavior of your application: source-language debugging or the interactive debugger.

For source-language debugging COBOL provides several language elements, compiler options, and listing outputs that make debugging easier.

Besides using the features inherent in COBOL, you can use the IBM Distributed Debugger, a graphical debugging tool you will find useful for debugging programs.

- Interactive debugging (in full-screen or line mode) or in batch mode

During an interactive full-screen mode session, you can use the Debug Tool's full-screen services and session panel windows on a 3270 device to debug your program as it is running.

- COBOL-like commands

For each high-level language supported, commands for coding actions to be taken at breakpoints are provided in a syntax similar to that programming language. (This feature is not available to workstation users.)

- Mixed-language debugging

You can debug an application that contains programs written in different language. Debug Tool automatically determines the language of the program or subprogram being run.

- COBOL-CICS debugging

Debug Tool supports the debugging of CICS applications in both interactive and batch mode.

- Support for remote debugging

Workstation users can use the IBM VisualAge COBOL product for debugging programs residing on OS/390. VisualAge COBOL is available as a separate product or as the Enterprise Workstation feature of this compiler.

### RELATED TASKS

"Debugging with source language"

"Debugging using compiler options" on page 225

"Getting listings" on page 231

"Debugging user exits" on page 240

"Debugging assembler routines" on page 241

### RELATED REFERENCE

Distributed debugger: overview (online help)

---

## Debugging with source language

You can use several COBOL language features to pinpoint the cause of a failure in your program. If the program is part of a large application already in production (precluding source updates), write a small test case to simulate the failing part of the program. Code certain 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 223  
 “Validating data” on page 223  
 “Finding uninitialized data” on page 223  
 “Generating information about procedures” on page 223

#### RELATED REFERENCES

Source language debugging (*IBM COBOL 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 have used explicit scope terminators to end statements in your program, the logic of your program 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 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 17

#### RELATED REFERENCES

DISPLAY statement (*IBM COBOL 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, include a test after each input-output statement to check for a nonzero value in the status key. If the value is nonzero (as reported in an error message), you should 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 have determined 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-0`, or `EXTEND`.

Code each `USE AFTER STANDARD ERROR` statement in a section immediately after the `DECLARATIVE SECTION` keyword of the `PROCEDURE DIVISION`.

### RELATED TASKS

“Coding `ERROR` declaratives” on page 129

### RELATED REFERENCES

Status key values and meanings (*IBM COBOL Language Reference*)

Status key (*IBM COBOL Language Reference*)

## Validating data

If you suspect that your program is trying to perform arithmetic on nonnumeric data or is somehow receiving the wrong type of data on an input record, use the class test to validate the type of data. The class test checks whether data is alphabetic, alphabetic-lower, alphabetic-upper, MBCS, KANJI, national or numeric.

### RELATED REFERENCES

Class condition (*IBM COBOL Language Reference*)

## Finding uninitialized data

Use `INITIALIZE` or `SET` statements to initialize a table or variable when you suspect that the problem might be caused by residual data left in those fields.

If the problem you are having happens intermittently and not always with the same data, the problem could be that a switch is not initialized but generally is set to the right value (0 or 1) by accident. By including a `SET` statement to ensure that the switch is initialized, you can either determine that the uninitialized switch is the problem or remove that as a possible cause.

### RELATED REFERENCES

`INITIALIZE` statement (*IBM COBOL Language Reference*)

`SET` statement (*IBM COBOL Language Reference*)

## Generating information about procedures

Generate information about your program or test case and how it is running with 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 check 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 control passes to that procedure. You can use the counter technique to check items such as these:

- How many times a `PERFORM` 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 have debugging lines or debugging statements or both in your program.

### Debugging lines

Debugging lines are statements that are identified by a `D` in column 7. To make debugging lines in your program active, include the `WITH DEBUGGING MODE` clause on the `SOURCE-COMPUTER` line in the `ENVIRONMENT DIVISION`. Otherwise debugging lines are treated as comments.

### Debugging statements

Debugging statements are the statements 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 contained in nested programs.

To use debugging statements in your program, you must include both the `WITH DEBUGGING MODE` clause and the `DEBUG` run-time 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

Debugging line (*IBM COBOL Language Reference*)

Coding debugging sections (*IBM COBOL Language Reference*)

`DEBUGGING` declarative (*IBM COBOL Language Reference*)

### Example: `USE FOR DEBUGGING`

These program segments show what kind of statements are needed to use a `DISPLAY` statement and a `USE FOR DEBUGGING` declarative to test a program. The `DISPLAY` statement generates information on the terminal or in the 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

```

In this example 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 shows 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.
```

Now a message is displayed before every nondebugging procedure in the outermost program is run.

---

## Debugging using compiler options

Use certain compiler options to generate information to help you find errors in your program such as these:

- Syntax errors such as duplicate data names (NOCOMPILE)
- Missing sections (SEQUENCE)
- Invalid subscript values (SSRANGE)

In addition, you can use certain compiler options to help you find these elements in your program:

- Error messages and where the errors occurred (FLAG)

- Program entity definitions and references (XREF)
- Items you defined in the DATA DIVISION (MAP)
- Verb references (VBREF)
- Assembler-language expansion (LIST)

You can get a copy of your source (by using the SOURCE compiler option) or a listing of generated code (by using the LIST compiler option).

There is also a compiler option (TEST) that you need to use to prepare your program for debugging.

#### RELATED TASKS

- “Selecting the level of error to be diagnosed” on page 228
- “Finding coding errors”
- “Finding line sequence problems” on page 227
- “Checking for valid ranges” on page 227
- “Finding program entity definitions and references” on page 230
- “Listing data items” on page 230
- “Getting listings” on page 231
- “Preparing to use the debugger” on page 231

#### RELATED REFERENCES

- “COMPILE” on page 166
- “SEQUENCE” on page 186
- “SSRANGE” on page 189
- “FLAG” on page 174
- “XREF” on page 196
- “MAP” on page 180
- “VBREF” on page 195
- “LIST” on page 179
- “TEST” on page 190

## Finding coding errors

Use the NOCOMPILE compiler option for compiling conditionally or for checking syntax only. When used with the SOURCE option, this option produces a listing that will help you find your COBOL coding mistakes, such as missing definitions, improperly defined data names, and duplicate data names.

### Checking syntax only

Use NOCOMPILE without parameters to have the compiler only syntax-check the source program and produce no object code. If you also specify the SOURCE option, the compiler produces a listing after completing the syntax check.

The following compiler options are suppressed when you use NOCOMPILE without parameters: LIST, OBJECT, OPTIMIZE, SSRANGE, and TEST.

### Compiling conditionally

When you use NOCOMPILE(*x*), where *x* is one of the severity levels for errors, your program is compiled if all the errors are of a lower severity than the *x* level. The severity levels (from highest to lowest) that you can use are S (severe), E (error), and W (warning).

If an error of *x* level or higher occurs, the compilation stops and your program is syntax-checked only. You receive a source listing if you have specified the SOURCE option.

RELATED REFERENCES  
"COMPILE" on page 166

## Finding line sequence problems

Use the SEQUENCE compiler option to find statements that are out of sequence. These breaks in sequence indicate that a section of your source program was moved or deleted.

When you use SEQUENCE, the compiler checks the source statement numbers you have supplied to see whether they are in ascending sequence. Two asterisks are placed beside statement numbers that are out of sequence. Also, the total number of these statements is printed as the first line of the diagnostics after the source listing.

RELATED REFERENCES  
"SEQUENCE" on page 186

## Checking for valid ranges

Use the SSRANGE compiler option to check the following ranges:

- 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?

When the SSRANGE option is specified, checking is performed at run time when both of the following are true:

- The COBOL statement containing the indexed, subscripted, variable-length, or reference-modified data item is actually performed.
- The CHECK run-time option is ON.

If a check finds that an address is generated outside the address range of the data item containing the referenced data, an error message is generated and the program stops. The error 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, or reference modifiers are literals in a given data reference and they result in a reference outside the data item, the error is diagnosed at compile time, regardless of the setting of the SSRANGE compiler option.

**Performance consideration:** SSRANGE can degrade the performance of your program somewhat because of the extra overhead to check each subscripted or indexed item.

RELATED REFERENCES  
"SSRANGE" on page 189  
"Performance-related compiler options" on page 460

## Selecting the level of error to be diagnosed

Use the FLAG compiler option to select the level of error to be diagnosed during compilation and to indicate whether syntax-error messages are embedded in the listing. Use FLAG(I) or FLAG(I,I) to be notified of all errors in your program.

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.

Severity level	What you get when you specify the corresponding level
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 the 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 containing 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 CONCEPTS

“Severity codes for compiler error messages” on page 149

### RELATED TASKS

“Generating a list of compiler error messages” on page 150

### RELATED REFERENCES

“FLAG” on page 174

“Messages and listings for compiler-detected errors” on page 150

## Example: embedded messages

The following example shows the embedded messages generated by specifying a second parameter on the FLAG option. Some messages in the summary apply to more than one COBOL statement.

```

000977 /
000978 *****
000979 *** INITIALIZE PARAGRAPH ***
000980 *** Open files. Accept date, time and format header lines. ***
000981 IA4690*** Load location-table. ***
000982 *****
000983 100-initialize-paragraph.
000984 move spaces to ws-transaction-record IMP 339
000985 move spaces to ws-commuter-record IMP 315
000986 move zeroes to commuter-zipcode IMP 326
000987 move zeroes to commuter-home-phone IMP 327
000988 move zeroes to commuter-work-phone IMP 328
000989 move zeroes to commuter-update-date IMP 332
000990 open input update-transaction-file 203
==000990==> IGYPS2052-S An error was found in the definition of file "LOCATION-FILE". The
reference to this file was discarded.
000991 location-file 192
000992 i-o commuter-file 180
000993 output print-file 216
000994 if loccode-file-status not = "00" or 248
000995 update-file-status not = "00" or 247
000996 updpriint-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 = " updpriint-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-vsam-error 1387
001007 1 display "Commuter File Status (OPEN) = "
001008 1 commuter-file-status 240
001009 1 perform 900-abnormal-termination 1433
001010 IA4790 end-if
001011 accept ws-date from date UND
==001011==> IGYPS2121-S "WS-DATE" was not defined as a data-name. The statement was discarded.
001012 IA4810 move corr ws-date to header-date UND 463
==001012==> IGYPS2121-S "WS-DATE" was not defined as a data-name. The statement was discarded.
001013 accept ws-time from time UND
==001013==> IGYPS2121-S "WS-TIME" was not defined as a data-name. The statement was discarded.
001014 IA4830 move corr ws-time to header-time UND 457
==001014==> IGYPS2121-S "WS-TIME" was not defined as a data-name. The statement was discarded.
001015 IA4840 read location-file 192

```

DATA VALIDATION AND UPDATE PROGRAM FLAGOUT Date 09/30/1999 Time 12:26:53 Page 69

```

LineID Message code Message text
192 IGYDS1050-E File "LOCATION-FILE" contained no data record descriptions.
899 IGYPS2052-S The file definition was discarded.
An error was found in the definition of file "LOCATION-FILE".
The reference to this file was discarded.
Same message on line: 990
1011 IGYPS2121-S "WS-DATE" was not defined as a data-name. The statement was discarded.
Same message on line: 1012
1013 IGYPS2121-S "WS-TIME" was not defined as a data-name. The statement was discarded.
Same message on line: 1014
1015 IGYPS2053-S An error was found in the definition of file "LOCATION-FILE".
This input/output statement was discarded.
Same message on line: 1027
1026 IGYPS2121-S "LOC-CODE" was not defined as a data-name. The statement was discarded.
1209 IGYPS2121-S "COMMUTER-SHIFT" was not defined as a data-name. The statement was discarded.
Same message on line: 1230
1210 IGYPS2121-S "COMMUTER-HOME-CODE" was not defined as a data-name. The statement was discarded.
Same message on line: 1231
1212 IGYPS2121-S "COMMUTER-NAME" was not defined as a data-name. The statement was discarded.
Same message on line: 1233
1213 IGYPS2121-S "COMMUTER-INITIALS" was not defined as a data-name. The statement was discarded.
Same message on line: 1234
1223 IGYPS2121-S "WS-NUMERIC-DATE" was not defined as a data-name. The statement was discarded.
Messages Total Informational Warning Error Severe Terminating
Printed: 19 1 18
* Statistics for COBOL program FLAGOUT:
* Source records = 1755
* Data Division statements = 279
* Procedure Division statements = 479
End of compilation 1, program FLAGOUT, highest severity: Severe.
Return code 12

```

## 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 variables, use the XREF(SHORT) option.

Use both the XREF (with FULL or SHORT) and the SOURCE options to get a modified cross-reference printed to the right of the source listing. This embedded cross-reference gives 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 character set (MBCS) 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 237

“Example: XREF output - program-name cross-references” on page 238

“Example: embedded cross-reference” on page 239

### RELATED TASKS

“Getting listings” on page 231

### RELATED REFERENCES

“XREF” on page 196

## Listing data items

Use the MAP compiler option to produce a listing of the items you defined in the DATA DIVISION, plus all items implicitly declared.

In addition, when you use the MAP option, an embedded MAP summary (which contains condensed data 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 or \*CONTROL NOMAP statements (\*CBL MAP or \*CBL NOMAP statements) throughout the source. For example:

*CONTROL NOMAP	*CBL NOMAP
01 A	01 A
02 B	02 B
*CONTROL MAP	*CBL MAP

“Example: MAP output” on page 235

### RELATED TASKS

“Getting listings” on page 231

### RELATED REFERENCES

“MAP” on page 180

## Preparing to use the debugger

Use the TEST option to prepare your executable COBOL program for use with the debugger.

IBM Distributed Debugger is a debugger that is provided with the VisualAge COBOL compiler. To use the IBM Distributed Debugger to step through a run of your program, specify the TEST compiler option.

The TEST option is required for remote debugging (with the Debug Tool engine running on OS/390). For debugging code that is on the same operating system, you can use the -g option of the cob2 command.

### RELATED TASKS

“Compiling from the command line” on page 144

### RELATED REFERENCES

“TEST” on page 190

---

## 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.

Use	Listing	Contents	Compiler option
To check diagnostic messages about the compilation, a list of the options in effect for the program, and statistics about the content of the program	Short listing	Diagnostic messages about the compile <sup>1</sup> (page 232); list of options in effect for the program; statistics about the content of the program	NOSOURCE, NOXREF, NOVBREF, NOMAP, NOOFFSET, 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 188
To find certain data items; to see the final storage allocation after reentrancy or optimization has been accounted for; to see where programs are defined and check their attributes	Map of DATA DIVISION items	All DATA DIVISION items and all implicitly declared variables	“MAP” on page 180 <sup>2</sup>
		Embedded map summary (in the right margin of the listing for lines in the DATA DIVISION that contain data declarations)	
		Nested program map (if the program contains nested programs)	

Use	Listing	Contents	Compiler option
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	Data names, procedure names, and program names; references to these names	"XREF" on page 196 <sup>2 3</sup> (page 232)
		Embedded modified cross-reference: provides the line number where the data name or procedure name was defined	
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> (page 232)	Generated code	"LIST" on page 179 <sup>2 4</sup> (page 232)
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 195
<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. You can control the selective listing of generated object code by placing *CONTROL LIST and *CONTROL NOLIST statements (*CBL LIST and *CBL NOLIST) in your source. Note that the *CONTROL statement is different from the PROCESS (or CBL) statement.</li> </ol> <p>The assembler listing is written to a file with the same name as the source program with the extension "asm" except for batch compiles with the SEPOBJ option.</p>			

"Example: short listing"

"Example: SOURCE and NUMBER output" on page 234

"Example: MAP output" on page 235

"Example: embedded map summary" on page 235

"Example: nested program map" on page 237

"Example: XREF output - data-name cross-references" on page 237

"Example: XREF output - program-name cross-references" on page 238

"Example: embedded cross-reference" on page 239

"Example: VBREF compiler output" on page 240

#### RELATED TASKS

"Generating a list of compiler error messages" on page 150

#### RELATED REFERENCES

"Messages and listings for compiler-detected errors" on page 150

"SEPOBJ" on page 185

## Example: short listing

The numbers used in the explanation after the listing correspond to those annotating the listing. For illustrative purposes, some errors that cause diagnostic messages to be issued were deliberately introduced.

Invocation parameters: (2)  
 quote  
 PROCESS(CBL) statements:  
 CBL FLAG(I,I),MAP,TEST  
 CBL NONUMBER,QUOTE,SEQ,XREF,VBREF (3)  
 Options in effect: (4)  
 ADATA  
 QUOTE  
 BINARY(NATIVE)  
 CALLINT(SYSTEM,NODESCRIPTOR)  
 CHAR(NATIVE)  
 NOCICS  
 COLLSEQ(BINARY)  
 NOCOMPILE(S)  
 NOCURRENCY  
 NODATEPROC  
 NODYNAM  
 ENTRYINT(SYSTEM)  
 EXIT(NOINEXIT,NOPRTEXT,NOLIBEXIT,ADEXIT(IWZRMGUX))  
 FLAG(I,I)  
 NOFLAGSTD  
 FLOAT(NATIVE)  
 NOIDLGEN  
 LIB  
 LINECOUNT(60)  
 NOLIST  
 MAP  
 NONUMBER  
 NOOPTIMIZE  
 PGMNAME(LONGUPPER)  
 PROBE  
 NOPROFILE  
 SEPOBJ  
 SEQUENCE  
 SIZE(2097152)  
 SOURCE  
 SPACE(1)  
 SQL  
 NOSSRANGE  
 TERM  
 TEST  
 NOTHREAD  
 NOTILED  
 TRUNC(STD)  
 NOTYPECHK  
 VBREF  
 NOWORD  
 XREF(FULL)  
 YEARWINDOW(1900)  
 ZWB

DATA VALIDATION AND UPDATE PROGRAM (5) SLISTING Date 09/30/1999 Time 12:26:53 Page 2  
 LineID Message code Message text (6)  
 IGYDS0139-W Diagnostic messages were issued during processing of compiler options. These messages are  
 located at the beginning of the listing.  
 193 IGYDS1050-E File "LOCATION-FILE" contained no data record descriptions. The file definition was discarded.  
 889 IGYPS2052-S An error was found in the definition of file "LOCATION-FILE". The reference to this file  
 was discarded.  
 Same message on line: 983  
 993 IGYPS2121-S "WS-DATE" was not defined as a data-name. The statement was discarded.  
 Same message on line: 994  
 995 IGYPS2121-S "WS-TIME" was not defined as a data-name. The statement was discarded.  
 Same message on line: 996  
 997 IGYPS2053-S An error was found in the definition of file "LOCATION-FILE". This input/output statement  
 was discarded.  
 Same message on line: 1009  
 1008 IGYPS2121-S "LOC-CODE" was not defined as a data-name. The statement was discarded.  
 1219 IGYPS2121-S "COMMUTER-SHIFT" was not defined as a data-name. The statement was discarded.  
 Same message on line: 1240  
 1220 IGYPS2121-S "COMMUTER-HOME-CODE" was not defined as a data-name. The statement was discarded.  
 Same message on line: 1241  
 1222 IGYPS2121-S "COMMUTER-NAME" was not defined as a data-name. The statement was discarded.  
 Same message on line: 1243  
 1223 IGYPS2121-S "COMMUTER-INITIALS" was not defined as a data-name. The statement was discarded.  
 Same message on line: 1244  
 1233 IGYPS2121-S "WS-NUMERIC-DATE" was not defined as a data-name. The statement was discarded.  
 Messages Total Informational Warning Error Severe Terminating (7)  
 Printed: 21 2 1 18  
 \* Statistics for COBOL program SLISTING: (8)  
 Source records = 1765  
 Data Division statements = 277  
 Procedure Division statements = 513  
 End of compilation 1, program SLISTING, highest severity: Severe. (9)  
 Return code 12

- (1) COBOL default page header, including compiler level information from the LVLINFO installation-time compiler option.
- (2) Message about options passed to the compiler at compiler invocation. This message does not appear if no options were passed.

- (3) Options coded in the PROCESS (or CBL) statement.
- (4) Status of options at the start of this compilation.
- (5) Customized page header resulting from the COBOL program TITLE statement.
- (6) 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.
- (7) Count of diagnostic messages in this program, grouped by severity level.
- (8) Program statistics for the program SLISTING.
- (9) 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.

```

DATA VALIDATION AND UPDATE PROGRAM (1)                                IGYTCARA Date 09/30/1999 Time 12:26:53 Page 22
LineID PL SL -----*A-1-B--+-+---2---+---3---+---4---+---5---+---6---+---7-|-+---8 Cross-Reference (2)
(3)    (4)    (5)
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-vsam-master-file.
088151** 088150 display "perform 050-create-vsam-master finished".
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** 089650 else
2 089600 perform 400-print-transaction-errors
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) Customized page header resulting from the COBOL program TITLE statement.
- (2) Scale line labels Area A, Area B, and source code column numbers.
- (3) Source code line number assigned by the compiler.
- (4) Program (PL) and statement (SL) nesting level.
- (5) 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 annotating the output.

DATA VALIDATION AND UPDATE PROGRAM

IGYTCARA Date 09/30/1999 Time 12:26:53 Page 22

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) Source LineID	(3) (4) Hierarchy and Data Name	(5) Length(Displacement)	(6) Data Type	(7) Data Def 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-IGYTCARA . . . . .	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 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 236

## Example: embedded map summary

The following example shows an embedded map summary from specifying the MAP option. The summary appears in the right margin of the listing for lines in the DATA DIVISION that contain data declarations.

```

000002      Identification Division.
000003
000004      Program-id.    EMBMAP.
. . .
000176      Data division.
000177      File section.
000178
000179
000180      FD  COMMUTER-FILE
000181          record 80 characters.
000182          01 commuter-record.
000183              05 commuter-key      PIC x(16).
000184              05 filler            PIC x(64).
. . .
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 S9999 comp.
000231          77 ws-type      pic x(3)  value spaces.
000232
000233
000234          01 i-f-status-area.
000235              05 i-f-file-status      pic x(2).
000236              88 i-o-successful      value zeroes.
000237
000238
000239          01 status-area.
000240              05 commuter-file-status  pic x(2).
000241              88 i-o-okay      value zeroes.
000242              05 commuter-vsam-status.
000243                  10 vsam-r15-return-code  pic 9(2) comp.
000244                  10 vsam-function-code  pic 9(1) comp.
000245                  10 vsam-feedback-code  pic 9(3) comp.
000246
000247          77 update-file-status      pic xx.
000248          77 loccode-file-status      pic xx.
000249          77 updpri-print-file-status  pic xx.
. . .
000877      procedure division.
000878          000-do-main-logic.
000879              display "PROGRAM EMBMAP - Beginning".
000880              perform 050-create-vsam-master-file.
. . .

```

- (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 and 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

This table describes the terms used in the listings produced by the MAP option.

Usage	Description
ALPHA-EDIT	Alphabetic-edited
ALPHABETIC	Alphabetic
AN-EDIT	Alphanumeric-edited
BINARY	Binary (computational)
COMP-1	Internal floating point (single precision)

Usage	Description
COMP-2	Internal floating point (double precision)
DBCS	DBCS (display-1)
DBCS-EDIT	DBCS edited
DISP-NUM	External decimal
DISPLAY	Alphanumeric
File processing method (VSAM)	File (FD)
GROUP	Group fixed-length
GRP-VARLEN	Group variable-length
INDEX	Index
INDX-NAME	Index name
Level name	Condition (77)
Level name for condition-name	Condition (88)
Level name for RENAMEs	Condition (66)
NUM-EDIT	Numeric-edited
OBJECT REFERENCE	Object reference
PACKED-DEC	Internal decimal (computational-3)
POINTER	Pointer
PROCEDURE-POINTER	Pointer to an externally callable program (or function)
Sort file definition	Sort definition (SD)

### Example: nested program map

This example shows a map of nested procedures produced by specifying the MAP compiler option.

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
LineID Level Program Name from PROGRAM-ID paragraph Program
      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, produced by the XREF compiler option.

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-VSAM-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 P1395 P1399
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.

(1)	(2)	(3)
Defined	Cross-reference of programs	References
EXTERNAL	EXTERNAL1. . . . .	25
2	X. . . . .	41
12	X1. . . . .	33 7
20	X11. . . . .	25 16
27	X12. . . . .	32 17
35	X2. . . . .	40 8

- (1) Line number where the program name was defined. If the program is external, the word EXTERNAL is displayed instead of a definition line number.
- (2) Program name.
- (3) Line numbers where the program is referenced.

### Example: embedded cross-reference

The following example shows a modified cross-reference embedded in the source listing, 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-vsam-master-file.          932 (1)
000882      perform 100-initialize-paragraph.             984
000883      read update-transaction-file into ws-transaction-record
000884      at end                                         204 340
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 your program and 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
		103 109 117 117 118 119 138 139 139 139 139 139 139 140 140 140
		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
		123 123 123 123 123 124 124 124 124 125 125 125 125 125 125 126
		126 126 126 126 127 127 127 127 128 128 129 129 130 130 130 130
		131 131 131 131 131 132 132 132 132 132 132 133 133 133 133 133
		134 134 134 134 134 135 135 135 135 135 135 136 136 137 137 137
		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 96
		96 96 97 97 97 100 100 101 102 104 109 109 111 116 116 117 117
		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
		155 156 156 156 156 157 157 157 157 157 158 158 158 158 159 159
		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
		147 151 165 165 166 166 167 174 174 174 174 174 174 175 175

(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.

1. Use cob2 with the -# option to see how cob2 calls the main compiler module and what options it passes. For example, the following cob2 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
```

This is what you will see:

```
igyccob2 -q"EXIT(ADEXIT(IWZRMGUX))" pgmname.cbl  
ilink /free /no1 /pm:vio pgmname.obj
```

This call to IGYCCOB2 is what calls your user exit.

2. Debug the user exit as follows:

```
idbug igyccob2 -q"EXIT(ADEXIT(IWZRMGUX))" pgmname.cbl
```

The debugger automatically stops at the beginning of your user exit, assuming 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, during startup the debugger 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:

```
IDBUG -i progrname
```



---

## Part 3. Accessing databases

### Chapter 15. Programming for a DB2

<b>environment</b>	245
DB2 coprocessor	245
Coding SQL statements	245
Using SQL INCLUDE	246
Using binary items	246
Determining the success of SQL statements	246
Starting DB2 before compiling	246
Compiling with the SQL option	247
Separating SQL suboptions	247
Using package and bind file names	247
Ignored options	248

### Chapter 16. Developing COBOL programs for CICS

Coding COBOL applications to run under CICS	250
Coding for ASCII-EBCDIC differences	251
Getting the system date under CICS	251
Making dynamic calls under CICS	251
DLL considerations	252
Calling between COBOL and C/C++ under CICS	253
Compiling and running CICS programs	253
EBCDIC-enabled COBOL programs under CICS	253
Selecting run-time options	254
Debugging CICS programs	254

### Chapter 17. Open Database Connectivity

<b>(ODBC)</b>	255
Comparison of ODBC and embedded SQL	255
Background	256
Installing and configuring software for ODBC	256
Coding ODBC calls from COBOL: overview	256
Using data types appropriate for ODBC	256
Passing pointers as arguments in ODBC calls	257
Example: passing pointers as arguments in ODBC calls	258
Accessing function return values in ODBC calls	259
Testing bits in ODBC calls	259
Using COBOL copybooks for ODBC APIs	260
Example: sample program using ODBC copybooks	261
Example: copybook for ODBC procedures	262
Example: copybook for ODBC data definitions	265
ODBC names truncated or abbreviated for COBOL	266
Compiling and linking programs that make ODBC calls	267
Understanding ODBC error messages	267
Errors from an ODBC driver	267
Errors from the data source	267
Errors from the driver manager	268



---

## Chapter 15. Programming for a DB2 environment

In general, the coding for your COBOL program will be the same whether or not you want it 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, you need to do the following:

- Code any SQL statements 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.

### RELATED CONCEPTS

*"DB2 coprocessor"*

### RELATED TASKS

*"Coding SQL statements"*

*"Starting DB2 before compiling" on page 246*

*"Compiling with the SQL option" on page 247*

*DB2 Application Development Guide*

### RELATED REFERENCES

*DB2 SQL Reference*

---

## DB2 coprocessor

When you use the DB2 coprocessor, the compiler handles your source program containing embedded SQL statements without your having to use a separate preprocessor. When the compiler encounters SQL statements at significant points in the source program, it interfaces with the DB2 coprocessor. This coprocessor takes appropriate actions on the SQL statements and indicates to the compiler what native COBOL statements to generate for them.

Certain restrictions on the use of COBOL language that applied when the preprocessor was used no longer apply:

- You can identify host variables used on 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.

### RELATED TASKS

*"Compiling with the SQL option" on page 247*

---

## Coding SQL statements

Delimit SQL statements with EXEC SQL and END-EXEC statements. You also need to take these special steps:

- Declare an SQL communication area (SQLCA) in the WORKING-STORAGE SECTION.

- Declare all host variables that you use in SQL statements in the WORKING-STORAGE or LINKAGE sections. However, you do not need to identify them with EXEC SQL BEGIN DECLARE SECTION and EXEC SQL END DECLARE SECTION.

You can use SQL statements even for large objects (such as BLOB and CLOB) and compound SQL. The size of large objects is limited to 16 MB for a group or elementary data item.

## Using SQL INCLUDE

An SQL INCLUDE statement is treated identically to a native COBOL COPY statement, including the path search and the file extensions used. Therefore, the following two lines are treated the same way:

```
EXEC SQL INCLUDE name
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* without a REPLACING clause.

COBOL does not use the DB2 environment variable DB2INCLUDE for SQL INCLUDE processing. However, if you use the standard DB2 copy files, there are no other settings that you must make. If the search rules call for using SYSLIB as the library name, the compiler will find the copy files 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.

## Using binary items

For binary data items that you specify in an SQL statement, use either of these techniques:

- Declare them as USAGE COMP-5.
- Use the TRUNC(BIN) option and the BINARY(NATIVE) option if USAGE BINARY, COMP, or COMP-4 is specified. (This might have a larger impact on performance than using USAGE COMP-5 on individual data items.)

If you specify a USAGE BINARY, COMP, or COMP-4 item when option TRUNC(OPT) or TRUNC(STD) or both are in effect, the compiler will accept the item 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.

## 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 SQLCA to indicate whether the operation succeeded or failed. Your program should test the return code and take any necessary action.

### RELATED CONCEPTS

“Formats for numeric data” on page 34

---

## Starting DB2 before compiling

Because the compiler works in conjunction with the DB2 coprocessor, you must start DB2 before you compile your program.

To connect to the target database for the compile, you can connect before you start the compile or 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 on the SQL compiler option is made available to the DB2 coprocessor. Only the DB2 coprocessor views the content of the string. The following cob2 command passes the database name SAMPLE and the DB2 options USER and USING to the coprocessor:

```
cob2 -q"sql('database sample user myname using mypassword')" mysql.cbl. . .
```

The SQL options that you include in the suboption string are cumulative.

## Separating SQL suboptions

Because of the concatenation of multiple SQL option specifications, you can separate SQL suboptions (which might not fit into a single CBL statement) into multiple CBL statements.

The SQL options specified from multiple sources are concatenated in the order of the specifications. For example, suppose that your mypgm.cbl source file has the following code:

```
cb1 . . . SQL("string2") . . .  
cb1 . . . SQL("string3") . . .
```

When you give the command `cob2 mypgm.cbl -q:"SQL('string1')"` the following SQL option string is passed to the DB2 coprocessor:

```
"string1 string2 string3"
```

The concatenated strings are delimited with single spaces. When multiple instances of the same SQL suboption are found, the last specification of the suboption in the concatenated string will be in effect. The compiler limits the length of the concatenated DB2 option string to 4K bytes.

## 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.

## Ignored options

The following options, which were meaningful to and used by the preprocessor, are ignored by the coprocessor:

- MESSAGES
- NOLINEMACRO
- OPTLEVEL
- OUTPUT
- SQLCA
- TARGET
- WCHARTYPE

### RELATED REFERENCES

*DB2 Command Reference*

---

## Chapter 16. Developing COBOL programs for CICS

You can write CICS applications in COBOL and run them using the following CICS environments:

- VisualAge CICS Enterprise Application Development: primarily intended for developing CICS applications that will ultimately run in a mainframe environment (for convenience, this tool, which is part of VisualAge COBOL, will be referred to as *VisualAge CICS*)
- CICS for Windows NT: intended for developing and running workstation-based applications

After you have installed and configured CICS, you need to prepare COBOL applications to run under CICS. In general, the process is the same regardless of which CICS system you use, but there are some differences in the details. The steps are outlined here, along with differences between the CICS systems.

1. Initialize the CICS environment.

For VisualAge CICS, use the `CICSENV` command file. You can edit `CICSENV.CMD` to configure your CICS and COBOL environment variables. For example, make sure that the operating system can access your programs, files, and copybooks. You do not normally need to run `CICSENV` before starting a CICS application, because `CICSENV` will be run automatically on the first invocation of a CICS command. However, if you have both VisualAge CICS and CICS for Windows NT installed on your system, you should run `CICSENV` before you build your program (that is, before you run `CICSTCL` in step 4) if you want VisualAge CICS to be used instead of CICS for Windows NT.

For CICS for Windows NT, the environment variables are set when you start the CICS region. To set environment variables that are unique to a particular region, set them in `\var\cics_regions\xxx\environment` (where `xxx` is the name of the region). To set environment variables that are in effect for all CICS regions, set them as system environment variables for Windows NT. No CICS for Windows NT region can see environment variables that you set as user environment variables for Windows NT.

2. Create the application by using an editor to do the following:

- Code your program using COBOL statements and CICS commands.
- Create COBOL copybooks.
- Create the CICS screen maps that your program uses.

3. Use the CICS command `CICSMAP` to process the screen maps.

4. Use the CICS command `CICSTCL` to translate the CICS commands to valid COBOL statements and to compile and link the program.

5. If you use VisualAge CICS, start CICS using `CICSRUN`.

For CICS for Windows NT, starting CICS is deferred until step 8.

6. Define the resources for your application, such as transactions, application programs, and files:

- For VisualAge CICS, use the CEDA transaction.
- For CICS for Windows NT, use the CICS Administration Utility.

7. If you use VisualAge CICS, activate the CICS resources (such as PCT and FCT entries) in either of two ways:

- Shut down CICS by using the CQIT transaction, and then restart it by using CICSRUN.
- Use the Install action of the CEDA transaction.

For CICS for Windows NT, you will activate the resources when you start the CICS region (in the next step).

8. If you use CICS for Windows NT, start your CICS region by using the CICS Administration Utility.

For VisualAge CICS, your CICS system should have already been started.

9. At your CICS terminal, run the application by entering the four-character transaction ID 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 executing.

#### RELATED TASKS

*CICS Customization Guide*

*CICS Application Programming Guide*

---

## Coding COBOL applications to run under CICS

In general, the COBOL language is supported in a CICS environment. However, there are certain restrictions and considerations you should be aware of when you code COBOL applications to run on CICS.

Do not use EXEC, CICS, DLI, or END-EXEC for variable names, or user-specified parameters to the main program.

In addition, it is recommended that you not use:

- FILE-CONTROL entry in the ENVIRONMENT DIVISION
- FILE SECTION of the DATA DIVISION
- USE declaratives (except USE FOR DEBUGGING)

The following COBOL language statements are not recommended for use in a CICS environment:

- ACCEPT (format 1, or format 2 with two-digit dates)
- CLOSE
- DELETE
- DISPLAY
- 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 COBOL language elements, be aware that:

- The application is not completely portable to the mainframe CICS environment.
- In the case of a CICS failure, a backout (restoring the resources associated with the failed task) will not be possible.

When you code nested (contained) programs, pass DFHEIBLK and DFHCOMMAREA as parameters to any nested programs that contain EXEC commands or references to the EIB. You must also pass the same parameters to any program that forms part of the control hierarchy between such a program and its top-level program.

## Coding for ASCII-EBCDIC differences

If your CICS program is to run on an ASCII platform (such as NT or AIX) and it accesses EBCDIC data, be aware that neither the CICS run time nor the COBOL run time automatically converts the data to the ASCII collating sequence.

Some data access methods (such as VSAM) can carry out such conversions automatically, but you should not assume that the data will be converted. If your program is designed to access mainframe data and you do not build with the host data type options, you might want to add logic to your program to test whether or not the data is EBCDIC and, if necessary, explicitly convert the collating sequence.

## Getting the system date under CICS

To retrieve the system date in a CICS program, use format-2 ACCEPT with the four-digit year options:

```
ACCEPT identifier FROM DATE YYYYMMDD
ACCEPT identifier FROM DAY YYYYDDD
```

Alternatively, you can use the CURRENT-DATE intrinsic function. These methods work in both CICS and non-CICS environments.

Do not use a format-1 ACCEPT statement in a CICS program. Also, the following forms of the ACCEPT statement to receive two-digit year dates are not supported under CICS:

```
ACCEPT identifier FROM DATE
ACCEPT identifier FROM DAY
```

### RELATED TASKS

“Compiling and running CICS programs” on page 253

“Making dynamic calls under CICS”

Calling between COBOL and C/C++ under CICS (“Calling between COBOL and C/C++ under CICS” on page 253)

“Debugging CICS programs” on page 254

## Making dynamic calls under CICS

You can use dynamic calls in the CICS environment. However, you must set the COBPATH environment variable correctly. Consider the following example.

```
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.
. . .
```

Because alpha is a COBOL program that contains CICS statements, CICS control blocks DFHEIBLK and DFHCOMMAREA must be passed (as shown) to alpha. The source for alpha is in the file alpha.ccp. The CICS command CICSTCL is used to translate, compile, and link alpha.ccp.

With VisualAge CICS, CICSTCL creates a DLL called alpha.dll. You must include the directory where alpha.dll resides in the COBPATH environment variable. CICS documentation describes how to set environment variables for CICS.

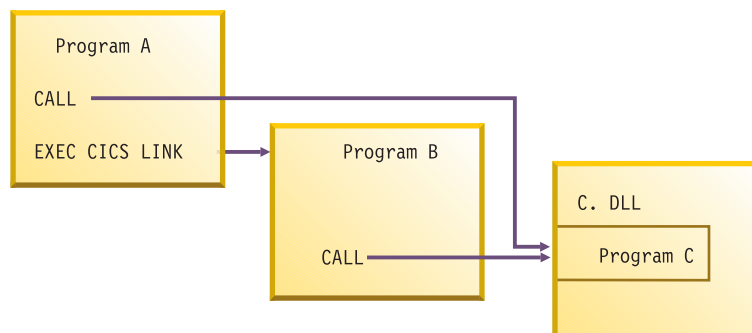
With CICS for Windows NT, CICSTCL creates a DLL called alpha.ibm cob. You must include the directory where alpha.ibm cob resides in the COBPATH environment variable. You must also take the following actions:

- Use the -LIBMCOB option with CICSTCL.
- Set the COBPATH environment variable in the system environment variables, because the CICS for Windows NT region does not recognize user environment variable settings. Alternatively, you can set COBPATH in the \var\cics\_regions\xxx\environment file, in which case it would be 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 are unpredictable. The following figure shows a CICS transaction where 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 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 will be initialized only on the first call.

If you need the subprogram initialized on the first call from each main program, you should statically link a separate copy of the subprogram with each calling program. If you need the subprogram initialized on every call, you should use one of the following methods:

- Put data to be reinitialized in the LOCAL-STORAGE section of the subprogram rather than in WORKING-STORAGE. 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.

## Calling between COBOL and C/C++ under CICS

The following rules govern calls between COBOL and C/C/C++ programs under CICS:

- COBOL programs that contain CICS commands can call C/C/C++ programs as long as the called C/C/C++ programs do not contain any CICS commands.
- C/C/C++ programs that contain CICS commands can call COBOL programs as long as the called COBOL programs do not contain any CICS commands.
- COBOL programs can issue an EXEC CICS LINK or EXEC CICS XCTL to a C/C/C++ program regardless of whether the C/C/C++ program contains CICS commands.

Therefore, if your COBOL program invokes a C/C/C++ program that contains CICS commands (or vice versa), use EXEC CICS LINK or EXEC CICS XCTL rather than the COBOL CALL statement.

---

## Compiling and running CICS programs

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, using TRUNC(OPT) could improve program performance.

You can use a COMP-5 data item instead of BINARY, COMP, or COMP-4 item as an EXEC CICS command argument. COMP-5 data items are treated like BINARY, COMP, or COMP-4 data items are when BINARY(NATIVE) and TRUNC(BIN) are in effect, regardless of whether you explicitly set those options.

TXSeries CICS applications must use the thread-safe version of the COBOL run-time library. Compile such applications with the THREAD compiler option.

You must use the PGMNAME(MIXED) option for applications that use CICS Client.

Avoid using NOLIB and TRUNC(STD) when you compile programs to run in a CICS environment. All other COBOL compiler options are supported.

## EBCDIC-enabled COBOL programs under CICS

VisualAge CICS provides support for running COBOL programs as EBCDIC-enabled programs, but CICS for Windows NT does not.

To prepare your IBM COBOL program to run under CICS as an EBCDIC-enabled program, you must take the following steps:

1. Translate the COBOL program using the BINARY(S370) and the EBCDIC translator options.
2. Compile the program using the CHAR(EBCDIC), COLLSEQ(EBCDIC), and the BINARY(S390) compiler options.

## Selecting run-time options

Use the FILESYS run-time 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 TASKS

*CICS Application Programming Guide*

### RELATED REFERENCES

Compiler options

“Appendix B. System/390 host data type considerations” on page 479

“FILESYS” on page 218

“THREAD” on page 191

---

## Debugging CICS programs

Before you debug your CICS programs, you need to translate them into COBOL. Then you can debug them the same way you would debug any other COBOL program.

Alternatively, you can debug CICS programs using the graphical debugger shipped with the product. Be sure to instruct the compiler to produce symbolic information used by the graphical debugger.

### RELATED CONCEPTS

“Chapter 14. Debugging” on page 221

### RELATED TASKS

“Compiling from the command line” on page 144

*CICS Application Programming Guide*

---

## Chapter 17. 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 preprocessor or coprocessor for a particular database and have 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 IBM VisualAge COBOL 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 OS/2 and UNIX systems.

---

## Installing and configuring software for ODBC

To enable ODBC for data access in IBM VisualAge COBOL, you must:

1. Install an ODBC driver manager and drivers.
2. Add the ODBC database drivers necessary for your installation.
3. 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 NT can host multiple users, each user must configure their own data sources. For detailed configuration information for the specific driver you wish 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 consider data descriptions to fit what ODBC expects and how to pass arguments, access return values for functions, and test bits. IBM VisualAge COBOL also provides copybooks to help you use ODBC calls.

### RELATED TASKS

"Using data types appropriate for ODBC"

"Passing pointers as arguments in ODBC calls" on page 257

"Accessing function return values in ODBC calls" on page 259

"Testing bits in ODBC calls" on page 259

"Using COBOL copybooks for ODBC APIs" on page 260

"Compiling and linking programs that make ODBC calls" on page 267

## 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 corresponding to the indicated ODBC C types of the arguments as shown in the table below:

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)

ODBC C type	COBOL form	Description
SQLFLOAT	COMP-2	Floating point (8 bytes)
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 is in COBOL a pointer to a null-terminated string. You define the target (of the pointer) item with PIC X(*n*), where *n* is large enough to represent the null-terminated field. Some ODBC APIs require you to pass null-terminated character strings as arguments.

Do not use OS/390 host data type options. ODBC APIs expect their parameters to be in native format.

#### RELATED TASKS

“Passing pointers as arguments in ODBC calls”

“Accessing function return values in ODBC calls” on page 259

## Passing pointers as arguments in ODBC calls

If you specify an argument as a pointer to one of the data types acceptable to ODBC, then you need to do one of the following:

- Pass the target item of the pointer BY REFERENCE.
- Define a pointer data item that points to the target item and pass that BY VALUE.
- Pass the ADDRESS OF the target item BY VALUE.

To illustrate, assume the function is defined as

```
RETCODE SQLSomeFunction(PSomeArgument)
```

Here PSomeArgument is defined as an argument pointing 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 fragment of a sample program 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
                                      RETURNING SQL-RC
. . .

```

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 overridden by 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 on 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 query bits. You can use the callable library routine `iwzODBCTestBits` for querying bits. (You must link the `IWZODBC.LIB` import library with an application that calls `iwzODBCTestBits`.)

Call this routine as follows:

```
CALL "iwzODBCTestBits" USING identifier-1, identifier-2 RETURNING identifier-3
```

### *identifier-1*

This is 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*

This is 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*

This is the return value for the test and has the following return values:

- 0**        None of the bits indicated by *identifier-2* is ON in *identifier-1*.
- 1**        All the 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 is used as *identifier-1*).

You must define it as USAGE COMP-5 with PICS9(4).

You can set multiple bits in a field using COBOL arithmetic expressions with the bit masks defined in the ODBCCOB copybook. For example, you can set the 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 of the arithmetic expression above represent disjoint bits from each other as defined for each of such bit mask symbols in ODBCCOB copybook. You should be careful not to repeat the same bit in an arithmetic expression for this purpose (since the operations are arithmetic additions, not logical ORs). For example, the following code results in InfoValue not having the 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

Included with the IBM VisualAge COBOL are copybooks that make it easier for you to access databases with ODBC drivers by using ODBC calls from your COBOL programs. You can use these copybooks with or without modifications.

The copybooks described here 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.

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

Including the paths for the INCLUDE and ODBC folders in your SYSLIB environment variable will ensure that the copybooks are available to the compiler.

The copybook 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 (input and output) and function return values.

The copybook 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). The copybook ODBC3D.CPY contains data declarations used by ODBC3.CPY in the WORKING-STORAGE SECTION (or LOCAL-STORAGE SECTION).

Some COBOL-specific adaptations have been made in these copybooks:

- Underscores (\_) are 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 outer-most 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 265

"Example: copybook for ODBC procedures" on page 262

#### RELATED REFERENCES

"ODBC names truncated or abbreviated for COBOL" on page 266

## Example: sample program using ODBC copybooks

The following sample illustrates the use of three 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
* set licensing information
  PERFORM SQL-SetLicInfo-Function
* 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 sample has 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
                                BY VALUE      or SQL-OV-ODBC2 *
                                BY VALUE      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.

*** SQL-SetLicInfo SECTION *****
SQL-SetLicInfo-Function SECTION.
SQL-SetLicInfo.
    CALL "iwoDBCLicInfo" USING BY VALUE Hdbc.
*** 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
            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

This sample has 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 that ODBC names are longer than 30 characters and their corresponding COBOL names.

ODBC C #define symbol > 30 characters long	Corresponding COBOL name
SQL_AD_ADD_CONSTRAINT_DEFERRABLE	SQL-AD-ADD-CONSTRAINT-DEFER
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

ODBC C #define symbol > 30 characters long	Corresponding COBOL name
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
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 at least two compiler options:

- CALLINT(SYSTEM) compiler option or the >>CALLINT SYSTEM directive for ODBC calls.
- PGMNAME(MIXED) compiler option. The ODBC entry points are case sensitive.

---

## Understanding ODBC error messages

Error messages for ODBC calls can come from any of the following sources:

- ODBC driver
- Database source
- Driver manager

### Errors from an ODBC driver

An error reported on an ODBC driver has the following format:

[*vendor*] [ODBC\_*component*] message

ODBC\_*component* is the component in which the error occurred. For example, an error message from INTERSOLV's SQL Server driver would look like this:

[INTERSOLV] [ODBC SQL Server driver] Login incorrect.

If you get this type of error, check the last ODBC call your application made for possible problems or contact your ODBC application vendor.

### Errors from the data source

An error that occurs in the data source includes the data source name, in the following format:

[*vendor*] [ODBC\_*component*] [*data\_source*] message

With this type of message, ODBC\_component is the component that received the error from the data source indicated. For example, you might get the following message from an Oracle data source:

```
[INTERSOLV] [ODBC Oracle driver] [Oracle] ORA-0919: specified length  
too long for CHAR column
```

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.

## Errors from the driver manager

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
```

*vendor* can be Microsoft or INTERSOLV. For example, an error from the Microsoft driver manager might look like this:

```
[Microsoft] [ODBC DLL] Driver does not support this function
```

To deal with this type of error, check the documentation for the driver manager.

---

## Part 4. Developing object-oriented programs

<b>Chapter 18. Writing object-oriented programs</b>	271
Example: mail-order catalog	271
Subclasses	273
Defining a class	274
CLASS-ID paragraph for defining a class	274
REPOSITORY paragraph for defining a class	275
WORKING-STORAGE SECTION for defining a class	275
Example: defining a class	276
Defining a class method	277
METHOD-ID paragraph for defining a class method	277
Overriding a method	277
INPUT-OUTPUT SECTION for defining a method	278
DATA DIVISION for defining a method	278
PROCEDURE DIVISION for defining a method	279
Coding special methods	279
Attribute methods	279
Method somInit for customizing object creation	280
Method somUninit for customizing object destruction	280
Example: defining a method	280
Defining a client program	285
REPOSITORY paragraph for defining a client	285
WORKING-STORAGE SECTION for defining a client	285
Creating and freeing instances of classes	286
Manipulating object references	286
Invoking methods	287
Example: defining a client	288
Defining a subclass	289
CLASS-ID paragraph for defining a subclass	290
REPOSITORY paragraph for defining a subclass	290
WORKING-STORAGE SECTION for defining a subclass	290
Defining a subclass method	291
METHOD-ID paragraph for defining a subclass method	291
Example: defining a subclass (with methods)	292
Defining a metaclass	301
CLASS-ID paragraph for defining a metaclass	301
REPOSITORY paragraph for defining a metaclass	302
WORKING-STORAGE SECTION for defining a metaclass	302
Defining a metaclass method	302
Invoking a metaclass constructor method	303
Changing the definition of a class or subclass	303
Changing a client program	303
Example: defining a metaclass (with methods)	304
<b>Chapter 19. System Object Model</b>	311
SOM Interface Repository	311
Accessing the SOM Interface Repository	311
Populating the SOM Interface Repository	312
SOM environment variables	312
SOM services	313
SOM methods and functions	313
Compiling and linking programs that call SOM functions	314
Class initialization	314
Changing SOM class interfaces	315
<b>Chapter 20. Using SOM IDL-based class libraries</b>	317
SOM objects	317
SOM IDL	318
Mapping IDL to COBOL	319
Using IDL operations	319
Expressing IDL attributes	320
Common IDL types	321
Examples: common IDL types	321
enum	322
interface	322
long (signed and unsigned)	322
short (signed and unsigned)	323
string	323
Complex IDL types	324
any	324
array	325
sequence	325
struct	326
union	326
Passing COBOL arguments and return values	327
Passing literal arguments	327
Passing arguments of complex types	327
Conventions for passing arguments and return values	328
Passing enumerated arguments	329
Passing string arguments	329
Example: passing string arguments	331
Example: using a SOM IDL-based class library	332
Handling errors and exceptions	334
Passing environment variables	335
Checking the exception type field	335
Handling exceptions	335
Example: checking SOM exceptions	335
Creating and initializing object instances	337
Looking at the IDL file	338
Avoiding memory leaks	339
Example: COBOL variable-length string class	340
Source code for helper routines	342
<b>Chapter 21. Wrapping or converting procedure-oriented programs</b>	343
OO view of COBOL programs	343
Wrapping procedure-oriented programs	344
Coordinating procedural code with interface actions	344
Integrating procedural code into OO systems	344

Changing procedural code . . . . .	345
Converting from procedure-oriented to OO programs. . . . .	345
Identifying objects. . . . .	346
Analyzing data flow and usage . . . . .	346
Reallocating code to objects . . . . .	347
Writing the object-oriented code . . . . .	347

---

## Chapter 18. Writing object-oriented programs

When you write an object-oriented (OO) program, you need to determine the classes and the methods that the classes need to do their work. OO programs are based on classes and methods for objects. A class is a template that defines the data structure and capabilities of an object. The data structure is commonly called instance data and the capabilities are commonly called methods. Usually, a program creates and works with multiple *object instances* of a class. Each instance has its own instance data and uses the methods defined for its class.

“Example: mail-order catalog”

### RELATED TASKS

“Defining a class” on page 274

“Defining a class method” on page 277

---

### Example: mail-order catalog

Consider a mail-order catalog business in which customers call service representatives to place orders. The service representatives are working with a user interface on the computer and creating an order. Therefore, in this situation there are two classes: `UserInterface` and `Orders`. Because there are many service representatives each processing a different customer’s order, there are multiple instances of the two classes existing simultaneously.

Once classes are determined, the next step is to determine the methods the classes need to do their work. The `Orders` class must provide the following services:

- Add items to the order
- Delete items from the order
- Compute the cost of the order
- Provide the order number to the service representative
- Write the final order for later processing

The following methods for the `Orders` class meet the above need:

#### **AddItem**

Add an item to the order

#### **DeleteItem**

Delete an item from the order

#### **ComputeCost**

Compute the cost of the order

#### **getOrderNumber**

Provide the order number

#### **WriteOrder**

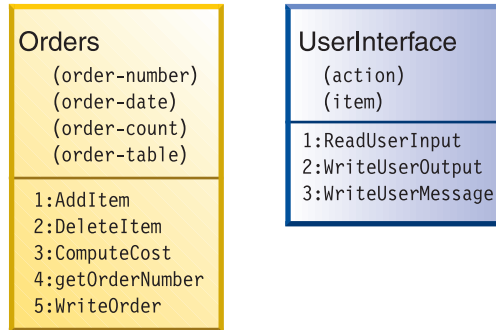
Write the final order

As you design your class and its methods, you discover the need for the class to keep some instance data. Typically, an `orders` class needs the following instance data:

- Order number

- Order date
- Number of items in the order
- Table of items ordered

Diagrams are very helpful when designing classes and their methods. The following diagrams depict the Orders and UserInterface classes.

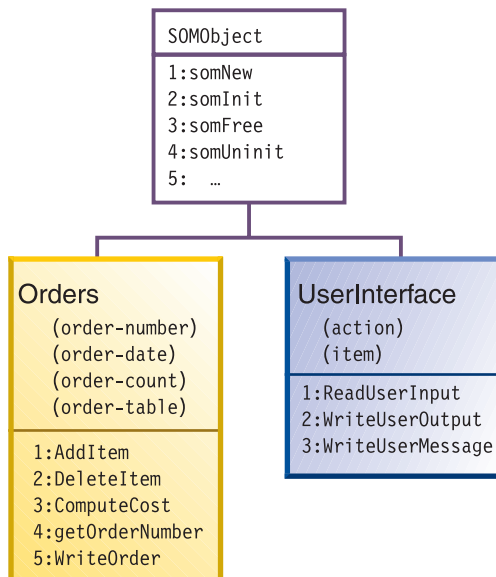


The words in parentheses are instance data and the words after the number and colon are methods.

The class structure of this object-oriented system is a tree structure. This structure shows how classes are related to each other and is known as the *inheritance hierarchy*. Orders and UserInterface are basic classes, so they inherit directly from the System Object Model (SOM) base class, SOMObject.

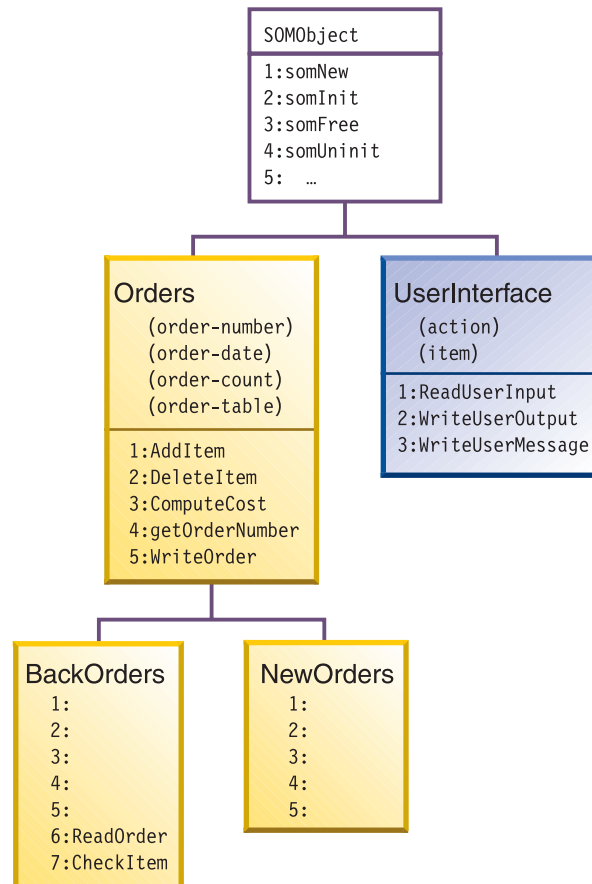
All classes in COBOL inherit directly or indirectly from SOMObject. When multiple inheritance is used, the class structure might not be a tree; it could be a graph. However, the SOMObject class will always be at the root of the tree or graph.

The complete class structure for the mail-order catalog system is shown below:



## Subclasses

In the mail-order catalog example, `Orders` is a general class. One of the first things you discover working with `Orders` is that there are two kinds of orders: new orders and back orders. Although both `NewOrders` and `BackOrders` have all the characteristics of `Orders`, `BackOrders` must also read the order from the file, and check the status of its items. It might make sense to make `NewOrders` and `BackOrders` subclasses of `Orders`, as shown below:



A number and colon with nothing after them represent a method inherited from a superclass.

### RELATED TASKS

- “Defining a class” on page 274
- “Defining a class method” on page 277
- “Defining a subclass” on page 289
- “Defining a metaclass” on page 301

---

## Defining a class

A COBOL class definition consists of four divisions (like a COBOL program), followed by an END CLASS header.

Division	Purpose	Syntax
IDENTIFICATION (required)	Name a class. Provide inheritance information for it.	"CLASS-ID paragraph for defining a class" (required) AUTHOR paragraph (optional) INSTALLATION paragraph (optional) DATE-WRITTEN paragraph (optional) DATE-COMPILED paragraph (optional)
ENVIRONMENT (required)	Describe the computing environment. Relate class names to external SOM names.	CONFIGURATION section (required) "REPOSITORY paragraph for defining a class" on page 275 (required) SOURCE-COMPUTER paragraph (optional) OBJECT-COMPUTER paragraph (optional) SPECIAL-NAMES paragraph (optional)
DATA (optional)	Describe instance data the class needs.	"WORKING-STORAGE SECTION for defining a class" on page 275 (optional)
PROCEDURE (optional)	Define methods.	"Defining a class method" on page 277

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 the INPUT-OUTPUT SECTION.

"Example: mail-order catalog" on page 271

"Example: defining a class" on page 276

### RELATED TASKS

"Defining a subclass" on page 289

"Defining a metaclass" on page 301

"Changing the definition of a class or subclass" on page 303

"Describing the computing environment" on page 7

## CLASS-ID paragraph for defining a class

You use the CLASS-ID paragraph, within the IDENTIFICATION DIVISION, to name a class and provide inheritance information for it. For example:

```
Identification Division.           Required  
Class-Id.  Orders INHERITS SOMObject.  Required
```

Use the CLASS-ID paragraph to:

- Name a class.  
In the example above, Orders is the class name.
- Specify the System Object Model (SOM) base class or user-written class from which this class inherits its characteristics.  
In the example above, INHERITS SOMObject indicates Orders inherits its basic characteristics from the base SOM class SOMObject.
- Name a metaclass.

You must specify SOMObject in the REPOSITORY paragraph in the ENVIRONMENT DIVISION. Optionally, you can specify Orders in the REPOSITORY paragraph.

## REPOSITORY paragraph for defining a class

The REPOSITORY paragraph declares to the compiler that the specified user-defined word is a class name and optionally relates the class name to an external class name in the SOM interface repository. For example:

Environment Division.	<b>Required</b>
Configuration Section.	<b>Required</b>
Repository.	<b>Required</b>
Class SOMObject is 'SOMObject'	
Class Orders is 'Orders'.	

You must specify, in the REPOSITORY paragraph, any class name that you explicitly reference in your class definition. For example:

- SOM base classes.

In the example above, CLASS SOMObject IS 'SOMObject' indicates that what you are calling SOMObject in your COBOL program is also called SOMObject in the SOM interface repository. All SOM names are mixed case, so SOMObject spelled in mixed case is required to properly handle SOM case sensitivity.

- The classes from which your class is inheriting.
- The metaclass to which your class belongs.
- Any class referenced in methods introduced by the class.

You can optionally include the name of the class that you are defining. If you do not include the name of your class, it is treated as all uppercase regardless of how you typed it in the CLASS-ID paragraph. In the above example, Orders is stored in the SOM interface repository in mixed case.

## WORKING-STORAGE SECTION for defining a class

You use the WORKING-STORAGE SECTION, within the DATA DIVISION, to describe the instance data the class needs. For example:

```
Data Division.  
Working-Storage Section.  
01 order-number PIC 9(5).  
01 order-date PIC X(8).  
01 order-count PIC 99.  
01 order-table.  
    02 order-entry OCCURS 10 TIMES.  
        03 order-item PIC X(5).
```

A class WORKING-STORAGE SECTION describes instance data that is statically allocated when the instance is created and exists until the instance is freed. By default, the data is global to all the methods that the class introduces. Instance data in a COBOL class is *private*. Therefore, no other class or subclass can reference it directly.

The syntax of the class WORKING-STORAGE SECTION is generally the same as in a program, with these exceptions:

- You cannot use the VALUE clause to initialize the data.

Class instance data is initialized by overriding the somInit method.

You can have level-88 numbers with the VALUE clause.

- You cannot use the EXTERNAL attribute.
- You can use the GLOBAL attribute, but it has no effect.

## Example: defining a class

The class definition (except the method definitions) for the Orders class:

```
IDENTIFICATION DIVISION.  
*  
* Orders is the name of the class  
* Orders inherits from SOMObject (SOM base class)  
*  
CLASS-ID. Orders INHERITS SOMObject.  
ENVIRONMENT DIVISION.  
CONFIGURATION SECTION.  
REPOSITORY.  
*  
* SOMObject is known as SOMObject in SOM repository  
  CLASS SOMObject IS 'SOMObject'  
*  
* Orders is known as Orders in SOM repository  
  CLASS Orders IS 'Orders'.  
DATA DIVISION.  
WORKING-STORAGE SECTION.  
*  
* Instance data for Orders class  
*  
01 order-number PIC 9(5).  
01 order-date   PIC X(8).  
01 order-count  PIC 99.  
01 order-table.  
    02 order-entry OCCURS 10 TIMES.  
        03 order-item PIC X(5).  
PROCEDURE DIVISION.  
*  
* method definitions in here  
*  
END CLASS Orders.
```

The class definition (except the method definitions) for the UserInterface class:

```
IDENTIFICATION DIVISION.  
*  
* UserInterface is the name of the class  
* UserInterface inherits from SOMObject (SOM base class)  
*  
CLASS-ID. UserInterface INHERITS SOMObject.  
ENVIRONMENT DIVISION.  
CONFIGURATION SECTION.  
REPOSITORY.  
*  
* SOMObject is known as SOMObject in SOM repository  
  CLASS SOMObject IS 'SOMObject'  
*  
* UserInterface is known as UserInterface in SOM repository  
  CLASS UserInterface IS 'UserInterface'.  
DATA DIVISION.  
WORKING-STORAGE SECTION.  
*  
* Instance data for UserInterface class  
*  
01 uif-action PIC X(10).  
    88 uif-add VALUE 'AddItem'.  
    88 uif-delete VALUE 'DeleteItem'.  
    88 uif-quit VALUE 'Quit'.  
01 uif-item PIC X(5).  
PROCEDURE DIVISION.  
*  
* method definitions in here  
*  
END CLASS UserInterface.
```

---

## Defining a class method

You can define a COBOL method only inside the PROCEDURE DIVISION of a class definition. You must make each method name within a class unique. A COBOL method definition consists of four divisions (like a COBOL program), followed by an END METHOD header.

Division	Purpose	Syntax
IDENTIFICATION (required)	Name a method. Say whether it overrides a method from a superclass.	"METHOD-ID paragraph for defining a class method" (required) AUTHOR paragraph (optional) INSTALLATION paragraph (optional) DATE-WRITTEN paragraph (optional) DATE-COMPILED paragraph (optional)
ENVIRONMENT (optional)	Relate the method files to the external file names known by the operating system.	"INPUT-OUTPUT SECTION for defining a method" on page 278 (optional)
DATA (optional)	Define external files. Allocate a copy of the data.	"DATA DIVISION for defining a method" on page 278 (optional)
PROCEDURE (optional)	Code the executable statements to complete the service provided by the method.	"PROCEDURE DIVISION for defining a method" on page 279 (optional)

"Example: mail-order catalog" on page 271

"Example: defining a method" on page 280

### RELATED TASKS

"Defining a subclass method" on page 291

"Defining a metaclass method" on page 302

"Overriding a method"

"Invoking methods" on page 287

"Coding special methods" on page 279

## METHOD-ID paragraph for defining a class method

Use the METHOD-ID paragraph to name the method. For example:

```
Identification Division.  
Method-Id. WriteOrder.
```

In this example, WriteOrder is the method name. Other methods or programs use this name to invoke the method.

## Overriding a method

Occasionally, a class defines a method whose name exists in a superclass. In this case, you need to override the superclass method with the OVERRIDE clause on the METHOD-ID. System Object Model (SOM) provides two methods designed to be overridden. These SOM methods allow you to customize the creation and disposal of an object instance. For example, you can initialize instance data when an instance is created, or save instance data when an instance is freed. The methods are called somInit and somUninit respectively.

To override somInit, code the IDENTIFICATION DIVISION as follows:

Identification Division.	<b>Required</b>
Method-Id. "somInit" Override.	<b>Required</b>

## INPUT-OUTPUT SECTION for defining a method

The method ENVIRONMENT DIVISION has only one section, the INPUT-OUTPUT SECTION. For example:

```
Environment Division.  
Input-Output Section.  
File-Control.  
    Select order-file Assign OrdFile.
```

The INPUT-OUTPUT SECTION relates your method files to the external file names known by the operating system. The syntax for a method INPUT-OUTPUT SECTION is the same as for a program INPUT-OUTPUT SECTION.

### RELATED TASKS

“Describing the computing environment” on page 7

## DATA DIVISION for defining a method

A method DATA DIVISION consists of any of four sections:

### FILE SECTION

The same as a program FILE SECTION except that a method FILE SECTION can define only EXTERNAL files.

### LOCAL-STORAGE SECTION

Allocates a separate copy of the data defined in the method LOCAL-STORAGE SECTION for each invocation of the method and frees it on return from the method.

If you specify the VALUE clause, the data item is initialized to the value on every invocation of the method.

The method LOCAL-STORAGE SECTION is similar to a program LOCAL-STORAGE SECTION, except that the GLOBAL attribute has no effect.

### WORKING-STORAGE SECTION

Allocates a single copy of the data defined in the method WORKING-STORAGE SECTION when the run unit begins and persists in its last-used state until the run unit terminates. Uses the same single copy of the WORKING-STORAGE data whenever the method is invoked, regardless of the invoking object.

If you specify the VALUE clause, the data item is initialized to the value on the first invocation of the method. You may specify the EXTERNAL clause for method WORKING-STORAGE data items.

A method WORKING-STORAGE SECTION is similar to a program WORKING-STORAGE SECTION except that the GLOBAL attribute has no effect.

### LINKAGE SECTION

The same as a program LINKAGE SECTION except that the GLOBAL attribute has no effect.

If you define the same data item in both the class DATA DIVISION and the method DATA DIVISION, a reference in the method to the data name refers to the data item in the method DATA DIVISION. The method DATA DIVISION takes precedence.

### RELATED TASKS

“Describing the data” on page 11

“Sharing data by using the EXTERNAL clause” on page 383

## PROCEDURE DIVISION for defining a method

In the PROCEDURE DIVISION of a method, you code the executable statements to implement the service that the method is expected to provide.

The EXIT METHOD statement returns control to the invoking program or method. GOBACK has the same effect as EXIT METHOD. If you specify the RETURNING clause when the method is invoked, the EXIT METHOD or GOBACK returns the value of the data item to the invoking program or method. You may specify STOP RUN in a method; however, it terminates the run unit.

An implicit EXIT METHOD is generated as the last statement of every method PROCEDURE DIVISION.

You can code any COBOL statements in a method except the following:

- ENTRY
- EXIT PROGRAM
- The following obsolete elements of ANSI COBOL-85:
  - ALTER
  - GOTO without a specified procedure name
  - SEGMENT-LIMIT
  - USE FOR DEBUGGING

You must properly terminate a method definition with an END METHOD statement. For example, the following statement marks the end of the WriteOrder method:  
End Method WriteOrder.

## Coding special methods

There are several special methods that you might want to code when defining a class, as discussed in the sections below.

### Attribute methods

Instance variables in COBOL are all *private* in the sense that the class fully encapsulates them, and only the methods that are introduced by the class that defines the instance variables 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 other object-oriented languages, nor the concept of a class *attribute*, as defined by SOM and CORBA. (A CORBA attribute behaves like an instance variable that has get and set methods to access and modify the value of the instance variable from outside the class definition.) You can provide this capability by coding getX and setX methods for any instance variables X for which direct access from outside the class is required.

The recommended naming convention for these methods is either getX and setX or perhaps get\_X and set\_X. Direct specification of attribute method names as defined by the CORBA C language binding (such as \_get\_X) is not recommended because such names are not valid in IDL. If you use such method names and specify the COBOL IDLGEN compiler option, the SOM compiler cannot compile the IDL file.

For example, the following method passes the order number to any program that invokes getOrderNumber.

```

Identification Division.
Method-Id. 'getOrderNumber'.
Data Division.
Linkage Section.
01 ord-num PIC 9(5).
Procedure Division returning ord-num.
    Move order-number To ord-num.
    Exit Method.
End Method 'getOrderNumber'.

```

### Method somInit for customizing object creation

Creating an object instance automatically invokes the somInit method. The default somInit in SOM does nothing. However, you can override it to customize the way that your object instances are created. A typical use would be to initialize your instance variables. For example:

```

Identification Division.
Method-Id. "somInit" Override.
Procedure Division.
    Move Function Current-Date(1:8) To order-date.
    Move 0 To order-count.
    Initialize order-table.
    Exit Method.
End Method "somInit".

```

### Method somUninit for customizing object destruction

Freeing an object automatically invokes the somUninit method. The default somUninit in SOM does nothing. However, you can override it to customize the way that your object instance is freed, perhaps to save the instance data or free any additional storage that your instance points to. For example:

```

Identification Division.
Method-Id. "somUninit" Override.
Data Division.
Local-Storage Section.
01 sub Pic 99.
Procedure Division.
    Display order-date.
    Perform varying sub from 1 by 1 until sub > order-count
        Display order-table (sub)
    End-Perform.
    Exit Method.
End Method "somUninit".

```

#### RELATED CONCEPTS

“How logic is divided in the PROCEDURE DIVISION” on page 15

#### RELATED TASKS

“Processing the data” on page 14

## Example: defining a method

The class definition for the Orders class, including the method definitions:

```

IDENTIFICATION DIVISION.
CLASS-ID. Orders INHERITS SOMObject.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
* Declare classes used in class definition
REPOSITORY.
    CLASS SOMObject IS 'SOMObject'
    CLASS Orders IS 'Orders'.

DATA DIVISION.
* Define instance data

```

```

WORKING-STORAGE SECTION.
01 order-number PIC 9(5).
01 order-date   PIC X(8).
01 order-count  PIC 99.
01 order-table.
    02 order-entry OCCURS 10 TIMES.
        03 order-item PIC X(5).
PROCEDURE DIVISION.

* Method to initialize instance data
* - this overrides the default somInit method
IDENTIFICATION DIVISION.
METHOD-ID. 'somInit' OVERRIDE.

PROCEDURE DIVISION.
    MOVE FUNCTION CURRENT-DATE(1:8) TO order-date.
    COMPUTE order-number = FUNCTION RANDOM ( 99999 ).
    MOVE 0 TO order-count.
    INITIALIZE order-table.
    EXIT METHOD.
END METHOD 'somInit'.

* Method to add an item to an order
IDENTIFICATION DIVISION.
METHOD-ID. AddItem.

DATA DIVISION.
* Use LOCAL-STORAGE for items that should be allocated
* and initialized for each invocation of the method
LOCAL-STORAGE SECTION.
77 sub PIC 99.
01 found-flag PIC 9 VALUE 1.
   88 found VALUE 0.
LINKAGE SECTION.
01 in-item PIC X(5).
01 add-flag PIC 9.

PROCEDURE DIVISION USING in-item
    RETURNING add-flag.
    MOVE 1 TO add-flag.
    PERFORM VARYING sub FROM 1 BY 1
        UNTIL (sub > 10) OR (found)
        IF order-item (sub) = SPACES
            MOVE in-item TO order-item (sub)
            ADD 1 TO order-count
            MOVE 0 TO add-flag
            SET found TO TRUE
        END-IF
    END-PERFORM.
    EXIT METHOD.
END METHOD AddItem.

* Method to delete an item from an order
IDENTIFICATION DIVISION.
METHOD-ID. DeleteItem.

DATA DIVISION.
* Use LOCAL-STORAGE for items that should be allocated
* and initialized for each invocation of the method
LOCAL-STORAGE SECTION.
77 sub PIC 99.
01 found-flag PIC 9 VALUE 1.
   88 found VALUE 0.
LINKAGE SECTION.
01 out-item PIC X(5).
01 delete-flag PIC 9.

```

```

PROCEDURE DIVISION USING out-item
    RETURNING delete-flag.
    MOVE 1 TO delete-flag.
    PERFORM VARYING sub FROM 1 BY 1
        UNTIL (sub > 10) OR (found)
        IF order-item (sub) = out-item
            MOVE SPACES TO order-item (sub)
            SUBTRACT 1 FROM order-count
            MOVE 0 TO delete-flag
            SET found TO TRUE
        END-IF
    END-PERFORM.
    EXIT METHOD.
END METHOD DeleteItem.

* Method to compute the total cost of an order
IDENTIFICATION DIVISION.
METHOD-ID. ComputeCost.

DATA DIVISION.
* Use LOCAL-STORAGE for items that should be allocated
* and initialized for each invocation of the method
LOCAL-STORAGE SECTION.
77 sub PIC 99.
77 cost PIC 9(5)V99.
LINKAGE SECTION.
01 total-cost PIC 9(7)V99.

PROCEDURE DIVISION USING total-cost.
    MOVE 0 TO total-cost.
    PERFORM VARYING sub FROM 1 BY 1
        UNTIL sub > order-count
    * Call a subroutine
    * NOTE: The subroutine code is not
    * included in this example.
        CALL 'InventoryGetCost'
            USING order-item (sub) cost
            ADD cost TO total-cost
        END-PERFORM.
    EXIT METHOD.
END METHOD ComputeCost.

* Method to return the order number
IDENTIFICATION DIVISION.
METHOD-ID. 'getOrderNumber'.

DATA DIVISION.
LINKAGE SECTION.
01 ord-num PIC 9(5).

PROCEDURE DIVISION RETURNING ord-num.
    MOVE order-number TO ord-num.
    EXIT METHOD.
END METHOD 'getOrderNumber'.

* Method to write completed order to file
IDENTIFICATION DIVISION.
METHOD-ID. WriteOrder.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT order-file ASSIGN OrdrFile.

DATA DIVISION.
FILE SECTION.
* Methods support only EXTERNAL files

```

```

FD order-file EXTERNAL.
01 order-record PIC X(80).
* Use LOCAL-STORAGE for items that should be allocated
* and initialized for each invocation of the method
LOCAL-STORAGE SECTION.
01 print-line.
02 print-order-number PIC 9(5).
02 print-order-date PIC X(8).
02 print-order-count PIC 99.
02 print-order-table.
03 print-order-entry OCCURS 10 TIMES.
04 print-order-item PIC X(5).

PROCEDURE DIVISION.
OPEN OUTPUT order-file.
MOVE order-number TO print-order-number.
MOVE order-date TO print-order-date.
MOVE order-table TO print-order-table.
MOVE order-count TO print-order-count.
WRITE order-record FROM print-line.
CLOSE order-file.
EXIT METHOD.
END METHOD WriteOrder.

END CLASS Orders.

```

The class definition for the UserInterface class, including the method definitions:

```

IDENTIFICATION DIVISION.
CLASS-ID. UserInterface INHERITS SOMObject.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
* Declare classes used in class definition
REPOSITORY.
CLASS SOMObject IS 'SOMObject'
CLASS UserInterface IS 'UserInterface'.

DATA DIVISION.
* Define instance data
WORKING-STORAGE SECTION.
01 uif-action PIC X(10).
88 uif-add VALUE 'AddItem'.
88 uif-delete VALUE 'DeleteItem'.
88 uif-quit VALUE 'Quit'.
01 uif-item PIC X(5).

PROCEDURE DIVISION.

* Method to get input from customer - action and item
IDENTIFICATION DIVISION.
METHOD-ID. ReadUserInput.

DATA DIVISION.
LINKAGE SECTION.
01 action PIC X(10).
01 item PIC X(5).

PROCEDURE DIVISION USING item action.
DISPLAY 'Enter the action: add, delete, quit'.
ACCEPT action FROM SYSIN.
MOVE FUNCTION UPPER-CASE (action) TO action.
EVALUATE TRUE
WHEN action = 'ADD'
SET uif-add TO TRUE
PERFORM Get-Item
WHEN action = 'DELETE'

```

```

        SET uif-delete TO TRUE
        PERFORM Get-Item
        WHEN action = 'QUIT'
            SET uif-quit TO TRUE
        END-EVALUATE.
        MOVE uif-action TO action.
        EXIT METHOD.

Get-Item.
    DISPLAY 'Enter the item'.
    ACCEPT item FROM SYSIN.
    MOVE item TO uif-item.

END METHOD ReadUserInput.

* Method to inform customer how action was completed
IDENTIFICATION DIVISION.
METHOD-ID. WriteUserMessage.

DATA DIVISION.
LINKAGE SECTION.
01 flag PIC 9.

PROCEDURE DIVISION USING flag.
    IF flag = 0
        DISPLAY uif-action
            ' successfully completed on '
            uif-item
    ELSE
        DISPLAY uif-action
            ' unsuccessfully completed on '
            uif-item
    END-IF.
    EXIT METHOD.

END METHOD WriteUserMessage.

* Method to display final order information
IDENTIFICATION DIVISION.
METHOD-ID. WriteUserOutput.

DATA DIVISION.
LOCAL-STORAGE SECTION.
77 formatted-cost PIC $Z,ZZZ,ZZ9.99.
LINKAGE SECTION.
01 total-cost PIC 9(7)V99.
01 order-number PIC 9(5).

PROCEDURE DIVISION USING total-cost order-number.
    MOVE total-cost TO formatted-cost.
    DISPLAY 'Your order costs ' formatted-cost.
    DISPLAY 'Your order number is ' order-number.
    EXIT METHOD.

END METHOD WriteUserOutput.

END CLASS UserInterface.

```

---

## Defining a client program

A program or method that requests services from a class via its methods is called a *client* of the class. A client program consists of the usual four divisions:

Division	Purpose	Syntax
IDENTIFICATION (required)	Name the client.	Coded as usual
ENVIRONMENT (required)	Describe the computer environment. Relate class names to external SOM names.	CONFIGURATION section (required) "REPOSITORY paragraph for defining a client" (required)
DATA (optional)	Describe the data the client needs (object references).	"WORKING-STORAGE SECTION for defining a client" (optional)
PROCEDURE (optional)	Create and free instances of classes. Manipulate object reference data items. Invoke methods.	Code using somNew, somFree, IF statements, and INVOKE.

A method can request services from another method. Therefore a method can be a client and use the statements discussed in this section.

"Example: mail-order catalog" on page 271

"Example: defining a client" on page 288

### RELATED TASKS

"Creating and freeing instances of classes" on page 286

"Manipulating object references" on page 286

"Invoking methods" on page 287

"Changing a client program" on page 303

## REPOSITORY paragraph for defining a client

The REPOSITORY paragraph in the CONFIGURATION SECTION of the ENVIRONMENT DIVISION declares to the compiler that the specified user-defined word is a class name and optionally relates the class name to an external class name in the SOM interface repository. For example:

```
Environment Division.      Required
Configuration Section.    Required
```

```
Repository.               Required
  Client UserInterface is 'UserInterface'
  Client Orders is 'Orders'.
```

You must specify any class name you explicitly reference in your program in the REPOSITORY paragraph. In the example above, Orders and UserInterface are the only two classes this program references.

## WORKING-STORAGE SECTION for defining a client

You can use any of the sections of the DATA DIVISION to describe the data that the client needs. For example:

```
Data Division.  
Working-Storage Section.  
01 orderObj Usage Object Reference Orders.  
01 userObj Usage Object Reference UserInterface.  
01 univObj Usage Object Reference.
```

Because the client is using classes, it needs one or more special data items called *object references*. Object references are handles to instances of classes that the program creates. All requests to a method are handled through object references to instances of the class that defined the method.

In the above example, the phrase USAGE OBJECT REFERENCE indicates that the data item is used as a handle for an object instance. The example defines three object references. The first two, orderObj and userObj, are *typed* object references because a class name appears after the OBJECT REFERENCE phrase. Therefore, orderObj can be used to reference only instances of the Orders class or one of its subclasses. Likewise, userObj can be used to reference only instances of the UserInterface class or one of its subclasses. The other object reference, univObj, does not have a class name after its OBJECT REFERENCE phrase. It is a *universal* object reference and can reference instances of any class.

**Remember:** You must define, in the REPOSITORY paragraph of the CONFIGURATION SECTION, class names that you use on the OBJECT REFERENCE phrase.

## Creating and freeing instances of classes

Before you can use the methods in a class, you must create an instance of the class. SOM provides a method, somNew, to create an instance of a class. The following example creates an instance of the Orders class and assigns its handle to the object reference orderObj.

```
Invoke Orders 'somNew' Returning orderObj.
```

When somNew executes, it automatically invokes somInit, another SOM method, which you can override to initialize your instance data.

**Remember:** You must define the class name, in this case Orders, in the REPOSITORY paragraph of the CONFIGURATION SECTION. You must define the object reference, in this case orderObj, as USAGE OBJECT REFERENCE in the DATA DIVISION.

When you finish with an instance of a class, you should free it. Again, SOM provides a method, somFree, to free the instance. The following example frees the instance of orderObj, after which orderObj has an undefined value.

```
Invoke orderObj 'somFree'.
```

When somFree executes, it automatically invokes somUninit, another SOM method that you can override to customize the way that your instance is freed.

## Manipulating object references

You can compare object references in conditional statements. The following examples are all valid uses of object references in an IF statement:

```
If orderObj = Null . . .  
If orderObj = Nulls . . .  
If orderObj = univObj . . .
```

The first and second IF statements check whether orderObj is a null object reference (refers to no instance). The third IF statement checks whether orderObj and univObj refer to the same instance.

In a method there is a fourth form of conditional statement for comparing object references:

```
If orderObj = Self . . .
```

This IF statement checks whether the instance on which the method was invoked, SELF, refers to the same instance as orderObj.

You may need to make an object reference null or to make one object reference refer to the same instance as another object reference. The SET statement takes care of these situations:

```
Set orderObj To Null.  
Set univObj To orderObj.
```

In the first SET statement, orderObj is set to NULL.

In the second SET statement, univObj is made to refer to the instance to which orderObj refers. In this syntax, if the receiver (univObj) is a universal object reference, the sender (orderObj) can be either a universal or typed object reference. However, if the receiver is a typed object reference, the sender must also be a typed object reference and typed to the same class or a subclass.

In a method you can use a third form of SET object reference:

```
Set orderObj To Self.
```

This SET statement makes the receiver (orderObj) refer to the same instance as that on which the method was invoked, SELF.

## Invoking methods

To use the service defined by a method, you must invoke the method with the INVOKE statement. For example:

```
Invoke Orders 'somNew' Returning orderObj.  
Invoke orderObj 'AddItem' Using item Returning flag.
```

In the first INVOKE statement, a class name is used to create a new instance whose handle is returned in the object reference orderObj. You must define the class name, Orders, in the REPOSITORY paragraph of the CONFIGURATION SECTION. You must define the object reference, orderObj, as either a universal object reference or as an object reference of type class Orders.

In the second INVOKE statement, an object reference, orderObj, is used to invoke the method AddItem. The general syntax of this form of INVOKE is one of the following:

```
Invoke objref 'literal-name'.  
Invoke objref identifier-name.
```

In both cases you must define the invoked method in the class for which the object reference (objref) is an instance. If you use the identifier-name form of the method, the object reference (objref) must be a universal object reference.

Conformance between the invoked method and the object reference is checked at compile time if the following three items are all true:

1. objref is a typed object reference.
2. The literal form of the method name is used in the INVOKE statement.
3. The TYPECHK compile option is specified.

Otherwise, conformance requirements are checked at run time. Run-time checking, however, is not as thorough as compile-time checking.

INVOKE has the optional scope terminator END-INVOKE. The USING and RETURNING phrases on the INVOKE work the same as they do on the CALL statement. Also, INVOKE has the optional ON EXCEPTION and NOT ON EXCEPTION phrases like the CALL statement.

The RETURN-CODE special register is not set by a method invocation.

#### RELATED REFERENCES

INVOKE statement (*IBM COBOL Language Reference*)

### Example: defining a client

The following shows a possible client program for the mail-order catalog using the Orders and UserInterface classes:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. 'PhoneOrders'.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
*
* Declare the classes used in the program
REPOSITORY.
    CLASS Orders IS 'Orders'
    CLASS UserInterface IS 'UserInterface'.

DATA DIVISION.
WORKING-STORAGE SECTION.
*
* Declare the object references used in the program
77 orderObj  USAGE OBJECT REFERENCE Orders.
77 userObj   USAGE OBJECT REFERENCE UserInterface.
*
* Declare other data items used in the program
77 order-number PIC 9(5).
77 total-cost PIC 9(7)V99.
77 item PIC X(5).
77 action PIC X(10).
77 flag PIC 9.

PROCEDURE DIVISION.
*
* Create an instance of the UserInterface class - userObj
    INVOKE UserInterface 'somNew' RETURNING userObj.
*
* Create an instance of the Orders class - orderObj
    INVOKE Orders 'somNew' RETURNING orderObj.
*
* Read customer input - action and item
    INVOKE userObj 'ReadUserInput' USING item action.
*
* Begin customer driven loop based on action
    PERFORM UNTIL action = 'Quit'
*
* Do appropriate action
        IF action (1:3) = 'Add'
            INVOKE orderObj 'AddItem' USING item
                RETURNING flag
        ELSE
            INVOKE orderObj 'DeleteItem' USING item
                RETURNING flag
```

```

        END-IF
*
* Display result of action
    INVOKE userObj 'WriteUserMessage' USING flag
*
* Read customer input - action and item
    INVOKE userObj 'ReadUserInput' USING item action
    END-PERFORM.
* End customer driven loop based on action
*

*
* Calculate the total cost of the order
    INVOKE orderObj 'ComputeCost' USING total-cost.
*
* Determine the order number
    INVOKE orderObj 'getOrderNumber'
        RETURNING order-number.
*
* Display information about the order
    INVOKE userObj 'WriteUserOutput'
        USING total-cost order-number.

*
* Write the order to a file
    INVOKE orderObj 'WriteOrder'.

*
* Free the object instances - orderObj and userObj
    INVOKE orderObj 'somFree'.
    INVOKE userObj 'somFree'.

    STOP RUN.
END PROGRAM 'PhoneOrders'.

```

---

## Defining a subclass

You can make a class (called a subclass or child class) a specialization of another class (called a superclass or parent class). The subclass is related to its superclass by an *is-a* relationship. For example, “Subclass S is a type of superclass P.”

Using subclasses has several advantages:

- Reuse of code. Through inheritance, a subclass can reuse methods that already exist in another class.
- More specific class. In a subclass you can add new methods to handle specific instances that the superclass does not handle.
- Change in action. You can override a method that a subclass inherits from its superclass. Overriding can vary from a few minor changes in how the method works to a complete overhaul of what the method does.

Division	Purpose	Syntax
IDENTIFICATION (required)	Name a subclass. Provide inheritance information for it.	“CLASS-ID paragraph for defining a subclass” on page 290 (required)
ENVIRONMENT (required)	Relate the subclass and subclass names to external SOM names.	CONFIGURATION SECTION (required) REPOSITORY paragraph for defining a subclass (required)
DATA (optional)	Describe instance data the subclass needs.	“WORKING-STORAGE SECTION for defining a subclass” on page 290 (optional)

Division	Purpose	Syntax
PROCEDURE (optional)	Define methods.	Method definitions: “Defining a subclass method” on page 291

You must properly terminate a subclass definition with an END CLASS statement.

“Example: mail-order catalog” on page 271

“Example: defining a subclass (with methods)” on page 292

## CLASS-ID paragraph for defining a subclass

The CLASS-ID paragraph in the IDENTIFICATION DIVISION names the subclass and indicates from what superclass or superclasses the subclass inherits. For example:

```
Identification Division.           Required
Class-Id.  BackOrders INHERITS Orders. Required
```

Use the CLASS-ID paragraph to name the subclass and indicate from what superclass (or superclasses) the subclass inherits.

In the above example, BackOrders is the class name. It inherits all the methods from Orders. Also, it can access Orders instance data if Orders provides methods to get and set its instance data.

You must specify the names of the superclasses in the REPOSITORY paragraph in the CONFIGURATION SECTION of the ENVIRONMENT DIVISION. Optionally, you can specify BackOrders in the REPOSITORY paragraph.

## REPOSITORY paragraph for defining a subclass

The REPOSITORY paragraph relates your subclass and class names to the subclass and class names in the SOM interface repository. For example:

```
Environment Division.           Required
Configuration Section.         Required
Repository.                    Required
    Class BackOrders is 'BackOrders'
    Class Orders is 'Orders'.
```

Use the REPOSITORY paragraph to specify:

- The classes from which your subclass is inheriting
- The metaclass to which your subclass belongs
- Any class referenced in methods introduced by the subclass

Optionally, you can include the name of the subclass you are defining. If you do not include the name of your subclass, it is treated as all uppercase, regardless of how you typed it in the CLASS-ID paragraph. In the above example, BackOrders is stored in the SOM interface repository in mixed case.

## WORKING-STORAGE SECTION for defining a subclass

You use the WORKING-STORAGE SECTION of the subclass DATA DIVISION to describe any extra instance data the subclass needs. For example:

```
Data Division.
Working-Storage Section.
01 order-status PIC X(3).
```

A subclass WORKING-STORAGE SECTION describes instance data that is statically allocated when the instance is created and exists until the instance is freed. By default, the data is global to all the methods that the subclass introduces. Instance data in a COBOL subclass is *private*. No other class or subclass can reference it directly.

## Defining a subclass method

A subclass inherits the methods from its superclass. Using the OVERRIDE clause on the METHOD-ID, you can change, or override, one or more methods that a subclass inherits from its superclass. Also, you can add new methods that a subclass needs.

In COBOL, instance data is private, and so the superclass must provide methods to allow the subclass to access superclass instance data, if necessary. Using the methods that the superclass provides, a subclass can retrieve values from, or store values in, the superclass instance data.

COBOL allows multiple inheritance, inheriting from more than one superclass. If a conflict in method names occurs between two superclasses due to multiple inheritance, the conflict is resolved according to the System Object Model (SOM) rules discussed in the related reference below.

Division	Purpose	Syntax
IDENTIFICATION (optional)	Name a method. Give other identifying information.	"METHOD-ID paragraph for defining a subclass method" (optional)
ENVIRONMENT (optional)	Same as class method	Same as for a class method
DATA (optional)	Same as class method	Same as for a class method
PROCEDURE (optional)	Same as class method	Same as for a class method

You must properly terminate a subclass method definition with an END METHOD statement.

"Example: mail-order catalog" on page 271

"Example: defining a subclass (with methods)" on page 292

### RELATED REFERENCES

Multiple inheritance (*SOMObjects Programmer's Guide*)

## METHOD-ID paragraph for defining a subclass method

Use the METHOD-ID paragraph to name the method. Other methods or programs use this name to invoke the method. For example:

```
Identification Division.  
Method-ID.  ReadOrder.
```

If the subclass defines a method whose name exists in a superclass, you must specify the OVERRIDE clause on the METHOD-ID. For example:

```
Identification Division.  
Method-Id.  AddItem Override.
```

When an object reference that is a handle to the BackOrders subclass invokes AddItem, this method is invoked rather than the method in the superclass Orders.

In a subclass method, you can invoke an overridden method of a superclass by using the following form of the INVOKE statement:

```
Invoke Super 'AddItem'.
```

This example invokes the method AddItem that is defined in the superclass rather than the method AddItem that is defined in the subclass.

In the case of multiple inheritance, a subclass may inherit several methods with the same name from different parents. To specify precisely which method from which parent is invoked, use the following form of the INVOKE statement:

```
Invoke Class-A of Super 'AddItem'.
```

This example invokes the method AddItem that is defined in the superclass Class-A rather than the method AddItem that is defined in any other superclass or in the subclass.

## Example: defining a subclass (with methods)

The new class and method definitions for the UserInterface class:

```
IDENTIFICATION DIVISION.
CLASS-ID. UserInterface INHERITS SOMObject.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
* Declare classes used in class definition
REPOSITORY.
    CLASS SOMObject IS 'SOMObject'
    CLASS UserInterface IS 'UserInterface'.

DATA DIVISION.
* Define instance data
WORKING-STORAGE SECTION.
01 uif-action PIC X(10).
    88 uif-add    VALUE 'AddItem'.
    88 uif-delete VALUE 'DeleteItem'.
    88 uif-quit   VALUE 'Quit'.
01 uif-item     PIC X(5).
PROCEDURE DIVISION.

* Method to read customer input - request
IDENTIFICATION DIVISION.
METHOD-ID. ReadUserRequest.

DATA DIVISION.
LINKAGE SECTION.
01 request PIC X(6).

PROCEDURE DIVISION USING request.
    DISPLAY 'Enter the request: new, status'.
    ACCEPT request FROM SYSIN.
    MOVE FUNCTION UPPER-CASE (request) TO request.
    EXIT METHOD.
END METHOD ReadUserRequest.

* Method to read customer input for new request - action and item
IDENTIFICATION DIVISION.
METHOD-ID. ReadUserInput1.

DATA DIVISION.
LINKAGE SECTION.
01 action PIC X(10).
01 item   PIC X(5).
```

```

PROCEDURE DIVISION USING item action.
    DISPLAY 'Enter the action: add, delete, quit'.
    ACCEPT action FROM SYSIN.
    MOVE FUNCTION UPPER-CASE (action) TO action.
    EVALUATE TRUE
        WHEN action = 'ADD'
            SET uif-add TO TRUE
            PERFORM Get-Item
        WHEN action = 'DELETE'
            SET uif-delete TO TRUE
            PERFORM Get-Item
        WHEN action = 'QUIT'
            SET uif-quit TO TRUE
    END-EVALUATE.
    MOVE uif-action TO action.
    EXIT METHOD.

Get-Item.
    DISPLAY 'Enter the item'.
    ACCEPT item FROM SYSIN.
    MOVE item TO uif-item.
END METHOD ReadUserInput1.

* Method to read customer input for status request - order number
IDENTIFICATION DIVISION.
METHOD-ID. ReadUserInput2.

DATA DIVISION.
LINKAGE SECTION.
01 order-numb PIC 9(5).

PROCEDURE DIVISION USING order-numb.
    DISPLAY 'Enter the order number'.
    ACCEPT order-numb FROM SYSIN.
    EXIT METHOD.
END METHOD ReadUserInput2.

* Method to inform customer how action was completed
IDENTIFICATION DIVISION.
METHOD-ID. WriteUserMessage.

DATA DIVISION.
LINKAGE SECTION.
01 flag PIC 9.

PROCEDURE DIVISION USING flag.
    IF flag = 0
        DISPLAY uif-action
            ' successfully completed on '
            uif-item
    ELSE
        DISPLAY uif-action
            ' unsuccessfully completed on '
            uif-item
    END-IF.
    EXIT METHOD.
END METHOD WriteUserMessage.

* Method to display order information
IDENTIFICATION DIVISION.
METHOD-ID. WriteUserOutput.
DATA DIVISION.
LOCAL-STORAGE SECTION.
77 formatted-cost PIC $Z,ZZZ,ZZ9.99.
LINKAGE SECTION.
01 total-cost PIC 9(7)V99.
01 order-number PIC 9(5).

```

```

PROCEDURE DIVISION USING total-cost order-number.
    MOVE total-cost TO formatted-cost.
    DISPLAY 'Your order costs ' formatted-cost.
    DISPLAY 'Your order number is ' order-number.
    EXIT METHOD.
END METHOD WriteUserOutput.

* Method to display out of stock items
IDENTIFICATION DIVISION.
METHOD-ID. WriteUserStatus.

DATA DIVISION.
LOCAL-STORAGE SECTION.
77 sub PIC 99.
LINKAGE SECTION.
01 out-table.
    02 out-entry OCCURS 10 TIMES.
        03 out-item PIC X(5).
01 out-count PIC 99.

PROCEDURE DIVISION USING out-table out-count.
    IF out-count > 0
        PERFORM VARYING sub FROM 1 BY 1
            UNTIL sub > out-count
                DISPLAY 'Out of stock '
                    out-item (sub)
            END-PERFORM
    END-IF.
    EXIT METHOD.
END METHOD WriteUserStatus.

END CLASS UserInterface.

```

The new class and method definitions for the Orders class:

```

IDENTIFICATION DIVISION.
CLASS-ID. Orders INHERITS SOMObject.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
* Declare classes used in program
REPOSITORY.
    CLASS SOMObject IS 'SOMObject'
    CLASS Orders IS 'Orders'.

DATA DIVISION.
* Define instance data
WORKING-STORAGE SECTION.
01 order-number PIC 9(5).
01 order-date PIC X(8).
01 order-count PIC 99.
01 order-table.
    02 order-entry OCCURS 10 TIMES.
        03 order-item PIC X(5).
PROCEDURE DIVISION.

* Method to initialize instance data
* - this overrides the default somInit method
IDENTIFICATION DIVISION.
METHOD-ID. 'somInit' OVERRIDE.

PROCEDURE DIVISION.
    MOVE FUNCTION CURRENT-DATE(1:8) TO order-date.
    COMPUTE order-number = FUNCTION RANDOM ( 99999 ).
    MOVE 0 TO order-count.
    INITIALIZE order-table.
    EXIT METHOD.

```

```

END METHOD 'somInit'.

* Method to set instance data read by subclass
IDENTIFICATION DIVISION.
METHOD-ID. 'setInstanceData'.

DATA DIVISION.
LINKAGE SECTION.
01 in-order.
    02 in-order-number PIC 9(5).
    02 in-order-date   PIC X(8).
    02 in-order-count  PIC 99.
    02 in-order-table.
        03 in-order-entry OCCURS 10 TIMES.
            04 in-order-item PIC X(5).

PROCEDURE DIVISION USING in-order.
    MOVE in-order-number TO order-number.
    MOVE in-order-date   TO order-date.
    MOVE in-order-count  TO order-count.
    MOVE in-order-table  TO order-table.
    EXIT METHOD.
END METHOD 'setInstanceData'.

* Method to get instance data and give it to subclass
IDENTIFICATION DIVISION.
METHOD-ID. 'getInstanceData'.

DATA DIVISION.
LINKAGE SECTION.
01 out-order.
    02 out-order-number PIC 9(5).
    02 out-order-date   PIC X(8).
    02 out-order-count  PIC 99.
    02 out-order-table.
        03 out-order-entry OCCURS 10 TIMES.
            04 out-order-item PIC X(5).

PROCEDURE DIVISION USING out-order.
    MOVE order-number TO out-order-number.
    MOVE order-date   TO out-order-date.
    MOVE order-count  TO out-order-count.
    MOVE order-table  TO out-order-table.
    EXIT METHOD.
END METHOD 'getInstanceData'.

* Method to add an item to an order
IDENTIFICATION DIVISION.
METHOD-ID. AddItem.
DATA DIVISION.

LOCAL-STORAGE SECTION.
77 sub PIC 99.
01 found-flag PIC 9 VALUE 1.
   88 found VALUE 0.
LINKAGE SECTION.
01 in-item PIC X(5).
01 add-flag PIC 9.

PROCEDURE DIVISION USING in-item
    RETURNING add-flag.
    MOVE 1 TO add-flag.
    PERFORM VARYING sub FROM 1 BY 1
        UNTIL (sub > 10) OR (found)
    IF order-item (sub) = SPACES
        MOVE in-item TO order-item (sub)
        ADD 1 TO order-count

```

```

        MOVE 0 TO add-flag
        SET found TO TRUE
    END-IF
END-PERFORM.
EXIT METHOD.
END METHOD AddItem.

* Method to delete an item from an order
IDENTIFICATION DIVISION.
METHOD-ID. DeleteItem.

DATA DIVISION.
LOCAL-STORAGE SECTION.
77 sub PIC 99.
01 found-flag PIC 9 VALUE 1.
   88 found VALUE 0.
LINKAGE SECTION.
01 out-item PIC X(5).
01 delete-flag PIC 9.

PROCEDURE DIVISION USING out-item
    RETURNING delete-flag.
    MOVE 1 TO delete-flag.
    PERFORM VARYING sub FROM 1 BY 1
        UNTIL (sub > 10) OR (found)
        IF order-item (sub) = out-item
            MOVE SPACES TO order-item (sub)
            SUBTRACT 1 FROM order-count
            MOVE 0 TO delete-flag
            SET found TO TRUE
        END-IF
    END-PERFORM.
    EXIT METHOD.
END METHOD DeleteItem.

* Method to compute the total cost of an order
IDENTIFICATION DIVISION.
METHOD-ID. ComputeCost.

DATA DIVISION.
LOCAL-STORAGE SECTION.
77 sub PIC 99.
77 cost PIC 9(5)V99.
LINKAGE SECTION.
01 total-cost PIC 9(7)V99.

PROCEDURE DIVISION USING total-cost.
    MOVE 0 TO total-cost.
    PERFORM VARYING sub FROM 1 BY 1
        UNTIL sub > order-count
    * Call a subroutine
    * NOTE: The subroutine code is not
    * included in this example.
        CALL 'InventoryGetCost'
            USING order-item (sub) cost
            ADD cost TO total-cost
    END-PERFORM.
    EXIT METHOD.
END METHOD ComputeCost.

* Method to return the order number
IDENTIFICATION DIVISION.
METHOD-ID. 'getOrderNumber'.

DATA DIVISION.
LINKAGE SECTION.
01 ord-num PIC 9(5).

```

```

PROCEDURE DIVISION RETURNING ord-num.
    MOVE order-number TO ord-num.
    EXIT METHOD.
END METHOD 'getOrderNumber'.

* Method to write completed order to a file
IDENTIFICATION DIVISION.
METHOD-ID. WriteOrder.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT order-file ASSIGN OrdrFile.

DATA DIVISION.
FILE SECTION.
FD order-file EXTERNAL.
01 order-record PIC X(80).
LOCAL-STORAGE SECTION.
01 print-line.
    02 print-order-number PIC 9(5).
    02 print-order-date PIC X(8).
    02 print-order-count PIC 99.
    02 print-order-table.
        03 print-order-entry OCCURS 10 TIMES.
            04 print-order-item PIC X(5).

PROCEDURE DIVISION.
    OPEN OUTPUT order-file.
    MOVE order-number TO print-order-number.
    MOVE order-date TO print-order-date.
    MOVE order-count TO print-order-count.
    MOVE order-table TO print-order-table.
    WRITE order-record FROM print-line.
    CLOSE order-file.
    EXIT METHOD.
END METHOD WriteOrder.

END CLASS Orders.

```

The subclass and method definitions for the NewOrders subclass:

```

IDENTIFICATION DIVISION.
CLASS-ID. NewOrders INHERITS Orders.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
* Declare classes used in subclass definition
REPOSITORY.
    CLASS NewOrders IS 'NewOrders'
    CLASS Orders IS 'Orders'.

DATA DIVISION.

PROCEDURE DIVISION.

* All methods are inherited from superclass

END CLASS NewOrders.

```

The subclass and method definitions for the BackOrders subclass:

```

IDENTIFICATION DIVISION.
CLASS-ID. BackOrders INHERITS Orders.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.

```

```

* Declare classes used in subclass definition
REPOSITORY.
    CLASS BackOrders IS 'BackOrders'
    CLASS Orders IS 'Orders'.
DATA DIVISION.

PROCEDURE DIVISION.

* Method to read back order from file
IDENTIFICATION DIVISION.
METHOD-ID. ReadOrder.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT backorder-file ASSIGN BackFile.

DATA DIVISION.
FILE SECTION.
FD backorder-file EXTERNAL.
01 backorder-record PIC X(80).
LOCAL-STORAGE SECTION.
01 backorder.
    02 backorder-number PIC 9(5).
    02 backorder-date PIC X(8).
    02 backorder-count PIC 99.
    02 backorder-table.
        03 backorder-entry OCCURS 10 TIMES.
            04 backorder-item PIC X(5).
77 eof-flag PIC 9 VALUE 1.
88 eof VALUE 0.
LINKAGE SECTION.
01 order-number PIC 9(5).

PROCEDURE DIVISION USING order-number.
    OPEN INPUT backorder-file.
    PERFORM UNTIL eof
        READ backorder-file INTO backorder
        AT END
            SET eof TO TRUE
        NOT AT END
            IF order-number = backorder-number
                INVOKE SELF 'setInstanceData' USING backorder
            END-IF
        END-READ
    END-PERFORM.
    CLOSE backorder-file.
    EXIT METHOD.
END METHOD ReadOrder.

* Method to check whether item is still not in stock
IDENTIFICATION DIVISION.
METHOD-ID. CheckItem.

DATA DIVISION.
LOCAL-STORAGE SECTION.
01 backorder.
    02 backorder-number PIC 9(5).
    02 backorder-date PIC X(8).
    02 backorder-count PIC 99.
    02 backorder-table.
        03 backorder-entry OCCURS 10 TIMES.
            04 backorder-item PIC X(5).
77 sub PIC 99.
77 status-flag PIC 9.
88 in-stock VALUE 0.
88 out-stock VALUE 1.

```

```

LINKAGE SECTION.
01 out-table.
    02 out-entry OCCURS 10 TIMES.
        03 out-item PIC X(5).
01 out-count PIC 99.

PROCEDURE DIVISION USING out-table out-count.
    INVOKE SELF 'getInstanceData' USING backorder.
    MOVE 0 TO out-count.
    PERFORM VARYING sub FROM 1 BY 1
        UNTIL sub > backorder-count
*   Call a subroutine
*   NOTE: The subroutine code is not
*   included in this example.
        CALL 'InventoryGetItem'
            USING backorder-item (sub) status-flag
        IF out-stock
            ADD 1 TO out-count
            MOVE backorder-item (sub) TO out-item (out-count)
        END-IF
    END-PERFORM.
    EXIT METHOD.
END METHOD CheckItem.

END CLASS BackOrders.

```

A possible new client program:

```

IDENTIFICATION DIVISION.
PROGRAM-ID. 'PhoneOrders'.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
*
* Declare the classes used in the program
REPOSITORY.
    CLASS NewOrders IS 'NewOrders'
    CLASS BackOrders IS 'BackOrders'
    CLASS UserInterface IS 'UserInterface'.

DATA DIVISION.
WORKING-STORAGE SECTION.
*
* Declare the object references used in the program
* Note: univObj is a universal object reference
77 univObj USAGE OBJECT REFERENCE.
77 userObj USAGE OBJECT REFERENCE UserInterface.
*
* Declare other data items used in the program
77 order-number PIC 9(5).
77 total-cost PIC 9(7)V99.
77 out-count PIC 9(2).
77 request PIC X(6).
77 action PIC X(10).
77 flag PIC 9.
77 item PIC X(5).
01 item-table.
    02 item-entry OCCURS 10 TIMES.
        03 item-element PIC X(5).

PROCEDURE DIVISION.
*
* Create an instance of the UserInterface class - userObj
    INVOKE UserInterface 'somNew' RETURNING userObj.
*
* Read customer input - request
    INVOKE userObj 'ReadUserRequest' USING request.

```

```

*
* What is the customer's request?
  IF request = 'STATUS'
    PERFORM CheckBackOrder
  ELSE
    PERFORM CreateNewOrder
  END-IF.
*
* Free the instance of the UserInterface class - userObj
  INVOKE userObj 'somFree'.

  STOP RUN.

CreateNewOrder.
*
* Create an instance of the NewOrders class - univObj
  INVOKE NewOrders 'somNew' RETURNING univObj.
*
* Read customer input - action and item
  INVOKE userObj 'ReadUserInput1' USING item action.

*
* Begin customer driven loop based on action
  PERFORM UNTIL action = 'Quit'
*
* Do appropriate action
  IF action (1:3) = 'Add'
    INVOKE univObj 'AddItem' USING item
                                RETURNING flag
  ELSE
    INVOKE univObj 'DeleteItem' USING item
                                RETURNING flag
  END-IF
*
* Display result of action
  INVOKE userObj 'WriteUserMessage' USING flag
*
* Read customer input - action and item
  INVOKE userObj 'ReadUserInput1' USING item action
  END-PERFORM.
* End customer driven loop based on action
*

*
* Calculate the total cost of the order
  INVOKE univObj 'ComputeCost' USING total-cost.
*
* Determine the order number
  INVOKE univObj 'getOrderNumber'
    RETURNING order-number.
*
* Display information about the order
  INVOKE userObj 'WriteUserOutput'
    USING total-cost order-number.

*
* Write the order to a file
  INVOKE univObj 'WriteOrder'.

*
* Free the NewOrders instance - univObj
  INVOKE univObj 'somFree'.

CheckBackOrder.
*
* Create an instance of the BackOrders class - univObj
  INVOKE BackOrders 'somNew' RETURNING univObj.

```

```

*
* Read customer input - order number
    INVOKE userObj 'ReadUserInput2' USING order-number.
*
* Read the back-ordered information from a file
    INVOKE univObj 'ReadOrder' USING order-number.
*
* Check whether the back-ordered items are now in stock
    INVOKE univObj 'CheckItem' USING item-table out-count.
*
* Display the status of the back-ordered items
    INVOKE userObj 'WriteUserStatus' USING item-table out-count.
*
* Free the BackOrders instance - univObj
    INVOKE univObj 'somFree'.

END PROGRAM 'PhoneOrders'.

```

## Defining a metaclass

A metaclass is a special type of class whose instances are called *class objects*. Class objects are the run-time objects that represent SOM classes. You can provide explicit metaclass definitions for specialized purposes. Otherwise, object-oriented COBOL applications use the default metaclasses provided automatically by the SOM environment.

Metaclasses have their own methods and can have their own instance data. The most common use of a metaclass is to control how an instance of a class is created. Metaclasses are also useful when multiple instances of a class are created and data must be gathered from all the instances.

Division	Purpose	Syntax
IDENTIFICATION (required)	Name the metaclass. Provide inheritance information for it.	"CLASS-ID paragraph for defining a metaclass" (required)
ENVIRONMENT (required)	Relate the metaclass and class names to external SOM names.	CONFIGURATION section (required) "REPOSITORY paragraph for defining a metaclass" on page 302 (required)
DATA (optional)	Describe instance data the metaclass needs.	"WORKING-STORAGE SECTION for defining a metaclass" on page 302 (optional)
PROCEDURE (optional)	Define methods.	"Defining a metaclass method" on page 302

You must properly terminate a metaclass definition with an END CLASS statement.

"Example: mail-order catalog" on page 271

"Example: defining a metaclass (with methods)" on page 304

### RELATED CONCEPTS

SOMClass Class Object (*SOMObjects Programmer's Guide*)

## CLASS-ID paragraph for defining a metaclass

The CLASS-ID paragraph names the metaclass and indicates from what base System Object Model (SOM) class the metaclass inherits. For example:

```

Identification Division.
Class-Id. MetaBackOrders INHERITS SOMClass.

```

**Required**  
**Required**

In the above example, `MetaBackOrders` is the class name. It inherits from the base SOM class `SOMClass`.

All metaclasses inherit directly or indirectly from `SOMClass`.

You must specify `SOMClass` in the `REPOSITORY` paragraph of the `ENVIRONMENT` `DIVISION`. Optionally, you can specify `MetaBackOrders` in the `REPOSITORY` paragraph.

## REPOSITORY paragraph for defining a metaclass

The `REPOSITORY` paragraph relates your metaclass and class names to the metaclass and class names in the SOM interface repository. For example:

```
Environment Division.      Required
Configuration Section.    Required
Repository.               Required
    Class MetaBackOrders is 'MetaBackOrders'
    Class SOMClass is 'SOMClass'.
```

You must include the following classes:

- SOM base classes.  
In the above example, `CLASS SOMClass IS 'SOMClass'` indicates what you are calling `SOMClass` in your COBOL program is also called `SOMClass` in the SOM repository.
- The classes from which your metaclass is inheriting.
- Any class referenced in methods introduced by the metaclass.

You can optionally include the name of the metaclass that you are defining. If you do not include the name of your metaclass, it is treated as all uppercase regardless of how you typed it in the `CLASS-ID` paragraph. In the above example, `MetaBackOrders` is stored in the SOM interface repository in mixed case.

## WORKING-STORAGE SECTION for defining a metaclass

A metaclass `WORKING-STORAGE SECTION` describes instance data that is statically allocated when the first instance of an object in the metaclass is created and exists until the COBOL run unit terminates. For example:

```
Data Division.
Working-Storage Section.
01 total-orders PIC X(3).
```

By default, the data is global to all the methods that the metaclass introduces. Instance data in a COBOL metaclass is *private*. No other class or metaclass can reference it.

## Defining a metaclass method

You define metaclass methods the same way that you define regular class methods. The only real difference is that metaclass methods are focused on object creation and initialization, or class-wide information, whereas ordinary class methods are instance-specific.

Division	Purpose	Syntax
IDENTIFICATION (optional)	Same as subclass method	"METHOD-ID paragraph for defining a subclass method" on page 291 (optional)
ENVIRONMENT (optional)	Same as class method	Same as for a class method

Division	Purpose	Syntax
DATA (optional)	Same as class method	Same as for a class method
PROCEDURE (optional)	Same as class method	Same as for a class method

You must properly terminate a metaclass method definition with an `END METHOD` statement.

“Example: mail-order catalog” on page 271

“Example: defining a metaclass (with methods)” on page 304

#### RELATED TASKS

“Invoking a metaclass constructor method”

“Defining a class method” on page 277

“Defining a subclass method” on page 291

### Invoking a metaclass constructor method

A metaclass method that creates an instance of a class is called a *constructor* method. A metaclass constructor method is usually invoked with a class name. Therefore, use the following form of the `INVOKE` statement in the constructor method to create an instance of the class:

```
Invoke Self 'somNew' Returning anObj.
```

This example creates an instance of the class on which the method was invoked, `Self`, and returns the handle to that instance in the object reference `anObj`.

## Changing the definition of a class or subclass

When a class or subclass uses an explicit metaclass, you must specify the name of the metaclass with the `METACLASS IS` clause in the `CLASS-ID` paragraph. For example:

```
Identification Division.
Class-Id. BackOrders Inherits Orders
          Metaclass is MetaBackOrders.
```

Also, you must specify the name of the metaclass in the `REPOSITORY` paragraph of the `CONFIGURATION SECTION`. For example:

```
Environment Division.
Configuration Section.
Repository.
  Class MetaBackOrders Is 'MetaBackOrders'
  Class BackOrders Is 'BackOrders'
  Class Orders Is 'Orders'.
```

## Changing a client program

To use the metaclass constructor method, the client program invokes the constructor method instead of `somNew`. For example:

```
Invoke BackOrders 'CreateBackOrders' Using order-number Returning anObj.
```

The method `CreateBackOrders` is defined in the metaclass for `BackOrders`. This method invokes `somNew` to create an instance, reads the data from the file using the order number, and returns the handle to the instance in the object reference `anObj`.

You can invoke any method in a metaclass with the class name. For example:

```
Invoke BackOrders 'CountBackOrders' Returning out-count.
```

Or you can define a metaclass object reference as a handle to the metaclass. For example:

```
Working-Storage Section.  
01 metaObj Usage Object Reference Metaclass BackOrders.
```

The object reference metaObj is a handle to the metaclass for BackOrders, not a handle to BackOrders itself.

The metaclass object reference is used as follows:

```
Procedure Division.  
.  
.  
.  
    Invoke backObj 'somGetClass' Returning metaObj.  
    Invoke metaObj 'CountBackOrders' Returning out-count.
```

The first INVOKE statement invokes a SOM method somGetClass which takes an object reference, backObj, to an instance and returns an object reference, metaObj, to the metaclass to which backObj belongs.

The second INVOKE statement uses the object reference to the metaclass, metaObj to invoke the method CountBackOrders which is defined in the metaclass.

“Example: defining a metaclass (with methods)”

## Example: defining a metaclass (with methods)

BackOrders requires the reading of a file to establish its instance data. Reading the file cannot be done by somInit because an order number is needed as a parameter. This is a good place to use a metaclass with a constructor method to create the instance of BackOrders and read the file.

The metaclass and method definitions for the BackOrders subclass:

```
IDENTIFICATION DIVISION.  
CLASS-ID.  MetaBackOrders INHERITS SOMClass.  
  
ENVIRONMENT DIVISION.  
CONFIGURATION SECTION.  
* Declare classes used in metaclass definition  
REPOSITORY.  
    CLASS MetaBackOrders IS 'MetaBackOrders'  
    CLASS BackOrders IS 'BackOrders'  
    CLASS SOMClass IS 'SOMClass'.  
  
DATA DIVISION.  
* Define instance data  
WORKING-STORAGE SECTION.  
01 status-count PIC 99.  
PROCEDURE DIVISION.  
  
* Method to initialize instance data  
IDENTIFICATION DIVISION.  
METHOD-ID.  'somInit' OVERRIDE.  
  
PROCEDURE DIVISION.  
    MOVE 0 TO status-count.  
    EXIT METHOD.  
END METHOD 'somInit'.  
  
* Method to create and initialize instances of BackOrders  
IDENTIFICATION DIVISION.  
METHOD-ID.  CreateBackOrders.  
  
DATA DIVISION.
```

```

LINKAGE SECTION.
01 order-number PIC 9(5).
01 anObj USAGE OBJECT REFERENCE.

PROCEDURE DIVISION USING order-number RETURNING anObj.
    INVOKE SELF 'somNew' RETURNING anObj.
    INVOKE anObj 'ReadOrder' USING order-number.
    ADD 1 TO status-count.
    EXIT METHOD.
END METHOD CreateBackOrders.

* Method to return the number of back orders processed
IDENTIFICATION DIVISION.
METHOD-ID. CountBackOrders.

DATA DIVISION.
LINKAGE SECTION.
01 out-count PIC 9(2).

PROCEDURE DIVISION RETURNING out-count.
    MOVE status-count TO out-count.
    EXIT METHOD.
END METHOD CountBackOrders.

END CLASS MetaBackOrders.

```

The new subclass and method definitions for the BackOrders subclass:

```

IDENTIFICATION DIVISION.
CLASS-ID. BackOrders INHERITS Orders
        METAClass MetaBackOrders.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
* Declare classes used in subclass definition
REPOSITORY.
    CLASS MetaBackOrders IS 'MetaBackOrders'
    CLASS BackOrders IS 'BackOrders'
    CLASS Orders IS 'Orders'.

DATA DIVISION.

PROCEDURE DIVISION.

* Method to read back order from file
IDENTIFICATION DIVISION.
METHOD-ID. ReadOrder.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT backorder-file ASSIGN BackFile.

DATA DIVISION.
FILE SECTION.
FD backorder-file EXTERNAL.
01 backorder-record PIC X(80).
LOCAL-STORAGE SECTION.
01 backorder.
    02 backorder-number PIC 9(5).
    02 backorder-date PIC X(8).
    02 backorder-count PIC 99.
    02 backorder-table.
        03 backorder-entry OCCURS 10 TIMES.
            04 backorder-item PIC X(5).
77 eof-flag PIC 9 VALUE 1.
88 eof VALUE 0.

```

```

LINKAGE SECTION.
01 order-number PIC 9(5).

PROCEDURE DIVISION USING order-number.
  OPEN INPUT backorder-file.
  PERFORM UNTIL eof
    READ backorder-file INTO backorder
    AT END
      SET eof TO TRUE
    NOT AT END
      IF order-number = backorder-number
        INVOKE SELF 'setInstanceData' USING backorder
      END-IF
    END-READ
  END-PERFORM.
  CLOSE backorder-file.
  EXIT METHOD.
END METHOD ReadOrder.

* Method to check whether item is still not in stock
IDENTIFICATION DIVISION.
METHOD-ID. CheckItem.

DATA DIVISION.
LOCAL-STORAGE SECTION.
01 backorder.
  02 backorder-number PIC 9(5).
  02 backorder-date PIC X(8).
  02 backorder-count PIC 99.
  02 backorder-table.
    03 backorder-entry OCCURS 10 TIMES.
    04 backorder-item PIC X(5).
77 sub PIC 99 VALUE 0.
77 status-flag PIC 9.
88 in-stock VALUE 0.
88 out-stock VALUE 1.
LINKAGE SECTION.
01 out-table.
  02 out-entry OCCURS 10 TIMES.
  03 out-item PIC X(5).
01 out-count PIC 99.

PROCEDURE DIVISION USING out-table out-count.
  INVOKE SELF 'getInstanceData' USING backorder.
  MOVE 0 TO out-count.
  PERFORM VARYING sub FROM 1 BY 1
    UNTIL sub > backorder-count
  * Call a subroutine
  * NOTE: The subroutine code is not
  * included in this example.
  CALL 'InventoryGetItem'
  USING backorder-item (sub) status-flag
  IF out-stock
    ADD 1 TO out-count
    MOVE backorder-item (sub) TO out-item (out-count)
  END-IF
  END-PERFORM.
  EXIT METHOD.
END METHOD CheckItem.

END CLASS BackOrders.

```

A possible new client program:

```

IDENTIFICATION DIVISION.
PROGRAM-ID. 'PhoneOrders'.

```

```

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
*
* Declare the classes used in the program
REPOSITORY.
    CLASS NewOrders IS 'NewOrders'
    CLASS BackOrders IS 'BackOrders'
    CLASS UserInterface IS 'UserInterface'.

DATA DIVISION.
WORKING-STORAGE SECTION.
*
* Declare the object references used in the program
77 univObj  USAGE OBJECT REFERENCE.
* Note: metaObj is an object reference to a metaclass
77 metaObj  USAGE OBJECT REFERENCE METAClass BackOrders.
77 userObj  USAGE OBJECT REFERENCE UserInterface.
*
* Declare other data items used in the program
77 order-number  PIC 9(5).
77 total-cost    PIC 9(7)V99.
77 out-count     PIC 9(2).
77 request       PIC X(6).
77 action        PIC X(10).
77 flag          PIC 9.
77 item          PIC X(5).
01 item-table.
    02 item-entry OCCURS 10 TIMES.
        03 item-element PIC X(5).

PROCEDURE DIVISION.
*
* Create an instance of the UserInterface class - userObj
    INVOKE UserInterface 'somNew' RETURNING userObj.
*
* Read customer input - request
    INVOKE userObj 'ReadUserRequest' USING request.
*
* What is the customer's request?
    IF request = 'STATUS'
        PERFORM CheckBackOrder
    ELSE
        PERFORM CreateNewOrder
    END-IF.
*
* Free the instance of the UserInterface class - userObj
    INVOKE userObj 'somFree'.

    STOP RUN.

CreateNewOrder.
*
* Create an instance of the NewOrders class - univObj
    INVOKE NewOrders 'somNew' RETURNING univObj.
*
* Read customer input - action and item
    INVOKE userObj 'ReadUserInput1' USING item action.
*
* Begin customer driven loop based on action
    PERFORM UNTIL action = 'Quit'
*
* Do appropriate action
        IF action (1:3) = 'Add'
            INVOKE univObj 'AddItem' USING item
                RETURNING flag
        ELSE

```

```

            INVOKE univObj 'DeleteItem' USING item
                                RETURNING flag
        END-IF
*
*   Display result of action
        INVOKE userObj 'WriteUserMessage' USING flag
*
*   Read customer input - action and item
        INVOKE userObj 'ReadUserInput1' USING item action
        END-PERFORM.
*   End customer driven loop based on action
*
*
*   Calculate the total cost of the order
        INVOKE univObj 'ComputeCost' USING total-cost.
*
*   Determine the order number
        INVOKE univObj 'getOrderNumber'
                                RETURNING order-number.
*
*   Display information about the order
        INVOKE userObj 'WriteUserOutput'
                                USING total-cost order-number.
*
*   Write the order to a file
        INVOKE univObj 'WriteOrder'.
*
*   Free the NewOrders instance - univObj
        INVOKE univObj 'somFree'.

CheckBackOrder.
*
*   Read customer input - order number
        INVOKE userObj 'ReadUserInput2' USING order-number.
*
*   Begin customer driven loop based order number
        PERFORM UNTIL order-number = 0
*
*   Create an instance of the BackOrders class (univObj) and
*   read the back order from a file using a metaclass method
        INVOKE BackOrders 'CreateBackOrders'
                                USING order-number RETURNING univObj
*
*   Check whether the back-ordered items are now in stock
        INVOKE univObj 'CheckItem'
                                USING item-table out-count
*
*   Display the status of the back-ordered items
        INVOKE userObj 'WriteUserStatus'
                                USING item-table out-count
*
*   Read customer input - order number
        INVOKE userObj 'ReadUserInput2'
                                USING order-number
        END-PERFORM.
*   End customer driven loop based on order number
*
*
*   Get an object reference to the metaclass
*   Note: somGetClass is a SOM method
        INVOKE univObj 'somGetClass' RETURNING metaObj.
*
*   How many back orders were processed?

```

```

* Note: Metaclass object reference to invoke metaclass method
  INVOKE metaObj 'CountBackOrders' RETURNING out-count.
  DISPLAY out-count ' back orders were processed.'.
*
* Free the metaclass instance - metaObj
* Note: This also frees all BackOrders instances
  INVOKE metaObj 'somFree'.

END PROGRAM 'PhoneOrders'.

```



---

## Chapter 19. System Object Model

The System Object Model (SOM) is an object-oriented programming technology for building, packaging, and using class libraries. With SOM, class implementers can describe the interface to a class in a standard language called the *Interface Definition Language (IDL)*. Unlike the object model found in most object-oriented programming languages, SOM is language-neutral. It provides encapsulation, inheritance, and polymorphism without requiring both the implementer and the user of a SOM class to use the same programming language.

### RELATED CONCEPTS

“SOM Interface Repository”

“SOM services” on page 313

### RELATED TASKS

“Defining a class” on page 274

“Chapter 20. Using SOM IDL-based class libraries” on page 317

### RELATED REFERENCES

“SOM methods and functions” on page 313

---

## SOM Interface Repository

The SOM Interface Repository (IR) is a database in which the SOM compiler optionally creates and maintains class interface definitions. The COBOL compiler uses the SOM IR when compiling object-oriented COBOL programs. When you compile a class definition or client program with the IDLGEN or the TYPECHK option, the interface information for referenced classes must be present in the IR. (You declare all referenced classes in the REPOSITORY paragraph of the CONFIGURATION SECTION.)

### RELATED TASKS

“Accessing the SOM Interface Repository”

“Compiling and linking programs that call SOM functions” on page 314

“Populating the SOM Interface Repository” on page 312

## Accessing the SOM Interface Repository

You specify which interface repository (IR) files to use by setting the SOM environment variable SOMIR to the names of the files. For example:

```
set SOMIR=c:\mydir\mycls.ir
```

In the following example, som.ir is SOM's IR that is not updated, dept.ir is a stable department IR that is not updated, and work.ir is the working IR that is updated:

```
set SOMIR=c:\som\som.ir;c:\dept\dept.ir;c:\work\work.ir
```

You can set SOMIR at the time you use it. However, it is easier to set it in the System window.

If you do not set the SOMIR environment variable, the IR emitter creates a file named som.ir in the current directory.

You might need to update the SOMIR environment variable if you delete and reinstall IBM VisualAge COBOL, or install another product that updates it.

#### RELATED CONCEPTS

"SOM Interface Repository" on page 311

"SOM services" on page 313

#### RELATED REFERENCES

"SOM environment variables"

"SOM methods and functions" on page 313

## Populating the SOM Interface Repository

To populate the IR with interface information from COBOL classes:

1. Compile the COBOL class definition with the COBOL compiler, specifying the IDLGEN compiler option.
2. Compile the IDL source files with the SOM compiler, using the IR emitter.

You might have to compile COBOL class definitions that have complex interdependencies in two steps. For example, there could be circular compilation order dependencies, such as when two class definitions each contain references to the other. To compile such complex configurations:

1. Compile all of the COBOL class definition source files using the IDLGEN, NOTYPECHK, and NOCOMPILE compiler options. This compile generates IDL files for the class interfaces, but does not perform type checking or generate an object file.
2. Compile the IDL files with the SOM compiler, using the IR emitter. This compilation populates the IR with the class interface information. For example, this TSO command:

```
sc -usir myclass.idl
```

starts the SOM compiler, `sc`, against the file `myclass.idl` with the `-usir` option, which updates the rightmost file in the list of IR files specified for the SOMIR environment variable.

3. Compile the COBOL class definitions again, using the NOIDLGEN and TYPECHK compiler options. This final compile performs full type checking and generates object files.

#### RELATED CONCEPTS

"SOM Interface Repository" on page 311

#### RELATED REFERENCES

"Class initialization" on page 314

---

## SOM environment variables

Environment variables provide information needed by the COBOL compiler and run-time environment and the SOM compiler and run-time environment.

### SMEMIT

Specifies which emitters the SOM compiler runs.

For a COBOL class the most frequently used emitter is the `.h` emitter which produces a header file for use by a C client of the COBOL class.

For example, the following series of statements directs the SOM compiler to produce `myclass.h`, and populate the IR from the `myclass.idl` input specification:

```
set SMEMIT="h"  
sc -usir myclass.idl
```

#### **SMINCLUDE**

Specifies where to look for `#include` files included by the `.idl` file being compiled. For example:

```
set SMINCLUDE=.;c:\toolkit20\include;c:\som\include
```

#### **SMTMP**

Specifies where to put intermediate output files. Use a different directory from the ones where input and output files are located. For example:

```
set SMTMP=c:\tmp\garbage
```

#### **SOMIR**

Specifies the location of the interface repositories. For example:

```
set SOMIR=c:\mydir\mycls.ir
```

You can type these environment variables when you need them. However, it is easier to set them in the System window.

---

## **SOM services**

IBM COBOL implements a subset of the ANSI Object-Oriented COBOL syntax, based on the SOM object-oriented engine. Not all essential object-oriented capabilities are implemented in native COBOL syntax. Instead, IBM COBOL uses SOM application programming interfaces, methods, and functions. For example, native COBOL syntax is available for class definitions, object-reference data type, and method invocation. However, you accomplish object creation, destruction, initialization, and termination by invoking SOM methods provided by the `SOMObject` and `SOMClass` classes. Many other SOM facilities are available to you either for direct use or for overriding and customizing.

#### **RELATED REFERENCES**

"SOM methods and functions"

"Class initialization" on page 314

## **SOM methods and functions**

The following SOM methods and functions are especially important to COBOL programmers:

**somNew** A method in `SOMClass` to create a new object instance of a class. During creation, `somInit` is invoked for customized initialization of the object.

#### **somFree**

A method in `SOMObject` to free an object instance releasing the storage used. Prior to freeing storage, `somUninit` is invoked for customized uninitialization.

`somFree` must not be invoked to destroy an active object, that is, an object upon which a method has been invoked that has not yet returned control to the invoker.

**somInit**

A method in SOMObject that has no default function, but can be overridden explicitly in a COBOL class definition to perform customized initializations when an object is created.

**somUninit**

A method in SOMObject that has no default function, but can be overridden explicitly in a COBOL class definition to perform customized uninitialization (typically the inverse of the function performed by a customized somInit).

**somGetClass**

A method in SOMObject that returns an object reference to the class object of an object instance. The class of an object is useful for obtaining information about the object.

**somIsObj**

A function that determines whether an object-reference refers to a valid object.

somIsObj returns a result of type Boolean. Though COBOL has no BOOLEAN data type, COBOL programmers can declare the return value as PIC X and test the value using a symbolic character or hex literal.

Data Division.

Working-Storage Section.

01 somBoolean Pic X.

88 invalid-obj Value X'00'.

88 valid-obj Value X'01'.

Procedure Division.

...

Call 'somIsObj' Using by Value anObj Returning somBoolean.

If invalid-obj

Display 'Object reference does not refer to a valid object'

End-if.

...

## Compiling and linking programs that call SOM functions

When you compile a program that calls a SOM function (such as somIsObj), specify these compiler options:

- PGMNAME(MIXED), because API names are case sensitive. Otherwise, the compiler will translate somIsObj to SOMISOBJ, and you will get an unresolved external reference.
- CALLINT(SYSTEM), because SOM API functions use the SYSTEM linkage convention. Alternatively, the >>CALLINT SYSTEM directive must be in effect for the CALL statement.

Your invocations of SOM methods do not require any special considerations. The correct linkage conventions are used automatically for method invocations.

## Class initialization

Every SOM class provides an initialization function <classname>NewClass. Normally COBOL programmers do not use this function directly, but the function is available in all COBOL classes. The COBOL run-time system automatically initializes all classes referenced within a COBOL program by calling their class initialization functions before the first user-written COBOL statement in the PROCEDURE DIVISION is executed.

The class initialization function has a case-sensitive name. Therefore, you must use PGMNAME(LONGMIXED) when compiling a COBOL program that explicitly calls a class initialization function.

If you specify an external class-name in the REPOSITORY paragraph for a class, COBOL uses this class-name to form the initialization function name. If you do not specify an external class-name, COBOL uses the class-name to form a CORBA-compliant external class name for the class initialization function. The translation to a CORBA-compliant external class-name follows these rules:

- The name becomes uppercase.
- Hyphens in the name become zeros.
- If the first character in the name is a digit, 1 through 9 become A through I and 0 becomes J.

The following example includes an external name for the class:

```
Identification Division.  
Class-Id. Employee inherits SOMObject.  
Environment Division.  
Configuration Section.  
Repository.  
    Class Employee is "Employee"  
    Class SOMObject is "SOMObject".  
    . . .  
End Class Employee.
```

The class initialization function names in the above cases are:

- EmployeeNewClass
- SOMObjectNewClass

The following example does not include an external name and so the class name is adapted:

```
Identification Division.  
Class-Id. Employee inherits SOMObject.  
Environment Division.  
Configuration Section.  
Repository.  
    Class SOMObject is "SOMObject".  
    . . .  
End Class Employee.
```

The class initialization function names in the above cases are:

- EMPLOYEENewClass
- SOMObjectNewClass

#### RELATED CONCEPTS

"SOM services" on page 313

#### RELATED REFERENCES

"SOM methods and functions" on page 313

## Changing SOM class interfaces

One of the benefits of SOM is that classes can change over time and yet retain backward binary compatibility. You need not recompile programs and classes that refer to a changed class. You can make the following changes to classes without having to recompile the clients of the classes:

- Add new methods.

- Change the size of an object by adding or deleting instance data.
- Insert new parent classes above a class in the inheritance hierarchy.
- Relocate methods upward in the class hierarchy.

The SOM engine provides several alternatives for method resolution. IBM COBOL uses SOM name-lookup resolution to invoke methods. Therefore, when COBOL code invokes COBOL methods, the somewhat more stringent recompilation requirements of the SOM offset-resolution mechanism do not apply. For example, you can relocate a COBOL method anywhere in a class hierarchy without having to recompile the COBOL programs that invoke the method.

You can invoke methods defined in COBOL classes from other languages (such as C code built with the SOM C emitter) that use offset resolution. In this case, the standard SOM requirements apply. COBOL does not provide language comparable to the SOM “release-order” mechanism, which is used to ensure methods can be added to a class definition without requiring recompilation of code that invokes the methods using offset resolution. When you add methods to an existing COBOL class, add them at the end of the PROCEDURE DIVISION of the class definition, after existing methods. This placement ensures that any existing client code invoking the original methods does not require recompilation.

---

## Chapter 20. Using SOM IDL-based class libraries

You can use SOM IDL-based class libraries, either as a client of the library as is, or by specializing the library classes using subclassing.

There are several forms of Interface Definition Language (IDL), such as those for the Distributed Computing Environment (DCE) or the Object Management Group's Common Object Request Broker Architecture (OMG CORBA). This information concerns only IBM's System Object Model Interface Definition Language (SOM IDL). SOM IDL is consistent with CORBA but allows some additional data types such as pointers.

You need to understand the System Object Model (SOM), at least conceptually, and know where to find more detailed documentation about SOM when you need it. You also need access to the documentation for the class library that you are intending to use.

To get started, you need one of the object-enabled IBM COBOL products, together with the executables for the class library and its documentation.

"Example: using a SOM IDL-based class library" on page 332

### RELATED CONCEPTS

"SOM objects"

"SOM IDL" on page 318

### RELATED TASKS

"Mapping IDL to COBOL" on page 319

"Handling errors and exceptions" on page 334

"Creating and initializing object instances" on page 337

"Avoiding memory leaks" on page 339

### RELATED REFERENCES

"Common IDL types" on page 321

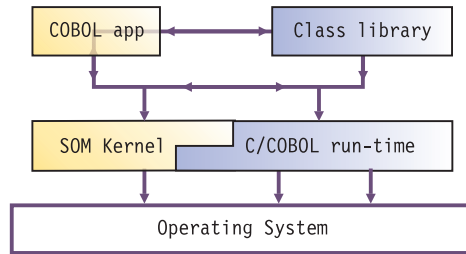
---

## SOM objects

A SOM class library consists of:

- Executable code
- Interface information that defines the operations that the library supports, including the parameters for invoking the operations (known as the operation *signatures*)

When the library is being used at run time, the components that are present in memory are shown below:



#### RELATED TASKS

“Using IDL operations” on page 319

## SOM IDL

The interface information for a SOM class library can be in various forms:

- IDL files
- An Interface Repository (IR), a machine-readable form of IDL used during compilation
- Documentation defining the method interfaces in IDL, and describing the semantics and rules for using the methods

IDL expresses the contract between the provider of object services (in this case the class library) and the user of these services (COBOL program, method, or subclass). The interface description is formally independent of the language in which either the user of the service or the service itself is implemented. This property is known as *language neutrality*. The separation of the interface from the implementation also allows flexibility in the deployment of the objects on the nodes of a network.

IDL data types have their origins in the C and C++ data model. Because many of them do not have an exact counterpart in the COBOL language, there needs to be a translation (or mapping) between IDL and COBOL. The mapping recommended here assumes that the data structures can be passed directly between the COBOL and C/C++ mappings to SOM IDL.

The standard CORBA model presumes a stub routine between the invoking and invoked object to do argument translation, marshalling, and so on. Passing the structures directly yields very significant gains in efficiency, but some of the mappings might not seem as natural to the COBOL programmer as they would if the transfer were mediated by a stub routine. Passing directly also means that you must have the correct alignment and padding of structures that are passed across an interface. In general the recommended way to achieve this for IDL-based interfaces is to specify the `SYNCHRONIZED` clause for COBOL mappings to any IDL structs or arrays that directly contain structs.

#### RELATED TASKS

“Mapping IDL to COBOL” on page 319

#### RELATED REFERENCES

“Common IDL types” on page 321

“Complex IDL types” on page 324

---

## Mapping IDL to COBOL

To use an IDL-based class library from COBOL, you must map the elements of IDL in which the interface to the library is expressed into the COBOL language. Typically, you find the description of the class library in a user's guide and reference, along with guidelines for using the class library and the calling sequences for each method.

You need to map IDL identifiers, operations, and attributes to their COBOL counterparts. Some of the IDL types (such as `boolean`, `long`, and `string`) occur very commonly in interfaces, while complex IDL types (such as `array` and `struct`) are quite rare. You might be able to avoid knowing about the complex IDL types. Refer to the related references below for each category of IDL types. You do need to know the conventions for passing arguments and return values.

The only IDL names that must be identical in COBOL are the class (IDL interface) and method (IDL operation) names. You specify these names exactly with literals:

- For a class, in the `CLASS IS` clause of the `REPOSITORY` paragraph.
- For a method, by using the literal form of the method name in the `METHOD-ID` paragraph. When you invoke a method, you use either the literal form of the name or a data name initialized with the exact method name.

The other identifiers, such as parameter, constant, and exception names, are internal to your program or class, and do not have to be identical to the IDL. However, it is a good practice to keep these names close to the IDL originals to enhance the readability and maintainability of your programs.

### RELATED TASKS

"Using IDL operations"

"Expressing IDL attributes" on page 320

"Passing COBOL arguments and return values" on page 327

### RELATED REFERENCES

"Common IDL types" on page 321

"Complex IDL types" on page 324

## Using IDL operations

IDL operations correspond to COBOL methods and represent the services that an IDL interface provides.

To use an operation, you code an `INVOKE` statement with the appropriate `USING` and `RETURNING` phrases that correspond to the parameters and the return value of the operation. If these parameters are simple scalar types, the operation definition is self-contained. But if an operation uses a "constructed" type, you might need the definition of the parameter type to specify your `INVOKE` statement completely.

Consider the IDL operation definition:

```
void addColor(in color that);
```

The single input argument is of type `color`, which is not a basic scalar IDL type. Suppose that `color` is an IDL enum (enumerated item) with the following definition (typically found in a different section of the library documentation):

```
typedef enum color{red, white, blue};
```

Then you would write the COBOL code to map the operation, adding the color blue to an object, as follows:

```
1 color binary pic 9(9).
  88 red value 1.
  88 white value 2.
  88 blue value 3.
  . . .
  Set blue to true
  Invoke anobject 'addColor' using by value evp color
```

The evp argument is the *environment pointer*, which precedes the explicit operation arguments. It is used for communicating to the caller any exceptions that the operation encounters.

#### RELATED TASKS

“Passing COBOL arguments and return values” on page 327

“Handling errors and exceptions” on page 334

#### RELATED REFERENCES

“Common IDL types” on page 321

## Expressing IDL attributes

An IDL attribute behaves like instance data that you can see outside the class definition (but there need not be any actual instance variable corresponding to it).

SOM models attributes as a pair of operations, one to set and one to get the attribute value. Attribute operations return errors by means of standard exceptions.

Consider the following IDL specification:

```
interface foo {
  struct position_t {
    float x, y;
  };

  attribute float radius;
  readonly attribute position_t position;
};
```

This is exactly equivalent to the following IDL specification (which is illegal because IDL identifiers are not permitted to start with an underscore):

```
interface foo {
  struct position_t {
    float x, y;
  };

  float _get_radius();
  void _set_radius(in float r);
  position_t _get_position();
};
```

The COBOL code to use these operations is straightforward:

```
1 radius comp-1.
1 position-t.
  2 x comp-1.
  2 y comp-1.
  . . .
  Invoke a-foo '_get_radius' using by value evp returning radius
  Invoke a-foo '_set_radius' using by value evp radius
  Invoke a-foo '_get_position' using by value evp
    returning position-t
```

#### RELATED TASKS

"Passing COBOL arguments and return values" on page 327

"Handling errors and exceptions" on page 334

## Common IDL types

These are the IDL types that you normally encounter in SOM IDL interfaces.

IDL type	COBOL equivalent	Data item
boolean	DISPLAY PICTURE X [+ level-88s. . .]	1-byte alphanumeric; level-88 condition names are recommended
char	DISPLAY PICTURE X	1-byte alphanumeric
double	COMPUTATIONAL-2	64-bit floating point
"enum" on page 322	COMPUTATIONAL-5 PICTURE 9(9) [+ level-88s]	Unsigned binary fullwords followed by a condition-name entry for each enumeration member
float	COMPUTATIONAL-1	32-bit floating point
"interface" on page 322	OBJECT REFERENCE	
"long (signed and unsigned)" on page 322	COMPUTATIONAL-5 PICTURE S9(9)	Unsigned forms of binary data
octet	DISPLAY PICTURE X	8-bit quantity that is guaranteed to be unchanged during transmission between objects; most closely matched to a 1-byte alphanumeric data item
pointer	POINTER	
"short (signed and unsigned)" on page 323	COMPUTATIONAL-5 PICTURE S9(4)	Unsigned forms of binary data
"string" on page 323	DISPLAY PIC X(n+1), Z'value' or variable-length alphanumeric table	
void	Omit the RETURNING phrase in the corresponding INVOKE statements or PROCEDURE DIVISION headers	

"Examples: common IDL types"

### Examples: common IDL types

The following table shows examples of the common IDL types and the COBOL equivalent.

IDL type	IDL example	COBOL equivalent
boolean	boolean that;	1 that display pic x. 88 that-false value x'00'. 88 that-true value x'01' thru x'ff'.
char	char that;	1 that display pic x.
double	double that;	1 that comp-2.
enum	enum that {red, white, blue, green};	1 that comp-5 pic 9(9). 88 that-red value 1. 88 that-white value 2. 88 that-blue value 3. 88 that-green value 4.
float	float that;	1 that comp-1.

IDL type	IDL example	COBOL equivalent
interface	interface that {. . .}	Repository. class that 'that'. . . . 1 a-that object reference that.  (You pass the instance of the class according to the rules for passing arguments.)
long	long that;	1 that comp-5 pic s9(9).
octet	octet that;	1 that displays pic x.
pointer	pointer that;	1 that pointer.
short	short that;	1 that comp-5 pic s9(4).
string (bounded)	string<100> that;	1 that-l-max comp-5 pic 9(9) value 101. 1 that-l comp-5 pic 9(9). 1 that. 2 that-v pic x occurs 1 to 101 depending that-l.
string (unbounded)	string that;	1 that-l-max comp-5 pic 9(9) value 4096. 1 that-l comp-5 pic 9(9). 1 that. 2 that-v pic x occurs 1 to 4096 depending that-l.
unsigned long	unsigned long that;	1 that comp-5 pic 9(9).
unsigned short	unsigned short that;	1 that comp-5 pic 9(4).

## enum

The closest COBOL equivalent to a SOM IDL enum is an unsigned binary fullword, together with condition-name entries for each of the enumeration members.

A SOM IDL enum is different from a C/C++ enum:

1. It is always exactly 4 bytes long, whereas a C/C++ enum is 1, 2, or 4 bytes long, depending on the maximum enum value and on compiler options.
2. The members are numbered sequentially starting from one, whereas a C/C++ enum starts at zero by default, or can optionally have specific values assigned to the enumeration members.

The way that you refer to a particular enum value in your PROCEDURE DIVISION depends on whether the value is supplied to an operation or returned by it.

### RELATED TASKS

“Passing enumerated arguments” on page 329

## interface

The use of an IDL interface as an argument to, or result of, an operation denotes an object reference to an instance of the class to which the interface has been mapped. Therefore, if a method has an interface type as one of its parameters, specify an OBJECT REFERENCE to an instance of a class that supports that interface.

## long (signed and unsigned)

The SOM IDL long type describes 32-bit signed binary quantities, and is mapped by a USAGE COMPUTATIONAL-5 data item with a PICTURE clause of (S9(5) through) S9(9).

The unsigned form of binary data is mapped as for the SOM IDL long type, except that the PICTURE clause does not specify the character S. If you map this IDL type to USAGE BINARY, you must either know that the PICTURE clause accommodates the expected range of values, or use the TRUNC(BIN) compiler option and (on the workstation) ensure that the BINARY(NATIVE) compiler option is in effect.

Also be aware that there are significant performance effects associated with the use of COMP-5 data items or the TRUNC(BIN) compiler option (primarily, though, for doubleword binary data items, that is for items declared USAGE BINARY with a PICTURE clause containing 10 or more 9s).

### **short (signed and unsigned)**

The SOM IDL short type defines 16-bit signed binary data, and is mapped by a USAGE COMPUTATIONAL-5 data item with a PICTURE clause of (S9(1) through) S9(4).

The unsigned form of binary data is mapped as for the SOM IDL short type, except that the PICTURE clause does not specify the character S.

### **string**

The SOM IDL type string is one of the most important types, because it is widely used in operations and interfaces. It is also one of the most difficult to match in COBOL, because SOM IDL strings are modeled on those of C and C++. These have a null terminator to determine the length of the string, and are passed by a typed pointer. IBM COBOL does not support such null-terminated strings as a native data type. However, the null-terminated literal Z'*string-value*' alleviates some of the problems and, when it can be passed BY CONTENT, is an exact match to a SOM IDL in string. (For inout and out strings, you must pass a pointer to the string data or buffer BY REFERENCE.) You can also use a null-terminated literal to set the initial VALUE of a data item to be used as a string argument.

In general, IDL strings are mapped to pointers to the appropriate character string data or buffer. With the exception of in strings, however, just the pointer is actually passed in a method invocation. But, for this to work operationally, the pointer must be set to the address of the *underlying* character string data or buffer in PICTURE X format. There are several styles of data definition, depending on whether the parameter is an in, inout, or out argument, or a return value. The declarations described below can be used to represent both the COBOL and SOM IDL view of a variable-length string simultaneously.

The main cases to distinguish are bounded and unbounded strings. Bounded strings have a fixed upper limit on their size. The following IDL declaration represents a SOM IDL string of no more than 100 characters in length:

```
string<100> that;
```

This IDL declaration can be approximated by the following COBOL data declarations:

```
1 that-l-max binary pic 9(9) value 101.  
1 that-l binary pic 9(9).  
1 that.  
2 that-v pic x occurs 1 to 101 depending that-l.
```

The -l suffix denotes the length of the string, the -v its value. The extra position allows for the null terminator. The data item that, in addition to being a valid SOM IDL string, is also variable length in COBOL, because of the OCCURS DEPENDING clause.

For unbounded strings, the maximum length must be inferred from ancillary information about the interface and the semantics of its operations. The following IDL declaration represents an unbounded SOM IDL string:

```
string that;
```

You might, for example, know that the strings that are passed across the interface do not in practice exceed 4095 characters. Then the following COBOL declarations would be appropriate:

```
1 that-l-max binary pic 9(9) value 4096.  
1 that-l binary pic 9(9).  
1 that.  
2 that-v pic x occurs 1 to 4096 depending that-l.
```

You can use a pair of helper routines (see the related references below) to synchronize the two representations, for example, either the bounded or unbounded that, above:

#### **'IDLStringToCOBOL' using that that-l**

This routine sets the COBOL OCCURS object that-l from the position of the mandatory null terminator.

If you prefer, you can achieve the same result in COBOL:

```
Move that-l-max to that-l  
Move zero to tally  
Inspect that tallying tally for characters before x'00'  
Move tally to that-l
```

#### **'IDLStringFromCOBOL' using that that-l**

This routine inserts the null terminator at the string position indicated by the COBOL OCCURS object.

If you prefer, you can do this quite easily yourself in COBOL:

```
Move x'00' to that-v(that-l)
```

#### **RELATED TASKS**

"Passing string arguments" on page 329

#### **RELATED REFERENCES**

"Source code for helper routines" on page 342

## **Complex IDL types**

Although defined in SOM IDL, complex IDL types are rarely found as a type definition or as an argument to or result of an operation.

IDL type	COBOL equivalent
"any"	Group + COBOL type
"array" on page 325	Table
"sequence" on page 325	Group + variable-length table
"struct" on page 326	Group
"union" on page 326	Group + redefinitions

### **any**

The IDL any type is a self-describing representation of any of the IDL types, including another IDL any. The descriptor is mapped to COBOL as a group item, which includes a pointer to the actual data item of the particular type. Suppose you want to map the following IDL declaration:

any that;

In COBOL, this declaration is represented by the following group item:

```
1 that.  
  2 that-type pointer.  
  2 that-value pointer.
```

The that-type field is a pointer to a TypeCode structure whose actual representation is opaque. SOM provides a set of functions to create and interrogate TypeCodes. A simple numeric type code is insufficient to describe an IDL type, because some types have additional information. For example, the type information for an IDL bounded string includes the size of the upper bound.

The that-value field points to the start of the data for the item that the any represents.

### array

IDL arrays map to COBOL tables—groups whose subordinate items contain the OCCURS clause. The underlying IDL type can be any of the IDL types, including array itself, and is mapped according to the rules for the individual IDL type.

A simple instance of the IDL array type is:

```
long that[4][5];
```

This array is represented in COBOL as:

```
1 that.  
  2 occurs 4.  
    3 that-v binary pic s9(9) occurs 5.
```

The -v suffix denotes the individual element values. The unsuffixed name is used to pass the entire array as an argument to a method; the suffixed name is used to refer to individual elements of the array.

If any level of the array contains a struct or union, then you must specify the SYNCHRONIZED clause on the group item. This is to ensure that the subordinate items are aligned on their natural boundaries, in conformance with the default alignment of SOM IDL structures.

### sequence

An IDL sequence is a one-dimensional array with a descriptor that specifies a maximum and current size for the sequence. If the maximum size is explicitly declared, the sequence is said to be *bounded*. Otherwise, the sequence is *unbounded*, and the maximum size is determined at run time (in an application-specific way) and is set prior to passing the sequence to an operation.

There are no restrictions on the element type of a sequence. It is possible to have a sequence of another sequence type.

Here is a simple example, a bounded sequence of IDL type long:

```
sequence<long,10> that;
```

The descriptor for the maximum and current size and address of this sequence is represented in COBOL as a group item:

```
1 that.  
  2 that-maximum binary pic 9(9).  
  2 that-length binary pic 9(9).  
  2 that-buffer pointer.
```

Then the element data is mapped as a variable-length table of the appropriate type, in this case, an array of IDL longs:

```
1 that-t.  
2 that-v comp-5 pic s9(9) occurs 1 to 10  
   depending that-length.
```

## struct

An IDL struct type corresponds to a COBOL group item containing the individually mapped components of the struct as subordinate data items.

Consider the following IDL struct:

```
struct that {  
    long x;  
    double y;  
};
```

The struct could be represented in COBOL as:

```
1 that sync.  
2 x binary pic s9(9).  
2 y comp-2.
```

The SYNCHRONIZED clause is required so that the alignment of the subordinate items approximates the default alignment of SOM IDL structures. In most practical cases, the alignment would be correct either way, but specifying SYNCHRONIZED is a sensible precaution.

## union

A SOM IDL union has a discriminator that indicates which format variant to use. In COBOL, this type is mapped to a group item containing the discriminator, plus the union itself represented by using the REDEFINES clause. Then, in the procedure division, use an EVALUATE statement to determine which format is currently in effect.

Suppose you have the following IDL:

```
union that switch (long) {  
    case 2:char x;  
    case 5:long y;  
    default:float z;  
};
```

The data declaration part of the COBOL mapping could be written as follows:

```
1 that sync.  
2 that-d binary pic s9(9).  
2 that-u display pic x(4).  
2 that-x redefines that-u display pic x.  
2 that-y redefines that-u binary pic s9(9).  
2 that-z redefines that-u comp-1.
```

The SYNCHRONIZED clause makes sure that COBOL mimics the default SOM IDL alignment rules. Thus, in the unlikely event that any IDL structures have “holes,” COBOL would insert slack bytes in the record as appropriate.

The size of the extra member of the union, that-u, is the maximum of the sizes of the explicit union members. This extra data item is needed because of the COBOL restriction that a data item being redefined must be at least as large as the item redefining it. Alternatively, you could declare the union members in order of decreasing size, but that might lose the correspondence between the COBOL declaration and the original IDL.

Whichever style you adopt, you can use an EVALUATE construct such as the following to determine which of the union members is in effect:

```
Evaluate that-d
  When 2
    Display 'case 2:IDL-char: ' that-x
  When 5
    Display 'case 5:IDL-long: ' that-y
  When other
    Display 'default case:IDL-float: ' that-z
End-evaluate
```

## Passing COBOL arguments and return values

The way you write COBOL argument-passing constructs (such as BY REFERENCE or BY VALUE) must comply with the IDL access intent specifiers in, inout, and out.

These specifiers do not correspond exactly to COBOL BY VALUE, BY CONTENT, and BY REFERENCE phrases. The IDL access intent determines only the semantics of the parameter, without necessarily implying a particular mechanism for passing arguments. In COBOL, both BY VALUE and BY CONTENT have input-only semantics but use different mechanisms, whereas BY REFERENCE parameters could have either input-output or output-only semantics, depending on how they are used. Some kinds of output parameters (IDL strings, for example) cannot be expressed directly in COBOL, but must be mapped to pointers.

### Passing literal arguments

For return values and for inout and out arguments, you must pass a data item. For input arguments however, you might be able to specify a literal, passed BY VALUE:

- Integer-valued fixed-point numeric literals and the figurative constant ZERO are formally equivalent to fullword binary data items, and thus match signed or unsigned long IDL types.
- Floating-point literals are formally equivalent to doubleword floating-point (COMPUTATIONAL-2) data items, and thus match the IDL double type.
- Single-byte alphanumeric literals, symbolic characters, and figurative constants other than ZERO match boolean, char, and octet.

You can specify null-terminated literals of the form Z'value' (which match the IDL string type), passed BY CONTENT.

Literal arguments are not supported for the enum type because of the risk of the source getting out of sync with the enumeration list.

### Passing arguments of complex types

For the complex types, not including string, you pass the level-1 group item. In the examples above, this is always the COBOL data name that. Where the conventions expect a pointer, this is set:

- For an argument, prior to executing the INVOKE statement
- For a return value, on return from the method

The rules for passing these types are quite involved. Generally, you provide the storage for all in and inout arguments, all modes of struct and union parameters, and, curiously enough, for out array arguments. The called method allocates some or all of the storage for all other out arguments and return values. You are not allowed to modify this returned storage, though you can of course use it otherwise (to copy it, for example). You must free it using OMMFree when you have finished with it.

If you have to supply inout arguments of any of the complex types, you would do well to allocate the storage dynamically using OMMA1locate, and declare the COBOL equivalent type in the LINKAGE SECTION. This recommendation is because later versions of CORBA allow the called routine to reallocate inout arguments when the output value is inconsistent with the type or size of the input data item. For this reallocation to work, both the caller and the called routine must use a standard memory management protocol.

#### RELATED TASKS

“Passing enumerated arguments” on page 329

“Passing string arguments” on page 329

#### RELATED REFERENCES

“Conventions for passing arguments and return values”

“Source code for helper routines” on page 342

*SOMobjects Developer's Toolkit Programming Guide* (rules for passing arguments of complex types)

### Conventions for passing arguments and return values

IDL type	in	inout/out	Return value
any	content <sup>1</sup>	reference <sup>1</sup>	type <sup>3</sup>
array	content <sup>1</sup>	reference <sup>1</sup>	pointer <sup>4</sup>
boolean	value	reference <sup>1</sup>	type <sup>3</sup>
char	value	reference <sup>1</sup>	type <sup>3</sup>
double	value	reference <sup>1</sup>	type <sup>3</sup>
enum	value	reference <sup>1</sup>	type <sup>3</sup>
float	value	reference <sup>1</sup>	type <sup>3</sup>
long	value	reference <sup>1</sup>	type <sup>3</sup>
object ref	value	reference <sup>1</sup>	type <sup>3</sup>
octet	value	reference <sup>1</sup>	type <sup>3</sup>
pointer	value	reference <sup>1</sup>	type <sup>3</sup>
short	value	reference <sup>1</sup>	type <sup>3</sup>
sequence	content <sup>1</sup>	reference <sup>1</sup>	type <sup>3</sup>
string	content <sup>1</sup>	pointer <sup>2</sup>	pointer <sup>4</sup>
struct	content <sup>1</sup>	reference <sup>1</sup>	type <sup>3</sup>
union	content <sup>1</sup>	reference <sup>1</sup>	type <sup>3</sup>
unsigned long	value	reference <sup>1</sup>	type <sup>3</sup>
unsigned short	value	reference <sup>1</sup>	type <sup>3</sup>

IDL type	in	inout/out	Return value
<ol style="list-style-type: none"> <li>1. For IBM COBOL for OS/390 &amp; VM you can use BY CONTENT or BY REFERENCE only if the argument is not the last. This limitation is due to the System/390 linkage conventions of the high-order bit of the last argument address being set on to indicate the end of the argument list. This convention confuses C and C++ programs that attempt to manipulate the address as a pointer. An alternative to BY REFERENCE (for this situation and in general) is to pass BY VALUE a pointer that has previously been set to the address of the data item.</li> <li>2. For inout and out strings, you must pass a pointer to the string data or buffer BY REFERENCE.</li> <li>3. The term <i>type</i> here means the COBOL equivalent of the IDL type, specified directly in the RETURNING phrase of the INVOKE statement.</li> <li>4. The term <i>pointer</i> here means a COBOL POINTER that has been set to the address of the equivalent data item or output buffer.</li> </ol>			

## Passing enumerated arguments

The access intent of an enum parameter affects the way you refer to its value, not just on the INVOKE statement, but also elsewhere in your program. Consider an operation that expects an enum to be passed in both directions, as an input value and as the operation result:

```
that changeColor(in that hue);
typedef enum that{red, white, blue};
```

To supply a particular color to the operation, you use the corresponding condition-name in a SET statement. For example, to pass the input value white, specify:

```
1 that-input binary pic 9(9).
  88 that-red-input value 1.
  88 that-white-input value 2.
  88 that-blue-input value 3.
  . . .
  Set that-white-input to true
  Invoke anObject 'changeColor' using by value evp that-input
    returning that-output
```

Then, to inspect the returned color, use conditional statements:

```
1 that-output binary pic 9(9).
  88 that-red-output value 1.
  88 that-white-output value 2.
  88 that-blue-output value 3.
  . . .
  Evaluate true
    When that-red-output
      Perform red-stuff
  . . .
  End-evaluate
```

## Passing string arguments

The examples in this section all presume this SOM IDL declaration:

```
string<100> that;
```

**For in string types:** You pass in string types in several ways. Where you know the content, you can specify it as a null-terminated literal, either directly or as the value of a data item:

```
1 that pic x(101) value z'initial value'.
. . .
  Invoke anObject 'method'
```

```

        using by value evp by content z'this or that'
    . . .
    Invoke anObject 'method' using by value evp by content that

```

For variable-content strings, you might find it convenient to use a plain (PICTURE X) alphanumeric data declaration for the string buffer. You can use reference modification if you want to see the valid part of the string:

```

1 that-1 binary pic 9(9).
1 that pic x(101).
. . .
Display 'Content of "that" = "' that(1:that-1) '''

```

However, if you want the string to behave naturally as a variable-length string in both COBOL and the SOM IDL-based library, use the dual representations:

```

1 that-1 binary pic 9(9).
1 that.
2 that-v pic x occurs 1 to 101 depending that-1.

```

You can synchronize either the reference-modified or the OCCURS DEPENDING form of the COBOL string representation with the IDL representation by using the `IDLStringToCOBOL` and `IDLStringFromCOBOL` helper routines.

**For inout and out strings:** You must pass the string buffer with an extra level of indirection. The way that you express the extra level of indirection in COBOL is to pass a pointer that for inout strings has previously been set to the address of the string data. As usual, you have a choice of passing this pointer BY REFERENCE, or declaring a second pointer that you have set to the address of the first and passing this second pointer BY VALUE:

```

1 ptr1 pointer.
1 ptr2 pointer.
1 that-1 binary pic 9(9).
. . .
Linkage section.
1 that.
2 that-v pic x occurs 1 to 101 depending that-1.
. . .
Set ptr1 to address of that
Invoke anObject 'method' using by value evp by reference ptr1 . . .
. . .
Set ptr2 to address of ptr1
Invoke anObject 'method' using by value evp ptr2

```

The extra level of indirection is needed for out strings, because they are allocated by the method and can be arbitrarily long. But the output size of an inout string is limited by the input size: the upper bound for a bounded string; or the actual input size for an unbounded string.

**For a return value:** Specify a pointer, which the method sets to the address of the output string before it returns.

**For all the output modes of string, including inout:** Declare the string buffer itself in the LINKAGE SECTION to allow the method to allocate, or reallocate, the storage for the string. You should acquire storage for an inout string calling `OMMA1locate`, so that (in future) methods can resize the string if necessary.

You can look at the storage by declaring appropriate LINKAGE SECTION data items as usual, but do not attempt to modify it. The storage might be protected, and you cannot assume that you have appropriate write privileges. Instead, make a copy

and modify the copy. Also, you are responsible for freeing the storage for the original returned string when you have finished with it, by calling the `OMMFree` routine.

“Example: passing string arguments”

“Example: using a SOM IDL-based class library” on page 332

#### RELATED REFERENCES

“Source code for helper routines” on page 342

### Example: passing string arguments

This example assumes the following IDL definition:

```
interface this {
  string that (
    in string in_string,
    inout string inout_string,
    out string out_string
  );
};
```

Assuming an arbitrary limit of 99 characters on the string sizes, the following COBOL fragments illustrate the techniques for passing strings. This code is written in a very simple style, does not check for errors, and might not be complete.

Data division.

LOCAL-/WORKING-STORAGE section.

```
1 inout-string-p pointer.
1 out-string-p pointer.
1 return-string-p pointer.
. . .
1 work-string-l binary pic 9(9).
1 inout-string-l binary pic 9(9).
1 out-string-l binary pic 9(9).
1 return-string-l binary pic 9(9).
. . .
1 work-string pic x(100).
1 in-string pic(100) value z'Nothing strange for in strings'.
1 evp pointer.
. . .
```

Linkage section.

```
1 inout-string.
2 inout-string-v pic x occurs 1 to 100 depending inout-string-l.
1 out-string.
2 out-string-v pic x occurs 1 to 100 depending out-string-l.
1 return-string.
2 return-string-v pic x occurs 1 to 100 depending return-string-l.
1 ev.
2 major binary pic 9(9).
88 no-exception value 0.
. . .
```

Procedure division.

```
. . .
* Acquire storage for, and initialize, inout-string:
  Move 100 to inout-string-l
  Call 'OMMAllocate' using content length of inout-string
    returning inout-string-p
  Set address of inout-string to inout-string-p
  Move z'Initial value for inout-string' to inout-string
  Call 'IDLStringToCOBOL' using inout-string inout-string-l
* Invoke method 'that' on an instance of the 'this' class:
  Invoke a-this 'that' using by value evp
    by reference in-string inout-string-p out-string-p
    returning return-string-p
* De-reference the returned pointers and copy one string:
  Set address of inout-string to inout-string-p
```

```

Call 'IDLStringToCOBOL' using inout-string inout-string-1
Set address of out-string to out-string-p
Call 'IDLStringToCOBOL' using out-string out-string-1
Set address of return-string to return-string-p
Call 'IDLStringToCOBOL' using return-string return-string-1
Move out-string to work-string
* Operate on copy, and free allocated storage when done:
Move function reverse(work-string) to work-string
If work-string = out-string then
    Display '"" out-string "" is palindromic.'
End-if
. . .
Call 'OMMFree' using inout-string-p
Call 'OMMFree' using out-string-p
Call 'OMMFree' using return-string-p
. . .

```

## Example: using a SOM IDL-based class library

This example shows the COBOL coding to use a very simple class library.

Let us begin by looking at the documentation for our class library, which provides a bucket class. A bucket is a container that lets you add or remove objects, and that can report the number of objects it contains. Buckets have no special initializer methods, and can thus be created and initialized correctly just by invoking the `somNew` method on the class. Normally, the documentation would define and describe each operation separately, but for this simple example, we will give the complete interface definition of a bucket:

```

interface Bucket {
    readonly attribute unsigned long count;
    void add(in SOMObject element) raises(BucketFull);
    SOMObject remove() raises(BucketEmpty);
};

```

The `raises` clause specifies the exceptions that the operation can incur.

The things that we put into our buckets have no external behavior beyond their existence. That is, they can be created and destroyed, and are identifiable by their object references, but they have no methods or attributes.

The COBOL program in this example shows how you might use this class library. The COBOL coding is very simplistic. For example, it does not check for errors realistically or free all the objects that it creates. But it does cover most of the things that you have to do to start using a class library. It performs the following steps:

1. Create an instance of a bucket.
2. Create and add some things to the bucket.
3. Print the number of things in the bucket.
4. Remove a thing from the bucket.
5. Print the number of things in the bucket.

```

(1) Process pgmname(longmixed)
*****
* Client program for Bucket.
*****
Identification division.
    Program-id. 'TryBucket'.
Environment division.
    Configuration section.
        Repository.
            class thing 'Thing'

```

```

        class bucket 'Bucket'
        class somobject 'SOMObject'.
Data division.
Working-Storage section.
(2)    1 evp pointer.
        1 abucket object reference bucket.
        1 athing object reference thing.
        1 asomobject redefines athing object reference somobject.
        1 cntnts binary pic 9(9).
Linkage section.
(3)    1 ev.
        2 major binary pic 9(9).
        88 no-exception value 0.
        88 any-exception value 1 thru 999999999.
Procedure division.
(4)    display 'Trying Bucket. . .'
        call 'somGetGlobalEnvironment' returning evp
        set address of ev to evp

(5)    invoke bucket 'somNew' returning abucket
        perform 5 times
(6)    invoke thing 'somNew' returning athing
(7)    invoke abucket 'add' using by value evp athing
        perform chkxcp
        end-perform

(8)    invoke abucket '_get_count' using by value evp returning cntnts
        perform chkxcp
        display 'Our bucket now has ' cntnts ' things in it.'
(9)    invoke abucket 'remove' using by value evp returning asomobject
        perform chkxcp
        invoke abucket '_get_count' using by value evp returning cntnts
        perform chkxcp
        display 'We took one out, so now it has only ' cntnts
            ' things in it.'

(10)   invoke abucket 'somFree'
        display 'Done with Bucket.'
        stop run.

chkxcp.
        if any-exception
            display 'An exception occurred; quitting the program!'
            stop run
        end-if.

End program 'TryBucket'.

```

#### Notes:

- (1) Regardless of what you call your program, you need to specify the PGMNAME(LONGMIXED) compiler option to be able to call SOM APIs such as somGetGlobalEnvironment. The option doesn't affect INVOKE statements, but it does apply to program names in CALL statements.

**(2), (3), and (4)**

If not stated otherwise, SOM IDL class libraries use callstyle id1. With this convention, every operation has an implicit *environment pointer* preceding the explicit IDL arguments for the operation. Although this argument is implicit in the IDL, you code it explicitly on your INVOKE statements.

You must, at a minimum, define the environment pointer in the WORKING-STORAGE or LOCAL-STORAGE SECTION. If you want to examine any exceptions that are returned, you must also define the *exception type* in the

LINKAGE SECTION, and set its base address to the value returned by `somGetGlobalEnvironment`. In the example, the exception type field is named `major`.

- (5) The `somNew` method creates an instance of the bucket class, and returns an object reference to the instance. Notice that this method does not take an environment pointer as its first argument.
- (6) Each time through the loop, a new thing is returned in the same variable. This is acceptable for the example, but normally it would be very bad practice to lose addressability to one's objects. Among other reasons, the storage they use remains allocated and, without the object reference, cannot be freed.
- (7) For the methods that correspond to the IDL operations, the environment pointer is included as the first argument, `env`, in the argument list. It's important to check for exceptions after invoking these methods. The ensuing `PERFORM` statement shows one way to do that.
- (8) This statement shows how attributes are mapped to get and set methods. In this case, the attribute is read-only, so only the get method is defined.
- (9) A problem peculiar to container classes is that they must allow arbitrary types for the elements that they contain. Thus the return type of the `remove` operation is specified as a `SOMObject`. We want to use the returned element with its proper description, to assure type safety. But coding a thing as the `RETURNING` value on the `INVOKE` statement would be a type violation. So the returned value, `asomobject`, is specified as a redefinition of `athing`. This redefinition allows the `INVOKE` statement to match the signature of the IDL operation. By using the redefined variable, `athing`, for subsequent operations on the object, we can ensure that these operations are type safe.
- (10) This statement reminds us that all object instances that the program creates should be freed to avoid memory leaks. However, in this example none of the things in the bucket are freed.

---

## Handling errors and exceptions

SOM uses two error or exception mechanisms: `SOMError` and CORBA-style exceptions.

`SOMError` is used for internal errors in the kernel classes, and is not relevant to the average user. Methods of the kernel classes create an object (`somNew` and `somNewNoInit`) or destroy it (`somFree`). The main implication of `SOMError` for these methods is that you do not need to provide an environment argument when you invoke them, and you do not need to check for exceptions after they return.

However you do need to know how to use the SOM exception mechanism, which is used for most other methods. Exceptions are not necessarily errors, but errors *do* use the SOM exception mechanism.

SOM exceptions are not the same as C++ exceptions, but instead set the value of an exception structure, which you can think of as a special kind of return code, accessed through the environment variable.

## Passing environment variables

There are two ways of passing the environment variable, depending on the call style of the method you want to invoke, and check. For each of them, provide a global (per thread) environment variable:

- For callstyle `oidl` methods, there is no explicit environment variable parameter. Such methods use the global environment variable implicitly.
- Callstyle `idl` methods use the same global environment variable, but pass it explicitly, as the first argument to the method.

The environment variable is opaque, except for the exception type field (major) at the beginning of the structure. This is a 4-byte C/C++ enum, origin zero, with three values: `NO_EXCEPTION`, `USER_EXCEPTION`, and `SYSTEM_EXCEPTION`. It is coded in COBOL as `BINARY PIC 9(9)`, with suitable level-88 condition-names.

## Checking the exception type field

Every callstyle `idl` operation (that is, a method whose first parameter is an environment structure) can return one of the standard system exceptions. A callstyle `idl` operation can return a standard system exception even if it does not declare any explicit exceptions with a `raises` expression in the operation declaration. Therefore you must check the exception type field of the environment variable after *every* invocation of a method of a class defined with callstyle `idl`. Do not assume that a method completed successfully. You will not get a reliable implementation of your application unless you do check.

## Handling exceptions

When a callstyle `idl` method that you have invoked detects a condition that is to be expressed as an exception, it uses the `somSetException` function to supply the exception name and an exception structure.

If you decide to handle the exception, perhaps by printing a message and continuing, you must reset the environment variable and free the associated exception structure by using the `somExceptionFree` function. Of course, there are other ways to handle exceptions. You might change the state of one of the input arguments to the method and retry it, or you might terminate your program rather than attempt to continue.

“Example: checking SOM exceptions”

### RELATED REFERENCES

*SOMObjects Developer's Toolkit Programming Guide* (CORBA standard exceptions)

## Example: checking SOM exceptions

The program fragments in this example show in some detail how you can handle SOM exceptions in IBM COBOL. The data names are only suggestions and are not mandatory.

In the `WORKING-STORAGE` or `LOCAL-STORAGE` section:

```
*****  
* Declare the environment variable pointer: *  
*****  
    1 evp pointer.
```

In the `LINKAGE SECTION`:

```

*****
* Declare the environment variable itself:                                     *
*****
1 ev.
2 major binary pic 9(9).
88 no-exception value 0.
88 any-exception value 1 thru 999999999.
88 user-exception value 1.
88 system-exception value 2.

```

In the PROCEDURE DIVISION:

```

*****
* Acquire a global environment variable                                     *
*****
    Call 'somGetGlobalEnvironment' returning evp
    Set address of ev to evp
    . . .
*****
* Check environment after invoking a method                               *
*****
    Invoke anObject 'op1' using by value evp other-args . . .
    If any-exception then
*       respond to exception appropriately, perhaps by using:
        Call 'Print-ev' using evp by content z'op1 on anObject'
    End-if
    . . .

```

Here is a sample subroutine for printing exceptions:

```

*****
* Subroutine for printing exceptions                                     *
*****
Identification division.
    Program-id.
        'Print-ev'.
Data division.
    WORKING-STORAGE section.
        1 counter binary pic 9(9) value 0.
    Local-storage section.
        1 d pic x(130).
        1 eip pointer.
        1 i binary pic 9(9).
        1 p binary pic 9(9).
        1 s pic 9(9).
    Linkage section.
        1 evp pointer.
        1 kind pic x(40).
        1 ev global.
        2 major binary pic 9(9).
        88 user-exception value 1.
        88 system-exception value 2.
        1 ei pic x(100).
    Procedure division using evp kind.
        Add 1 to counter
        Set address of ev to evp
        Call 'SLZ' using counter s i
        Move 1 to p
        String 'Check #' s(i : ) ': method invocation "'
            delimited size into d pointer p
        Move 0 to i
        Inspect kind tallying i for characters before initial x'00'
        String kind(1 : i) '"' returned '
            delimited size into d pointer p
        Evaluate true
            When user-exception
                String 'a user' delimited size into d pointer p
            When system-exception

```

```

        String 'a system' delimited size into d pointer p
    When other
        String 'an unknown' delimited size into d pointer p
    End-evaluate
    Call 'SLZ' using major s i
    String ' exception (major = ' s(i : ) ' )'
        delimited size into d pointer p
    Display d(1 : p - 1)
    Call 'somExceptionId' using by value evp returning eip
    Set address of ei to eip
    Move 0 to i
    Inspect ei tallying i for characters before initial x'00'
    Display ' Exception ID: <' ei(1 : i) '>'
    Call 'somExceptionFree' using by value evp
    Goback
    . . .
End program 'Print-ev'.

*****
* Subroutine to strip leading zeroes *
*****
Identification division.
    Program-id.
        'SLZ'.
Data division.
Linkage section.
    1 uint binary pic 9(9).
    1 str pic x(9).
    1 pos binary pic 9(9).
Procedure division using uint str pos.
    Move uint to str
    Move 0 to pos
    Inspect str(1 : length str - 1)
        tallying pos for leading '0'
    Add 1 to pos
    Goback
    . . .
End program 'SLZ'.

```

---

## Creating and initializing object instances

IBM COBOL directly supports the existing `somInit` and `somUninit` protocols. For classes that use `somInit` (these include all pure COBOL classes), the `somNew` method both creates and initializes an object instance. This technique is appropriate when all instances have the same initial value or do not have an explicit initial value. If, however, you want each instance to have a unique initial value, you might prefer to use a metaclass.

You can execute the nondefault initializer methods (as a client) of a class by invoking first `somNewNoInit` and then the appropriate initializer method. This is the recommended way to create an instance of one of the SOM-enabled collections, for example.

You do need to specify the `somInitCtrl` structure that is used to control the progress of the initializer as it traverses the class hierarchy. For a client of a class initializer method (as opposed to a subclass that provides its own initializer methods), this structure is initially null, represented in COBOL as an OMITTED argument. Suppose that the IDL for the initialization method is:

```
void ISHeap_withNumber(inout somInitCtrl ctrl, in long number);
```

COBOL code for invoking this initializer might be:

```

1 a-heap object reference isheap.
. . .
Invoke isheap 'somNewNoInit' returning a-heap
Invoke a-heap 'ISHeap_withNumber'
using by value evp by reference omitted by value 10000

```

For COBOL subclasses of classes that use explicit initializers, use metaclass methods to instantiate and initialize the COBOL object. After creating the instance, the metaclass method invokes the initializer for each parent (with multiple inheritance, there could be several). It then initializes any instance data that the subclass introduced. You could create and initialize these objects directly in the client code. However, encapsulating the logic in a metaclass method is more reliable and convenient, especially when the subclass inherits from multiple parents.

Using a metaclass method is also a good way to create and initialize your own pure COBOL objects in a single step, particularly where each object has a unique initial value.

---

## Looking at the IDL file

Generally, the documentation for a class library has all the information you need to use (as a client) or specialize (subclass) the classes. In particular, you would expect to find the following information:

- Interfaces (types and operations or methods) expressed in IDL
- Semantics of the operations including the required data types and conventions for passing arguments
- Descriptions of the rules for using the library such as which objects must be instantiated and in what order; what methods must be invoked to initialize the classes; and what the relation is between the classes

Sometimes, you might need more detailed information about a class library, as when you specialize the library by subclassing. To get this additional information, you might need to look at the IDL or header files. It is helpful to know their structure: what is relevant and what you can ignore. Typically, the IDL file consists of some IDL definitions, guarded so that they are processed only once per IDL compilation, plus some implementation-specific information, also guarded so that it is conditionally included.

Consider the sample IDL file spread from the collection class library.

```

#ifndef _ISPRED_IDL (1)
#define _ISPRED_IDL

#include <somobj.idl> (2)

interface ISPredicate : SOMObject { (3)

    boolean evaluateFor (in SOMObject element); (4)

#ifdef __SOMIDL__ (5)
    implementation {
        releaseorder: evaluateFor;

        somDefaultInit: override,init;
        somDestruct:    override;

        callstyle      = idl;
        majorversion    = 1;
        minorversion    = 0;
    }
#endif
}

```

```

        filestem      = spread;
        dllname       = "sccl.dll";
        functionprefix = sISPredicate_;

#ifdef __PRIVATE__ (6)
        passthru C_xh_before = "#include <ssglobal.xih>";
#endif
    };
#endif
};
#endif

```

- (1) One of three conditional sections in the file; its purpose is to ensure that the IDL definitions in the file are processed no more than once during the IDL compilation. The matching `#endif` statement is at the end of the file.
- (2) The `#include` statement incorporates another IDL file that you might have to refer to.
- (3) The compound statement specifies the IDL element that this file defines, the `ISPredicate` interface.
- (4) This definition is for the (single) new operation `evaluateFor` that `ISPredicate` introduces.
- (5) The start of some SOM-specific implementation information, which needn't concern you; its matching `#endif` is the second to last.
- (6) A directive that is needed only by the implementation itself. Again it is not relevant to you, as a client of the class.

---

## Avoiding memory leaks

Typically, many individual object instances are created and destroyed during execution of an object-oriented application. It is important to ensure that, when an object is destroyed or assigned, all of its associated storage is also freed. The source code is provided for two C routines that you can use to allocate and free dynamic storage for data that an object points to, for an `inout` argument to a method, and so on:

- `'OMMAllocate'` using `storage-size` returning a-pointer  
to allocate storage, where `storage-size` is the 4-byte unsigned binary number of bytes to allocate
- `'OMMFree'` using a-pointer  
to free the previously allocated storage element that a-pointer addresses

You must use `OMMFree` to free output storage allocated and returned to you by SOM class libraries.

You can also use these routines to manage dynamic storage (as opposed to instance data) for your own classes.

“Example: COBOL variable-length string class” on page 340

### RELATED TASKS

“Passing COBOL arguments and return values” on page 327

### RELATED REFERENCES

“Source code for helper routines” on page 342

## Example: COBOL variable-length string class

Let's look at an example of a variable-length string class, where the string data is not an explicit part of the instance, but is instead a separate storage area that the instance refers to.

Here's the COBOL definition for the class:

```
*****
* COBOL variable-length string class definition.
*****
Identification division.
  Class-id.
    varstring inherits somobject.
Environment division.
  Configuration section.
    Repository.
      Class varstring 'VarString'
      Class somobject 'SOMObject'
  . . .

*****
* Variable-length string class instance data.
*****
Data division.
  Working-storage section.
    1 vstlen binary pic 9(9).
    1 vstptr pointer.

*****
* Variable-length string class method: default initialization; *
* set the instance to a predictable state.
*****
Identification division.
  Method-id.
    'somInit' override.
Procedure division.
  (1) Set vstptr to null
      Move 0 to vstlen
      Goback
  . . .
  End method 'somInit'.

*****
* Variable-length string class method: assignment from a literal*
*****
Identification division.
  Method-id.
    'SetVarstring'.
Data division.
  Local-Storage section.
    1 strsize pic 9(9) binary.
  Linkage section.
    1 valptr pointer.
    1 setval pic x(100).
    1 vstval pic x(100).
Procedure division using by value valptr.
  (2) If vstptr not = null then
      Call 'OMMFree' using vstptr
    End-if
    Move 0 to vstlen
    Set address of setval to valptr
    Inspect setval tallying vstlen
      for characters before initial x'00'
    Add 1 to vstlen giving strsize
    Call 'OMMAllocate' using strsize returning vstptr
    Set address of vstval to vstptr
```

```

        Move setval(1:strsize) to vstval(1:strsize)
        Goback
    . . .
End method 'SetVarstring'.

*****
* Variable-length string class method: return string (pointer). *
*****
Identification division.
    Method-id.
        'GetVarstring'.
    Data division.
    Linkage section.
        1 valptr pointer.
    Procedure division returning valptr.
        Set valptr to vstptr
        Goback
    . . .
End method 'GetVarstring'.

*****
* Variable-length string class method: assign from another string*
*****
Identification division.
    Method-id.
        'AssignVarstring'.
    Data division.
    Local-Storage section.
        1 strsize binary pic 9(9).
        1 valptr pointer.
    Linkage section.
        1 str object reference varstring.
    Procedure division using by value str.
(3)    If self not = str then
        Invoke str 'GetVarstring' returning valptr
        Invoke self 'SetVarstring' using by value valptr
    End-if
        Goback
    . . .
End method 'AssignVarstring'.

*****
* Variable-length string class method: free associated storage. *
*****
Identification division.
    Method-id.
        'somUninit' override.
    Procedure division.
(4)    If vstptr not = null then
        Call 'OMMFree' using vstptr
        Set vstptr to null
        Move 0 to vstlen
    End-if
        Goback
    . . .
End method 'somUninit'.

End class varstring.

```

- (1) All VarString instances are created in a predictable initial state, with the length set to zero and the string pointer set to null.
- (2) Before a new value is assigned to an instance, the storage allocated for the current value is freed. If this storage were not freed, it would be “orphaned,” causing a memory leak.
- (3) When assigning one string to another, you have to check whether the

sender is identical to the receiver before doing the assignment and thereby prematurely freeing the sender's storage.

- (4) Although `somFree` deallocates the storage for the instance data, it does *not* free storage that the instance refers to. Thus it is critical to free any such storage when the instance is uninitialized.

---

## Source code for helper routines

You can use the following C source to implement the helper functions for representing strings and avoiding memory leaks. You can either statically link the functions into your application or generate a dynamic load library (DLL) for the functions and bind your application to the DLL.

```
/* **** */
/* Helper functions for using SOM IDL-based class libraries.      */
/* **** */

/* OS/390 pragma to generate long, mixed-case names              */
#pragma longname

/* Macro to clear the high-order bit of the argument address      */
#define Clean(p,q) p=(void*)((int)q&0xfffff)
#include <som.h>

/* Object Memory Management: allocate memory.                    */
somToken OMMAAlloc(size_t *size){
    size_t *s;
    Clean(s,size);
    return SOMMAalloc(*s);
}

/* Object Memory Management: free allocated memory.              */
void OMMFree(somToken *ptr){
    somToken *p;
    Clean(p,ptr);
    SOMFree(*p);
    return;
}

/* Set COBOL representation (ODO object) from IDL string length  */
void IDLStringToCOBOL(char *str, long *len) {
    char *s;
    long *l;
    Clean(s,str);
    Clean(l,len);
    (*l)=strlen(s);
    return;
}

/* Set IDL string length (null byte) from COBOL (ODO) representation */
void IDLStringFromCOBOL(char *str, long *len) {
    char *s;
    long *l;
    Clean(s,str);
    Clean(l,len);
    s[*l]=0;
    return;
}
```

---

## Chapter 21. Wrapping or converting procedure-oriented programs

If you use *wrappers* (objects that provide an interface between object-oriented and procedure-oriented code), the object-oriented and procedure-oriented parts of your system can exist quite well together. Certainly, you want to reuse your existing code as long as it continues to meet your needs. You can add new function to your system using object-oriented enhancements.

Only if your existing code no longer meets your needs or its maintenance cost is too high should you consider converting the entire procedure-oriented system to an object-oriented system. Then the conversion entails identifying objects, analyzing the data flow and usage, reallocating code to objects, and writing the object-oriented code.

### RELATED CONCEPTS

“OO view of COBOL programs”

### RELATED TASKS

“Wrapping procedure-oriented programs” on page 344

“Converting from procedure-oriented to OO programs” on page 345

---

## OO view of COBOL programs

Conventional COBOL programs belong to one of three types:

- Batch
- Online
- Subprogram

Batch programs are often constructed to access files or databases or both, and to produce reports. The file or database is the object of the accessing action (UPDATE, INSERT, DELETE), which determines the structure of the program. The report produced by a batch program can be viewed as an object, but the structure of the batch program reflects the structure of the report.

Online programs are built around the transactions which they process, and which are reflected in user interface maps and panels. Online transactions can access several files or databases from one panel. There is a one-to-many relationship between the source of the action and the targets of the action, all of which can be viewed as objects.

Subprograms typically provide a function too large or complex to include in the main program. Some subprograms provide reusable code by implementing general-purpose functions that many programs require. Subprograms can provide actions such as:

- Changing the values of some parameters based on the values of other parameters
- Accessing files or databases
- Printing reports

In the last two cases, the parameter list can be viewed as a message to trigger an action on a file or database object.

---

## Wrapping procedure-oriented programs

You can use wrapping to integrate existing procedure-oriented code with new object-oriented code. Two of the definitions for *wrap* are:

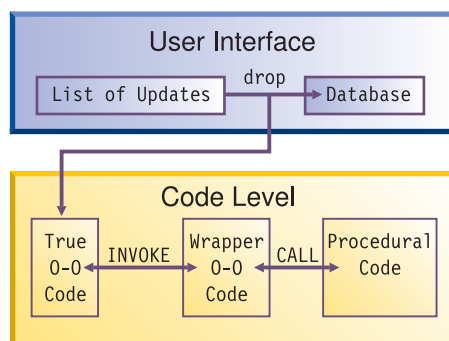
- To enclose as if with a protective covering
- To conceal as if by enveloping

*Wrappers* are objects that provide an interface between object-oriented code and procedure-oriented code. They enclose the procedure-oriented code in a package, concealing its true nature and making it seem like object-oriented code. Wrappers are useful in coordinating actions on the user interface, and integrating procedure-oriented code into OO systems.

## Coordinating procedural code with interface actions

You can use wrappers to integrate old and new code at the user interface or “glass-top” level. As user interfaces move toward an object-oriented approach, they use direct manipulation more. Users can, for example, drag and drop objects onto other objects, and the objects must work together to take the appropriate action, such as updating or printing. If one of the objects is implemented by procedure-oriented code, the wrapper is an interface to this underlying implementation.

Suppose you have a stable set of procedural code for updating a database, but you want to include the database as part of a graphical user interface. You want users to be able to drop a list object representing an update to the database on the database object and have the update performed. To achieve this, you need to write a wrapper class to accept messages from the list object; that is, the list object invokes methods in the wrapper. The methods in the wrapper class interpret the information from the list object and use the CALL statement to call the appropriate subprogram in the old procedural code, as shown in the figure:

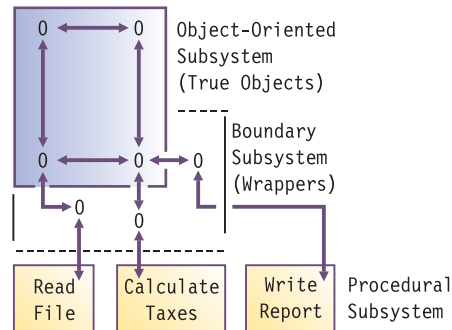


## Integrating procedural code into OO systems

Boundary interface wrappers create objects for procedural code outside the boundaries of the new object-oriented subsystem. These wrappers allow the object-oriented part of the system to use the procedure-oriented part of the system as if it too were object-oriented.

By using wrappers, you can phase in new object-oriented code and continue to use your existing procedure-oriented code. You can, for example, write a wrapper for

each subprogram in the procedural code. Or if several subprograms are working with the same object, processing the same file, or producing the same report, you can write a single wrapper for all the related subprograms. A true object invokes the appropriate method in the wrapper, and the method in turn calls the appropriate subprogram, as shown in the figure:



## Changing procedural code

If you decide to use wrappers, there is one change that you must make to your procedural code. Because methods are always recursive, the following series of events is possible:

1. Method A is invoked and calls program B.
2. While program B is executing, method A is invoked again and calls program B again.

The second call is a recursive call. Therefore, any procedural code that a method invokes should have the RECURSIVE clause on the PROGRAM-ID statement. For example:

```
Identification Division.
Program-Id.  ProgB Recursive.
Environment Division.
. . .
```

### RELATED TASKS

“Converting from procedure-oriented to OO programs”

## Converting from procedure-oriented to OO programs

If you use a typical procedure-oriented COBOL batch or online program as the starting point, your aim is to produce a formal specification of the program in object-oriented form. The conversion involves four steps:

1. “Identifying objects” on page 346: create a list of objects with instance data for each object.
2. “Analyzing data flow and usage” on page 346: create an object relationship table that lists all the inheritance and collaboration relationships between objects.
3. “Reallocating code to objects” on page 347: complete the method definitions.
4. “Writing the object-oriented code” on page 347: write a class definition and client program.

### RELATED CONCEPTS

“OO view of COBOL programs” on page 343

#### RELATED TASKS

“Defining a class” on page 274

“Defining a class method” on page 277

“Defining a client program” on page 285

## Identifying objects

1. Partition the DATA DIVISION into potential objects. Identify every record as an object and every field in the record as its instance data.

For example, you can organize different record structures as follows, using names of your choosing for the first five letters:

Potential objects	Object names
Record structures that define files	FffffFile
Record structures that define database views	VvvvvView
Map or panel record structures	UuuuuInterface
Record structures in the WORKING-STORAGE SECTION not related to files, databases, maps, or panels	WwwwWork
Record structures in the LINKAGE SECTION	PppppParameter

2. Now you have many potential objects, some of which are redundant. Study the potential objects and decide whether two or more are slight variations of the same object.
3. Where possible, combine two potential objects into one object. Maybe you have two detail lines as potential objects that differ in only one or two of their fields. If possible, use REDEFINES or some other technique to combine the two detail lines into one.

As a result of identifying objects, you have an object list with the name of each object and its instance data.

## Analyzing data flow and usage

1. Analyze the file and database accesses to collect the access operations (such as SELECT, UPDATE, INSERT, DELETE, READ, and WRITE) for each object. Use the access sequence to look for the relationships between objects.

For example, if a record is read from the input file and then results in a detail line written to a report, a relationship exists between the file object (the source) and the report object (the target).

2. Trace the data flow between objects to identify the objects that use instance data from another object.

If the two objects have a superclass-subclass (parent-child) relationship, the subclass inherits methods from the superclass and can share instance data through get and set methods defined in the superclass.

If the two objects are separate and distinct, they are *collaborators*. Collaborators do not inherit anything from each other. Instance data that needs to be shared between two collaborators is typically passed by means of parameters on an INVOKE statement.

As a result of analyzing the data flow and usage, you have an object relationship table that lists all the inheritance and collaboration relationships between objects.

#### RELATED TASKS

“Coding special methods” on page 279

## Reallocating code to objects

For each object you identified, collect all references to it from the PROCEDURE DIVISION. Look for procedural code that changes the state of the object's instance data. If a statement affects several data items in different objects, you must change or duplicate the statement to associate the intended actions with the correct objects.

For example:

Move 0 To input-z output-z.

Change this statement to:

Move 0 To input-z.

Move 0 To output-z.

The first MOVE statement is associated with an input object and the second with an output object.

Now couple the procedural statements from the PROCEDURE DIVISION with the objects to form methods. Take the code you pulled from the program and organize it into task-oriented methods.

Refer to the object relationship table from step two (analyzing data flow and usage) and determine if you must write any new methods to facilitate passing data between objects.

The result of this step is completed method definitions.

### RELATED TASKS

"Identifying objects" on page 346

"Analyzing data flow and usage" on page 346

"Defining a class method" on page 277

## Writing the object-oriented code

Write a class definition using the object list from step one (identifying objects) and the methods from step three (reallocating code to objects).

Also, write the client program to create instances of the classes and invoke methods.

Your client program might be a modification of your original procedure-oriented program, but with added method invocations and manipulation of object references where needed. (This is the case when all the procedure-oriented code was not placed into methods.) However, if all the procedure-oriented code was placed into methods, then your client program is a new program that you write from scratch.

### RELATED TASKS

"Identifying objects" on page 346

"Reallocating code to objects"

"Defining a class" on page 274

"Defining a client program" on page 285



---

## Part 5. Working with more complex applications

### Chapter 22. Porting applications between platforms

Getting mainframe applications to compile	351
Choosing the right compiler options.	351
Allowing for language features of mainframe COBOL	351
Using the COPY statement to help port programs.	352
Getting mainframe applications to run: overview	353
Fixing differences caused by data representations	353
ASCII versus EBCDIC	353
Native versus nonnative.	354
IEEE versus hexadecimal	354
EBCDIC DBCS versus ASCII multibyte strings.	355
Fixing environment differences affecting portability	355
File names	355
Control codes	355
Device-dependent control codes	355
Fixing differences caused by language elements	356
Writing code to run on the mainframe	356
Language features supported only by the workstation compiler.	356
Compiler options supported only on the workstation	356
Names supported only on the workstation	357
Differences with THREAD	357
Writing applications that are portable between the workstation and AIX	357

### Chapter 23. Using subprograms

Main programs, subprograms, and calls	359
Ending and reentering main programs or subprograms	360
Calling nested COBOL programs	360
Nested programs	361
Example: structure of nested programs	362
Scope of names.	363
Local names.	363
Global names	363
Searching for name declarations	363
Calling nonnested COBOL programs	364
CALL identifier and CALL literal.	364
Calling between COBOL and C/C/C++ programs	364
Initializing environments	365
Passing data.	365
Setting linkage conventions.	365
Collapsing stack frames and terminating run units or processes	366
Handling exceptions	366
COBOL and C/C/C++ data types	367
Example: COBOL program calling C/C/C++ functions	367

Example: C programs that are called by and call COBOL programs	368
Example: COBOL program called by a C program	370
Example: results of compiling and running examples	370
Making recursive calls	370

### Chapter 24. Sharing data

Passing data.	373
Describing arguments in the calling program	374
Describing parameters in the called program	375
Coding the LINKAGE SECTION	375
Coding the PROCEDURE DIVISION for passing arguments	375
Grouping data to be passed	376
Handling null-terminated strings.	376
Using pointers to process a chained list	377
Example: using pointers to process a chained list	377
Using procedure pointers to pass data	380
Dealing with a Windows restriction	380
Coding multiple entry points	381
Passing return code information	382
Understanding the RETURN-CODE special register	382
Using PROCEDURE DIVISION RETURNING	382
Specifying CALL . . . RETURNING	382
Sharing data by using the EXTERNAL clause.	383
Sharing files between programs (external files)	383
Example: using external files	383
Using command-line arguments	386
Example: command-line arguments with -host option.	387

### Chapter 25. Building dynamic link libraries

Static linking and dynamic linking	389
How the linker resolves references to DLLs	390
Creating DLLs	390
Example: DLL source file and related files.	391
Module definition file	392
Resolving DLL references at run time	392
Resolving DLL references at link time	392
Compiling and linking the DLL	393
Creating object-oriented DLLs	393
Creating module definition files	394
Reserved words for module statements.	395
Summary of module statements	395
BASE	396
DESCRIPTION	396
Example	397
EXPORTS	397
Example	398
HEAPSIZE	398
Example	398

LIBRARY . . . . .	398	Resolving date-related logic problems . . . . .	427
Example . . . . .	399	Using a century window . . . . .	428
NAME . . . . .	399	Example: century window . . . . .	429
Example . . . . .	400	Using internal bridging . . . . .	429
STACKSIZE . . . . .	400	Example: internal bridging . . . . .	430
Example . . . . .	400	Moving to full field expansion. . . . .	430
STUB . . . . .	401	Example: converting files to expanded date	
Example . . . . .	401	form . . . . .	431
VERSION . . . . .	401	Using year-first, year-only, and year-last date fields	432
Example . . . . .	401	Compatible dates . . . . .	433
		Example: comparing year-first date fields . . . . .	434
<b>Chapter 26. Preparing COBOL programs for</b>		Using other date formats . . . . .	434
<b>multithreading . . . . .</b>	<b>403</b>	Example: isolating the year . . . . .	434
Multithreading . . . . .	403	Manipulating literals as dates . . . . .	435
Working with language elements with		Assumed century window . . . . .	436
multithreading . . . . .	404	Treatment of nondates . . . . .	437
Working with elements that have run-unit scope	405	Setting triggers and limits . . . . .	437
Working with elements that have program		Example: using limits . . . . .	438
invocation instance scope . . . . .	405	Using sign conditions . . . . .	439
Scope of COBOL language elements with		Sorting and merging by date . . . . .	439
multithreading . . . . .	405	Example: sorting by date and time . . . . .	440
Choosing THREAD to support multithreading . . . . .	406	Performing arithmetic on date fields. . . . .	441
Transferring control with multithreading . . . . .	406	Allowing for overflow from windowed date	
Controlling the state . . . . .	406	fields . . . . .	441
Ending a program . . . . .	406	Specifying the order of evaluation . . . . .	442
Preinitializing the COBOL environment . . . . .	407	Controlling date processing explicitly . . . . .	443
Handling COBOL limitations with multithreading	407	Using DATEVAL . . . . .	443
Example: using COBOL in a multithreaded		Using UNDATE . . . . .	443
environment. . . . .	408	Example: DATEVAL . . . . .	444
Source code for thr cob.c . . . . .	408	Example: UNDATE . . . . .	444
Source code for subd.cbl. . . . .	410	Analyzing and avoiding date-related diagnostic	
Source code for sube.cbl. . . . .	410	messages . . . . .	444
		Avoiding problems in processing dates . . . . .	446
<b>Chapter 27. National language support . . . . .</b>	<b>411</b>	Avoiding problems with packed-decimal fields	446
Setting the locale . . . . .	411	Moving from expanded to windowed date fields	446
Code page . . . . .	411		
Messages . . . . .	412		
Collating sequence . . . . .	412		
Locales and code sets supported . . . . .	413		
Using DBCS user-defined words and comments	414		
Restrictions on certain user-defined words. . . . .	415		
Declaring DBCS data . . . . .	415		
Specifying DBCS literals. . . . .	416		
Using ALL . . . . .	416		
Comparing literals. . . . .	417		
Controlling the collating sequence . . . . .	417		
DBCS collating sequence . . . . .	417		
ASCII collating sequence . . . . .	417		
Intrinsic functions that are sensitive to collating			
sequence . . . . .	418		
Testing for valid DBCS characters . . . . .	418		
<b>Chapter 28. Preinitializing the COBOL run-time</b>			
<b>environment . . . . .</b>	<b>419</b>		
Initializing persistent COBOL environment . . . . .	419		
Terminating preinitialized COBOL environment	420		
Example: preinitializing the COBOL environment	421		
<b>Chapter 29. Processing two-digit-year dates</b>	<b>425</b>		
Millennium language extensions (MLE) . . . . .	426		
Principles and objectives of these extensions . . . . .	426		

---

## Chapter 22. Porting applications between platforms

Your workstation has a different hardware and operating system architecture than IBM mainframes or AIX workstations. Because of fundamental platform differences, some problems can arise as you move COBOL programs between the workstation, and mainframe environments.

The sections below describe some of the differences between development platforms, and provide instructions to help you minimize portability problems.

### RELATED TASKS

“Getting mainframe applications to compile”

“Getting mainframe applications to run: overview” on page 353

“Writing code to run on the mainframe” on page 356

“Writing applications that are portable between the workstation and AIX” on page 357

### RELATED REFERENCES

“Appendix A. Summary of differences with host COBOL” on page 475

---

## Getting mainframe applications to compile

As you move programs to the workstation from the mainframe, one of your first goals is to get the applications you have already been using to compile in the new environment without errors.

### Choosing the right compiler options

Some mainframe COBOL compiler options are not applicable on the workstation, and are treated as comments.

Two compiler options might yield unpredictable results and the VisualAge COBOL compiler flags them with W-level messages:

**CMPR2** This compiler option affects the interpretation of language elements. The VisualAge COBOL compiler does not support VS COBOL II Release 2 language elements that conflict with the ANSI 85 COBOL standard. A program that depends on the CMPR2 option is not portable.

**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 it such that language specification does not assume a printer control character as the first character of the 01 record for the file.

### Allowing for language features of mainframe COBOL

The following language features are valid under mainframe COBOL but can create errors or unpredictable results in your workstation compilation. Where possible, the following table provides a solution to the potential problem:

Language feature on mainframe	VisualAge COBOL 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.
ASSIGN clause	Uses a different syntax and mapping to the system file name based on ASSIGNMENT name	See the ASSIGN clause ( <i>IBM COBOL 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 OS/390 QSAM supports.
POINTER (defined as 4 bytes) and PROCEDURE-POINTER (defined as 8 bytes) data items	Treats the size of these data items consistently with the native pointer definition (such as 4 bytes for a 32-bit machine)	
RERUN clause	Treats RERUN clause as a comment	
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	User CHAR(EBCDIC) compiler option, because these registers do not apply on the workstation.
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 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

COPY statement (*IBM COBOL Language Reference*)

“Run-time environment variables” on page 140

“Appendix A. Summary of differences with host COBOL” on page 475

---

## Getting mainframe applications to run: overview

After you have downloaded your source program from the mainframe and compiled it on the workstation without errors, the next step is to run the program. In many cases, you can get the same results from the workstation run as from the mainframe COBOL run without greatly modifying the source.

To assess whether to modify your 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 as referenced below.

#### RELATED TASKS

“Fixing differences caused by data representations”

“Fixing environment differences affecting portability” on page 355

“Fixing differences caused by language elements” on page 356

## Fixing differences caused by data representations

COBOL stores USAGE PACKED-DECIMAL data in the same manner on both the mainframe and the workstation, but most other computational data is, by default, represented differently. Most programs act the same on the workstation as on the mainframe regardless of the data representation. To ensure the same behavior for your programs, you should understand the differences in the following ways of representing data and take appropriate action: ASCII and EBCDIC, native and nonnative, IEEE and hexadecimal, and EBCDIC DBCS and ASCII multibyte strings.

### ASCII versus EBCDIC

The workstation uses the ASCII-based character set, and the mainframe uses the EBCDIC character set. Therefore, most characters have a different hexadecimal value. Also, code that depends on the EBCDIC hexadecimal values of character data probably fails when run using ASCII, as shown in the following table:

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
Comparison	Evaluation if ASCII	Evaluation if EBCDIC
'a' < 'A'	False	True
'A' < '1'	False	True
'x' >= '0'	x probably is not a digit	x probably is a digit
x = X'40'	Fails to test whether x is a blank	Successfully tests whether x is a blank

Because of these differences, the results of sorting character strings are different under 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 using PROGRAM COLLATING SEQUENCE IS EBCDIC or the COLLSEQ(EBCDIC) compiler option.

To avoid problems with the different data representation between ASCII and EBCDIC characters, use the CHAR(EBCDIC) compiler option.

### Native versus nonnative

The workstation holds integers 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*, as in “big-end-in.” In other words, the most significant digit of the number is stored first. The workstation representation is known, conversely, as *little-endian*, as in “little-end-in.” On the workstation, the least significant digit of the number is stored first.

For most programs this difference should create no problems. However, if your program depends on the hexadecimal value 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 the program 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 specified.

### IEEE versus hexadecimal

The workstation represents floating-point data using the IEEE format, and the mainframe uses the System/390 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:

Specification	COMP-1 data		COMP-2 data	
	IEEE	Hexadecimal	IEEE	Hexadecimal
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 in porting if your program depends on hexadecimal representation of data.

To avoid most problems with the different representation between IEEE and hexadecimal floating-point data, use the FLOAT(S390) compiler option.

## **EBCDIC DBCS versus ASCII multibyte strings**

Mainframe double-byte character strings (DBCS) are enclosed in shift codes, while workstation multibyte character strings (including DBCS and Extended UNIX Code, EUC) are not enclosed in shift codes. The hexadecimal values used to represent the same characters are also different.

For most programs these differences should not make porting difficult. However, if your program depends on the hexadecimal value of a graphic string or on a character string containing mixed character and graphic data, use caution in your coding practices.

On the workstation, DBCS data can contain single-byte as well as double-byte characters.

“Examples: numeric data and internal representation” on page 36

### **RELATED REFERENCES**

“CHAR” on page 163

“BINARY” on page 161

“FLOAT” on page 176

## **Fixing environment differences affecting portability**

Differences in file names and control codes between workstation and mainframe platforms can also affect the portability of your programs.

### **File names**

File naming conventions on the workstation are very different from those on the mainframe. The following file name, for example, is valid on the workstation but not on the mainframe:

```
/users/joesmith/programs/cobol/myfile.cbl
```

This difference can affect portability if you use file names in your COBOL source programs.

### **Control codes**

Some characters that have no particular meaning on the mainframe are interpreted as control characters on the 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”)

### **Device-dependent control codes**

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 your portable COBOL programs to behave the same way on the workstation that they do on the mainframe. However, be aware of the differences in FILE STATUS values used for I/O processing of input and output.

If your program is written to respond to status key data items, you should be concerned with two issues, depending on whether the program is written to respond to the first status key or the second status key:

- If your program is written to respond to the first file status data item (*data-name-1*), be aware that values returned in the 9X range depend on the platform. If your program relies on the interpretation of a particular 9X value (for example, 97), do not expect the value to have the same meaning on the workstation that it does on the mainframe. Instead, revise your program so that it responds to any 9X value as a generic I/O failure.
- If your program is written to respond 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, VSAM returns values with a different record structure on the mainframe than it does on the workstation. If your program relies on the interpretation of the second file status data item, it is probably not portable.

### RELATED REFERENCES

FILE STATUS clause (*IBM COBOL Language Reference*)

---

## Writing code to run on the mainframe

You can use IBM VisualAge COBOL to write new applications, taking advantage of the productivity gains and increased flexibility of using your workstation. You need to be aware of how to avoid using IBM VisualAge COBOL features not supported by mainframe COBOL.

## Language features supported only by the workstation compiler

IBM VisualAge COBOL supports several language features not supported by mainframe COBOL compilers. As you write code on your workstation that is intended to run on the mainframe, avoid using these features:

- ORGANIZATION LINE SEQUENTIAL
- Format-5 SET statement extension that allows setting of pointers or ADDRESS OF special register to an address of level 01, 02-49, or 77 in the LINKAGE SECTION or the WORKING-STORAGE SECTION
- LOCK MODE IS AUTOMATIC
- ASSIGN USING *data-name*
- READ statement using PREVIOUS phrase
- START statement using < or <= in the KEY PHRASE

## Compiler options supported only on the workstation

A number of compile-time options are available only with IBM VisualAge COBOL. Do not use any of the following options in your source code if you intend to port this code to the mainframe COBOL compiler:

- BINARY
- CALLINT (also a compiler directive)

- CHAR
- ENTRYINT
- FLOAT
- PROBE
- PROFILE

## Names supported only on the workstation

Be aware of the difference in naming conventions supported on the workstation and other file systems. Avoid hard-coding the names of files in your source programs. Instead, use mnemonic names (in turn, mapped to mainframe ddnames or workstation environment variables) that you can define on each platform. 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*)

## Differences with THREAD

Multithreading 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 multithreading environment.

---

## Writing applications that are portable between the workstation and AIX

The workstation and AIX environments are similar, and their language support is almost identical. However, there are two key differences between these platforms that you should keep in mind when developing applications that are portable between the workstation and the AIX workstation.

- Hard-coded file names in your source programs can lead to problems. Instead of hard-coding the names, use mnemonic names so that you can compile your program without having to change the source code. In particular, 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*)
- The workstation represents integers in *little-endian* format. Like the mainframe, AIX workstations maintain integers in *big-endian* format. Therefore, if your workstation 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 workstation-format integers, use the BINARY(S390) compiler option and avoid the USAGE COMP-5 type.



---

## Chapter 23. Using subprograms

Many applications consist of several separately compiled programs linked together.

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 need to transfer control and usually need to have access to common data.

COBOL programs that are nested within each other can also communicate. All the required subprograms for an application can be contained in one program and thereby require only one compilation.

### RELATED CONCEPTS

"Main programs, subprograms, and calls"

### RELATED TASKS

"Calling nested COBOL programs" on page 360

"Calling nonnested COBOL programs" on page 364

Calling between COBOL and C/C/C++ programs ("Calling between COBOL and C/C/C++ programs" on page 364)

---

## Main programs, subprograms, and calls

If a COBOL program is the first program in the 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 a subprogram.

Whether a COBOL program is a main program or a 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 in COBOL terms), 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 called program processing is completed, the 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 360

"Calling nested COBOL programs" on page 360

"Calling nonnested COBOL programs" on page 364

Calling between COBOL and C/C/C++ programs ("Calling between COBOL and C/C/C++ programs" on page 364)

"Making recursive calls" on page 370

---

## Ending and reentering main programs or subprograms

To end execution in the main program, you must use 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 ending the run unit. 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 it is called in the run unit, its internal values will be as they were left, except that return values for PERFORM statements will be 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 with the INITIAL attribute will be in the initial state each time it is called.
- Data defined in the LOCAL-STORAGE SECTION will be in the initial state each time the outermost containing program is called. (For nested programs, LOCAL-STORAGE is in the initial state each time the nested program is called.)

### RELATED CONCEPTS

“Comparison of WORKING-STORAGE and LOCAL-STORAGE” on page 12

### RELATED TASKS

“Calling nested COBOL programs”

“Making recursive calls” on page 370

---

## Calling nested COBOL programs

By calling nested programs you can create applications using structured programming techniques. You can also use them in place of PERFORM procedures to prevent unintentional modification of data items.

Use either the CALL *literal* or CALL *identifier* statement to make calls to nested programs.

You can call a nested program only from its directly nesting program, unless you identify the nested program as COMMON in its PROGRAM-ID clause. 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:

- Use the IDENTIFICATION DIVISION in each program. All other divisions are optional.
- Make the name of a nested program unique. You can use any valid COBOL word or a nonnumeric literal.
- In the outermost program set any CONFIGURATION SECTION options that might be required. Nested programs cannot have a CONFIGURATION SECTION.
- Include each nested program in the nesting program immediately before its End Program header.
- Use an End Program header to terminate nested and nesting programs.

#### RELATED CONCEPTS

“Nested programs”

#### RELATED REFERENCES

“Scope of names” on page 363

CALL statement (*IBM COBOL Language Reference*)

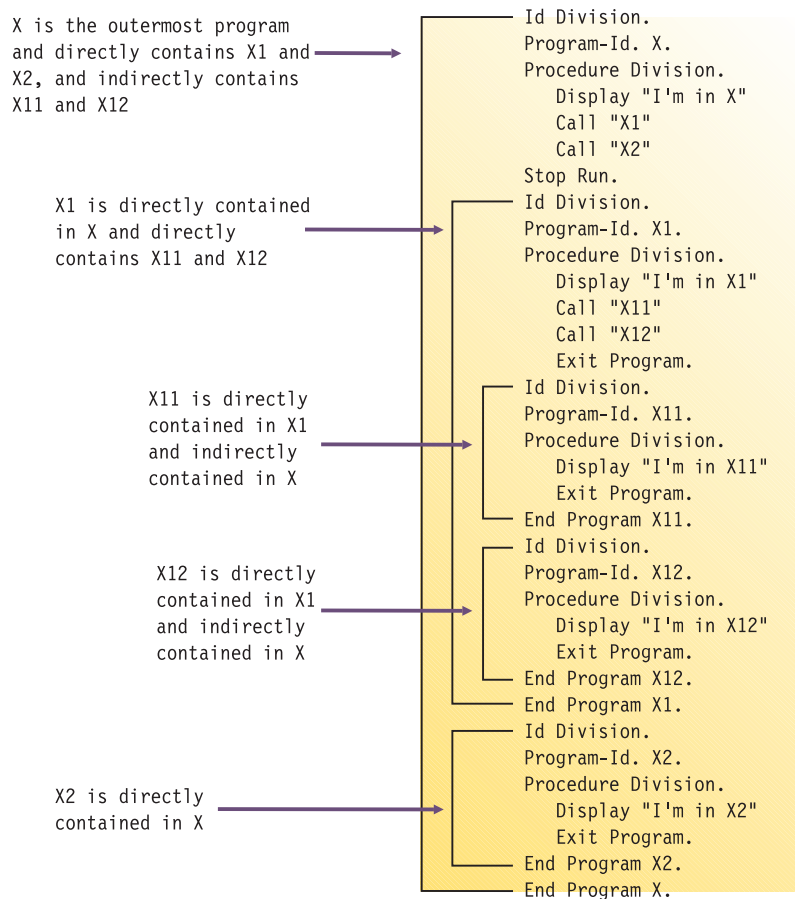
## Nested programs

A COBOL program can “nest,” or contain, other COBOL programs. The nested programs can themselves contain yet other programs. A nested program can be directly or indirectly contained in a program.

There are four main advantages to nesting called programs:

1. Nested programs give you a method to create modular functions for your application and maintain 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.
2. Nested programs allow for debugging a program before including it in the application.
3. Nested programs allow you to compile your application with a single invocation of the compiler.
4. Calls to nested programs have the best performance of all the forms of COBOL CALL statements.

The following example describes a nested program structure with directly and indirectly contained programs:



“Example: structure of nested programs”

#### RELATED TASKS

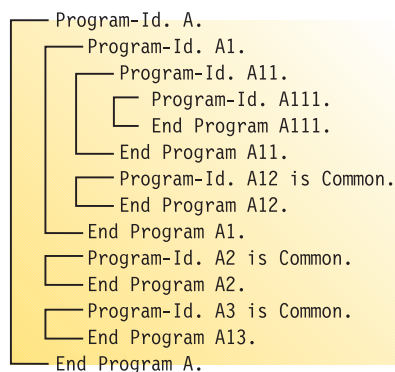
“Calling nested COBOL programs” on page 360

#### RELATED REFERENCES

“Scope of names” on page 363

## Example: structure of nested programs

The following example shows a nested structure with some nested programs identified as COMMON.



The following table describes the calling hierarchy for the structure that is shown in the example above. Programs A12, A2, and A3 are identified as COMMON, and the calls associated with them differ.

This program	Can call these programs	And can be called by these programs
A	A1, A2, A3	None
A1	A11, A12, A2, A3	A
A11	A111, A12, A2, A3	A1
A111	A12, A2, A3	A11
A12	A2, A3	A1, A11, A111
A2	A3	A, A1, A11, A111, A12, A3
A3	A2	A, A1, A11, A111, A12, A2

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 were declared. They are not visible or accessible to contained and containing programs.

### Global names

A name that is global (indicated 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 name of the item.

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, providing 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 of the same containing structure. However, this masking could cause problems during a search for a name declaration.

### Searching 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 containing programs until a match is found. The search follows this process:

1. Declarations in the program are searched first.
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; otherwise, an error exists if no match is found.

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

Your COBOL programs 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). IBM VisualAge COBOL also provides for run-time resolution of the target subprogram from a DLL.

If you link the target program statically, it is part of the executable module of the caller and is loaded with the caller. If you link it dynamically or resolve the call at run time, it is provided in a library and is loaded either when the caller is loaded or when it is called.

Static linking and dynamic linking are done for COBOL *CALL literal* subprograms only. Run-time resolution is always done for COBOL *CALL identifier* and is done for *CALL literal* when the DYNAM option is in effect.

### CALL identifier and CALL literal

The COBOL *CALL identifier*, where *identifier* is a data item that contains the name of a nonnested subprogram when the program is run, always results in the target subprogram being loaded when it is called. Also, the name of the DLL must match the name of the target entry point.

The COBOL *CALL literal*, where *literal* is the explicit name of a nonnested subprogram being called, can be resolved statically or dynamically. 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 the same 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.

#### RELATED CONCEPTS

"Static linking and dynamic linking" on page 389

#### RELATED REFERENCES

*CALL* statement (*IBM COBOL Language Reference*)

---

## Calling between COBOL and C/C/C++ programs

You can call functions written in C or C/C++ from your COBOL programs and vice versa. The following rules and guidelines for applications involving COBOL and C or C/C++ programs describe how to perform such interlanguage calls.

## Initializing environments

When you call C programs in an application where the main program is a COBOL program, you must initialize the C environment. To initialize the C environment, call the C initialization routine `_CRT_init` from either the COBOL program before the first C function is called or from the first C function called by COBOL.

Likewise, call the C termination routine `_CRT_term` when the C environment is no longer required.

If your main program is written in C and makes multiple calls to a COBOL program, you should preinitialize the COBOL environment in your C program. For example, if your C program repeatedly calls a COBOL program to carry out input and output tasks, you will probably want the COBOL program to remain in its last-used state.

## Passing data

Some COBOL data types have C/C/C++ equivalents, but others do not. When you pass data between COBOL and C/C/C++ functions, be sure to restrict data exchange to appropriate data types.

The COBOL default is that arguments are passed BY REFERENCE. If an argument is passed BY REFERENCE, C gets a pointer to the argument. If you pass an argument BY VALUE in the CALL statement, COBOL passes the actual argument. BY VALUE can be used only for the following data types:

- Alphanumeric DISPLAY items
- BINARY
- COMP
- COMP-1
- COMP-2
- COMP-4
- COMP-5
- OBJECT REFERENCE
- POINTER
- PROCEDURE-POINTER

C/C/C++ lets you call a given program with a varying number of parameters, 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. For example, some programs look for a null pointer to signify the end of the parameter list. COBOL does not terminate the parameter list with a null; you can supply it by passing BY VALUE 0 as the last argument.

## Setting linkage conventions

C/C/C++ and COBOL use different default linkage conventions. For calls between COBOL and C/C/C++ programs, the linkage convention must be set to the convention used by the calling program. You can set the linkage convention for your COBOL programs using COBOL compiler directives or compiler options.

Use the `>>CALLINT OPTLINK` compiler directive or `CALLINT(OPTLINK)` compiler option for COBOL programs that call C/C/C++ functions. Use the compiler directive when you want to change the linkage convention for a particular call rather than the entire program.

Use the ENTRYINT(OPTLINK) compiler option for COBOL programs that are called by C/C/C++ functions. This option (not the default) sets the linking convention to that of VisualAge C++.

## Collapsing stack frames and terminating run units or processes

Do not invoke functions in one language that collapse program stack frames of other languages. This guideline includes these situations:

- Collapsing some active stack frames from one language with active stack frames written in another language in the to-be-collapsed stack frames (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 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 longjmp() or COBOL STOP RUN and C exit() or \_exit() calls if the function does not collapse active stack frames of a language other than the one initiating the function.

For the languages not initiating the collapsing and the termination, these adverse effects might occur:

- Normal cleanup or exit functions of the language might not be performed, such as the closing of files by COBOL on a 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 atexit function.

## Handling exceptions

In general, exceptions incurred during the execution of a stack frame are handled according to the rules of the language incurring the exception.

IBM VisualAge COBOL saves the exception environment on entry to the COBOL run-time environment and restores it on termination of the COBOL environment. COBOL expects interfacing languages and tools to follow the same convention.

Because the COBOL implementation does not depend on the interception of exceptions through system services for the support of ANSI COBOL language semantics, you can specify the TRAP(OFF) run-time option to enable the exception handling semantics of the non-COBOL language.

### RELATED CONCEPTS

“Chapter 28. Preinitializing the COBOL run-time environment” on page 419

### RELATED TASKS

“Chapter 24. Sharing data” on page 373

### RELATED REFERENCES

COBOL and C/C/C++ data types (“COBOL and C/C/C++ data types” on page 367)

“Example: C programs that are called by and call COBOL programs” on page 368

Example: COBOL program calling C/C/C++ functions (“Example: COBOL program calling C/C/C++ functions” on page 367)

“TRAP” on page 219

“CALLINT” on page 162

“ENTRYINT” on page 168  
CALLINT compiler directive (*IBM COBOL Language Reference*)

## COBOL and C/C/C++ data types

The following table shows the correspondence between the data types available in COBOL and C/C/C++.

C/C/C++ data types	COBOL data types
wchar_t	DISPLAY-1 (PICTURE N, G)  wchar_t is the processing code whereas DISPLAY-1 is the file code.
char	PIC X.
signed char	No appropriate COBOL equivalent.
unsigned char	No appropriate COBOL equivalent.
short signed int	PIC S9-S9(4) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC(BIN) compiler option.
short unsigned int	PIC 9-9(4) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC(BIN) compiler option.
long int	PIC 9(5)-9(9) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC(BIN) compiler option.
long long int	PIC 9(10)-9(18) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC(BIN) compiler option.
float	COMP-1.
double	COMP-2.
enumeration	Equivalent to level 88, but not identical.
char(n)	PICTURE X(n).
array pointer (*) to type	No appropriate COBOL equivalent.
pointer(*) to function	PROCEDURE-POINTER.

### RELATED REFERENCES

“TRUNC” on page 192

## Example: COBOL program calling C/C/C++ functions

The following sample COBOL program calls C/C/C++ functions and illustrates the following concepts:

- C/C/C++ programs are called using the COBOL CALL statement. The CALL statement does not indicate if the called program is written in COBOL or C/C/C++.
- COBOL supports calling programs with mixed-case names.
- Arguments can be passed to C/C/C++ programs in different ways (for example, CALL BY REFERENCE, CALL BY VALUE).
- You must declare a function return value on the CALL statement that calls a C/C/C++ function.
- You must map COBOL data types to appropriate C/C/C++ data types.

```
CBL PGMNAME(MIXED) CALLINT(OPTLINK)
* These compiler options allow for
* case-sensitive names for called programs
* and set the call interface/linking
* convention to that of the IBM C/C++ default.
```

```

*
IDENTIFICATION DIVISION.
PROGRAM-ID. "COBCALLC".
*
DATA DIVISION.
WORKING-STORAGE SECTION.
*
01 N4          PIC 9(4)  COMP-5.
01 NS4         PIC S9(4) COMP-5.
01 N9          PIC 9(9)  COMP-5.
01 NS9         PIC S9(9) COMP-5.
01 NS18        USAGE COMP-2.
01 D1          USAGE COMP-2.
01 D2          USAGE COMP-2.
01 R1.
    02 NR1      PIC 9(8) COMP-5.
    02 NR2      PIC 9(8) COMP-5.
    02 NR3      PIC 9(8) COMP-5.
PROCEDURE DIVISION.
*
* Initialize C environment
*
    CALL "initC".
*
    MOVE 123 TO N4
    MOVE -567 TO NS4
    MOVE 98765432 TO N9
    MOVE -13579456 TO NS9
    MOVE 222.22 TO NS18
    DISPLAY "Call MyFun with n4=" N4 " ns4=" NS4 " N9=" n9
    DISPLAY "          ns9=" NS9 " ns18=" NS18
*
* The following CALL illustrates several ways to pass
* arguments.
*
    CALL "MyFun" USING N4 BY VALUE NS4 BY REFERENCE N9 NS9 NS18
    MOVE 1024 TO N4
*
* The following CALL returns the C function return value.
*
    CALL "MyFunR" USING BY VALUE N4 RETURNING NS9
    DISPLAY "n4=" N4 " and ns9= n4 times n4= " NS9
    MOVE -357925680.25 TO D1
    CALL "MyFunD" USING BY VALUE D1 RETURNING D2
    DISPLAY "d1=" D1 " and d2= 2.0 times d2= " D2
    MOVE 11111 TO NR1
    MOVE 22222 TO NR2
    MOVE 33333 TO NR3
    CALL "MyFunV" USING R1
*
* Terminate C environment
*
    CALL "termC".
    STOP RUN.

```

#### RELATED REFERENCES

CALL statement (*IBM COBOL Language Reference*)

## Example: C programs that are called by and call COBOL programs

The following sample C program illustrates that a called C function receives arguments in the order in which they were passed in the COBOL CALL statement.

The file MyFun.c contains the following C source code, which calls the COBOL program tprog1.cbl.

```
#include <stdio.h>

extern int _CRT_init(void);
extern void _CRT_term(void);
extern void TPROG1(double *);

void
initC(void)
{
    int rc;

    rc = _CRT_init();
    setbuf(stdout, NULL);
    if (rc)
        printf("Error occurred during C initialization\n");
}

void
termC(void)
{
    _CRT_term();
}

void
MyFun(short *ps1, short s2, long *k1, long *k2, double *m)
{
    double x;

    x = 2.0*(*m);
    printf("MyFun got s1=%d s2=%d k1=%d k2=%d x=%f\n",
           *ps1, s2, *k1,
           *k2, x);
}

long
MyFunR(short s1)
{
    return(s1 * s1);
}

double
MyFunD(double d1)
{
    double z;

    /* example of C calling COBOL */
    z = 1122.3344;
    (void) TPROG1(&z);
    /* example of C returning a value to COBOL */
    return(2.0 * d1);
}

void
MyFunV(long *pn)
{
    printf("MyFunV got %d %d %d\n", *pn, *(pn+1), *(pn+2));
}
```

MyFun.c consists of the following functions:

**MyFun** Illustrates passing a variety of arguments.

**MyFunR** Illustrates how to pass and return a long variable.

**MyFunD** Illustrates C calling a COBOL program and it also illustrates how to pass and return a double variable.

**MyFunV** Illustrates passing a pointer to a record and accessing the items of the record in a C program.

## Example: COBOL program called by a C program

The following example shows how to write COBOL programs that are called by C programs.

The file TPROG1.CBL is called by the C function MYFUND in the C program MyFun.c (see “Example: C programs that are called by and call COBOL programs” on page 368).

```
*
* Sets the calling convention to that of IBM C/C/C++
*
CBL ENTRYINT(OPTLINK)
*
IDENTIFICATION DIVISION.
PROGRAM-ID. TPROG1.
*
DATA DIVISION.
LINKAGE SECTION.
*
01 X                      USAGE COMP-2.
*
PROCEDURE DIVISION USING X.
    DISPLAY "TPROG1 got x= " X
    GOBACK.
```

### RELATED TASKS

Calling between COBOL and C/C/C++ programs (“Calling between COBOL and C/C/C++ programs” on page 364)

## Example: results of compiling and running examples

Compile and link the COBOL programs COBCALLC.CBL and TPROG.CBL and the C program MyFun.c and run COBCALLC using the following commands:

1. `icc -c MyFun.c`
2. `cob2 cobcallc.cbl MyFun.out tprog1.cbl -o cobcallc`  
Run the program by entering COBCALLC.

The results are as follows:

```
call MyFun with n4=00123 ns4=-00567 n9=0098765432
      ns9=-0013579456 ns18=.2222200000000000E 03
MyFun got s1=123 s2=-567 k1=98765432 k2=-13579456 x=444.440000
n4=01024 and ns9= n4 times n4= 0001048576
TPROG1 got x= .112233440000000000E+04
d1=-.357925680250000000E+09 and d2= 2.0 times d2= -.715851360500000000E+09
MyFunV got 11111 22222 33333
```

### RELATED REFERENCES

Example: COBOL program calling C/C/C++ functions (“Example: COBOL program calling C/C/C++ functions” on page 367)

“Example: C programs that are called by and call COBOL programs” on page 368

“Example: COBOL program called by a C program”

---

## 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 on 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 coded on its `PROGRAM-ID` paragraph, the run unit will end abnormally.

#### RELATED TASKS

“Identifying a program as recursive” on page 6

#### RELATED REFERENCES

`RECURSIVE` attribute (*IBM COBOL Language Reference*)

“`THREAD`” on page 191



---

## Chapter 24. 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 to have access to common data.

This material describes how you can write programs that can share data with other programs. For the purposes of this discussion, a *subprogram* is any program called by another program.

### RELATED TASKS

“Passing data”

“Coding the LINKAGE SECTION” on page 375

“Coding the PROCEDURE DIVISION for passing arguments” on page 375

“Using procedure pointers to pass data” on page 380

“Coding multiple entry points” on page 381

“Passing return code information” on page 382

“Specifying CALL . . . RETURNING” on page 382

“Sharing data by using the EXTERNAL clause” on page 383

“Sharing files between programs (external files)” on page 383

“Using command-line arguments” on page 386

---

## Passing data

You can choose among three ways of passing data between programs:

### BY REFERENCE

The subprogram refers to and processes the data items in storage of the calling program rather than working on a copy of the data.

### BY CONTENT

The calling program passes only the contents of the *literal*, or *identifier*. With a CALL . . . BY CONTENT, the called program cannot change the value of the *literal* or *identifier* in the calling program, even if it modifies the variable 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 in the called program or invoked method. However, because the subprogram or method has access only to a temporary copy of the sending data item, these changes do not affect the argument in the calling program.

Determine which of these three data-passing methods to use based on what you want your program to do with the data:

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.

Code	Purpose	Comments
CALL . . . BY CONTENT ADDRESS OF <i>record-name</i> .	To pass the address of a record area to a called program	The subprogram receives the ADDRESS special register for the record-name that you specify.  You must define the record-name as a level-01 or level-77 item in the LINKAGE SECTION of the called and calling programs. The compiler provides a separate ADDRESS special register for each record in the LINKAGE SECTION.
CALL . . . BY CONTENT <i>identifier</i> .	To prevent the contents of the argument of the CALL statement in the calling program and the contents of the parameter in the called subprogram from sharing the same memory	
CALL . . . BY CONTENT <i>literal</i> .	To pass a literal value to a called program	The called program cannot change the value of the literal.
CALL . . . BY CONTENT LENGTH OF <i>identifier</i> .	To pass 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 its length to a subprogram	
CALL . . . BY VALUE	To pass data to a C program that expects the value of the argument	Parameters must be of certain data types to be passed BY VALUE.
CALL . . . BY REFERENCE	To pass data to a C program that expects a reference (pointer) to the argument and that you want to modify the value of the argument	
CALL . . . BY CONTENT	To pass data to a C program that expects a reference (pointer) to the argument and that you do <i>not</i> want to modify the value of the argument	
CALL . . . RETURNING	To call a C/C++ function with a function return value	

## 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. Describe these data items in the LINKAGE SECTION of all the programs that it calls directly or indirectly.

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 and are described in the LINKAGE sections 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 clause of the CALL statement to pass the arguments.

## Describing parameters in the called program

You must know what is being passed from the calling program and describe it in the LINKAGE SECTION of the called program.

Code the USING clause after the PROCEDURE-DIVISION header to receive the parameters.

### RELATED TASKS

“Specifying CALL . . . RETURNING” on page 382

“Sharing data by using the EXTERNAL clause” on page 383

“Sharing files between programs (external files)” on page 383

### RELATED REFERENCES

CALL statement (*IBM COBOL Language Reference*)

INVOKE statement (*IBM COBOL Language Reference*)

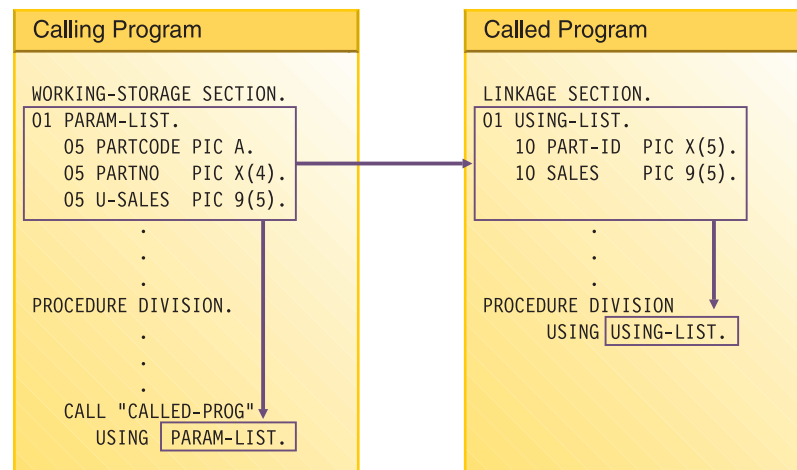
---

## Coding the LINKAGE SECTION

Code the same number of *data-names* in the *identifier* list of the calling program as the number of *data-names* in the *identifier* list of the called program. Synchronize them by position because the compiler passes the first *identifier* of 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 *data-names* in the *identifier* list of 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.



In the calling program, the code for parts (PARTCODE) and the part number (PARTNO) are referred to separately. In the called program, by contrast, the code for parts and the part number are combined into one data item (PART-ID). A reference in the called program 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, use the USING BY VALUE clause on the PROCEDURE DIVISION header of the subprogram:

```
PROCEDURE DIVISION USING BY VALUE
```

If you pass an argument BY REFERENCE or BY CONTENT, you do not need to indicate how the argument was passed on the PROCEDURE DIVISION.

You can code the header in either of the following ways:

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

#### RELATED REFERENCES

The procedure division header (*IBM COBOL Language Reference*)

## Grouping data to be passed

Consider grouping all the data items you want to pass between programs and putting them under one level-01 item. If you do this, you can pass a single level-01 record between programs.

To make the possibility of mismatched records even smaller, put the level-01 record into a copy library and copy it in 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 375

## Handling null-terminated strings

COBOL supports null-terminated strings when you use null-terminated literals, the hexadecimal literal X'00', and string-handling verbs.

You can manipulate null-terminated strings (passed from a C program, for example) by using string-handling mechanisms such as those shown for 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 display a null-terminated string, you can inspect N by tallying N-length for characters before the initial X'00':

```
Display 'N: ' N(1:N-length) ' Length: ' N-length
```

To move a null-terminated string to alphanumeric and strip null:

```
Unstring N    delimited by X'00'
              into X
```

To create a null-terminated string:

```
String Y      delimited by size
              X'00' delimited by size
              into N.
```

To concatenate two null-terminated strings:

```
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 83

#### RELATED REFERENCES

Null-terminated nonnumeric literals (*IBM COBOL Language Reference*)

## Using pointers to process a chained list

When you want to pass and receive addresses of record areas, you can manipulate pointer data items, which are a special type of data item to hold addresses. Pointer data items are data items that either 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 where 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 pointer items. You can assign the value `NULL` to a pointer data item in 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, the code to check for the end of the list is as follows:

```
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 375

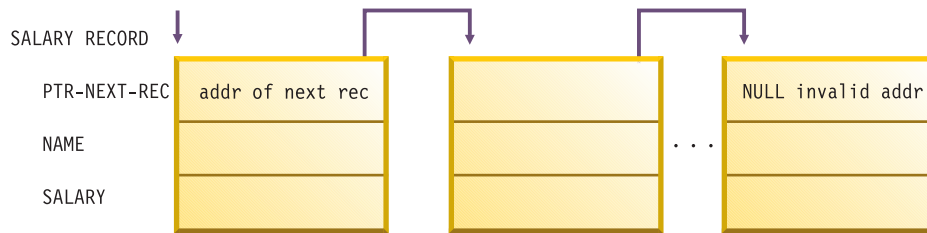
“Coding the `PROCEDURE DIVISION` for passing arguments” on page 375

#### RELATED REFERENCES

`SET` statement (*IBM COBOL Language Reference*)

### Example: using pointers to process a chained list

For this example, picture a chained list of data that consists of individual salary records. The following figure shows one way to visualize how these records are linked in storage:



The first item in each record points to the next record, except for the last record. The first item in the last record contains a null value instead of an address, to indicate that it is the last record.

The high-level logic of an application that processes these records might look as follows:

```

OBTAIN ADDRESS OF FIRST RECORD IN CHAINED LIST FROM ROUTINE
CHECK FOR END OF THE CHAINED LIST
DO UNTIL END OF THE CHAINED LIST
  PROCESS RECORD
  GO ON 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.
77 DEPT-TOTAL     PIC 9(4) VALUE IS 0.
*****
LINKAGE SECTION.
01 SALARY-REC.
   02 PTR-NEXT-REC  POINTER.
   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
    SET ADDRESS OF SALARY-REC TO PTR-FIRST
*****
    PERFORM WITH TEST BEFORE UNTIL ADDRESS OF SALARY-REC = NULL
*****
    IF DEPT = DEPT-X
      THEN ADD SALARY TO DEPT-TOTAL
      ELSE CONTINUE
    END-IF
    SET ADDRESS OF SALARY-REC TO PTR-NEXT-REC
  
```

```

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 amiss with the call, however, 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 377

---

## Using procedure pointers to pass data

Procedure pointers are data items defined with the `USAGE IS PROCEDURE-POINTER` clause. You can set procedure-pointer 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 procedure-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 your procedure-pointer item to an entry address in a load module called using `CALL identifier` and your program later cancels that called module. Your procedure-pointer item becomes undefined, and reference to it thereafter is not reliable.

### RELATED REFERENCES

`PROCEDURE-POINTER` phrase (*IBM COBOL Language Reference*)

`SET` statement (*IBM COBOL 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 if the calls have any arguments. This restriction is due to the associated convention for forming names (also 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 would be `_abc@8`. You cannot set a procedure 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 on 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 to programs with arguments, made using a procedure pointer, 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 do the link.

If the program being called is 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.

---

## Coding multiple entry points

You cannot always use the call interface convention SYSTEM (STDCALL) for calling programs with 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, using the STDCALL convention causes unpredictable results due to corruption of the stack.

The STDCALL 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 with 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 STDCALL linkage uses four bytes on the stack for each argument, differences in data type are immaterial.

---

## Passing return code information

Use the RETURN-CODE special register to pass and receive 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 on the PROCEDURE DIVISION header in your 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 of the program returning to the caller.

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 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 option. 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 on the PROCEDURE DIVISION header of your program to return information to the calling program:

```
PROCEDURE DIVISION RETURNING dataname2
```

When the called program 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 functions in C/C++ or to subroutines in COBOL.

It 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 COBOL 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 on the 01-level data description in the WORKING-STORAGE SECTION of your program or method. The following rules apply:

- Items that are subordinate to an EXTERNAL group item are themselves EXTERNAL.
- You cannot use the name for the 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 containing the item can access and process the data item. For example, suppose program A has the following data description:

```
01 EXT-ITEM1    EXTERNAL    PIC 99.
```

Program B could access that data item by having the identical data description in its WORKING-STORAGE SECTION.

Remember, any program that has access to an EXTERNAL data item can change its value. Do not use this clause for data items that you need to protect.

---

## Sharing files between programs (external files)

Use the EXTERNAL clause for files to allow separately compiled programs or methods in the run unit to access common files. 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 all programs that check the same file status field, code the EXTERNAL clause on the level-01 data definition for the file status field in each program.

Using external files has these benefits:

- Your main program can reference the record area of the file, although the main program does not contain any input or output statements.
- 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 REFERENCES

EXTERNAL clause (*IBM COBOL Language Reference*)

## Example: using external files

The table below describes the main program and subprograms used in the example that follows.

Name	Function
ef1	The main program, which calls all the subprograms and then verifies the contents of a record area

Name	Function
eflopeno	Opens the external file for output and checks the file status code
eflwrite	Writes a record to the external file and checks the file status code
eflopeni	Opens the external file for input and checks the file status code
eflread	Reads a record from the external file and checks the file status code
eflclose	Closes the external file and checks the file status code

Additionally, COPY statements ensure that each subprogram contains an identical description of the file.

Each program in the example declares a data item with the EXTERNAL clause in its WORKING-STORAGE SECTION. This item is used for checking file status codes and is also placed using the COPY statement.

Each program uses three copy library members:

- The first is named efselect and is placed in the FILE-CONTROL paragraph.  

```
Select efl
Assign To efl
File Status Is efs1
Organization Is Sequential.
```
- The second is named effile and is placed in the FILE SECTION.  

```
Fd efl Is External
    Record Contains 80 Characters
    Recording Mode F.
01 ef-record-1.
02 ef-item-1      Pic X(80).
```
- The third is named efwrkstg and is placed in the WORKING-STORAGE SECTION.  

```
01 efs1          Pic 99 External.
```

### Input-output using external files

```
Identification Division.
Program-Id.
    efl.
*
* This is the main program that controls the 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 efl.
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 efl
    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 your main programs using the `cob2` command. The operating system calls main programs with a string containing the arguments. How these are treated depends on whether you use the `-host` option.

If you specify the `-host` compiler option, Windows calls all main programs with a string in EBCDIC, which gives the command-line arguments. The length of the string is in *big-endian* format.

“Example: command-line arguments with -host option”

## Example: command-line arguments with -host option

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 following program:

```
cob2 -host testarg.cbl
```

```
testarg 95
```

The result is:

```
parm-len=002  
parm-string='95'  
case ninety five
```



---

## Chapter 25. Building dynamic link libraries

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.

You can use a dynamic link library (DLL) as a library of frequently used functions. Even if the DLL holds only one function, the purpose of the DLL is to provide that function as needed to calling programs.

In COBOL terms, a DLL is a collection of outermost programs. Although these outermost programs might contain nested programs, programs external to the DLL can call only the outermost programs (known as entry points). Just as you can compile and link several COBOL programs together as a single executable (.EXE), you 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 is referred to as a *subprogram* in the DLL. Even if a DLL provides only one program, that program is considered a subprogram of the DLL.

### RELATED CONCEPTS

“Static linking and dynamic linking”

“How the linker resolves references to DLLs” on page 390

### RELATED TASKS

“Creating DLLs” on page 390

“Creating object-oriented DLLs” on page 393

---

## Static linking and dynamic linking

Static linking occurs when a calling program is linked to a called program module, in a single executable module. 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. When the program is loaded, the operating system places into memory a single file that contains the executable code and data.

The primary advantage of static linking is that you can 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 duplicate copies of programs to be loaded in memory if more than one calling program needs to access them.

To overcome these disadvantages, use dynamic linking.

Dynamic linking allows several programs to use a single copy of an executable module. The executable module is completely separate from the programs that use it. You can build several subprograms into a DLL, and 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 run-time DLLs and with your own COBOL DLLs.

**RELATED CONCEPTS**

“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 your program 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 389

**RELATED TASKS**

“Creating DLLs”

---

## Creating DLLs

A DLL is built using compiled source code and a module definition (.DEF) file or export (.EXP) file.

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, imports, 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 and the IMPORTS statement to specify where the DLL programs are that your program needs.

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 run-time 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 your 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 (as in -dll:TEST).

“Example: DLL source file and related files”

RELATED CONCEPTS

“Static linking and dynamic linking” on page 389

“How the linker resolves references to DLLs” on page 390

RELATED TASKS

“Creating object-oriented DLLs” on page 393

“Creating module definition files” on page 394

## 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.cbl mydll.def
cob2 mydll.cbl -dll:mydll
```

The following cob2 command will build the program RTDLLRES.EXE, which calls the DLL subprogram MYDLL with run-time resolution:

```
cob2 rtdllres.cbl
```

The following cob2 command will build the program LTDLLRES.EXE, which calls the DLL subprogram MYDLL with link-time resolution:

```
cob2 ltdllres.cbl mydll.lib
```

---

## Creating object-oriented DLLs

COBOL supports the use of class programs as DLLs.

You create class program DLLs the same as you create procedural program DLLs, with one exception. Class program DLLs require a different set of entries in the EXPORTS section of the module definition file for the DLL.

To build the definition file for a class:

1. Compile the COBOL class with the IDLGEN compiler option.
2. Compile the IDL with the def emitter of the SOM compiler.

For example:

```
cob2 -qidlgen ClassA.cbl
sc -sdef ClassA.idl
```

Identify your class program as a SOM subclass in the REPOSITORY paragraph of the ENVIRONMENT DIVISION. In the .DEF file for the DLL, export the name of the class program exactly as it is identified in the REPOSITORY paragraph, with three required strings appended to the end of the class name:

```
class-nameNewClass
class-nameClassData
class-nameCClassData
```

Because the export specification is case sensitive, specify the class name and required strings in correct mixed case.

For example, suppose you have two class programs, CLASSA.CBL and CLASSB.CBL, specified as ClassA and ClassB in their respective REPOSITORY paragraphs. If you want to compile and link these two class programs as a single DLL, you need to specify the following items in the EXPORTS section of the module definition file of the DLL:

```
EXPORTS
    ClassANewClass
    ClassAClassData
```

```
ClassACClassData  
ClassBNewClass  
ClassBCClassData  
ClassBCCClassData
```

A variation on each class name is exported verbatim, with three different strings appended to the class name. In the case of `ClassA`, the resulting export statement includes:

```
ClassANewClass  
ClassACClassData  
ClassACClassData
```

#### RELATED TASKS

“Creating DLLs” on page 390

---

## Creating module definition files

A module definition file contains one or more module statements. You use these statements to:

- Define attributes of your executable output file
- Define attributes of code and data segments in the file
- Identify data and functions that are imported into or exported from your 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 link a file with a DLL without using an import library. You can use the `IMPORTS` module statement to define imports, instead of linking to an import library to resolve references to 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, follow these rules:

- 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 reserved words 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” on page 395

“Summary of module statements” on page 395

## Reserved words for module statements

The following words cannot be used as text parameters to a module statement. 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.

The words are either module definition keywords or reserved by the linker.

Although module definition keywords should always be entered in uppercase letters and only the uppercase forms are shown below, the mixed-case and lowercase forms of these words are also reserved. For example, `CONTIGUOUS`, `ContiGuous`, and `contiguous` are all reserved.

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

Linker module statements can be used to create module definition files.

The following table shows the linker module statements summary. Default parameters are underlined. The defaults for `NONE|SINGLE|MULTIPLE`, `SHARED|NONSHARED`, `INITGLOBAL|INITINSTANCE`, and `TERMGLOBAL|TERMINSTANCE` are described in the detailed description of the option.

Statement	Description	Parameters
"BASE" on page 396	Set preferred loading address.	Loading address
"DESCRIPTION" on page 396	Describe the executable.	Descriptive text

Statement	Description	Parameters
"EXPORTS" on page 397	Define exported functions and data.	Entry name Internal name Ordinal position DECORATED CONSTANT Parameter size
"HEAPSIZE" on page 398	Specify local heap size.	Virtual stack size Initial physical memory
"LIBRARY" on page 398	Identify output as dynamic link library (DLL). See detailed description for defaults of parameters.	Library name Loading address
"NAME" on page 399	Identify output as executable (EXE).	Application name Loading address
"STACKSIZE" on page 400	Specify local stack size.	Virtual stack size Initial physical memory
"STUB" on page 401	Add DOS executable file to module.	File name to add
"VERSION" on page 401	Add string to executable.	Version number to add

## BASE

»— BASE — = — *address* —«

Use the BASE statement to specify the preferred load address for the first load segment of the module. The number you give for the option is rounded up to the nearest multiple of 64K. The second load segment is then loaded at the next available multiple of 64K, and so on.

If the module's load segments cannot be loaded beginning at this preferred address, then the preferred address is ignored and the load segments are loaded according to the internal relocation records retained in the file data.

For .EXE files, accept the default base address of 64K (BASE=0x00010000). Any other address will result in a warning, and 64K will be used anyway.

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

»— DESCRIPTION — '*text*' —«

Use the DESCRIPTION statement to insert the specified text into the .EXE or .DLL file you are creating. The DESCRIPTION statement is useful for embedding source control or copyright information into your program or DLL.

The inserted text must be a one-line string enclosed in single quotation marks.

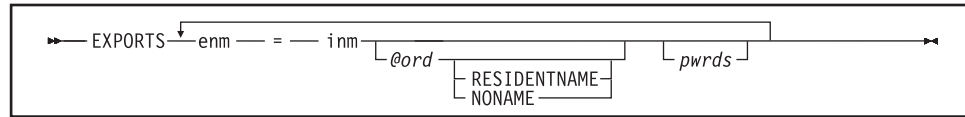
## Example

Given the following line in a .DEF file:

```
DESCRIPTION 'Template Program'
```

the linker inserts the text Template Program into the .EXE or .DLL file.

## EXPORTS



Use the EXPORT statement when you are creating a dynamic link library (DLL) to define the names and attributes of data and functions exported from the DLL, and of functions that run with I/O hardware privilege.

Exported data and functions are those available to other .EXE or .DLL files. Data and functions that are not exported can only be accessed within your DLL, and cannot be accessed by other .EXE or .DLL files.

Give export definitions for functions and data in your DLL that you want to make available to other .EXE or .DLL files.

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.
- inm*** The internal name of the data construct or function, which is its actual name as it appears within the DLL. 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 may save space.

You can specify one of two values:

### RESIDENTNAME

Indicates that you want the data construct or function's name kept resident in memory at all times. You only need to specify RESIDENTNAME if you gave an ordinal position in *ord*.

**NONAME** Indicates that you want the data construct or function to always be referenced by its ordinal number. If you specify NONAME, the data construct or function cannot be referenced by name: it can only be referenced by ordinal number.

You cannot specify both values.

***pwrds*** The total size of the function's parameters, as measured in words (bytes divided by two). This field is required only if the function executes with I/O privilege. When a function with I/O privilege is called, the operating

system consults *pwdrs* to determine how many words to copy from the caller's stack to the stack of the I/O-privileged function.

### Example

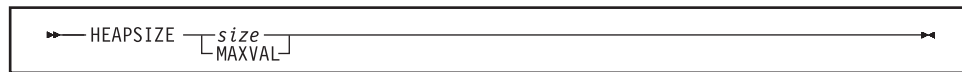
The following example defines three exported functions:

- 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. Note that in the module's own source code, these functions are actually defined as *read2bin* and *str1*, respectively. The last function runs with I/O privilege, and so has *pwdrs* (the total size of the parameters) defined for it: six words.

## HEAPSIZE



Use the `HEAPSIZE` statement to define the size of the application's local heap in bytes. This value affects the size of the automatic data segment (DGROUP), which contains the local stack and heap of the application.

You can enter any positive integer for the heap size.

Instead of entering the number of bytes, you can enter the keyword `MAXVAL`. This increases the size of DGROUP to 64K, if it is smaller than 64K. `MAXVAL` is useful in bound applications, when you want to force a 64K requirement for DGROUP. `MAXVAL` is not generally useful for 32-bit programs.

### Example

Given the following line in a .DEF file:

```
HEAPSIZE 4000
```

the linker sets the local heap to 4000 bytes.

## LIBRARY



Use the `LIBRARY` statement to identify the output file as a dynamic link library (DLL), and optionally define the name, library module initialization, and library module termination.

You can also identify the output file as a DLL with the `/DLL` option.

The following table shows defaults for the fields, depending on whether the DLL has 16-bit entry points, or 32-bit entry points:

Attribute	Default for DLLs with 16-bit entry points	Default for DLLs with 32-bit entry points
<i>libname</i>	Name of output file with DLL extension removed	Name of output file with DLL extension removed
Initialization routine	INITGLOBAL	Matches <i>term</i> , if termination given. Otherwise INITGLOBAL
Termination routine	None (applies only to DLLs with 32-bit entry points)	Matches initialization routine, if initialization given. Otherwise TERMGLOBAL

If you use the `LIBRARY` statement in your module definition (.DEF) file, it must be the first statement in the .DEF file, and you cannot use the `NAME` statement.

Specify one of the following library initialization routines to use:

#### INITGLOBAL

The library initialization routine is called only when the library module is initially loaded into memory.

#### INITINSTANCE

The library initialization routine is called each time a new process gains access to the library.

If you are generating a DLL with 32-bit entry points, you can set the type of library termination you want:

#### TERMGLOBAL

The library termination routine is called only when the library module is unloaded from memory.

#### TERMINSTANCE

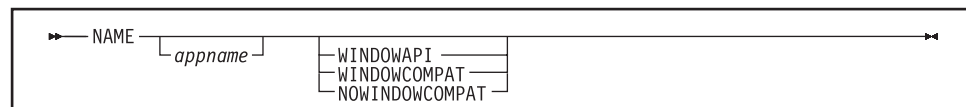
The library termination routine is called each time a process gives up access to the library.

### Example

The following example assigns the name `calendar` to the dynamic link library (DLL), and specifies that library initialization be performed each time a new process gains access. If `calendar` has 32-bit entry points, the linker will assume `TERMINSTANCE`.

```
LIBRARY calendar INITINSTANCE
```

## NAME



Use the `NAME` statement to identify the output file as an executable program (.EXE file), and optionally define the name and type of the .EXE file.

You can also identify the output file as an .EXE file with the `/EXEC` option.

If you use the `NAME` statement in your module definition (.DEF) file, it must be the first statement in the .DEF file, and you cannot use the `LIBRARY` statement.

If you specify *appname*, it becomes the name of the .EXE. The name can be any valid file name. If you do not provide a name, the name of the executable program is the same as the name of the output file, with the EXE extension removed.

The NAME statement also allows you to define the type of the program as shown below:

Type	Description	/PMTYPE option equivalent
WINDOWAPI	Presentation Manager application. The application uses the API provided by the Presentation Manager, and must run in the Presentation Manager environment.	PM
WINDOWCOMPAT	Application compatible with Presentation Manager. The application can run inside the Presentation Manager, or it can run in a separate screen group.	VIO
NOTWINDOWCOMPAT	Application that is not compatible with the Presentation Manager and must run in a separate screen group from the Presentation Manager.	NOVIO

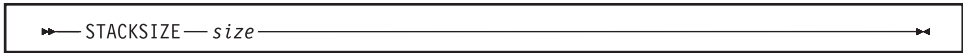
You can also use the /PMTYPE option to set the type. If conflicting types are defined by the option and in the NAME statement, the type defined by the NAME statement overrides the option value.

**Example**

The following example assigns the name calendar to the executable program, and specifies it as compatible with PM.

NAME calendar WINDOWCOMPAT

**STACKSIZE**



Use STACKSIZE to set the stack size (in bytes) of your program. 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.

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.

**Example**

The following example allocates 4K of local-stack space:

STACKSIZE 4096

## STUB

»— STUB — '*filename*' —————<«

Use the STUB statement to add a DOS .EXE file to the beginning of your .EXE or .DLL file. The stub function is then 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 linker searches for the file name you specify as the stub as follows:

1. In the directory you specify, or in the current directory if you did not give a path
2. In the directories listed in the PATH environment variable

### Example

The following example adds the DOS .EXE file STOPIT.EXE to the beginning of the file you are creating. STOPIT.EXE runs whenever your file is run under DOS.

```
STUB 'STOPIT.EXE'
```

## VERSION

»— VERSION — '*file number*' —————<«

Use the VERSION statement to add a version number to the header of the run file.

### Example

The following example adds the text “VERSION 2.3” to the executable.

```
VERSION '2.3'
```



---

## Chapter 26. Preparing COBOL programs for multithreading

Although you cannot initiate or manage program threads in a COBOL program, you can prepare your COBOL program to run in a multithreading environment. In the workstation environment, you can run your COBOL programs within the threads of processes. You do not use any explicit COBOL language; use the THREAD compiler option for multithreaded execution.

COBOL does not directly support initiating or managing program threads. However, COBOL programs can run as threads in multithreaded environments. In other words, other applications can call COBOL programs in such a way that the COBOL programs are running in multiple threads within a process or as multiple program invocation instances within a thread. Therefore, COBOL programs can run in multithreading environments like MQSeries Three Tier applications.

“Example: using COBOL in a multithreaded environment” on page 408

### RELATED CONCEPTS

“Multithreading”

### RELATED TASKS

“Working with language elements with multithreading” on page 404

“Choosing THREAD to support multithreading” on page 406

“Transferring control with multithreading” on page 406

“Handling COBOL limitations with multithreading” on page 407

---

## Multithreading

To use COBOL support for multithreading, you need to understand processes, threads, run units, and program invocation instances and how they relate to each other.

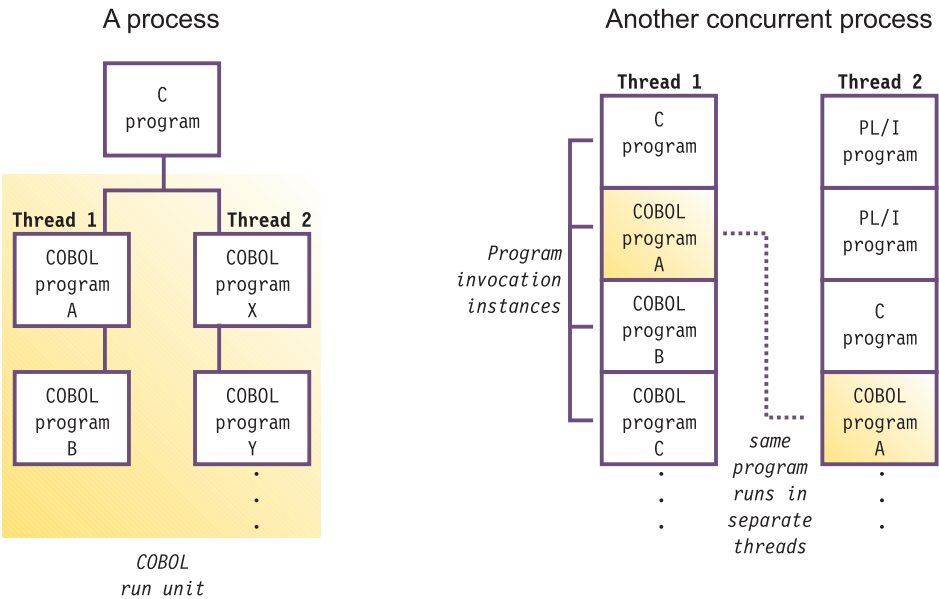
The operating system and multithreading applications can handle execution flow within a *process*, which is the course of events that occurs during the execution of all or part of a program. Multiple processes can run concurrently, and programs that run within a process can share resources. Processes can be manipulated. For example, they can be given a high or low priority in terms of the amount of time 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 is in control of a process. A multithreaded process begins with one stream of instructions (one thread) and can later create other instruction streams to perform tasks. Within a thread, control is transferred between executing programs.

In a multithreaded environment, a COBOL *run unit* is defined as the portion of the process including threads with 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*, which is generally used to describe the external behavior of applications. That is, 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 408

#### RELATED TASKS

"Working with language elements with multithreading"

"Choosing THREAD to support multithreading" on page 406

"Transferring control with multithreading" on page 406

"Handling COBOL limitations with multithreading" on page 407

#### RELATED REFERENCES

"THREAD" on page 191

## Working with language elements with multithreading

Because your COBOL programs can run as separate threads within a process, be aware that language elements can be interpreted in two different scopes:

### 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.

These two types of scope are important in determining where an item can be referenced from and how long the item persists in storage. An item can be referenced from the scope in which it was declared or 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 will remain in storage only while that instance is running.

## Working with elements that have run-unit scope

If you have resources with run-unit scope (such as GLOBAL data declared in the WORKING-STORAGE), you must synchronize access to that data from multiple threads using logic in the application. You can take one or both of the following actions:

- Structure the application so that you do not access simultaneously from multiple threads resources that have run-unit scope.
- If you are going to access resources simultaneously from separate threads, synchronize access using facilities provided by C or by platform functions.

If you have resources with 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 will be 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.

### RELATED REFERENCES

“Scope of COBOL language elements with multithreading”

## Scope of COBOL language elements with multithreading

The following table summarizes the scope of various COBOL language elements.

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

Language element	Can be referenced from:	Lifetime same as:
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

Select the THREAD compiler option for multithreading support. Choose THREAD only if you think your program will be called more than once in a single process by an application (such as MQSeries Three Tier applications). Compiling with THREAD prepares the COBOL run-time environment for threading support. However, compiling with THREAD might reduce the performance of your program due to automatically generated serialization logic.

You must compile all of the COBOL programs in the run unit with the THREAD option.

**Language restrictions:** When you use the THREAD option, there are certain language elements that you cannot use. See the complete list under the THREAD option.

**Recursion:** When you compile a program with the THREAD compiler option, you can call the program recursively in a threading or nonthreading environment. This recursive capability is available whether or not you have specified the RECURSIVE phrase in the PROGRAM-ID paragraph.

### RELATED TASKS

“Sharing data in recursive or multithreaded programs” on page 14

### RELATED REFERENCES

“THREAD” on page 191

---

## Transferring control with multithreading

When you write COBOL programs for a multithreaded environment, choose appropriate control statements.

### Controlling the state

As in single-threaded environments, a program called is in its initial state the first time it is called within a run unit and the first time it is called after a CANCEL to the called program.

### Ending a 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 also terminated.

Use EXIT PROGRAM as you would GOBACK, except from a main program where it has no effect.

If 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 on the GOBACK from the first program of this thread. This determination can be made if all COBOL programs called within the run unit have returned to their callers through GOBACK or EXIT PROGRAM.

This determination cannot be made when, for example:

- A thread with one or more active COBOL programs was terminated (for example, because of an exception or via pthread\_exit).
- A longjmp was executed and resulted in collapsing active COBOL programs in the invocation stack.

In general, it is recommended that the programs initiating and managing multiple threads use the COBOL preinitialization interface.

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 \_iwezCOBOLTerm after the run-time termination exit.

## Preinitializing the COBOL environment

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. Particularly, 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 28. Preinitializing the COBOL run-time environment” on page 419

### RELATED TASKS

“Ending and reentering main programs or subprograms” on page 360

---

## 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 are responsible for synchronizing access to resources visible to the application within a run unit. Exceptions to this are DISPLAY and ACCEPT, which you can use from multiple threads; all synchronization is provided by the run-time environment.

**DB2:** A DB2 application can be run 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. Similarly, input and output for VSAM files should be active from only one thread

at a time. However, the COBOL run-time environment does not enforce either of these restrictions. The application must therefore control sorting and merging as well as input and output for VSAM files.

#### RELATED TASKS

“Making recursive calls” on page 370

---

## Example: using COBOL in a multithreaded environment

This 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, follow these steps:

- To compile thrcob.c, enter `icc /Gm+ /C+ thrcob.c` at a command prompt.
- To compile subd.cbl, enter `cob2 -qthread -c -dll subd.cbl`.
- To compile sube.cbl, enter `cob2 -qthread -c -dll sube.cbl`.
- To link the two programs in a DLL, enter `cob2 -qthread -dll subd.obj sube.obj`.
- To link the executable thrcob.exe, enter `ilink thrcob.obj subd.lib iwzrlbm.lib`.
- To run the program thrcob, enter `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);                                // returns the thread identifier

    // 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 27. National language support

IBM VisualAge COBOL 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 user-defined names and DBCS comments
- Data type DBCS data type (with PICTURE N or G) is supported
- DBCS literals
- Collating sequence

Because the ANSI COBOL language elements dictate knowing the collating sequence at compile time, the locale setting in effect at compile time and at the run time should be consistent.

### RELATED TASKS

“Setting the locale”

“Using DBCS user-defined words and comments” on page 414

“Declaring DBCS data” on page 415

“Specifying DBCS literals” on page 416

“Controlling the collating sequence” on page 417

“Testing for valid DBCS characters” on page 418

---

## Setting the locale

A *locale* is a collection of data that encodes information about a cultural environment. You can establish a cultural environment for an application by selecting the active locale. Only one locale can be active at a time. The active locale affects the behavior on the locale-sensitive interfaces for the entire program.

Setting the locale affects the following aspects of an application:

- Code page
- Messages
- Collating sequence

The locale setting does not affect the following items:

- Decimal point and numeric separator
- Currency sign

Instead, the ANSI COBOL Standard defines specific COBOL language elements for controlling these items

### Code page

You can use the characters that are represented in a supported code page in COBOL names, data definitions, literals, and common entries.

The locale in effect determines the code set for compiling source programs (including nonnumeric literal values) and running them. That is, the code set that

is used for compilation is based on the locale setting at compile time, and the code set that is used for running the application program is based on the locale setting at run time.

The EBCDIC code set is based on the current locale setting.

If more than one EBCDIC code set is applicable for the current locale and you want to use a code page other than the default, do these two steps:

1. Set the CHAR compiler option to EBCDIC.
2. Set the EBCDIC\_CODEPAGE to establish the EBCDIC code to establish the EBCDIC code set applicable.

## Messages

The following messages are NLS enabled, and approximate message text and formats are used based on the locale setting:

- Compiler messages
- Run-time messages
- Compiler listing headers (including locale-sensitive date and time formats)
- Debugger user interface

## Collating sequence

Locale sensitivity for the collating sequence applies only when the collating sequence is NATIVE. The locale has no impact on the collating sequence if COLLSEQ(BIN) or COLLSEQ(EBCDIC) is in effect.

The collating sequence for single-byte alphanumeric characters for the program collating sequence is based on the compile-time or run-time locale. If you specify the PROGRAMMING COLLATING SEQUENCE clause in the source program, the collating sequence is set at compile time and is used regardless of the run-time locale. If instead you set the collating sequence by using the COLLSEQ compiler option, the run-time locale takes precedence.

The collating sequence for SORT or MERGE statements is always based on the run-time locale.

The run-time locale-based collating sequence is always applicable to DBCS data, independent of the COBOL source-level collating sequence specification (which applies to single-byte alphanumeric data), except for comparisons of literals.

The compile-time and run-time locale settings are assumed to be the same for other uses of the collating sequence.

### RELATED TASKS

“Specifying DBCS literals” on page 416

“Controlling the collating sequence” on page 417

### RELATED REFERENCES

“CHAR” on page 163

“Run-time environment variables” on page 140

“Locales and code sets supported” on page 413

“COLLSEQ” on page 165

## Locales and code sets supported

The following table shows the locales that VisualAge COBOL supports in a Windows environment.

Locale name	Language	Country or area	ASCII code sets	EBCDIC code sets
ar_AA	Arabic	Arabic Area	IBM-864	IBM-420
bg_BG	Bulgarian	Bulgaria	IBM-855	IBM-880 <sup>1</sup> , IBM-1025
cz_CZ	Czech	Czech Republic	IBM-852	IBM-870
da_DK	Danish	Denmark	IBM-850, IBM-437	IBM-277
de_CH	German	Switzerland	IBM-850, IBM-437	IBM-500
de_DE	German	Germany	IBM-850, IBM-437	IBM-273
el_GR	Greek	Greece	IBM-869	IBM-875
en_GB	English	United Kingdom	IBM-850, IBM-437	IBM-285
en_US	English	United States	IBM-850, IBM-437	IBM-037
es_ES	Spanish	Spain	IBM-850, IBM-437	IBM-284
fi_FI	Finnish	Finland	IBM-850, IBM-437	IBM-278
fr_BE	French	Belgium	IBM-850, IBM-437	IBM-500
fr_CA	French	Canada	IBM-850, IBM-437	IBM-037
fr_CH	French	Switzerland	IBM-850, IBM-437	IBM-500
fr_FR	French	France	IBM-850, IBM-437	IBM-297
hr_HR	Croatian	Croatia	IBM-852	IBM-870
hu_HU	Hungarian	Hungary	IBM-852	IBM-870
it_IT	Italian	Italy	IBM-850, IBM-437	IBM-280
iw_IL	Hebrew	Israel	IBM-856	IBM-424 <sup>1</sup> , IBM-803
ja_JP	Japanese	Japan	IBM-932, IBM-942, IBM-943	IBM-930 <sup>1</sup> , IBM-939
ko_KR	Korean	Korea	IBM-949	IBM-933
mk_MK	Macedonian	Macedonia, former Yugoslav Republic of	IBM-855	IBM-880 <sup>1</sup> , IBM-1025
nl_BE	Flemish	Belgium	IBM-850, IBM-437	IBM-500
nl_NL	Dutch	Netherlands	IBM-850, IBM-437	IBM-037

Locale name	Language	Country or area	ASCII code sets	EBCDIC code sets
no_NO	Norwegian	Norway	IBM-850, IBM-437	IBM-277
pl_PL	Polish	Poland	IBM-852	IBM-870
pt_PT	Portuguese	Portugal	IBM-850, IBM-437	IBM-037
ru_RU	Russian	Russia	IBM-855	IBM-880 <sup>1</sup> , IBM-1025
sh_SP	Serbo- Croatia	Serbia	IBM-852	IBM-870
sk_SK	Slovak	Slovakia	IBM-852	IBM-870
sl_SI	Slovene	Slovenia	IBM-852	IBM-870
sr_SP	Serbian	Serbia	IBM-855	IBM-880 <sup>1</sup> , IBM-1025
sv_SE	Swedish	Sweden	IBM-850, IBM-437	IBM-278
tr_TR	Turkish	Turkey	IBM-857	IBM-1026
zh_CN	Simplified Chinese	China	IBM-1381, IBM-1386	IBM-935
zh_TW	Traditional Chinese	Taiwan	IBM-938, IBM-948, IBM-950	IBM-937
1. Indicates default EBCDIC code set				

The following table shows the code set translations that VisualAge COBOL supports in a Windows environment.

Language group	From ASCII code set	To EBCDIC code set
Arabic	IBM-864	IBM-420, IBM-8612
Cyrillic	IBM-866	IBM-880, IBM-1025, IBM-1123
Latin-1	IBM-437, IBM-850, IBM-860, IBM-861, IBM-863, IBM-865	IBM-037, IBM-273, IBM-277, IBM-278, IBM-280, IBM-284, IBM-285, IBM-297, IBM-500, IBM-871
Latin-2	IBM-852, IBM-4948	IBM-870
Thai	IBM-874	IBM-838, IBM-9030
Turkey	IBM-857	IBM-905, IBM-1026

Other code set translations are possible. However, they might result in substitution characters (X'3F') being used for characters that are incompatible.

## Using DBCS user-defined words and comments

You can form user-defined words by using double-byte characters.

A user-defined word containing double-byte characters must not contain more than 15 characters.

A DBCS user-defined word can contain both multibyte and single-byte characters. When a character exists in both single-byte and multibyte forms, IBM COBOL does not regard its single-byte and multibyte representations as equivalent. For example, “A” represented in double bytes is not considered to match “A” represented in a single byte.

Alphabet-names, class-names, condition-names, data-names, file-names, mnemonic-names, record-names, and symbolic characters must contain at least one single-byte alphabetic character or one multibyte character.

You cannot continue a user-defined word containing multibyte characters.

## Restrictions on certain user-defined words

The IBM COBOL compiler supports the *level-number* user-defined word only when represented in SBCS digits.

Support for the *library-name*, *program-name*, and *text-name* user-defined words with DBCS depends on the DBCS name support of the platform. IBM COBOL allows double-byte or single-byte characters in these names.

---

## Declaring DBCS data

You define the DBCS class and category as shown in the following table:

Level of Item	Class	Category
Elementary	Alphabetic	Alphabetic
	Numeric	Numeric
	Alphanumeric	Numeric edited Alphanumeric edited Alphanumeric
	DBCS	DBCS
Nonelementary (Group)	Alphanumeric	Alphabetic Numeric Numeric edited Alphanumeric edited Alphanumeric DBCS

If you declare a data item is declared with PICTURE N or G, the selected locale must indicate a DBCS code page. In all other cases, the PICTURE characters N and G and USAGE DISPLAY-1 are flagged as errors.

Use the PICTURE clause and symbols G, G and B, or N to represent a DBCS data item. Each PICTURE symbol represents a DBCS character position. The number of bytes occupied by each double-byte character is assumed to be two. Therefore, do not include single-byte characters of a DBCS code page in a DBCS data item.

Operations on DBCS strings that do not conform to this rule might produce unpredictable results, such as the truncation of a string at a byte position in the middle of a double-byte character.

This rule will not be enforced at run time. For a code page with characters represented in double bytes, the following padding and truncation rules apply where COBOL language semantics specify padding with spaces or truncation:

- For operations involving DBCS data items, the padding is done using the double-byte space characters until the data area is filled. This padding is based on the number of byte positions allocated for the data area.

Where the padding might not be in the multiple of the code page width (for example, a group item moved to a DBCS data item), IBM COBOL pads with single-byte space characters.

- IBM COBOL truncates characters based on the size of the target data area on the byte boundary of the end of that data area. You must ensure that such truncation does not result in truncating bytes that represent a partial double-type character.

You can specify a DBCS data item with `USAGE DISPLAY-1`. When you use `PICTURE` symbol `G`, you must specify `USAGE DISPLAY-1`. When you use `PICTURE` symbol `N`, `USAGE DISPLAY-1` is implied and you can omit the `USAGE` clause.

If you use a `VALUE` clause along with the `USAGE` clause, you must specify a DBCS literal or the figurative constants `SPACE` or `SPACES`.

For the purpose of handling reference modifications, each character in a DBCS data item is considered to occupy the number of bytes corresponding to the code page width (that is, two).

---

## Specifying DBCS literals

You can specify either of two literal types to represent double-byte character constants: *N'dbcs characters'* and *G'dbcs characters'*.

In addition, you can use the alphanumeric literal syntax to specify any character in one of the supported code pages. However, such a literal is treated as alphanumeric in COBOL language semantics (that is, semantics appropriate for single-byte characters).

The literal delimiters can be apostrophes or quotes depending on the `APOST` or `QUOTE` compiler option setting.

You cannot continue a nonnumeric literal containing double-byte characters. The maximum length of an `N` or `G` literal is 28 double-byte characters. The maximum length of an `N` or `G` literal is limited only by the available space in Area B on a single source line.

## Using ALL

The figurative constants `[ALL]SPACE` and `[ALL]SPACES` represent space characters in SBCS or DBCS.

The `ALL` literal represents all or part of the string generated by successive concatenations of the single-byte characters or double-byte characters comprising the literal. The literal must be a nonnumeric literal or a DBCS literal. The literal must not be a figurative constant.

## Comparing literals

Comparisons of DBCS literals are based on the compile-time locale. Therefore, do not use DBCS literals in the source program within a statement with an implied relational condition between two DBCS literals (such as `VALUE G literal-1 THRU G literal-2`) unless the intended run-time locale is the same as the compile-time locale.

---

## Controlling the collating sequence

Any comparison that involves a group item is handled based on comparing of the byte-for-byte positions in hexadecimal.

You can specify the following clauses in the `SPECIAL-NAMES` paragraph only if the code page in effect is an ASCII code page:

- `ALPHABET` clause
- `SYMBOLIC CHARACTER` clause
- `CLASS` clause

If you specify these clauses with a DBCS code page in effect, they 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). Therefore, locale-sensitive collating can be expressed by assigning a weight to each character in the code set.

## DBCS collating sequence

You can use data items and literals of class DBCS in a relation condition with any relational operator. Each operand must be either of class DBCS, alphabetic, or alphanumeric (elementary or group). You can, for example, compare a DBCS item with an alphanumeric item. No conversion or editing is done. No distinction is made between items of category DBCS and items of category DBCS edited.

IBM COBOL performs DBCS comparisons based on the rules for nonnumeric comparisons. The comparison is based on the locale setting for the collating sequence if the operands are elementary DBCS or alphanumeric data items.

The `PROGRAM COLLATING SEQUENCE` clause has no effect on comparisons involving data items of class DBCS or DBCS literals.

## ASCII collating sequence

The ANSI COBOL rules on the `PROGRAM COLLATING SEQUENCE` clause and the `COLLATING SEQUENCE` clause on `SORT` and `MERGE` apply to the ASCII collating sequence.

If the collating sequence in effect is `NATIVE` (the default if you do not specify the `COLLATING SEQUENCE` clause or the `PROGRAM COLLATING SEQUENCE` clause), the collating sequence is based on the locale setting. This setting applies to `SORT` or `MERGE` statements as well as to the program collating sequence.

The collating sequence impacts the processing of the following items:

- `ALPHABET` clause (for example, `literal-1 THRU literal-2`)
- `SYMBOLIC CHARACTER` specifications

- VALUE range specifications for level-88 items as well as relation conditions and SORT and MERGE statements.

## Intrinsic functions that are sensitive to 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 DBCS code page, the ordinal positions of single-byte characters are assumed to correspond to the hex 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).

Intrinsic function	Returns:	Comments
CHAR	Character of the ordinal position given	
MAX	Contents of the argument that contains the maximum value	The arguments can be alphabetic or alphanumeric.
ORD	Original position of the given character	
ORD-MAX	Highest ordinal position of the characters given	ORD-MAX with numeric arguments is supported regardless of the code page in effect.
ORD-MIN	Lowest ordinal position of the characters given	ORD-MIN with numeric arguments is supported regardless of the code page in effect.

These intrinsic functions are not supported for the DBCS data type (for example, supported for single-byte characters, alphabetic, or numeric).

Any comparisons involving a group item will be handled based on the comparison of the byte-for-byte positions in hexadecimal.

---

## Testing for valid DBCS characters

Kanji and DBCS class test are defined to be consistent with their System/390 definitions. Both class tests are internally performed by converting the double-byte characters to the double-byte characters defined for OS/390. The converted double-byte characters are tested for DBCS and Japanese graphic characters.

Kanji class test results in testing for valid Japanese graphic characters. This testing includes Katakana, Hiragana, Roman, and Kanji character sets.

The Kanji class test is done by checking the converted characters for X'41' - X'7E' for the first byte and X'41' - X'FE' for the second byte plus the space character X'4040'.

DBCS class test results in testing for valid graphic characters for the code page.

The DBCS class test is done by checking the converted characters for X'41' - X'FE' for both the first and second byte of each character plus the space character X'4040'.

---

## Chapter 28. Preinitializing the COBOL run-time environment

*Preinitialization* allows an application to initialize the COBOL run-time environment once, perform multiple executions using the 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 or C/C++.

**Restriction:** Preinitialization is not supported under CICS.

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 run-time 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.

Use the interfaces described below to initialize and terminate a persistent COBOL run-time 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 421

### RELATED TASKS

“Initializing persistent COBOL environment”

“Terminating preinitialized COBOL environment” on page 420

Using preinitialization services (*Language Environment Programming Guide*)

---

## Initializing persistent COBOL environment

<b>Syntax</b> ➤ <code>CALL — <i>Init_routine</i> — (— <i>function_code</i> —, — <i>routine</i> —, — <i>error_code</i> — ➤ —, — <i>token</i> —)</code>
--

**CALL** Invocation of *Init\_routine*, using language elements appropriate to the language from which the call is being made.

### *Init\_routine*

The name of the initialization routine: `_iwbCOBOLInit` or `IWZCOBOLINIT` for Windows (using OPTLINK linkage convention); `_IwbCOBOLInit` for Windows (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 following 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 will be passed on 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 rather call longjmp or exit. This routine will be invoked with the SYSTEM linkage convention.

If you do not provide an exit routine address, an *error\_code* is generated indicating preinitialization failed.

***error\_code (output)—a 4-byte binary number***

*error\_code* can be:

- 0 Preinitialization was successful.
- 1 Preinitialization failed.

***token (input)***

4-byte token to be passed on to the exit routine specified in the earlier argument when that routine is invoked upon run-unit termination.

RELATED TASKS

“Terminating preinitialized COBOL environment”

---

## Terminating preinitialized COBOL environment

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 being made.

***Term\_routine***

The name of the termination routine: `_iwzCOBOLTerm` or `IWZCOBOLTERM` for Windows (using OPTLINK linkage convention); `_IwzCOBOLTerm` for Windows (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 run-time environment as if a COBOL STOP RUN statement had been 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.

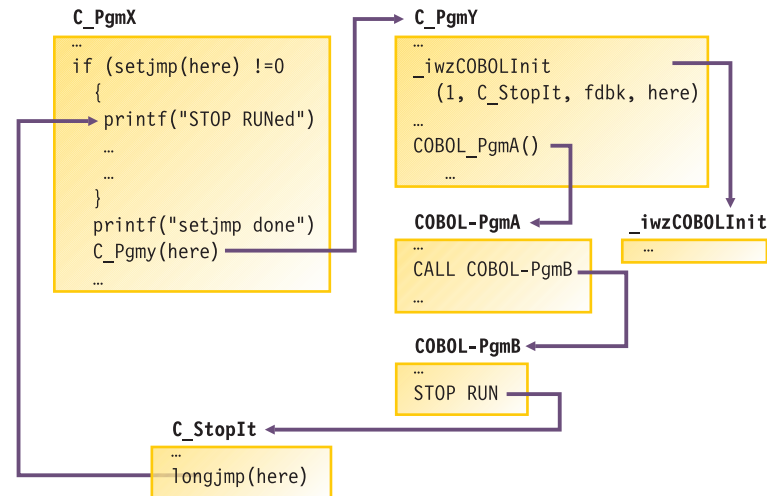
The first COBOL program called following the invocation of the preinitialization routine is treated as a subprogram. Thus, a GOBACK from this (initial) program *does not* trigger run-unit termination semantics (such as the closing of files). Note that run-unit termination (such as with STOP RUN) *does* 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 termination routine invocation has no effect on the execution, and control is returned to the invoker with an error code of 0.

“Example: preinitializing the COBOL environment”

## Example: preinitializing the COBOL environment

The following figure illustrates how the preinitialized COBOL environment works. The example shows a C program initializing the COBOL environment, calling COBOL programs, then terminating the COBOL environment.



The following example shows the use of COBOL preinitialization. A C main program calls the COBOL program XIO several times. The first call to XIO opens the file, the second call writes one record, and so on. The final call closes the file. The C program then uses C-stream I/O to open and read the file. It assumes the use of VisualAge for C++.

To test and run the program, enter the following commands from a command window:

```
cob2 -c xio.cbl
icc 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 XIO(long *k);

jmp_buf Jmpbuf;
long StopArg = 0;

int LINKAGE
StopFun(long *stoparg)
{
    printf("inside StopFun\n");
    *stoparg = 123;
    longjmp(Jmpbuf,1);
}

main()
{
    int rc;
    long k;
    FILE *s;
    int c;

    if (setjmp(Jmpbuf) ==0) {
        _iwzCOBOLInit(1, StopFun, &rc, &StopArg);
        printf( "_iwzCOBOLInit got %d\n",rc);
        for (k=0; k <= 4; k++) XIO(&k);
        k = 99; XIO(&k);
    }
    else printf("return after STOP RUN\n");
    printf("StopArg=%d\n", StopArg);
    _iwzCOBOLTerm(1, &rc);
    printf("_iwzCOBOLTerm expects rc=0 and got rc=%d\n",rc);
    printf("FILE1 contains ---- \n");
    s = fopen("FILE1", "r");
    if (s) {
        while ( (c = fgetc(s) ) != EOF ) putchar(c);
    }
    printf("---- end of FILE1\n");
}
```

The following COBOL program is in the file `xio.cbl`:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.      xio.
*****
ENVIRONMENT      DIVISION.
CONFIGURATION    SECTION.
INPUT-OUTPUT     SECTION.
FILE-CONTROL.
    SELECT file1 ASSIGN TO FILE1
    ORGANIZATION IS LINE SEQUENTIAL
```

```

        FILE STATUS IS file1-status.
. . .
DATA          DIVISION.
FILE SECTION.
FD FILE1.
01 file1-id pic x(5).
. . .
WORKING-STORAGE SECTION.
01 file1-status pic xx    value is zero.
. . .
LINKAGE SECTION.
*
01 x          PIC S9(8) COMP-5.
. . .
PROCEDURE DIVISION using x.
. . .
    display "xio entered with x=" x
    if x = 0 then
        OPEN output FILE1
    end-if
    if x = 1 then
        MOVE ALL "1" to file1-id
        WRITE file1-id
    end-if
    if x = 2 then
        MOVE ALL "2" to file1-id
        WRITE file1-id
    end-if
    if x = 3 then
        MOVE ALL "3" to file1-id
        WRITE file1-id
    end-if
    if x = 99 then
        CLOSE file1
    end-if
    GOBACK.

```



---

## Chapter 29. Processing two-digit-year dates

With the millennium language extensions, 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 426

### RELATED TASKS

"Resolving date-related logic problems" on page 427

"Using year-first, year-only, and year-last date fields" on page 432

"Manipulating literals as dates" on page 435

"Setting triggers and limits" on page 437

"Sorting and merging by date" on page 439

"Performing arithmetic on date fields" on page 441

"Controlling date processing explicitly" on page 443

"Analyzing and avoiding date-related diagnostic messages" on page 444

"Avoiding problems in processing dates" on page 446

### RELATED REFERENCES

DATE FORMAT clause (*IBM COBOL Language Reference*)

"DATEPROC" on page 167

"YEARWINDOW" on page 197

---

## Millennium language extensions (MLE)

The term *millennium language extensions* refers to the features of VisualAge COBOL that the DATEPROC compiler option activates to help with logic problems involving twenty-first century dates.

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.
- The reinterpretation of the function return value as a date field, for the following intrinsic functions:

```
DATE-OF-INTEGER  
DATE-TO-YYYYMMDD  
DAY-OF-INTEGER  
DAY-TO-YYYYDDD  
YEAR-TO-YYYY
```

- The reinterpretation as a date field of the conceptual data items DATE, DATE YYYYMMDD, DAY, and DAY YYYYDDD in the following forms of the ACCEPT statement:

```
ACCEPT identifier FROM DATE  
ACCEPT identifier FROM DATE YYYYMMDD  
ACCEPT identifier FROM DAY  
ACCEPT identifier FROM DAY YYYYDDD
```

- The intrinsic functions UNDATE and DATEVAL, used for selective reinterpretation of date fields and nondates.
- The intrinsic function YEARWINDOW, which retrieves the starting year of the century window set by the YEARWINDOW compiler option.

The DATEPROC compiler option enables special date-oriented processing of identified date fields. The YEARWINDOW compiler option specifies the 100-year window (the century window) to use for interpreting two-digit windowed years.

### RELATED CONCEPTS

"Principles and objectives of these extensions"

### RELATED REFERENCES

"DATEPROC" on page 167

"YEARWINDOW" on page 197

## 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 through the year 2000, 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:

### **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.

Aspect	Advantages and disadvantages by solution		
	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 429

“Example: internal bridging” on page 430

“Example: converting files to expanded date form” on page 431

#### RELATED TASKS

“Using a century window”

“Using internal bridging” on page 429

“Moving to full field expansion” on page 430

## 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(-48) defines a sliding window that starts 48 years before the year that the program is running. So if

the program is running in 2001, the century window is 1953-2052, and in 2002 it automatically becomes 1954-2053, and so on.

The compiler then 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”

### Example: century window

The following example shows (in bold) how to modify a program to use the automatic date windowing capability. The program checks whether a video tape was returned on time:

```
CBL  LIB,QUOTE,NOOPT,DATEPROC(FLAG),YEARWINDOW(-60)
      . . .
01  Loan-Record.
      05  Member-Number    Pic X(8).
      05  Tape-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.
```

In this example, 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 000102 (January 2, 2000) and Date-Returned contains 991231 (December 31, 1999), Date-Returned is less than (earlier than) Date-Due-Back, so the program does not perform the Fine-Member paragraph.

## Using internal bridging

For internal bridging, you can structure your program as follows:

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, and simply moving it 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 or not the date is within the century window.

“Example: internal bridging”

#### RELATED TASKS

“Using a century window” on page 428

“Performing arithmetic on date fields” on page 441

“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:

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 INFILE
        FILE STATUS IS INPUT-FILE-STATUS.

    SELECT OUTPUT-FILE
        ASSIGN TO OUTFILE
        FILE STATUS IS OUTPUT-FILE-STATUS.

DATA DIVISION.
FILE SECTION.
FD  INPUT-FILE
    RECORDING MODE IS F.
01  INPUT-RECORD.
    03  CUST-NAME.
        05  FIRST-NAME  PIC X(10).
        05  LAST-NAME   PIC X(15).
    03  ACCOUNT-NUM     PIC 9(8).
    03  DUE-DATE        PIC X(6) DATE FORMAT YYYYXX.      (1)
    03  REMINDER-DATE   PIC X(6) DATE FORMAT YYYYXX.
    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 both Gregorian dates with two-digit-year components. They are defined with a DATE FORMAT clause in this program so that the compiler will recognize 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 will treat them as four-digit-year date fields.
- (3) The MOVE CORRESPONDING statement moves each item in INPUT-RECORD individually to its matching item in OUTPUT-RECORD. When the two windowed date fields are moved to the corresponding expanded date fields, the compiler will expand 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. When you compare two date fields of either of these types, the two dates must be compatible; that is, they must have the same number of nonyear characters. The number of digits for the year component need not be the same.

A *year-last* date field is a date field whose DATE FORMAT clause specifies one or more Xs preceding the YY or YYYY. Such 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.

If your version of DFSORT (or equivalent) has the appropriate capabilities, year-last dates are supported as windowed keys in SORT or MERGE statements. Apart from sort and merge operations, 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 434

#### RELATED CONCEPTS

“Compatible dates”

#### RELATED TASKS

“Sorting and merging by date” on page 439

“Using other date formats” on page 434

## Compatible dates

The meaning of the term *compatible dates* depends on the COBOL division in which the usage occurs, as follows:

- The DATA DIVISION usage deals with the declaration of date fields, and the rules governing 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 yxxxxx.  
       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 with a date format of XXXXYY to another XXXXYY date field, but not to a date field with a format of XXXXYYYY.

You can perform operations on date fields, or on a combination of date fields and nondates, provided that the date fields in the operation are compatible. For example, assume the following definitions:

```
01 Date-Gregorian-Win Pic 9(6) Packed-Decimal Date Format yxxxxx.  
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 either is assumed to be compatible with the date field or is treated as a simple numeric value.

## Example: comparing year-first date fields

In this example, a windowed date field is compared with an expanded date field, so the century window is applied to Date-Due-Back:

```
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 . . .
```

Todays-Date must have a DATE FORMAT clause in this case 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 be between one and four characters, but its content is not important. For example, it can contain a three-digit Julian day, or a two-character identifier of some event specific to the enterprise.

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 consisting of a two-digit year, three characters containing 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”

## Example: isolating the year

In the following example, the two date fields contain dates of 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 this example, if Last-Review-Date contains 230197 (January 23, 1997), then Next-Review-Date will contain 230198 (January 23, 1998) 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 gives 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 an 88-level condition-name associated with it, the literal in the VALUE clause is windowed against the century window for 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 051220.
```

If the century window is 1950-2049 and the contents of Date-Due is 051220 (representing December 20, 2005), then the first condition below evaluates to true, but the second condition evaluates to false:

```
If Date-Target  
If Date-Due = 051220
```

The literal 051220 is treated as a nondate, and therefore it is windowed against the assumed century window of 1900-1999 to represent December 20, 1905. But where the same literal is specified in the VALUE clause of an 88-level 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, and the output from the intrinsic function will then 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 these conditions evaluate to true:

```
If Date-Due = Function DATEVAL (051220 "YYYYXX")  
If Date-Due = Function DATEVAL (20051220 "YYYYXX")
```

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 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(-60).

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 437

#### RELATED TASKS

"Controlling date processing explicitly" on page 443

## Assumed century window

When the program operates on windowed date fields, the compiler applies the century window for the compilation unit (that is, the one defined by the YEARWINDOW compiler option). When a windowed date field is used in conjunction with a nondate, and the context demands that the nondate also 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 is typically 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, the following three conditions need to be true to make the 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 443

## Treatment of nondates

The simplest kind of nondate is a literal value. The following items are also nondates:

- 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.

---

## Setting triggers and limits

Triggers and limits are special values that never match valid dates because either the value is nonnumeric or the nonyear part of the value cannot occur in an actual date. Triggers and limits are recognized in date fields and also in nondates used in combination with date fields.

Type of field	Special value
Alphanumeric windowed date or year fields	HIGH-VALUE, LOW-VALUE, and SPACE
Alphanumeric and numeric windowed date fields with at least one X in the DATE FORMAT clause (that is, date fields other than just a year)	All nines or all zeros

The difference between a trigger and a limit is not in the particular value, but in the way it is used. You can use any of the special values as either a trigger or a limit.

When used as triggers, special values can indicate a specific condition such as “date not initialized” or “account past due.” When used as limits, special values are intended to act as dates earlier or later than any valid date. LOW-VALUE, SPACE and zeros are lower limits; HIGH-VALUE and nines are upper limits.

You activate trigger and limit support by specifying the TRIG suboption of the DATEPROC compiler option. If the DATEPROC(TRIG) compiler option is in effect, automatic expansion of windowed date fields (before their use as operands in comparisons, arithmetic, and so on) is sensitive to these special values.

The DATEPROC(TRIG) option results in slower performing code when comparing windowed dates. The DATEPROC(NOTRIG) option is a performance option that assumes valid date values in all windowed date fields.

When an actual or assumed windowed date field contains a trigger, the compiler expands the trigger as if the trigger value were propagated to the century part of the expanded date result, rather than inferring 19 or 20 as the century value as in normal windowing. In this way, your application can test for special values or use them as upper or lower date limits. Specifying DATEPROC(TRIG) also enables SORT and MERGE statement support of the DFSORT special indicators, which correspond to triggers and limits.

Example: using limits

#### RELATED TASKS

“Using sign conditions” on page 439

## Example: using limits

Suppose that your application checks subscriptions for expiration, but you want some subscriptions to last indefinitely. You can use the expiration date field to hold either normal expiration dates or a high limit for the “everlasting” subscription. For example, consider the following code fragment:

```
Process DateProc(Flag,Trig). . .
. . .
01 SubscriptionRecord.
. . .
03 ExpirationDate PIC 9(6) Date Format yyxxxx.
. . .
77 TodaysDate Pic 9(6) Date Format yyxxxx.
. . .
    If TodaysDate >= ExpirationDate
        Perform SubscriptionExpired
```

Suppose that you have the following values:

- Today's date is January 4, 2000, represented in `TodaysDate` as 000104.
- One subscription record has a normal expiration date of December 31, 1999, represented as 991232.
- The special subscription expiration date is coded as 999999.

Because both dates are windowed, the first subscription is tested as if 20000104 were compared with 19991231, and so the test succeeds. However, when the compiler detects the special value, it uses trigger expansion instead of windowing. Therefore, the test proceeds as if 20000104 were compared with 99999999, and it fails and will always fail.

## 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 yyyxx.
. . .
   If Order-Date Equal Zero Then . . .
```

However, if you are compiling with the `NOTRIG` suboption of the `DATEPROC` compiler option, 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, or if compiling on the workstation, you can use a sign condition instead of a literal comparison as follows:

```
If Order-Date Is Zero Then . . .
```

With a sign condition, `Order-Date` is treated as a nondate, and the century window is not considered.

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 or the `TRIG` suboption of the `DATEPROC` compiler option to achieve the same result.

### RELATED CONCEPTS

"Treatment of nondates" on page 437

### RELATED TASKS

"Setting triggers and limits" on page 437

"Controlling date processing explicitly" on page 443

---

## Sorting and merging by date

DFSORT is the IBM licensed program for sorting and merging. Wherever DFSORT is mentioned here, you can use any equivalent product.

If your sort product supports the Y2PAST option and the windowed year identifiers (Y2B, Y2C, Y2D, Y2S, and Y2Z), you can perform sort and merge operations using windowed date fields as sort keys. Virtually all date fields that can be specified with a DATE FORMAT clause are supported, including binary year fields and year-last date fields. The fields will be sorted in windowed year sequence, according to the century window that you specify in the YEARWINDOW compiler option.

If your sort product also supports the date field identifiers Y2T, Y2U, Y2W, Y2X, and Y2Y, you can use the TRIG suboption of the DATEPROC compiler option. (Support for these date fields identifiers was added to DFSORT through APAR PQ19684.)

The special indicators that DFSORT recognizes match exactly those supported by COBOL: LOW-VALUE, HIGH-VALUE, and SPACE for alphanumeric date or year fields, and all zeros and all nines for numeric and alphanumeric date fields with at least one nonyear digit.

“Example: sorting by date and time”

#### RELATED TASKS

OPTION Control Statement (Y2PAST) (*DFSORT Application Programming Guide*)

#### RELATED REFERENCES

“DATEPROC” on page 167

“YEARWINDOW” on page 197

## Example: sorting by date and time

The following example shows a transaction file, with the transaction records sorted by date and time within account number. The field Trans-Date is a windowed Julian date field.

```
SD Transaction-File
   Record Contains 29 Characters
   Data Record is Transaction-Record

01 Transaction-Record.
   05 Trans-Account PIC 9(8).
   05 Trans-Type    PIC X.
   05 Trans-Date    PIC 9(5) Date Format yyxxx.
   05 Trans-Time    PIC 9(6).
   05 Trans-Amount  PIC 9(7)V99.
. . .
Sort Transaction-File
   On Ascending Key Trans-Account
                   Trans-Date
                   Trans-Time
   Using Input-File
   Giving Sorted-File.
```

COBOL passes the relevant information to DFSORT for it to perform the sort. In addition to the information COBOL always passes to DFSORT, COBOL also passes the following information:

- Century window as the Y2PAST sort option
- Windowed year field and date format of Trans-Date.

DFSORT uses this information to perform the sort.

---

## Performing arithmetic on date fields

You can perform arithmetic operations on numeric date fields in the same manner as any numeric data item, and, 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 arithmetic 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.

## 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. For example:

```
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.
```

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.

If the DATEPROC (TRIG) compiler option is in effect, trigger values of zeros or nines in the result also cause the SIZE ERROR condition, even though the year part of the result (00 or 99, respectively) falls within the century window.

## 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.

Consider the following example:

```
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 this example, 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.

### RELATED TASKS

“Using internal bridging” on page 429

---

## 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 this field cannot be windowed. Such date fields include:

- A key on a VSAM file
- A search field in a database system such as DB2
- A key field in a CICS command

Conversely, there might be times when you want a date field to be treated as a nondate in specific parts of the program.

COBOL provides two intrinsic functions to deal with these conditions:

### **DATEVAL**

Converts a nondate to a date field

**UNDATE** Converts a date field to a nondate

## 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 444

## Using **UNDATE**

The **UNDATE** intrinsic function converts 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”

### Example: DATEVAL

Assume that a program contains a field Date-Copied and that this field is referenced many times in the program, but that most of these references move it 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, the DATEVAL intrinsic function converts Date-Copied to a date field so that the comparison will be meaningful.

#### RELATED REFERENCES

DATEVAL (*IBM COBOL Language Reference*)

### Example: UNDATE

In the following example, the field Invoice-Date in Invoice-Record is a windowed Julian date. In some records, it contains a value of 00999 to indicate that this is not a true invoice record, but a record containing file control information.

Invoice-Date has been given a DATE FORMAT clause because most of its references in the program are date-specific. However, in the instance where it is checked for the existence of a control record, the value of 00 in the year component will lead to some confusion. A year of 00 in Invoice-Date will represent a year of 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 will therefore always represent 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 (*IBM COBOL 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 non-date 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 run-time 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 date data, even if they are not used in comparisons.
- 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 and you do not want a windowed comparison.

#### RELATED TASKS

“Controlling date processing explicitly” on page 443

#### RELATED REFERENCES

*COBOL Millennium Language Extensions Guide*(for details on the MLE messages)

---

## 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. This section outlines some of the areas that you might need to consider.

### 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, and this declaration will reserve 4 bytes of storage. But the programmer might have declared the field as PIC S9(7), knowing that this would reserve the same 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 give you 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 check carefully 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 6. Improving performance and productivity

<b>Chapter 30. Tuning your program . . . . .</b>	<b>451</b>
Using an optimal programming style . . . . .	451
Using structured programming . . . . .	452
Factoring expressions. . . . .	452
Using symbolic constants . . . . .	452
Grouping constant computations . . . . .	452
Grouping duplicate computations . . . . .	453
Choosing efficient data types . . . . .	453
Computational data items . . . . .	453
Consistent data types. . . . .	454
Arithmetic expressions . . . . .	454
Exponentiations . . . . .	454
Handling tables efficiently . . . . .	455
Optimization of table references . . . . .	456
Optimization of constant and variable items	457
Optimization of duplicate items . . . . .	457
Optimization of variable-length items . . . . .	457
Comparison of direct and relative indexing	457
Optimizing your code . . . . .	458
Optimization . . . . .	458
Contained program procedure integration	459
Choosing compiler features to enhance	
performance. . . . .	459
Performance-related compiler options . . . . .	460
Evaluating performance . . . . .	461
 <b>Chapter 31. Simplifying coding. . . . .</b>	 <b>463</b>
Eliminating repetitive coding . . . . .	463
Example: using the COPY statement. . . . .	464
Manipulating dates and times . . . . .	465
Getting feedback . . . . .	465
Handling conditions . . . . .	465
Example: manipulating dates . . . . .	466
Example: formatting dates for output . . . . .	466
Feedback token. . . . .	467
Picture character terms and strings . . . . .	468
Example: date-and-time picture strings . . . . .	470
Century window . . . . .	471
Example: querying and changing the century	
window . . . . .	471



---

## Chapter 30. 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:

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 five 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.
- Run-time environment: carefully consider your choice of run-time options and other run-time considerations that control how your compiled program runs.
- Running under CICS: convert EXEC CICS LINKs to COBOL CALLs to improve transaction response time.

### RELATED CONCEPTS

"Optimization" on page 458

### RELATED TASKS

"Using an optimal programming style"

"Choosing efficient data types" on page 453

"Handling tables efficiently" on page 455

"Optimizing your code" on page 458

"Choosing compiler features to enhance performance" on page 459

### RELATED REFERENCES

"Performance-related compiler options" on page 460

"Chapter 13. Run-time options" on page 217

---

## Using an optimal programming style

The coding style you use can, in certain circumstances, affect how the optimizer handles your code.

## Using structured programming

Using structured programming statements (such as EVALUATE and inline PERFORM) makes your program more comprehensible and generates a linear control flow. The optimizer can then operate over larger regions of the program, which gives you more efficient 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 a PERFORM procedure that prevents control from passing to the end of the procedure and returning to the PERFORM statement)

Use top-down programming constructs. Out-of-line PERFORM statements are a natural means of doing top-down programming. With the optimizer, out-of-line PERFORM statements can often be as efficient as inline PERFORM statements, because the optimizer can simplify or remove the linkage code.

## Factoring expressions

Factoring can save a lot of computation. For example, this 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
```

is more efficient than this code:

```
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 do factoring for you.

## 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 considers it to be 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 it as a constant, but only in a limited region of the program after the MOVE statement.

## Grouping constant computations

When several items of an expression are constant, ensure that the optimizer is permitted to optimize them. For evaluating expressions, 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, given that V1, V2, and V3 are variables and that C1, C2, and C3 are constants, the expressions that contain the constant computations are preferable to those that do not:

**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

Often, in production programming, there is a tendency to place constant factors on the right-hand side of expressions. However, this placement can result in less efficient code because optimization is lost.

## Grouping duplicate computations

When several components of different expressions are duplicates, make sure the compiler is permitted to optimize them. For evaluating 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.

Given that V1 through V5 are variables, the computation V2 \* V3 \* V4 is a duplicate (known as a common subexpression) between the following two statements:

```
COMPUTE A = V1 * (V2 * V3 * V4)
COMPUTE B = V2 * V3 * V4 * V5
```

In the following example, the common subexpression is V2 + V3:

```
COMPUTE C = V1 + (V2 + V3)
COMPUTE D = V2 + V3 + V4
```

No common subexpressions are in these examples:

```
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. The 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 data items in areas heavily used for computations. Consistent data types can reduce the need for conversions when performing operations on data items. You can also improve program performance by carefully determining when to use fixed-point and floating-point data types.

## 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 digits or fewer. 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 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

## 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 largely avoid conversions by using consistent data types, giving both operands the same usage and also appropriate PICTURE specifications. That is, you should give 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).

## Arithmetic expressions

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.

## Exponentiations

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.

### RELATED CONCEPTS

“Formats for numeric data” on page 34

---

## Handling tables efficiently

Pay close attention to table-handling operations, particularly when they are a major part of an application. Several techniques can improve the efficiency of these operations and can also influence the optimizer. The return for your efforts can be significant.

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 than with subscripts, even if the subscripts are `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 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 guidelines can help you get better performance in terms of how you code them:

- Put constant and duplicate indexes or subscripts on the left.

You can reduce or eliminate run-time computations by making the constant and duplicate indexes or subscripts the leftmost ones. 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 as well as 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. With equal element lengths, the stride for the last dimension of the tables will be equal. The optimizer can then 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 your indexes and subscripts, it might be faster to code your own checks in your COBOL program than to use the `SSRANGE` compiler option.

You can also improve the efficiency of tables in situations covered by the following 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 53

“Choosing efficient data types” on page 453

#### RELATED REFERENCES

“SSRANGE” on page 189

## Optimization of table references

For the table element reference `ELEMENT(S1 S2 S3)`, where S1, S2, and S3 are subscripts, the compiler evaluates the following expression:

```
comp_s1 * d1 + comp_s2 * d2 + comp_s3 * d3 + base_address
```

Here `comp_s1` is the value of S1 after conversion to binary, `comp_s2` is the value of S2 after conversion to binary, and so on. The strides for each dimension are d1, d2, and d3. The stride of a given dimension is the distance in bytes between table elements whose occurrence numbers in that dimension differ by 1 and whose other occurrence numbers are equal. For example, the stride, d2, of the second dimension in the above example is the distance in bytes between `ELEMENT(S1 1 S3)` and `ELEMENT(S1 2 S3)`.

Index computations are similar to subscript computations, except that no multiplication needs to be done. Index values have the stride factored into them. They involve loading the indexes into registers, and these data transfers can be optimized, much as the individual subscript computation terms are optimized.

Because the compiler evaluates expressions from left to right, the optimizer finds the most opportunities to eliminate computations when the constant or duplicate subscripts are the leftmost.

### Optimization of constant and variable items

Assume that  $C1, C2, \dots$  are constant data items and that  $V1, V2, \dots$  are variable data items. Then, for the table element reference  $\text{ELEMENT}(V1\ C1\ C2)$  the compiler can eliminate only the individual terms  $\text{comp\_c1} * d2$  and  $\text{comp\_c2} * d3$  as constant from the expression:

$$\text{comp\_v1} * d1 + \text{comp\_c1} * d2 + \text{comp\_c2} * d3 + \text{base\_address}$$

However, for the table element reference  $\text{ELEMENT}(C1\ C2\ V1)$  the compiler can eliminate the entire subexpression  $\text{comp\_c1} * d1 + \text{comp\_c2} * d2$  as constant from the expression:

$$\text{comp\_c1} * d1 + \text{comp\_c2} * d2 + \text{comp\_v1} * d3 + \text{base\_address}$$

In the table element reference  $\text{ELEMENT}(C1\ C2\ C3)$  all the subscripts are constant, and so no subscript computation is done at run time. The expression is:

$$\text{comp\_c1} * d1 + \text{comp\_c2} * d2 + \text{comp\_c3} * d3 + \text{base\_address}$$

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  $\text{ELEMENT}(V1\ V3\ V4)$  and  $\text{ELEMENT}(V2\ V3\ V4)$  only the individual terms  $\text{comp\_v3} * d2$  and  $\text{comp\_v4} * d3$  are common subexpressions in the expressions needed to reference the table elements:

$$\begin{aligned} &\text{comp\_v1} * d1 + \text{comp\_v3} * d2 + \text{comp\_v4} * d3 + \text{base\_address} \\ &\text{comp\_v2} * d1 + \text{comp\_v3} * d2 + \text{comp\_v4} * d3 + \text{base\_address} \end{aligned}$$

However, for the two table element references  $\text{ELEMENT}(V1\ V2\ V3)$  and  $\text{ELEMENT}(V1\ V2\ V4)$  the entire subexpression  $\text{comp\_v1} * d1 + \text{comp\_v2} * d2$  is common between the two expressions needed to reference the table elements:

$$\begin{aligned} &\text{comp\_v1} * d1 + \text{comp\_v2} * d2 + \text{comp\_v3} * d3 + \text{base\_address} \\ &\text{comp\_v1} * d1 + \text{comp\_v2} * d2 + \text{comp\_v4} * d3 + \text{base\_address} \end{aligned}$$

In the two references  $\text{ELEMENT}(V1\ V2\ V3)$  and  $\text{ELEMENT}(V1\ V2\ V3)$ , the expressions are the same:

$$\begin{aligned} &\text{comp\_v1} * d1 + \text{comp\_v2} * d2 + \text{comp\_v3} * d3 + \text{base\_address} \\ &\text{comp\_v1} * d1 + \text{comp\_v2} * d2 + \text{comp\_v3} * d3 + \text{base\_address} \end{aligned}$$

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  $\text{ELEMENT}\ (I5, J3, K2)$  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 455

---

## Optimizing your code

When your program is ready for final test, specify OPTIMIZE 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 your program.

For unit-testing your program, you will probably find it easier to debug code that has not been optimized.

To see how the optimizer works on your program, compile it with and without the OPTIMIZE option and compare the generated code. (Use the LIST compiler option to request the assembler language listing of the generated code.)

RELATED CONCEPTS  
"Optimization"

RELATED REFERENCES  
"LIST" on page 179  
"OPTIMIZE" on page 181

## Optimization

To improve the efficiency of the generated code, you can use the OPTIMIZE compiler option to cause the COBOL optimizer to do the following:

- 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 456

#### RELATED REFERENCES

"OPTIMIZE" on page 181

---

## 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.

1. To see what your system defaults are, get a short listing for any program and review the listed option settings.
2. Check with your system programmer as to which options are fixed as nonoverridable for your installation.
3. For the options not fixed by installation, select performance-related options for compiling your programs.

**Important:** Confer with your system programmer on how you should tune your COBOL programs. Doing so will ensure that the options you choose are appropriate for programs being developed at your site.

Another compiler feature to consider besides compiler options 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 458

**RELATED TASKS**

“Optimizing your code” on page 458

“Getting listings” on page 231

**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.

Compiler option	Purpose	Performance advantages	Performance disadvantages	Usage notes
“DYNAM” on page 168	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” on page 181	To optimize generated code for better performance	Generally results in more efficient run-time 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 easier debugging. For production runs, OPTIMIZE is recommended.
NOSSRANGE (see “SSRANGE” on page 189)	To eliminate code 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 with the CHECK(OFF) run-time option. For performance-sensitive applications, NOSSRANGE is recommended.

Compiler option	Purpose	Performance advantages	Performance disadvantages	Usage notes
NOTEST (see “TEST” on page 190)	To avoid the additional object code that would be produced to take full advantage of the Interactive Debugger.	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 192)	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 V2 R2.	TRUNC(STD) conforms to the COBOL 85 Standard, but TRUNC(BIN) and TRUNC(OPT) do not. With TRUNC(OPT), the compiler assumes that the data conforms to the PICTURE and USAGE specifications. TRUNC(OPT) is recommended where possible.

#### RELATED CONCEPTS

“Optimization” on page 458

#### RELATED TASKS

“Generating a list of compiler error messages” on page 150

“Choosing compiler features to enhance performance” on page 459

“Handling tables efficiently” on page 455

#### RELATED REFERENCES

“Sign representation and processing” on page 40

## Evaluating performance

Fill in this 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.

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, TEST(NONE,SYM) or TEST(NONE,SYM,SEPARATE) for productions runs?	
TRUNC	Do you use TRUNC(OPT) when possible?	

#### RELATED TASKS

“Choosing compiler features to enhance performance” on page 459

#### RELATED REFERENCES

“Performance-related compiler options” on page 460

---

## Chapter 31. Simplifying coding

This material provides techniques for improving 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 into a COBOL copy library. Then you can use the COPY statement to retrieve these code sequences from the library and have them included in your program at compile time. This 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 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 for your application in order to perform these calculations.

### RELATED CONCEPTS

“Numeric intrinsic functions” on page 42

### RELATED TASKS

“Eliminating repetitive coding”

“Converting data items (intrinsic functions)” on page 88

“Evaluating data items (intrinsic functions)” on page 90

“Manipulating dates and times” on page 465

---

## Eliminating repetitive coding

Use the COPY statement to include stored source statements in any part of your program. You can code them in any program division and at every code sequence level. You can nest COPY statements to any depth.

To specify more than one copy library, either set the environment variable SYSLIB to multiple path names separated by a semicolon (;) or define your own environment variables and include the following phrase on 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

“Compiler-directing statements” on page 198

## Example: using the COPY statement

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 member CFILEA by using the COPY statement in the DATA DIVISION of your source program code as follows:

```
FD FILEA
      COPY CFILEA.
```

The library entry is copied into your program, and the resulting program listing looks as follows:

```
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, and C precedes copied lines.

Assume that a member named DOWORK was stored with the following statements:

```
COMPUTE QTY-ON-HAND = TOTAL-USED-NUMBER-ON-HAND
MOVE QTY-ON-HAND to PRINT-AREA
```

To retrieve the stored member, DOWORK, write:

```
paragraph-name.
COPY DOWORK.
```

The statements included 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 presented here.

#### RELATED TASKS

“Eliminating repetitive coding” on page 463

#### RELATED REFERENCES

“Compiler-directing statements” on page 198

---

## Manipulating dates and times

To invoke a date and time callable service, use a CALL statement with the correct parameters for the service. For example:

```
CALL "CEEDATE" using argument, format, result, feedback-code.
```

You define the variables in the CALL statement in the DATA DIVISION of your program with the data definitions required by the function that you are calling:

```
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.
```

In this example, the date and time callable service CEEDATE converts a number representing a Lilian date in the variable `argument` to a date in character format, which is written to the variable `result`. The picture string contained in the variable `format` controls the format of the conversion.. Information about the success or failure of the call is returned in the variable `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.

## Getting feedback

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 date and time callable 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.

## Handling conditions

Condition handling by VisualAge COBOL is significantly different from that by IBM Language Environment on the host. VisualAge COBOL adheres to the native COBOL condition handling scheme and does not provide the level of support that is in Language Environment.

If a feedback token is passed as an argument, it will simply be returned after the appropriate information has been filled in. You can then code logic in the calling routine to examine the contents and perform any actions if necessary. The condition will not be signaled.

“Example: manipulating dates” on page 466

“Example: formatting dates for output” on page 466

#### RELATED REFERENCES

"Appendix E. Date and time callable services" on page 497  
"Feedback token" on page 467  
"Picture character terms and strings" on page 468  
"CALLINT" on page 162  
"Compiler-directing statements" on page 198  
CALL statement (*IBM COBOL Language Reference*)

## Example: manipulating dates

This example shows how to use date and time callable services to convert a date to a different format and do a simple calculation on the formatted date.

```
CALL CEEDAYS USING dateof_hire, 'YYMMDD', doh_lilian, fc.  
CALL CEELOCT USING today_lilian, today_seconds, today_gregorian, fc.  
COMPUTE servicedays = today_lilian - doh_lilian.  
COMPUTE serviceyears = service_days / 365.25.
```

This example 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:

- The CEEDAYS (Convert Date to Lilian Format) service converts the dates to a Lilian format.
- The CEELOCT (Get Current Local Time) service gets the current local time.
- 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 total number of days of employment.
- Divide that number by 365.25 to get the number of service years.

## Example: formatting dates for output

This sample COBOL program uses date and time callable services to format and display a date from the results of a COBOL ACCEPT statement.

Many callable services offer you function 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 for output.

```
CBL QUOTE  
  ID DIVISION.  
  PROGRAM-ID. HOHOHO.  
  *****  
  * FUNCTION:  DISPLAY TODAY'S DATE IN THE FOLLOWING FORMAT: *  
  *           WWWWWWWW, MMMMMMM DD, YYYY                    *  
  *           *                                               *  
  *           For example:  MONDAY, MARCH 10, 1997            *  
  *           *                                               *  
  *****  
  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 " WWWWWWZ, MMMMMMMM 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 will be returned to the calling routine, indicating whether the service completed successfully or not. VisualAge COBOL 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 following describes the contents of each field and identifies differences from IBM Language Environment on the host:

### 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 copy files that are provided define the condition tokens. The file CEEIGZCT.CPY contains the definitions of the condition tokens when you use native data types in your program. The file CEEIGZCT.EBC contains the copy file with the host data type support, you need to rename the file 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 types, 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 range is set to 31 December 9999. Lilian dates after this date are undefined. The reason for this limit is a four-digit year.

**RELATED REFERENCES**

"Appendix F. Run-time messages" on page 535

## Picture character terms and strings

You use picture character terms and strings to define the format of a date or time field that several of the date and time callable services use. A picture string is a template that indicates the format of the input of the data or the desired format of the output.

This table defines the supported picture character terms and string values.

Picture terms	Explanations	Valid values	Notes
Y	one-digit year	0-9	Y valid for output only.
YY	two-digit year	00-99	YY assumes range set by CEESCEN.
YYY	three-digit year	000-999	YYY/ZYY used with <JJJJ>, <CCCC>, and <CCCCCCCC>.
ZYY	three-digit year within era	1-999	
YYYY	four-digit year	1582-9999	

Picture terms	Explanations	Valid values	Notes
<JJJJ>	Japanese era name in DBCS characters	<i>Heisei</i> (X'95BD90AC') <i>Showa</i> (X'8FBA9861') <i>Taisho</i> (X'91E590B3') <i>Meiji</i> (X'96BE8EA1')	Affects YY field: if <JJJJ> specified, YY means the year within Japanese era; for example, 1988 equals Showa 63.
<CCCC> <CCCCCCCC>	Republic of China (ROC) era name in DBCS characters	<i>MinKow</i> (X'8D8196CD') <i>ChuHwaMinKow</i> (X'8C839ADC8D8196CD')	Affects YY field: if <CCCC> specified, YY means the year within ROC era, for example, 1988 equals Minkow 77. See example.
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 <sup>ooo</sup> -XII <sup>o</sup> (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 <sup>ooo</sup> -DECEMBER <sup>o</sup> January <sup>ooo</sup> -December <sup>o</sup> 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 (°) (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 <sup>ooo</sup> -SATURDAY <sup>o</sup> Sunday <sup>ooo</sup> -Saturday <sup>o</sup> 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.

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 °.			

This table defines Japanese eras used by date and time services when <JJJJ> is specified.

First date of Japanese era	Era name	Era name in Japanese DBCS code	Valid year values
1868-09-08	Meiji	X'96BE8EA1'	01-45
1912-07-30	Taisho	X'91E590B3'	01-15
1926-12-25	Showa	X'8FBA9861'	01-64
1989-01-08	Heisei	X'95BD90AC'	01-999 (01 = 1989)

This table defines the Republic of China eras used by date and time services when <CCCC> or <CCCCCCCC> is specified.

First date of ROC era	Era name	Era name in Chinese DBCS code	Valid year values
1912-01-01	MinKow	X'96BE8EA1'	01-999 (77 = 1988)
	ChuHwaMinKow	X'8C839ADC8D8196CD'	

“Example: date-and-time picture strings”

## Example: date-and-time picture strings

This is an example of picture strings that are recognized by date and time services.

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.
<CCCC> YY.MM.DD	<i>MinKow</i> 77.05.16	<i>MinKow</i> is an ROC Era name. <i>MinKow</i> 77 equals 1988.
MDDYY 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.

Picture strings	Examples	Comments
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.
YY.DDD YYDDD YYYY/DDD	88.137 88137 1988/137	Julian date
YYMDDHHMISS YYYYMDDHHMISS 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 must 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 by which all two-digit years are assumed to lie within a 100-year interval starting 80 years before the current system date:



One hundred years, spanning from 1920 to 2019 in the year 2000, is the default century window for the date and time callable services. For example, in the year 2000, years 20 through 99 are recognized as 1920-1999, and years 00 through 19 are recognized as 2000-2019.

By year 2080, all two-digit years will be recognized as 20nn. 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 01/31/25. The two-digit year 25 would be interpreted as 1925 using the current default century window. The CEESCEN callable service allows you to 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 demonstrates 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 (here, OLDCEN) that tells how many years ago the current century window began.

It then temporarily changes the starting point of the current century window to a new value (here, 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 497

---

## Part 7. Appendixes



---

## Appendix A. Summary of differences with host COBOL

IBM VisualAge COBOL implemented certain functions differently from the way IBM COBOL for OS/390 & VM implemented them. This section describes the differences in the following functions:

- “Compiler options”
- “Data representation”
- “Environment variables” on page 476
- “File specification” on page 476
- “Interlanguage communication (ILC)” on page 476
- “Input and output” on page 477
- “Run-time options” on page 477
- “Source code line” on page 477

---

### Compiler options

IBM VisualAge COBOL treats the following compiler options as comments: ADV, AWO, BUFSIZE, DATA, DECK, DBCS, FASTSRT, FLAGMIG, INTDATE, LANGUAGE, NAME, OUTDD, RENT (the compiler always produces reentrant code), and suboptions of TEST. These options are flagged with I-level messages.

IBM VisualAge COBOL treats the following compiler options as comments: NOADV and CMPR2. They are flagged with W-level messages. If you specify these options, the application might yield unpredictable results.

LIB is the IBM-supplied default on the workstation; NOLIB is the IBM-supplied default on the host.

---

### Data representation

IBM VisualAge COBOL handles binary data types based on the specification of the BINARY compiler option.

Sign representation for external decimal data are ASCII-based. Specifying NUMPROC(NOPFD) allows the full range of valid sign values for the numeric class test.

EBCDIC vs. ASCII:

- You can specify the EBCDIC collating sequence using the following language elements: ALPHABET clause, PROGRAM COLLATING SEQUENCE clause, and the COLLATING SEQUENCE phrase of the SORT and MERGE verbs.
- You can specify the CHAR(EBCDIC) compiler option to indicate that DISPLAY data items are in the System/390 data representation (EBCDIC).

You can use the FLOAT(S390) compiler option to indicate that floating-point data items are in the System/390 data representation (hexadecimal) as opposed to the native (IEEE) format.

Under AIX, and Windows, you do not use shift-in or shift-out delimiters for DBCS literals unless the CHAR(EBCDIC) compiler option is in effect.

Within an alphanumeric literal, using control characters X'00' through X'1F' can yield unpredictable results.

---

## Environment variables

IBM VisualAge COBOL recognizes the following as environment variables:

- *assignment-name*
- COBMSGS
- COBOPT
- COBPATH
- COBRTOPT
- DB2DBDFT
- EBCDIC\_CODEPAGE
- LANG
- LC\_COLLATE
- LC\_MESSAGES
- LC\_TIME
- LIBPATH
- *library-name* specified as a user-defined word
- LOCPATH
- NLSPATH
- SYSIN, SYSIPT, SYSOUT, SYSLIST, SYSLST, CONSOLE, SYSPUNCH, and SYSPCH
- SYSLIB
- TEMP
- *text-name* specified as a user-defined word
- TZ

---

## File specification

IBM VisualAge COBOL treats all files as single volume files. All other file specifications are treated as comments. This change affects the following: REEL, UNIT, MULTIPLE FILE TAPE clause, and CLOSE. . .UNIT/REEL.

---

## Interlanguage communication (ILC)

ILC is available with C, C/C++, and PL/I programs.

The following is a list of differences in ILC behavior on the 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. Use of a C longjmp() that crosses COBOL programs should be avoided on the workstation.
- 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 the workstation, the first COBOL program invoked within the process is considered to be the main program by COBOL. This change impacts the 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 will

result in the return of control to the invoker of the main program, which in a mixed language environment may be different as stated above.

---

## Input and output

IBM VisualAge COBOL supports input and output for sequential, relative, and indexed files by using STL file system, VSAM (only remote files supported on Windows), and Btrieve. Sizes and values are different for the *data-name* returned from the file system.

IBM VisualAge COBOL does not provide direct support for tape drives or diskette drives.

IBM VisualAge COBOL supports line-sequential input and output by using the native byte stream file support of the platform. The following language elements are treated as comments for line-sequential files, as well as for sequential, relative, and indexed files:

- ADVANCING phrase of WRITE statement
- APPLY WRITE ONLY clause
- AT END-OF-PAGE phrase of WRITE statement
- BLOCK CONTAINS clause
- CODE-SET clause
- DATA RECORDS clause
- FILE STATUS value 39 (fixed file attribute conflict)
- LABEL RECORDS clause
- LINAGE clause
- OPEN I-O option
- PADDING CHARACTER clause
- RECORD CONTAINS 0 clause
- RECORD CONTAINS clause (format 3)
- RECORD DELIMITER clause
- RECORDING MODE clause
- RERUN clause
- RESERVE clause
- REVERSED phrase of OPEN statement
- VALUE OF clause of file description entry

---

## Run-time options

IBM VisualAge COBOL does not recognize the following run-time 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 run-time option to initialize COBOL WORKING-STORAGE. With IBM VisualAge COBOL, you would use the WSCLEAR compiler option.

---

## Source code line

A COBOL source line can be less than 72 characters. A line ends on column 72 or where a carriage control character is found.

#### RELATED TASKS

“Chapter 22. Porting applications between platforms” on page 351

#### RELATED REFERENCES

Summary of language difference: host COBOL and workstation COBOL (*IBM COBOL Language Reference*)

---

## Appendix B. System/390 host data type considerations

The following are considerations, restrictions, and limitations which apply to the use of System/390 host data types. The BINARY, CHAR, and FLOAT compiler options determine if System/390 host data types or native data types are used.

---

### 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, etc.). Your use of host versus native data depends on such selections. Refer to the appropriate CICS documentation for specific information about how such choices can best be made.

System/390 host data type support is allowed only on VisualAge CICS Enterprise Application Development using the EBCDIC enablement support. It will not work for COBOL programs that are translated by the CICS translator and run on CICS for Windows NT or CICS for AIX.

---

### Date and time callable services

You can use the date and time callable services with the System/390 host data types. All of the parameters passed to the callable services must be in System/390 host data type format. You cannot mix native and host data types 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 workstation and the System/390 host platform, it is possible that a floating-point overflow exception can occur during conversion between the two formats. For example, you might get the following message on the workstation:

IWZ053S An overflow occurred on conversion to floating point

when running a program which executes successfully on the System/390 host platform.

To avoid this problem, you must be aware of the maximum floating-point values supported on either platform for the respective data types. The limits are shown in the following table.

Data type	Maximum workstation value	Maximum System/390 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 System/390 host can carry a larger COMP-1 value than the workstation and the workstation can carry a larger COMP-2 value than the System/390 host.

---

## DB2

The System/390 host data type compiler options can be used with DB2 programs.

---

## MQSeries

The System/390 host data type compiler options should not be used with MQSeries programs.

---

## Remote file access

- If you are accessing remote host files using SMARTdata Utilities (via COBOL file input and output statements), you do not need to specify A Data Language (ADL) for data conversion. You can access the data in the VSAM host files directly when you compile with the host data options.
- If you are already using ADL for conversion of remote file data, do not use the host data support.
- Note that file records (01 record under FD) implicitly take on the characteristics of the CHAR compiler option.

---

## Local file access

- EBCDIC data and hex binary data can be read from and written to with any local file system. 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 System/390 host data types can be used as sort keys.

### RELATED REFERENCES

“Chapter 11. Compiler options” on page 159

---

## 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 reserved.

Intermediate results are possible in the following cases:

- In an ADD or SUBTRACT statement containing more than one operand immediately following the verb
- In a COMPUTE statement specifying 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 using the GIVING option and multiple result fields
- In a statement using an intrinsic function as an operand

“Example: calculation of intermediate results” on page 482

### RELATED CONCEPTS

“Formats for numeric data” on page 34

“Fixed-point versus floating-point arithmetic” on page 45

### RELATED REFERENCES

“Fixed-point data and intermediate results” on page 482

“Floating-point data and intermediate results” on page 487

Arithmetic expressions in nonarithmetic statements (“Arithmetic expressions in nonarithmetic statements” on page 488)

---

## Terminology used for intermediate results

In the discussion of the number of integer and decimal places that the compiler reserves for intermediate results, the following terms are used:

- i* The number of integer places carried for an intermediate result. (If you use the ROUNDED option, 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 option, one more decimal place might be carried for accuracy if necessary.)
- dmax* In a particular statement, the largest of the following:
  - 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 a reference to a function, the largest of the following:

- 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.)

---

## 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 <i>ir1</i> . |
| 2. Multiply B          | by C            | yielding <i>ir2</i> . |
| 3. Divide E            | into D          | yielding <i>ir3</i> . |
| 4. Add A               | to <i>ir2</i>   | yielding <i>ir4</i> . |
| 5. Subtract <i>ir3</i> | from <i>ir4</i> | yielding <i>ir5</i> . |
| 6. Add <i>ir5</i>      | to <i>ir1</i>   | yielding Y.           |

**RELATED CONCEPTS**

“Arithmetic expressions” on page 41

**RELATED REFERENCES**

“Terminology used for intermediate results” on page 481

---

## Fixed-point data and intermediate results

The compiler determines the number of integer and decimal places in an intermediate result as discussed in the following sections.

### 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	$d1 \text{ or } d2$ , whichever is greater
*	$i1 + i2$	$d1 + d2$
/	$i2 + d1$	$(d2 - d1) \text{ or } dmax$ , 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 involving addition, subtraction, multiplication, or division:

Value of $i + d$	Value of $d$	Value of $i + dmax$	Number of places carried for $ir$
<30 =30	Any value	Any value	$i$ integer and $d$ decimal places
>30	< $dmax$ = $dmax$	Any value	$30 - d$ integer and $d$ decimal places
	> $dmax$	<30 =30	$i$ integer and $30 - i$ decimal places
		>30	$30 - dmax$ integer and $dmax$ decimal places

## Exponentiation

Exponentiation is represented by the expression  $op1 ** op2$ . Based on the characteristics of  $op2$ , the compiler handles exponentiation of fixed-point numbers in one of three ways:

- When  $op2$  is expressed with decimals, floating-point instructions are used.
- When  $op2$  is an integral literal or constant, the value  $d$  is computed as  

$$d = d1 * |op2|$$

and the value  $i$  is computed based on the characteristics of  $op1$ :

- When  $op1$  is a data name or variable,  

$$i = i1 * |op2|$$
- When  $op1$  is a literal or constant,  $i$  is set equal to the number of integers in the value of  $op1 ** |op2|$ .

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 $ir$ .

Value of $i + d$	Other conditions	Action taken
>30	Any.	Same action as when $op2$ is an integral data name or variable (shown below).

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  $30-dmax$  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 481

“Truncated intermediate results” on page 485

“Binary data and intermediate results” on page 485

“Floating-point data and intermediate results” on page 487

“Intrinsic functions evaluated in fixed-point arithmetic” on page 485

SIZE ERROR phrases (*IBM COBOL 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 following 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 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 481

“Fixed-point data and intermediate results” on page 482

## Truncated intermediate results

Whenever the number of digits in an intermediate result exceeds 30, the compiler truncates to 30 digits as shown in the tables referred to below, 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 34

#### RELATED REFERENCES

“Fixed-point data and intermediate results” on page 482

## Binary data and intermediate results

If an operation involving 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 of the operation from internal decimal to binary.

You use binary operands most efficiently when the intermediate results will not exceed nine digits.

#### RELATED REFERENCES

“Fixed-point data and intermediate results” on page 482

---

## 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-INTEGER	0	8
DATE-TO-YYYYMMDD	0	8
DAY-OF-INTEGER	0	7
DAY-TO-YYYYDDD	0	7
FACTORIAL	0	30
INTEGER-OF-DATE	0	7
INTEGER-OF-DAY	0	7

Function	<i>Inner-dmax</i>	Digit precision of function result
LENGTH	n/a	9
MOD	0	$\min(i1\ i2)$
ORD	n/a	3
ORD-MAX		9
ORD-MIN		9
YEAR-TO-YYYY	0	4
INTEGER		For a fixed-point argument: one more digit than in the argument. For a floating-point argument: 30.
INTEGER-PART		For a fixed-point argument: same number of digits as in the argument. For a floating-point argument: 30.

## 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:
  - 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:
  - 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:
  - 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 481

“Fixed-point data and intermediate results” on page 482

“Floating-point data and intermediate results”

---

## Floating-point data and intermediate results

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.

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.

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.

**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

Floating-point exponentiations are always evaluated using long 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

Floating-point intrinsic functions always return a long (64-bit) floating-point value.

Mixed functions with at least one floating-point argument are evaluated using floating-point arithmetic.

### RELATED REFERENCES

"Terminology used for intermediate results" on page 481

---

## Arithmetic expressions in nonarithmetic statements

Arithmetic expressions can appear in contexts other than arithmetic statements. For example, you can use an arithmetic expression with the IF or EVALUATE statement. In such statements, the rules for intermediate results with fixed-point data and for intermediate results with floating-point data apply, with the following changes:

- Abbreviated IF statements are handled as though the statements were not abbreviated.
- In an explicit relation condition where at least one of the comparands is an arithmetic expression, *dmax* is the maximum number of decimal places for any operand of either comparand, excluding divisors and exponents. The rules for floating-point arithmetic apply if any of the following conditions is true:
  - Any operand in either comparand is COMP-1, COMP-2, external floating point, or a floating-point literal.
  - An exponent contains decimal places.
  - An exponent is an expression that contains an exponentiation or division operator, and *dmax* is greater than zero.

For example:

```
IF operand-1 = expression-1 THEN . . .
```

If *operand-1* is a data name defined to be COMP-2, the rules for floating-point arithmetic apply to *expression-1* even if it contains only fixed-point operands, because it is being compared to a floating-point operand.

- When the comparison between an arithmetic expression and another data item or arithmetic expression does not use a relational operator (that is, there is no explicit relation condition), the arithmetic expression is evaluated without regard to the attributes of its comparand. For example:

```
EVALUATE expression-1  
  WHEN expression-2 THRU expression-3  
  WHEN expression-4  
  . . .  
END-EVALUATE
```

In the statement above, each arithmetic expression is evaluated in fixed-point or floating-point arithmetic based on its own characteristics.

### RELATED CONCEPTS

"Fixed-point versus floating-point arithmetic" on page 45

### RELATED REFERENCES

"Terminology used for intermediate results" on page 481

"Fixed-point data and intermediate results" on page 482

"Floating-point data and intermediate results" on page 487

IF statement (*IBM COBOL Language Reference*)

EVALUATE statement (*IBM COBOL Language Reference*)  
Conditional expressions (*IBM COBOL Language Reference*)



---

## Appendix D. Complex OCCURS DEPENDING ON

Complex OCCURS DEPENDING ON (ODO) is supported as an extension to the COBOL 85 Standard.

The basic forms of complex ODO permitted by the compiler are as follows:

- Variably located item or group: A data item described by an OCCURS clause with the DEPENDING ON option is followed by a nonsubordinate data item or 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.

Complex ODO can help you save disk space, but it can be tricky to use and can make maintaining your code more difficult.

“Example: complex ODO”

### RELATED TASKS

“Preventing index errors when changing ODO object value” on page 493

“Preventing overlay when adding elements to a variable table” on page 493

### RELATED REFERENCES

“Effects of change in ODO object value” on page 492

OCCURS DEPENDING ON clause (*IBM COBOL 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 following, but not subordinate to, a variable-length table in the same level-01 record.
- (2) A variably located table: TABLE-2 is a table following, but 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 containing 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 with variable-length elements.
- (5) An element, TABLE-ITEM, of a table with 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 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 described by an OCCURS clause with the DEPENDING ON option 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:

- The size of any group containing the relevant ODO clause reflects the new value of the ODO object.
- A MOVE to a group containing the ODO subject is made based on the new value of the ODO object.
- The location of any nonsubordinate items following 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 with 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 obtain 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, take these steps:

1. Save the index item in an integer data item. (Doing so causes an implicit conversion: the integer item receives the table element occurrence number corresponding to the offset in the index.)
2. Change the value of the ODO object.
3. Immediately restore the index item from the integer data item. (Doing so causes an implicit conversion: the index item receives the offset corresponding 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 (seen in “Example: complex ODO” on page 491) 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 corresponding 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
*   corresponding to INTEGER-DATA-ITEM-1, and
*   can be used with the expected results.
```

#### RELATED REFERENCES

“Effects of change in ODO object value” on page 492

SET statement (*IBM COBOL 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.

**RELATED REFERENCES**

“Effects of change in ODO object value” on page 492



---

## 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. Two callable services, CEEQCEN and CEESCEN, provide a predictable way to handle two-digit years, such as 91 for 1991 or 02 for 2002.

These are the available date and time callable services:

Callable service	Description
CEECBLDY ("CEECBLDY—convert date to COBOL integer format" on page 498)	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 502)	Converts dates in the Lilian format back to character values.
CEEDATM ("CEEDATM—convert seconds to character timestamp" on page 505)	Converts number of seconds to character timestamp.
CEEDAYS ("CEEDAYS—convert date to Lilian format" on page 509)	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 513)	Provides day of week calculation.
CEEGMT ("CEEGMT—get current Greenwich Mean Time" on page 515)	Gets current Greenwich Mean Time (date and time).
CEEGMTO ("CEEGMTO—get offset from Greenwich Mean Time to local time" on page 517)	Gets difference between Greenwich Mean Time and local time.
CEEISEC ("CEEISEC—convert integers to seconds" on page 519)	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 521)	Gets current date and time.
CEEQCEN ("CEEQCEN—query the century window" on page 523)	Queries the callable services century window.
CEESCEN ("CEESCEN—set the century window" on page 525)	Sets the callable services century window.
CEESECI ("CEESECI—convert seconds to integers" on page 526)	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 529)	Converts character timestamps (a date and time) to the number of seconds since 00:00:00 15 October 1582.
CEEUTC ("CEEUTC—get coordinated universal time" on page 533)	Same as CEEGMT.

Callable service	Description
IGZEDT4 ("IGZEDT4—get current date" on page 533)	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 COBOL for OS/390 & VM and COBOL for MVS & VM. There are, however, significant differences in the way conditions are handled.

"Example: formatting dates for output" on page 466

#### RELATED REFERENCES

CALL statement (*IBM COBOL Language Reference*)

"Feedback token" on page 467

## CEECBLDY—convert date to COBOL integer format

CEECBLDY converts a string representing a date into a COBOL integer format, which is the number of days since 31 December 1600. This service is similar to CEEDAYS, except that it provides a string in COBOL integer format, which is compatible with ANSI intrinsic functions. Use CEECBLDY to access the century window of the date and time callable services and to perform date calculations with ANSI intrinsic functions.

#### Syntax

```
CALL "CEECBLDY" USING input_char_date, picture_string,
output_Integer_date, fc.
```

#### *input\_char\_date* (input)

A halfword length-prefixed character string, representing 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 CEECBLDY skips exactly that many positions before parsing begins).

After parsing a valid date, as determined by the format of the date specified in *picture\_string*, CEECBLDY ignores all remaining characters. Valid dates range between and include 01 January 1601 to 31 December 9999.

#### *picture\_string* (input)

A halfword length-prefixed character string, indicating the format of the date specified in *input\_char\_date*.

Each character in the *picture\_string* corresponds to a character in *input\_char\_date*. For example, if you specify MMDDYY as the *picture\_string*, CEECBLDY reads an *input\_char\_date* of 060288 as 02 June 1988.

If delimiters such as the slash (/) appear in the picture string, you can omit leading zeros. For example, the following calls to CEECBLDY:

```
MOVE '6/2/88' TO DATEVAL-STRING.
MOVE 6 TO DATEVAL-LENGTH.
MOVE 'MM/DD/YY' TO PICSTR-STRING.
MOVE 8 TO PICSTR-LENGTH.
CALL CEECBLDY USING DATEVAL, PICSTR, COBINTDTE, FC.
```

```

MOVE '06/02/88' TO DATEVAL-STRING.
MOVE 8 TO DATEVAL-LENGTH.
MOVE 'MM/DD/YY' TO PICSTR-STRING.
MOVE 8 TO PICSTR-LENGTH.
CALL CEECBLDY USING DATEVAL, PICSTR, COBINTDTE, FC.

MOVE '060288' TO DATEVAL-STRING.
MOVE 6 TO DATEVAL-LENGTH.
MOVE 'MMDDYY' TO PICSTR-STRING.
MOVE 6 TO PICSTR-LENGTH.
CALL CEECBLDY USING DATEVAL, PICSTR, COBINTDTE, FC.

MOVE '88154' TO DATEVAL-STRING.
MOVE 5 TO DATEVAL-LENGTH.
MOVE 'YYDDD' TO PICSTR-STRING.
MOVE 5 TO PICSTR-LENGTH.
CALL CEECBLDY USING DATEVAL, PICSTR, COBINTDTE, FC.

```

would each assign the same value, 141502 (02 June 1988), to COBINTDTE.

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.

If *picture\_string* includes an ROC (Republic of China) Era symbol <CCCC> or <CCCCCCCC>, the YY position in *input\_char\_date* is replaced by the year number within the ROC Era. For example, the year 1988 equals the ROC year 77 in the MinKow Era.

#### ***output\_Integer\_date* (output)**

A 32-bit binary integer representing 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.

The following symbolic conditions can result from this service:

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 Japanese or Republic of China Era passed to CEEDAYS or CEESECS was not recognized.

Symbolic feedback code	Severity	Message number	Message text
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 Japanese (<JJJJ>) or Chinese (<CCCC>) 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. You should not use the returned value with other date and time callable services, nor should you call CEECBLDY from any non-COBOL programs. 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 function of CEECBLDY is provided by the DATE-OF-INTEGER and DAY-OF-INTEGER intrinsic functions.
- To perform calculations on dates earlier than 1 January 1601, add 4000 to the year in each date, convert the dates to COBOL integer format, then do the calculation. If the result of the calculation is a date, as opposed to a number of days, convert the result to a date string and subtract 4000 from the year.
- By default, two-digit years lie within the 100-year range starting 80 years prior to the system date. Thus in 2000, all two-digit years represent dates between 1920 and 2019, inclusive. You can change this default range by using the CEESCEN callable service.

### Example

```

CBL LIB,APOST
*****
**                                     **
** 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 'CEECBLDY' USING CHRDATE, PICSTR,
                    INTEGER, FC.
*****
** If CEECBLDY runs successfully, then compute **
** the date of the 90th day after the **
** input date using Intrinsic Functions **
*****
IF CEE000 of FC THEN
    COMPUTE INTEGER = INTEGER + 90
    COMPUTE NEWDATE = FUNCTION
        DATE-OF-INTEGER (INTEGER)
    DISPLAY NEWDATE
        ' is Lilian day: ' INTEGER
ELSE
    DISPLAY 'CEEBLDY failed with msg '
        Msg-No of FC UPON CONSOLE
    STOP RUN
END-IF.
*
GOBACK.

```

#### RELATED REFERENCES

“Picture character terms and strings” on page 468

## CEEDATE—convert Lilian date to character format

CEEDATE converts a number representing a Lilian date to a date written in character format. The output is a character string, such as 1996/04/23.

### Syntax

```
CALL "CEEDATE" USING input_Lilian_date, picture_string,
output_char_date, fc.
```

### *input\_Lilian\_date* (input)

A 32-bit integer representing 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, representing 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.

If *picture\_string* includes an ROC (Republic of China) Era symbol <CCCC> or <CCCCCCCC>, the YY position in *output\_char\_date* is replaced by the year number within the ROC Era. For example, the year 1988 equals the ROC year 77 in the MinKow 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.

The following symbolic conditions can result from this service:

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	Japanese (<JJJJ>) or Republic of China (<CCCC> or <CCCCCCCC>) Era 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.

Symbolic feedback code	Severity	Message number	Message text
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 notes

- The inverse of CEEDATE is CEEDAYS, which converts character dates to the Lilian format.

### Example

CBL LIB,APOST

```

*****
**                                     **
** 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 table shows the sample output from CEEDATE.

input_Lilian_date	picture_string	output_char_date
148138	YY	98
	YYMM	9805
	YY-MM	98-05
	YYMMDD	980516
	YYYYMMDD	19980516
	YYYY-MM-DD	1998-05-16
	YYYY-ZM-ZD	1998-5-16
	<JJJJ> YY.MM.DD	Showa 63.05.16 (in a DBCS string)
	<CCCC> YY.MM.DD	MinKow 77.05.16 (in a DBCS string)

input_Lilian_date	picture_string	output_char_date
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, Mmmmmmmmmm 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 470

RELATED REFERENCES

“Picture character terms and strings” on page 468

---

## CEEDATM—convert seconds to character timestamp

CEEDATM converts a number representing the number of seconds since 00:00:00 14 October 1582 to a character string format. The format of the output is a character string timestamp, such as 1988/07/26 20:37:00.

<b>Syntax</b>
➤ CALL "CEEDATM" USING <i>input_seconds</i> , <i>picture_string</i> , ➤
➤ <i>output_timestamp</i> , <i>fc</i> . ➤

***input\_seconds* (input)**

A 64-bit long floating-point number representing the number of seconds since 00:00:00 on 14 October 1582, not counting leap seconds.

For example, 00:00:01 on 15 October 1582 is second number 86,401 (24\*60\*60 + 01). The valid range of *input\_seconds* is 86,400 to 265,621,679,999.999 (23:59:59.999 31 December 9999).

***picture\_string* (input)**

A halfword length-prefixed character string, representing 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.

If *picture\_string* includes the ROC (Republic of China) Era symbol <CCCC> or <CCCCCCCC>, the YY position in *output\_timestamp* represents the year within ROC 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.

The following symbolic conditions can result from this service:

Symbolic Feedback Code	Severity	Message Number	Message Text
CEE000	0	—	The service completed successfully.
CEE2E9	3	2505	The input_seconds value in a call to CEEDATM or CEESECI was not within the supported range.
CEE2EA	3	2506	Japanese (<JJJJ>) or Republic of China (<CCCC> or <CCCCCCCC>) Era 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 notes**

- The inverse of CEEDATM is CEESECS, which converts a timestamp to number of seconds.

#### **Example**

CBL LIB,APOST

```
*****
**                                     **
** 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 Mmmmmmmmmmmmmz 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 table shows the sample output of CEEDATM.

input_seconds	picture_string	output_timestamp
12,799,191,601.000	YYMMDD	880516
	HH:MI:SS	19:00:01
	YY-MM-DD	88-05-16
	YYMMDDHHMISS	880516190001
	YY-MM-DD HH:MI:SS	88-05-16 19:00:01
	YYYY-MM-DD HH:MI:SS AP	1988-05-16 07:00:01 PM
12,799,191,661.986	DD Mmm YY	16 May 88
	DD MMM YY HH:MM	16 MAY 88 19:01
	WWW, MMM DD, YYYY	MON, MAY 16, 1988
	ZH:MI AP	7:01 PM
	Wwwwwwwwz, ZM/ZD/YY	Monday, 5/16/88
	HH:MI:SS.99	19:01:01.98

input_seconds	picture_string	output_timestamp
12,799,191,662.009	YYYY	1988
	YY	88
	Y	8
	MM	05
	ZM	5
	RRRR	V
	MMM	MAY
	Mmm	May
	Mmmmmmmmm	May
	Mmmmmmmmmz	May
	DD	16
	ZD	16
	DDD	137
	HH	19
	ZH	19
	MI	01
	SS	02
	99	00
	999	009
	AP	PM
	WWW	MON
	Www	Mon
	Wwwwwwwww	Monday
	Wwwwwwwwwz	Monday

“Example: date-and-time picture strings” on page 470

RELATED REFERENCES

“Picture character terms and strings” on page 468

---

## CEEDAYS—convert date to Lilian format

CEEDAYS converts a string representing a date into a Lilian format, which represents a date as the number of days from the beginning of the Gregorian calendar. CEEDAYS converts the specified *input\_char\_date* to a number representing the number of days since day zero in the Lilian format: Friday, 14 October, 1582.

Do not use CEEDAYS in combination with COBOL intrinsic functions. Use CEECBLDY for programs that use intrinsic functions.

<b>Syntax</b>
➤ CALL — "CEEDAYS" — USING — <i>input_char_date</i> — , — <i>picture_string</i> — , — ➤
➤ <i>output_Lilian_date</i> — , — <i>fc</i> — . — ➤

***input\_char\_date* (input)**

A halfword length-prefixed character string, representing 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:

```
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.
```

would each assign the same value, 148155 (02 June 1988), to *lildate*.

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.

If *picture\_string* includes an ROC (Republic of China) Era symbol <CCCC> or <CCCCCCCC>, the YY position in *input\_char\_date* is replaced by the year number within the ROC Era. For example, the year 1988 equals the ROC year 77 in the MinKow Era.

#### ***output\_Lilian\_date* (output)**

A 32-bit binary integer representing 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.

The following symbolic conditions can result from this service:

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.

Symbolic feedback code	Severity	Message number	Message text
CEE2ED	3	2509	The Japanese or Republic of China 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 Japanese (<JJJ>) or Chinese (<CCCC>) 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 starting 80 years prior to the system date. Thus in 2000, all two-digit years represent dates between 1920 and 2019, 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,APOST
*****
**                                     **
** 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 2000'
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 Mmmmmmmmmmmmmz 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 470

#### RELATED REFERENCES

“Picture character terms and strings” on page 468

# CEEDYWK—calculate day of week from Lilian date

CEEDYWK calculates the day of the week on which a Lilian date falls. The day of the week is returned to the calling routine as a number between 1 and 7.

The number returned by CEEDYWK is useful for end-of-week calculations.

Syntax

CALL — "CEEDYWK" — USING — *input\_Lilian\_date* — , — *output\_day\_no* — , — *fc* — .

### *input\_Lilian\_date* (input)

A 32-bit binary integer representing 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 representing *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.

The following symbolic conditions can result from this service:

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,APOST
*****
**                                     **
** 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. GMT is also known as Coordinated Universal Time (UTC). The returned values are compatible with those generated and used by the other date and time callable services.

<b>Syntax</b> CALL — "CEEGMT" — USING — <i>output_GMT_Lilian</i> —, — <i>output_GMT_seconds</i> — — <i>fc</i> —.
--

### *output\_GMT\_Lilian* (output)

A 32-bit binary integer representing 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 representing 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.

The following symbolic conditions can result from this service:

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,APOST
*****
**                                     **
** 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 137

# CEEGMTO—get offset from Greenwich Mean Time to local time

CEEGMTO returns values to the calling routine representing the difference between the local system time and Greenwich Mean Time (GMT).

Syntax

CALL — "CEEGMTO" — USING — *offset\_hours* — , — *offset\_minutes* — , — *offset\_seconds* — , — *fc* — .

## *offset\_hours* (output)

A 32-bit binary integer representing 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 representing 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 representing 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.

The following symbolic conditions can result from this service:

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 is used to convert *offset\_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.

### Example

```

CBL LIB,APOST
*****
**                                     **
** Function: Call CEEGMT0 to get offset from **
**           Greenwich Mean Time to local  **
**           time                          **
**                                     **
** In this example, a call is made to CEEGMT0 **
** 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 137

### RELATED REFERENCES

CEEGMT (“CEEGMT—get current Greenwich Mean Time” on page 515)

“Run-time environment variables” on page 140

---

## CEEISEC—convert integers to seconds

CEEISEC converts separate binary integers representing year, month, day, hour, minute, second, and millisecond to a number representing the number of seconds since 00:00:00 14 October 1582. Use CEEISEC instead of CEESECS when the input is in numeric format rather than character format.

### 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 representing the year.

The range of valid values for *input\_year* is 1582 to 9999, inclusive.

#### *input\_month* (input)

A 32-bit binary integer representing the month.

The range of valid values for *input\_month* is 1 to 12.

#### *input\_day* (input)

A 32-bit binary integer representing the day.

The range of valid values for *input\_day* is 1 to 31.

#### *input\_hours* (input)

A 32-bit binary integer representing the hours.

The range of valid values for *input\_hours* is 0 to 23.

#### *input\_minutes* (input)

A 32-bit binary integer representing the minutes.

The range of valid values for *input\_minutes* is 0 to 59.

#### *input\_seconds* (input)

A 32-bit binary integer representing the seconds.

The range of valid values for *input\_seconds* is 0 to 59.

#### *input\_milliseconds* (input)

A 32-bit binary integer representing milliseconds.

The range of valid values for *input\_milliseconds* is 0 to 999.

#### *output\_seconds* (output)

A 64-bit long floating-point number representing 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.

The following symbolic conditions can result from this service:

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 notes

- The inverse of CEEISEC is CEESECI, which converts number of seconds to integer year, month, day, hour, minute, second, and millisecond.

#### Example

```

CBL LIB, APOST
*****
**                                     **
** 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 in three formats:

- Lilian date (the number of days since 14 October 1582)
- Lilian seconds (the number of seconds since 00:00:00 14 October 1582)
- Gregorian character string (in the form YYYYMMDDHHMISS999).

These values are compatible with other date and time callable services, and with existing language intrinsic functions.

CEELOCT performs the same function as calling the CEEGMT, CEEGMTO, and CEEDATM date and time services separately. CEELOCT, however, performs the same services with much greater speed.

### Syntax

```

➤ CALL — "CEELOCT" — USING — output_Lilian — , — output_seconds — , —
➤ output_Gregorian — , — fc — .

```

### *output\_Lilian* (output)

A 32-bit binary integer representing 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 representing 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 representing 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.

The following symbolic conditions can result from this service:

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,APOST
*****
**                                     **
** Function: Call CEELOCT to get current **
**          local time                  **
**                                     **
** In this example, a call is made to CEELOCT **
** to return the current local time in Lilian **
** days (the number of days since 14 October **
** 1582), Lilian seconds (the number of **
** seconds since 00:00:00 14 October 1582), **
** and a Gregorian string (in the form **
** 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 'CEELOCT' USING LILIAN, SECONDS,
GREGORN, FC.
*****
** If CEELOCT runs successfully, display **
** Gregorian character string **
*****
IF CEE000 of FC THEN
DISPLAY 'Local Time is ' GREGORN
ELSE
DISPLAY 'CEELOCT failed with msg '
Msg-No of FC UPON CONSOLE
STOP RUN
END-IF.

GOBACK.

```

---

## CEEQCEN—query the century window

CEEQCEN queries the century which is a two-digit year value. When you want to change the setting, use CEEQCEN to get the setting and then use CEESCEN to save and restore the current setting.

### Syntax

```
CALL "CEEQCEN" USING century_start , fc .
```

### *century\_start* (output)

An integer between 0 and 100 indicating 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 starting 80 years prior to the system date. CEEQCEN then returns the value 80. An 80 value indicates to the date and time callable services that in the year 2000 all two-digit years lie within the 100-year window starting 80 years before the system date (between 1920 and 2019, inclusive).

### *fc* (output)

A 12-byte feedback code (optional) that indicates the result of this service.

The following symbolic conditions can result from this service:

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	—	The service completed successfully.

### Example

CBL LIB,APOST

```

*****
**                                                     **
** 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 to a two-digit year value for use by other date and time callable services. Use it in conjunction with CEEDAYS or CEESECS when:

- You process date values containing 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.

### Syntax

```
CALL — "CEESCEN" — USING — century_start — , — fc — .
```

### *century\_start*

An integer between 0 and 100, setting the century window.

A value of 80, for example, places all two-digit years within the 100-year window starting 80 years before the system date. In 2000, therefore, all two-digit years are assumed to represent dates between 1920 and 2019, inclusive.

### *fc* (output)

A 12-byte feedback code (optional) that indicates the result of this service.

The following symbolic conditions can result from this service:

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	—	The service completed successfully.
CEE2E6	3	2502	The UTC/GMT was not available from the system.
CEE2F5	3	2533	The value passed to CEESCEN was not between 0 and 100.

## Example

CBL LIB, APOST

```
*****
**
** 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 CEEECEN to change the start of the century
** window.
*****
CALL 'CEEECEN' USING STARTCW, FC.
IF NOT CEE000 of FC THEN
    DISPLAY 'CEEECEN 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 representing the number of seconds since 00:00:00 14 October 1582 to seven separate binary integers representing 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.

### 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 representing 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 representing the year.

The range of valid values for *output\_year* is 1582 to 9999, inclusive.

***output\_month* (output)**

A 32-bit binary integer representing the month.

The range of valid values for *output\_month* is 1 to 12.

***output\_day* (output)**

A 32-bit binary integer representing the day.

The range of valid values for *output\_day* is 1 to 31.

***output\_hours* (output)**

A 32-bit binary integer representing the hour.

The range of valid values for *output\_hours* is 0 to 23.

***output\_minutes* (output)**

A 32-bit binary integer representing the minutes.

The range of valid values for *output\_minutes* is 0 to 59.

***output\_seconds* (output)**

A 32-bit binary integer representing the seconds.

The range of valid values for *output\_seconds* is 0 to 59.

***output\_milliseconds* (output)**

A 32-bit binary integer representing 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.

The following symbolic conditions can result from this service:

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 representing 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,APOST

```
*****
**
** 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 representing a timestamp into the number of Lilian seconds (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.

### Syntax

```

➤ CALL — "CEESECS" — USING — input_timestamp — , — picture_string — , — ➤
➤ output_seconds — , — fc — . — ➤

```

### *input\_timestamp* (input)

A halfword length-prefixed character string, representing 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 variable *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.

If *picture\_string* includes a Republic of China (ROC) Era symbol <CCCC> or <CCCCCCCC>, the YY position in *input\_timestamp* represents the year number within the ROC Era. For example, the year 1988 equals the ROC year 77 in the MinKow Era.

***output\_seconds* (output)**

A 64-bit long floating-point number representing 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 ( $24 \times 60 \times 60 + 01$ ) 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.

The following symbolic conditions can result from this service:

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 Japanese or Republic of China 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.
CEE2EP	3	2521	The Japanese (<JJJ>) or Chinese (<CCCC>) 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 starting 80 years prior to the system date. Thus in 2000, all two-digit years represent dates between 1920 and 2019, inclusive. You can change this range by using the callable service CEESCEN.

### Example

CBL LIB,APOST

```

*****
**                                     **
** 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 '2000-01-01 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 470

#### RELATED REFERENCES

“Picture character terms and strings” on page 468

---

## CEEUTC—get coordinated universal time

CEEUTC is identical to CEEGMT.

#### RELATED REFERENCES

CEEGMT (“CEEGMT—get current Greenwich Mean Time” on page 515)

---

## IGZEDT4—get current date

In addition to the previous date and time callable services, VisualAge COBOL supports the VS COBOL II callable service IGZEDT4.

IGZEDT4 returns the current date with a four-digit year in the form YYYYMMDD.

<p><b>Syntax</b></p> <pre>CALL "IGZEDT4" USING output_char_date.</pre>
--

### *output\_char\_date (output)*

An 8-byte fixed-length character string in the form YYYYMMDD representing current year, month, and day.

### Usage notes

- IGZEDT4 is not supported under CICS.

### Example

CBL LIB, APOST

```
*****
** 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. Run-time messages

Messages for VisualAge COBOL contain a message prefix, message number, severity code, and descriptive text. The message prefix is always IWZ, followed by the message number. The severity code is either I (information), W (warning), S (severe), or C (critical). The message text provides a brief explanation of the condition.

In the following example message:

IWZ2519S The seconds value in a CEEISEC call was not recognized.

- 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. Note that the callable services messages contain a four-digit message number.

Within a project environment, by default, your applications are run in the foreground. If your application contains run-time errors, you can view the run-time message by running your application in the project monitor. To do this, select the EXE file, right-click, and select the **Run Monitored** action.

You can also capture run-time messages by redirecting stdout and stderr to a file when running your application from the command line. 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
```

Message number	Message text
IWZ006S (page 540)	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 (page 541)	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 (page 541)	Invalid run unit termination occurred while sort or merge is running.
IWZ013S (page 541)	Sort or merge requested while sort or merge is running in a different thread.
IWZ026W (page 542)	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 (page 542)	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 (page 542)	Argument-2 for function <i>function-name</i> in program <i>program</i> at line <i>line-number</i> was not a positive integer.
IWZ036W (page 543)	Truncation of high order digit positions occurred in program <i>program-name</i> on line number <i>line-number</i> .
IWZ037I (page 543)	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 (page 543)	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 (page 543)	An invalid overpunched sign was detected.
IWZ040S (page 544)	An invalid separate sign was detected.
IWZ045S (page 544)	Unable to invoke method <i>method-name</i> on line number <i>line number</i> in program <i>program-name</i> .
IWZ047S (page 545)	Unable to invoke method <i>method-name</i> on line number <i>line number</i> in class <i>class-name</i> .
IWZ048W (page 545)	A negative base was raised to a fractional power in an exponentiation expression. The absolute value of the base was used.
IWZ049W (page 545)	A zero base was raised to a zero power in an exponentiation expression. The result was set to one.
IWZ050S (page 545)	A zero base was raised to a negative power in an exponentiation expression.
IWZ053S (page 546)	An overflow occurred on conversion to floating point.
IWZ054S (page 546)	A floating point exception occurred.
IWZ055W (page 546)	An underflow occurred on conversion to floating point. The result was set to zero.
IWZ058S (page 546)	Exponent overflow occurred.
IWZ059W (page 547)	An exponent with more than nine digits was truncated.
IWZ060W (page 547)	Truncation of high order digit positions occurred.
IWZ061S (page 547)	Division by zero occurred.
IWZ063S (page 547)	An invalid sign was detected in a numeric edited sending field in <i>program-name</i> on line number <i>line-number</i> .
IWZ064S (page 548)	A recursive call to active program <i>program-name</i> in compilation unit <i>compilation-unit</i> was attempted.
IWZ065I (page 548)	A CANCEL of active program <i>program-name</i> in compilation unit <i>compilation-unit</i> was attempted.
IWZ066S (page 548)	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.
IWZ071S (page 549)	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 (page 549)	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 (page 549)	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 (page 549)	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 (page 550)	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 (page 550)	The number of characters in the INSPECT REPLACING CHARACTERS BY data-name was not equal to one. The first character was used.
IWZ077W (page 550)	The lengths of the INSPECT data items were not equal. The shorter length was used.
IWZ078S (page 551)	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 (page 551)	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 (page 552)	Argument-1 for function <i>function-name</i> contained no digits.
IWZ100S (page 552)	Argument-1 for function <i>function</i> was less than or equal to -1.
IWZ151S (page 552)	Argument-1 for function <i>function-name</i> contained more than 18 digits.
IWZ152S (page 552)	Invalid character <i>character</i> was found in column <i>column-number</i> in argument-1 for function <i>function-name</i> .
IWZ155S (page 552)	Invalid character <i>character</i> was found in column <i>column-number</i> in argument-2 for function <i>function-name</i> .
IWZ156S (page 553)	Argument-1 for function <i>function-name</i> was less than zero or greater than 28.
IWZ157S (page 553)	The length of Argument-1 for function <i>function-name</i> was not equal to 1.
IWZ159S (page 553)	Argument-1 for function <i>function-name</i> was less than 1 or greater than 3067671.
IWZ160S (page 553)	Argument-1 for function <i>function-name</i> was less than 16010101 or greater than 99991231.
IWZ161S (page 553)	Argument-1 for function <i>function-name</i> was less than 1601001 or greater than 9999365.

IWZ162S (page 554)	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 (page 554)	Argument-1 for function <i>function-name</i> was less than zero.
IWZ165S (page 554)	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 (page 554)	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 (page 555)	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 (page 555)	SYSPUNCH/SYSPCH will default to the system logical output device. The corresponding environment variable has not been set.
IWZ170S (page 555)	Illegal data type for DISPLAY operand.
IWZ171I (page 556)	<i>string-name</i> is not a valid run-time option.
IWZ172I (page 556)	The string <i>string-name</i> is not a valid suboption of the run-time option <i>option-name</i> .
IWZ173I (page 556)	The suboption string <i>string-name</i> of the run-time option <i>option-name</i> must be <i>number of</i> characters long. The default will be used.
IWZ174I (page 556)	The suboption string <i>string-name</i> of the run-time option <i>option-name</i> contains one or more invalid characters. The default will be used.
IWZ175S (page 556)	There is no support for routine <i>routine-name</i> on this system.
IWZ176S (page 557)	Argument-1 for function <i>function-name</i> was greater than <i>decimal-value</i> .
IWZ177S (page 557)	Argument-2 for function <i>function-name</i> was equal to <i>decimal-value</i> .
IWZ178S (page 557)	Argument-1 for function <i>function-name</i> was less than or equal to <i>decimal-value</i> .
IWZ179S (page 557)	Argument-1 for function <i>function-name</i> was less than <i>decimal-value</i> .
IWZ180S (page 557)	Argument-1 for function <i>function-name</i> was not an integer.
IWZ181I (page 558)	An invalid character was found in the numeric string <i>string</i> of the run-time option <i>option-name</i> . The default will be used.
IWZ182I (page 558)	The number <i>number</i> of the run-time option <i>option-name</i> exceeded the range of <i>min-range</i> to <i>max-range</i> . The default will be used.
IWZ183S (page 558)	The function name in _IWZCOBOLInit did a return.
IWZ200S (page 558)	Error detected during I/O operation for file <i>file-name</i> . File status is: <i>file-status</i> .
IWZ200S (page 559)	STOP or ACCEPT failed with an I/O error, <i>error-code</i> . The run unit is terminated.
IWZ201C (page 559)	
IWZ203W (page 559)	The code page in effect is not a DBCS code page.
IWZ204W (page 560)	An error occurred during conversion from ASCII DBCS to EBCDIC DBCS.
IWZ211S (page 560)	CBLTDLI detected a Remote DL/I error.
IWZ212S (page 560)	Too few arguments were passed to CBLTDLI.
IWZ213S (page 561)	Too many arguments were passed to CBLTDLI.
IWZ214S (page 561)	No function code was passed to CBLTDLI.

IWZ230W (page 561)	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 (page 561)	The EBCDIC codepage specified, <i>EBCDIC codepage</i> , is not consistent with the locale <i>locale</i> , but will be used as requested.
IWZ230W (page 562)	The EBCDIC codepage specified, <i>EBCDIC codepage</i> , is not supported. The default EBCDIC codepage, <i>EBCDIC codepage</i> , will be used.
IWZ230S (page 562)	The EBCDIC conversion table cannot be opened.
IWZ230S (page 562)	The EBCDIC conversion table cannot be built.
IWZ231S (page 562)	Query of current locale setting failed.
IWZ240S (page 563)	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 (page 563)	The current year was outside the 100-year window, <i>year-start</i> through <i>year-end</i> , for program <i>program-name</i> .
IWZ813S (page 564)	Insufficient storage was available to satisfy a get storage request.
IWZ901W (page 564)	Message variants include: <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>
IWZ902W (page 564)	The system detected a decimal-divide exception.
IWZ903S (page 564)	The system detected a data exception.
IWZ907W (page 565)	Message variants include: <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 (page 565)	Insufficient storage. Cannot find space for message <i>message-number</i> .
IWZ994W (page 565)	Cannot find message <i>message-number</i> in %s.
IWZ995C (page 565)	Message variants include: <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 (page 566)	The UTC/GMT was not available from the system.
IWZ2503S (page 566)	The offset from UTC/GMT to local time was not available from the system.
IWZ2505S (page 566)	The input_seconds value in a call to CEEDATM or CEESECI was not within the supported range.
IWZ2506S (page 567)	Japanese (<JJJJ>) or Republic of China (<CCCC> or <CCCCCCCC>) Era 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 (page 567)	Insufficient data was passed to CEEDAYS or CEESECS. The Lilian value was not calculated.
IWZ2508S (page 567)	The date value passed to CEEDAYS or CEESECS was invalid.

IWZ2509S (page 568)	The Japanese or Republic of China Era passed to CEEDAYS or CEESECS was not recognized.
IWZ2510S (page 568)	The hours value in a call to CEEISEC or CEESECS was not recognized.
IWZ2511S (page 568)	The day parameter passed in a CEEISEC call was invalid for year and month specified.
IWZ2512S (page 569)	The Lilian date value passed in a call to CEEDATE or CEEDYWK was not within the supported range.
IWZ2513S (page 569)	The input date passed in a CEEISEC, CEEDAYS, or CEESECS call was not within the supported range.
IWZ2514S (page 569)	The year value passed in a CEEISEC call was not within the supported range.
IWZ2515S (page 569)	The milliseconds value in a CEEISEC call was not recognized.
IWZ2516S (page 570)	The minutes value in a CEEISEC call was not recognized.
IWZ2517S (page 570)	The month value in a CEEISEC call was not recognized.
IWZ2518S (page 570)	An invalid picture string was specified in a call to a date/time service.
IWZ2519S (page 571)	The seconds value in a CEEISEC call was not recognized.
IWZ2520S (page 571)	CEEDAYS detected nonnumeric data in a numeric field, or the date string did not match the picture string.
IWZ2521S (page 571)	The Japanese (<JJJJ>) or Chinese (<CCCC>) year-within-Era value passed to CEEDAYS or CEESECS was zero.
IWZ2522S (page 572)	Japanese (<JJJJ>) or Republic of China (<CCCC> or <CCCCCCCC>) Era 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 (page 572)	CEESECS detected nonnumeric data in a numeric field, or the timestamp string did not match the picture string.
IWZ2526S (page 572)	The date string returned by CEEDATE was truncated.
IWZ2527S (page 573)	The timestamp string returned by CEEDATM was truncated.
IWZ2531S (page 573)	The local time was not available from the system.
IWZ2533S (page 573)	The value passed to CEESCEN was not between 0 and 100.
IWZ2534W (page 573)	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/or 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/Merge.

**Programmer response:** Determine why the Sort/Merge was unsuccessful and fix the problem. Possible return codes are:

501 Invalid function	517 Invalid output work buffer
502 Invalid record type	518 COBOL input ioerr
503 Invalid record length	519 COBOL output ioerr
504 Type length error	520 Unsupported function
505 Invalid type	521 Invalid key
506 Mismatched number of keys	522 Invalid using file
507 Type too long	523 Invalid giving file
508 Invalid key offset	524 No work directory supplied
509 Invalid scending	525 Work directory nonexistent
510 Invalid overlapping keys	526 Sortcommon not allocated
511 No key defined	527 No storage for sortcommon
512 No input specified	528 Binary buffer not allocated
513 No output specified	529 LS buffer not allocated
514 Mixed type input files	530 Work space allocation failed
515 Mixed type output files	531 FCB allocation failed
516 Invalid input work buffer	

**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* 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 reference 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 <i>method-name</i> on line number <i>line number</i> in program <i>program-name</i> .
---

**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.

#### 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 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 <i>program-name</i> on line number <i>line-number</i> .
---

**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 <i>program-name</i> in compilation unit <i>compilation-unit</i> 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 <i>program-name</i> in compilation unit <i>compilation-unit</i> 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 <i>data-record</i> in program <i>program-name</i> 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* depends on the execution platform as follows:

AIX:	The errno set by load.
Windows:	The last-error code value set by LoadLibrary.

**Programmer response:** Check that you have COBPATH defined. Check that the module exists: AIX and Windows have graphical interfaces for showing directories and files. You can also use the *ls* command on AIX or the *dir* command on Windows. 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.

#### 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 only one byte long.

**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 one byte long.

**System action:** The application was terminated.

#### IWZ159S

Argument-1 for function *function-name* was less than 1 or greater than 3067671.

**Explanation:** The input argument to DATE-OF-INTEGGER or DAY-OF-INTEGGER function is less than 1 or greater than 3067671.

**Programmer response:** Check that the function argument is in the valid range.

**System action:** The application was terminated.

#### IWZ160S

Argument-1 for function *function-name* was less than 16010101 or greater than 99991231.

**Explanation:** The input argument to function INTEGER-OF-DATE is less than 16010101 or greater than 99991231.

**Programmer response:** Check that the function argument is in the valid range.

**System action:** The application was terminated.

#### IWZ161S

Argument-1 for function *function-name* was less than 1601001 or greater than 9999365.

**Explanation:** The input argument to function INTEGER-OF-DAY is less than 1601001 or greater than 9999365.

**Programmer response:** Check that the function argument is in the valid range.

**System action:** The application was terminated.

#### IWZ162S

Argument-1 for function *function-name* was less than 1 or greater than the number of positions in the program collating sequence.

**Explanation:** The input argument to function CHAR is less than 1 or greater than the highest ordinal position in the program collating sequence.

**Programmer response:** Check that the function argument is in the valid range.

**System action:** The application was terminated.

#### IWZ163S

Argument-1 for function *function-name* was less than zero.

**Explanation:** The input argument to function RANDOM is less than 0.

**Programmer response:** Correct the argument for function RANDOM in the failing statement.

**System action:** The application was terminated.

#### IWZ165S

A reference modification start position value of *start-position-value* on line *line number* referenced an area outside the region of the function result of *function-result*.

**Explanation:** The value of the starting position in a reference modification specification was less than 1, or was greater than the current length of the function result that was being reference modified. The starting position value must be a positive integer less than or equal to the number of characters in the reference modified function result.

**Programmer response:** Check the value of the starting position in the reference modification specification and the length of the actual function result.

**System action:** The application was terminated.

#### IWZ166S

A nonpositive reference modification length value of *length* on line *line-number* was found in a reference to the function result of *function-result*.

**Explanation:** The length value in a reference modification specification for a function result was less than or equal to 0. The length value must be a positive integer.

**Programmer response:** Check the length value and make appropriate correction.

**System action:** The application was terminated.

#### IWZ167S

A reference modification start position value of *start-position* and length value of *length* on line *line-number* caused reference to be made beyond the rightmost character of the function result of *function-result*.

**Explanation:** The starting position and length value in a reference modification specification combine to address an area beyond the end of the reference modified function result. The sum of the starting position and length value minus one must be less than or equal to the number of characters in the reference modified function result.

**Programmer response:** Check the length of the reference modification specification against the actual length of the function result and make appropriate corrections.

**System action:** The application was terminated.

#### IWZ168W

SYSPUNCH/SYSPCH will default to the system logical output device. The corresponding environment variable has not been set.

**Explanation:** COBOL environment names (such as SYSPUNCH/SYSPCH) are used as the environment variable names corresponding to the mnemonic names used on ACCEPT and DISPLAY statements. Set them equal to files, not existing directory names. To set environment variables:

- On Windows, use the SET command.
- On AIX, use the export command.

You can set environment variables either persistently or temporarily.

**Programmer response:** If you do not want SYSPUNCH/SYSPCH to default to the screen, set the corresponding environment variable.

**System action:** No system action was taken.

#### 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 run-time option.

**Explanation:** *string-name* is not a valid option.

**Programmer response:** CHECK, DEBUG, ERRCOUNT, FILESYS, TRAP, and UPSI are valid run-time options.

**System action:** *string-name* is ignored.

#### IWZ172I

The string *string-name* is not a valid suboption of the run-time option *option-name*.

**Explanation:** *string-name* was not in the set of recognized values.

**Programmer response:** Remove the invalid suboption *string* from the run-time option *option-name*.

**System action:** The invalid suboption is ignored.

#### IWZ173I

The suboption string *string-name* of the run-time 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 run-time 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 run-time 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 <i>function-name</i> was greater than <i>decimal-value</i> .
--

**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 <i>function-name</i> was equal to <i>decimal-value</i> .
--

**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 <i>function-name</i> was less than or equal to <i>decimal-value</i> .
---

**Explanation:** An illegal value for argument-1 was used.

**Programmer response:** Ensure argument-1 is greater than *decimal-value*.

**System action:** The application was terminated.

#### IWZ179S

Argument-1 for function <i>function-name</i> was less than <i>decimal-value</i> .
---

**Explanation:** An illegal value for argument-1 was used.

**Programmer response:** Ensure argument-1 is equal to or greater than *decimal-value*.

**System action:** The application was terminated.

#### IWZ180S

Argument-1 for function <i>function-name</i> was not an integer.
--

**Explanation:** An illegal value for argument-1 was used.

**Programmer response:** Ensure argument-1 is an integer.

**System action:** The application was terminated.

#### IWZ181I

An invalid character was found in the numeric string *string* of the run-time 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 run-time option's string to contain all numeric characters.

**System action:** The default will be used.

#### IWZ182I

The number *number* of the run-time 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 run-time option's string to be within the valid range.

**System action:** The default will be used.

#### IWZ183S

The function name in `_iwzCOBOLInit` did a return.

**Explanation:** The run unit termination exit routine returned to the function that invoked the routine (the function specified in `function_code`).

**Programmer response:** Rewrite the function so that the run unit termination exit routine does a longjump or exit instead of return to the function.

**System action:** The application was terminated.

#### IWZ200S

Error detected during I/O operation for file *file-name*. File status is: *file-status*.

**Explanation:** An error was detected during a file I/O operation. No file status was specified for the file and no applicable error declarative is in effect for the file.

**Programmer response:** Correct the condition described in this message. You can specify the FILE STATUS clause for the file if you want to detect the error and take appropriate actions within your source program.

**System action:** The application was terminated.

## IWZ200S

STOP or ACCEPT failed with an I/O error, <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.

## IWZ201C

<b>Message variants include:</b> Access Intent List Error. Concurrent Opens Exceeds Maximum. Cursor Not Selecting a Record Position. Data Stream Syntax Error. Duplicate Key Different Index. Duplicate Key Same Index. Duplicate Record Number. File Temporarily Not Available. File system cannot be found. File Space Not Available. File Closed with Damage. Invalid Key Definition. Invalid Base File Name. Key Update Not Allowed by Different Index. Key Update Not Allowed by Same Index. No Update Intent on Record. Not Authorized to Use Access Method. Not Authorized to Directory. Not Authorized to Function. Not authorized to File. Parameter Value Not Supported. Parameter Not Supported. Record Number Out of Bounds. Record Length Mismatch. Resource Limits Reached in Target System. Resource Limits Reached in Source System.	Address Error. Command Check. Duplicate File Name. End of File Condition. Existing Condition. File Handle Not Found. Field Length Error. File Not Found. File Damaged. File is Full. File In Use. Function Not Supported. Invalid Access Method. Invalid Data Record. Invalid Key Length. Invalid File Name. Invalid Request. Invalid Flag. Object Not Supported. Record Not Available. Record Not Found. Record Inactive. Record Damaged. Record In Use. Update Cursor Error.
--	--

**Explanation:** An error was detected during a file I/O operation for a VSAM file. No file status was specified for the file and no applicable error declarative is in effect for the file.

**Programmer response:** Correct the condition described in this message. For details, see the SMARTdata Utilities VSAM manual for your platform:

- For Windows: *VSAM API Reference*
- For AIX: *VSAM in a Distributed Environment*

**System action:** The application was terminated.

## IWZ203W

The code page in effect is not a DBCS code page.
--

**Explanation:** References to DBCS data was 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:

	Windows (NT and 95)	AIX
Japan	IBM-943	IBM-932
Korea		IBM-1363
China (Simplified - Mainland)		IBM-1386
China (Traditional - Taiwan)	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.

#### IWZ211S

CBLTDLI detected a Remote DL/I error.
---------------------------------------

**Explanation:** The CBLTDLI routine invoked Remote DL/I and Remote DL/I returned with an error.

**Programmer response:** Look for Remote DL/I messages that provide information about the error.

**System action:** The application was terminated.

#### IWZ212S

Too few arguments were passed to CBLTDLI.
---

**Explanation:** The CBLTDLI routine must be passed at least one argument.

**Programmer response:** Add the missing arguments to the CBLTDLI call.

**System action:** The application was terminated.

#### IWZ213S

Too many arguments were passed to CBLTDLI.
--

**Explanation:** The CBLTDLI routine was passed more than 19 arguments.

**Programmer response:** Remove the extra arguments from the CBLTDLI call.

**System action:** The application was terminated.

#### IWZ214S

No function code was passed to CBLTDLI.
---

**Explanation:** The CBLTDLI routine recognized the first argument as a parm count field and a second argument was not provided.

**Programmer response:** Add the extra arguments to the CBLTDLI call.

**System action:** The application was terminated.

#### IWZ230W

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.
--

**Explanation:** The application has a module which 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-850 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 IBM COBOL for OS/390 & VM.

**System action:** No system action was taken.

#### IWZ230W

The EBCDIC codepage specified, <i>EBCDIC codepage</i> , is not consistent with the locale <i>locale</i> , but will be used as requested.
--

**Explanation:** The application has a module which 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 codepage specified, <i>EBCDIC codepage</i> , is not supported. The default EBCDIC codepage, <i>EBCDIC codepage</i> , will be used.
---

**Explanation:** The application has a module which was compiled with the CHAR(EBCDIC) compiler option. This error occurred because the specification of the EBCDIC\_CODE page 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 has failed.

**Programmer response:** Retry the execution from a new window.

**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
  - Not used on AIX.
  - On Windows, this environment variable should include the IBMCOBOL\LOCALE directory
- LANG
  - On Windows this should be set to the name of one of the directories located in the IBMCOBW\LOCALE directory. The default value is en\_US.
  - On AIX this should be set to a locale which has been installed on your machine. Type `locale -a` to get a list of the valid values. The default value is en\_US.

**System action:** The application was terminated.

#### 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.

## 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

## IWZ901W

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 have used the ERRCOUNT run-time 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.

## IWZ902W

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:** No system action was taken.

## 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:** No system action was taken.

#### IWZ907W

Message variants include:

- Insufficient storage.
- Insufficient storage. Cannot get *number-bytes* bytes of space for storage.

**Explanation:** The run-time 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.

#### IWZ993W

Insufficient storage. Cannot find space for message *message-number*.

**Explanation:** The run-time 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 %s.

**Explanation:** The run-time 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 NLSPATH is 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 CEEGMT0 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**

Japanese (<JJJJ>) or Republic of China (<CCCC> or <CCCCCCCC>) Era 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 a Japanese or Republic of China 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 Japanese or Republic of China 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 Japanese or Republic of China Era name.

**Programmer response:** Verify that the format of the input data matches the picture string specification and that the spelling of the Japanese or ROC 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 Japanese (<JJJJ>) and Republic of China (<CCCC>) being specified, 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 Japanese (<JJJJ>) or Chinese (<CCCC>) 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

Japanese (<JJJJ>) or Republic of China (<CCCC> or <CCCCCCCC>) Era 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 a Japanese or Republic of China 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 variable 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 variable 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, MMMMMMZ, 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 TASKS**

“Generating a list of compiler error messages” on page 150

“Setting environment variables” on page 137

**RELATED REFERENCES**

*COBOL Language Reference*

“Locales and code sets supported” on page 413

---

## 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  
J74/G4  
555 Bailey Avenue  
P.O. Box 49023  
San Jose, CA 95161-9023  
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.

---

## Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AD/Cycle	MQSeries Three Tier
AIX	MVS
AS/400	OS/2
C/370	OS/390
CICS	Presentation Manager
COBOL/370	RS/6000
DATABASE 2	S/390
DB2	SOM
DFSMS/MVS	SOMobjects
IBM	System Object Model
IMS	System/390
IMS/ESA	TXSeries
Language Environment	VisualAge
MQSeries	

Intel is a registered trademark of Intel Corporation in the United States and/or other countries.

Pentium is a registered trademark of Intel Corporation in the United States and/or other countries.

Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product or service names may be the trademarks or service marks of others.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

---

## Glossary

The terms in this glossary are defined in accordance with their meaning in COBOL and cover all platforms where IBM COBOL is used. These terms may or may not have the same meaning in other languages.

This glossary includes terms and definitions from the following publications:

- *American National Standard Programming Language COBOL, ANSI X3.23-1985* (copyright 1985 American National Standards Institute, Inc.) (ISO 1989: 1985), as amended by X3.23a-1989 (ISO 1989/Amendment 1) and X3.23b-1993 (ISO1989/Amendment 2).
- *American National Standard Dictionary for Information Systems ANSI X3.172-1990*, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42nd Street, New York, New York 10036.

American National Standard definitions are preceded by an asterisk (\*).

### A

\* **abbreviated combined relation condition.** The combined condition that results from the explicit omission of a common subject or a common subject and common relational operator in a consecutive sequence of relation conditions.

**abend.** Abnormal termination of a program.

\* **access mode.** The manner in which records are to be operated upon within a file.

\* **actual decimal point.** The physical representation, using the decimal point characters period (.) or comma (,), of the decimal point position in a data item.

**advanced program-to-program communication (APPC).** A communications protocol between the workstation and the host. CICS, IMS, and SMARTdataUtilities for remote development use the APPC communications protocol.

\* **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.

\* **alphanumeric character.** Any character in the character set of the computer.

**alphanumeric-edited character.** A character within an alphanumeric character string that contains at least one B, 0 (zero), or / (slash).

\* **alphanumeric function.** A function whose value is composed of a string of one or more characters from 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.

**APPC.** See *advanced program-to-program communication (APPC)*.

\* **argument.** An identifier, a literal, an arithmetic expression, or a function-identifier that specifies a value to be used in the evaluation of a function.

\* **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.** In Language Environment, 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 consists of seven-bit coded characters (eight 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.

**Extension:** IBM has defined an extension to ASCII code (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

**big-endian.** The default format that the mainframe and the AIX workstation use to store binary data. In this format, the least significant digit is on 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. IBM COBOL supports COBOL sequential and indexed file input-output language through Btrieve (available from Pervasive Software). You can use the Btrieve file system to access files created by VisualAge CICS Enterprise Application Development.

**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.

## 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 (page 580)

**century window.** A century window is 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*.
- For Language Environment callable services, you specify the century window in CEESCEN.

\* **character.** The basic indivisible unit of the language.

**character position.** The amount of physical storage required to store a single standard data format character described as USAGE IS DISPLAY.

| **character set.** A defined set of characters with no coded representation assumed..

**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, 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 library.** A collection of classes.

\* **class-name.** 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 run-time object representing a class.

\* **clause.** An ordered set of consecutive COBOL character strings whose purpose is to specify an attribute of an entry.

**CMS (Conversational Monitor System).** A virtual machine operating system that provides general interactive, time-sharing, problem-solving, and program-development capabilities, and that operates only under the control of the VM/SP control program.

\* **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 eight-bit code, and another code page could assign characters and meanings to 128 code points for seven-bit code. For example, the usual IBM code page for English on the workstation is IBM-850 and on the host is IBM-1047.

| **code point.** A unique bit pattern defined in a coded character set (code page). Code points are assigned to graphic characters in a coded character set.

| **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. Character sets represented by ASCII or EBCDIC code pages, or by the UTF-16 representation of Unicode are examples of a coded character sets.

**coded character set identifier (CCSID).** A 16 bit number (five-digit decimal) that includes a specific set of encoding scheme identifiers, character set identifiers, code page identifiers, and other information that uniquely identifies the coded graphic character representation.

**\* 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 character position within a print line. The columns are numbered from 1, by 1, starting at the leftmost character position of the print line and extending to the rightmost position of the print line.

**\* 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 in a program. 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 YYYYX is compatible with:

- Another windowed date field with DATE FORMAT YYYYX
- An expanded date field with DATE FORMAT YYYYXXXX

**\* 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 a COBOL source program is translated, by a COBOL compiler, to a COBOL object program.

**compiler.** A program that translates a program written in a higher-level language into a machine-language object program.

**compiler-directing statement.** A statement (or directive), beginning with a compiler-directing verb, that causes the compiler to take a specific action during compilation. Compiler directives are contained in the COBOL source program. Therefore, you can specify different suboptions of a directive within the source program by using multiple compiler-directing 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.
- 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 Visual Builder, a GUI project whose target is built as a DLL instead of as an EXE.

\* **computer-name.** A system-name that identifies the computer where the program is to be compiled or run.

**condition.** An exception that has been enabled, or recognized, by Language Environment and thus is eligible to activate user and language condition handlers. 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. When condition-name is used in the general formats, it represents a unique data item reference that consists of a syntactically correct combination of a condition-name, qualifiers, and subscripts, as required for uniqueness of reference.

\* **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.

\* **contiguous items.** Items that are described by consecutive entries in the DATA DIVISION, and that bear a definite hierarchic relationship to each other.

**copy file.** A file or library member containing 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 *copybook*.

**CORBA.** The Common Object Request Broker Architecture established by the Object Management Group. IBM's Interface Definition Language, which is used to describe the interface for SOM classes, is fully compliant with CORBA standards. See also *Interface Definition Language*.

\* **counter.** A data item used for storing numbers or number representations in a manner that permits these numbers to be increased or decreased by the value of another number, or to be changed or reset to zero or to an arbitrary positive or negative value.

**cross-reference listing.** The portion of the compiler listing that contains information on where files, fields, and indicators are defined, referenced, and modified in a program.

**currency-sign value.** A character string that identifies the monetary units stored in a numeric-edited item. Typical examples are \$, USD, and EUR. A currency-sign value can be defined by either the CURRENCY compiler option or the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION. If the CURRENCY SIGN clause is not specified and the

NOCURRENCY compiler option is in effect, the dollar sign (\$) is used as the default currency-sign value. See also *currency symbol*.

**currency symbol.** A character used in a PICTURE clause to indicate the position of a currency sign value in a numeric-edited item. A currency symbol can be defined by either the CURRENCY compiler option or the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION. If the CURRENCY SIGN clause is not specified and the NOCURRENCY compiler option is in effect, the dollar sign (\$) is used as the default currency sign value and currency symbol. Multiple currency symbols and currency sign values can be defined. See also *currency sign value*.

\* **current record.** In file processing, the record that is available in the record area associated with a file.

\* **current volume pointer.** A conceptual entity that points to the current volume of a sequential file.

## D

\* **data clause.** A clause, appearing in a data description entry in the DATA DIVISION of a COBOL program, that provides information describing a particular attribute of a data item.

\* **data description entry.** An entry in the DATA DIVISION of a COBOL program that is composed of a level-number followed by a data-name, if required, and then followed by a set of data clauses, as required.

**DATA DIVISION.** One of the four main components of a COBOL program, class definition, or method definition. The DATA DIVISION describes the data to be processed by the object program, class, or method: 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. (A class DATA DIVISION contains only the WORKING-STORAGE SECTION.)

\* **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 in *IBM COBOL Language Reference*).

The term *date field* refers to both *expanded date field* and *windowed date field*. See also *nondate*.

**date format.** The date pattern of a date field, specified in either of the following ways:

- Explicitly, by the DATE FORMAT clause or DATEVAL intrinsic function argument-2
- Implicitly, by statements and intrinsic functions that return date fields (for details, see Date field in *IBM COBOL Language Reference*)

**DBCS.** See *double-byte character set (DBCS)*.

\* **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.

**Distributed Debugger.** A client-server application that enables you to detect and diagnose errors in programs that run on systems accessible through a network connection or that run on your workstation. The Distributed Debugger uses a graphical user interface where you can issue commands to control the execution (remote or local) of your program.

\* **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.** a load module or program object that exports program, function, or variable definitions to other DLLs or DLL applications.

**DLL application.** an application that references imported program, function, or variables.

**DLL linkage.** a CALL in a program compiled with the DLL and NODYNAM options, that resolves to an exported name in a separate module; or an INVOKE of a method that is defined in a separate module.

**do construction.** In structured programming, a DO statement is used to group a number of statements in a procedure. In COBOL, an in-line 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.

**double-byte character set (DBCS).** A set of characters in which each character is represented by two 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 two 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 compiled with the DYNAM and NODLL options; or a CALL identifier statement in a program compiled with the NODLL option.

**dynamic link library (DLL).** A file containing 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 .EXE file to run properly.

**dynamic storage area (DSA).** Dynamically acquired storage composed of a register save area and an area available for dynamic storage allocation (such as program variables). DSAs are generally allocated within STACK segments managed by Language Environment.

## E

\* **EBCDIC (Extended Binary-Coded Decimal Interchange Code).** A coded character set consisting of eight-bit coded characters.

**EBCDIC character.** Any one of the symbols included in the eight-bit 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*.

**enclave.** When running under the Language Environment product, an enclave is analogous to a run unit. An enclave can create other enclaves on OS/390 and CMS by a LINK, on CMS by CMSCALL, and the use of the system() function of C.

**\*end class header.** A combination of words, followed by a separator period, that indicates the end of a COBOL class definition. The end class header is:  
END CLASS class-name.

**\*end method header.** A combination of words, followed by a separator period, that indicates the end of a COBOL method definition. The end method header 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 header.** A combination of words, followed by a separator period, that indicates the end of a COBOL source program. The end program header 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 describe the way an operating system is going to run and the devices it is going to recognize.

**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 *run-time 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.

**extensions.** Certain COBOL syntax and semantics supported by IBM compilers in addition to those described in ANSI Standard.

\* **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 item.** A format for representing numbers in which the digit is contained in bits 4 through 7 and the sign is contained in bits 0 through 3 of the rightmost byte. Bits 0 through 3 of all other bytes contain 1 (hex F). For example, the decimal value of +123 is represented as 1111 0001 1111 0010 1111 0011. Synonymous with *zoned decimal item*.

\* **external file connector.** A file connector that is accessible to one or more object programs in the run unit.

**external floating-point item.** 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.

**external program.** The outermost program. A program that is not nested.

\* **external switch.** A hardware or software device, defined and named by the implementor, which is used to indicate that one of two alternate states exists.

## F

\* **figurative constant.** A compiler-generated value referenced through the use of certain reserved words.

\* **file.** A collection of logical records.

\* **file attribute conflict condition.** An unsuccessful attempt has been made to execute an input-output operation on a file and the file attributes, as specified for that file in the program, do not match the fixed attributes for that file.

\* **file clause.** A clause that appears as part of any of the following DATA DIVISION entries: file description entry (FD entry) and sort-merge file description entry (SD entry).

\* **file connector.** A storage area that contains information about a file and is used as the linkage between a file-name and a physical file and between a file-name and its associated record area.

**FILE-CONTROL.** The name of an ENVIRONMENT DIVISION paragraph in which the data files for a given source program are declared.

\* **file control entry.** A SELECT clause and all its subordinate clauses that declare the relevant physical attributes of a file.

\* **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 and file management structures on a physical or logical mass storage device, such as a diskette or minidisk.

\* **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 character positions.

**fixed-point number.** 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 number.** 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.

## G

\* **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.** A data item that is composed of subordinate data items.

## 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*.

**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.

**IBM COBOL extension.** Certain COBOL syntax and semantics supported by IBM compilers in addition to those described in ANSI Standard.

**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 VisualAge COBOL bootstrap routine. It must be link-edited with any module that contains a VisualAge COBOL 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 one or more classes as the basis for another class. A subclass inherits from one or more superclasses. By definition the inheriting class conforms to the inherited classes.

**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 during execution of the object program or method definition.

\* **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.** (1) 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 of the class definition. The state of an object also includes the state of the instance variables introduced by base classes that are inherited by the current class. A separate copy of the instance data is created for each object instance. (2) In Visual Builder, private data that belongs to a given object and is hidden from direct access by all other objects. Data members can be accessed only by the methods of the defining class and its subclasses.

\* **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.

**Interactive System Productivity Facility (ISPF).** An IBM software product that provides a menu-driven interface for the TSO or VM user. ISPF includes library utilities, a powerful editor, and dialog management.

**interface.** The information that a client must know to use a class—the names of its attributes and the signatures of its methods. With direct-to-SOM compilers such as COBOL, the native language syntax for class definitions can define the interface to a class. Classes implemented in other languages might have their interfaces defined directly in SOM Interface Definition Language (IDL). The COBOL compiler has a compiler option, IDLGEN, to automatically generate IDL for a COBOL class.

**Interface Definition Language (IDL).** The formal language (independent of any programming language) by which the interface for a class of objects is defined in a IDL file, which the SOM compiler then interprets to create an implementation template file and binding files. The Interface Definition Language is fully compliant with standards established by the Object Management Group's Common Object Request Broker Architecture (CORBA).

**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 item.** A format in which each byte in a field except the rightmost byte represents two numeric digits. The rightmost byte contains one digit and the sign. For example, the decimal value +123 is represented as 0001 0010 0011 1111. Synonymous with *packed decimal*.

\* **internal file connector.** A file connector that is accessible to only one object program in the run unit.

\* **intrarecord data structure.** The entire collection of groups and elementary data items from a logical record that a contiguous subset of the data description entries defines. These data description entries include all entries whose level-number is greater than the level-number of the first data description entry describing the intra-record data structure.

**intrinsic function.** A predefined function, such as a commonly used arithmetic function, called by a built-in function reference.

\* **invalid key condition.** A condition, at run time, caused when a specific value of the key associated with an indexed or relative file is determined to be not valid.

\* **I-O-CONTROL.** The name of an ENVIRONMENT DIVISION paragraph in which object program requirements for rerun points, sharing of same areas by several data files, and multiple file storage on a single input-output device are specified.

\* **I-O-CONTROL entry.** An entry in the I-O-CONTROL paragraph of the ENVIRONMENT DIVISION; this entry contains clauses that provide information required for the transmission and handling of data on named files during the execution of a program.

\* **I-O mode.** The state of a file after execution of an OPEN statement, with the I-O phrase specified, for that file and before the execution of a CLOSE statement without the REEL or UNIT phase for that file.

\* **I-O status.** A conceptual entity that contains the two-character value indicating the resulting status of an input-output operation. This value is made available to the program through the use of the FILE STATUS clause in the file control entry for the file.

**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.

**ISPF.** See *Interactive System Productivity Facility (ISPF)*.

**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.

## 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.

**Language Environment-conforming.** A characteristic of compiler products (such as VisualAge COBOL COBOL for OS/390 & VM, COBOL for MVS & VM, C/C++ for MVS and VM, PL/I for MVS and VM) that produce object code conforming to the Language Environment format.

**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 pseudotext 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 that describes data items available from the calling program. Both the calling program and the called program can refer to these data items.

**literal.** A character string whose value is specified either by the ordered set of characters comprising the string or by the use of a figurative constant.

**little-endian.** The default format that the PC uses to store binary data. In this format, the most significant digit is on the highest address. Compare with *big-endian*.

**locale.** A set of attributes for a program execution environment indicating 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.

**make file.** 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 magnetic disk and magnetic drum.

\* **mass storage file.** A collection of records that is assigned to a mass storage medium.

\* **megabyte (M).** 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.

**metaclass.** A SOM class whose instances are SOM class-objects. The methods defined in metaclasses are executed without requiring any object instances of the class to exist, and are frequently used to create instances of the class.

**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 unit that defines a method.

\* **method identification entry.** An entry in the METHOD-ID paragraph of the IDENTIFICATION DIVISION; this entry contains clauses that specify the method-name and assign selected attributes to the method definition.

**method invocation.** A communication from one object to another that requests the receiving object to execute a method. In Visual Builder, a method invocation consists of a method name that indicates the requested method, the parameters to be used in executing the method, and, if required, a return variable.

\* **method-name.** A user-defined word that identifies a method. The name is used in a call to specify the requested operation.

\* **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.

**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.

\* **native character set.** The implementor-defined character set associated with the computer specified in the OBJECT-COMPUTER paragraph.

\* **native collating sequence.** The implementor-defined collating sequence associated with the computer specified in the OBJECT-COMPUTER paragraph.

\* **negated combined condition.** The NOT logical operator immediately followed by a parenthesized combined condition. See also *condition* and *combined condition*.

\* **negated simple condition.** The NOT logical operator immediately followed by a simple condition. See also *condition* and *simple condition*.

**nested program.** A program that is directly contained within another program.

\* **next executable sentence.** The next sentence to which control will be transferred after execution of the current statement is complete.

\* **next executable statement.** The next statement to which control will be transferred after execution of the current statement is complete.

\* **next record.** The record that logically follows the current record of a file.

\* **noncontiguous items.** Elementary data items in the WORKING-STORAGE SECTION and LINKAGE SECTION that bear no hierarchic relationship to other data items.

**nondate.** Any of the following:

- 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

\* **nonnumeric item.** A data item whose description permits its content to be composed of any combination of characters from the character set of the computer. Certain categories of nonnumeric items can be formed from more restricted character sets.

\* **nonnumeric literal.** A literal that is bounded by quotation marks. The string of characters can include any character in the character set of the computer.

**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-edited item.** A numeric item that is in a form appropriate for use in printed output. It can consist of external decimal digits from 0 through 9, the decimal point, commas, the dollar sign, editing sign control symbols, plus other editing symbols.

\* **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 item.** 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.

\* **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.** (1) 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. (2) In Visual Builder, a computer representation of something that a user can work with to perform a task. An object can appear as text or as an icon.

**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 deck.** A portion of an object program suitable as input to a linkage editor. Synonymous with *object module* and *text deck*.

**object instance.** See *object*.

**object module.** Synonym for *object deck* or *text deck*.

\* **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. 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 applies 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 that is to be deleted from the next revision of Standard COBOL.

**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 program.

\* **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.

**override.** To redefine a method (inherited from a parent class) in a subclass.

## P

**packed decimal item.** See *internal decimal item*.

\* **padding character.** An alphanumeric character that is used to fill the unused character positions in a physical record.

**page.** A vertical division of output data that represents a physical separation of the data. The separation is based on internal logical requirements or external characteristics of the output medium or both.

\* **page body.** That part of the logical page in which lines can be written or spaced or both.

\* **paragraph.** In the PROCEDURE DIVISION, a paragraph-name followed by a separator period and by zero, one, or more sentences. In the IDENTIFICATION DIVISION and ENVIRONMENT DIVISION, a paragraph header followed by zero, one, or more entries.

\* **paragraph header.** A reserved word, followed by the separator period, that indicates the beginning of a paragraph in the IDENTIFICATION DIVISION and ENVIRONMENT DIVISION. The permissible paragraph headers in the IDENTIFICATION DIVISION are:

PROGRAM-ID. (Program IDENTIFICATION DIVISION)  
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 clause of a method call. Arguments provide additional information that the invoked method can use to perform the requested operation.

\* **phrase.** A phrase is 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 run-time 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.** Accessible only by methods of the class that defines the instance 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 program. 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).

**PROCEDURE DIVISION.** One of the four main component parts of a COBOL program, class definition, or method definition. The PROCEDURE DIVISION contains instructions for solving a problem. The PROCEDURE DIVISION for a program or method can contain imperative statements, conditional statements, compiler-directing statements, paragraphs, procedures, and sections. The PROCEDURE DIVISION for a class contains only method definitions.

**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.

**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 run-time 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 header, a user-defined word that identifies a COBOL source program.

**project environment.** The central location where you launch your COBOL tools such as the editor and job monitor and work with COBOL files or data sets.

\* **pseudotext.** A sequence of text words, comment lines, or the separator space in a source program or COBOL library bounded by, but not including, pseudotext delimiters.

\* **pseudotext delimiter.** Two contiguous equal sign characters (==) used to delimit pseudotext.

**public.** Accessible by getX and setX methods from outside the class that defines the instance data X.

\* **punctuation character.** A character that belongs to the following set:

Character	Meaning
,	Comma
;	Semicolon
:	Colon
.	Period (full stop)
"	Quotation mark
(	Left parenthesis
)	Right parenthesis
	Space
=	Equal sign

## Q

**QSAM (Queued Sequential Access Method).** An extended version of the basic sequential access method (BSAM). When this method is used, a queue is formed of input data blocks that are awaiting processing or of output data blocks that have been processed and are awaiting transfer to auxiliary storage or to an output device.

\* **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 VisualAge COBOL 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 alphanumeric data item by specifying the leftmost character and length relative to the leftmost character of another alphanumeric 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

Character	Meaning
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.

**reusable environment.** A reusable environment is created when you establish an assembler program as the main program by using either ILBOSTP0 programs, IGZERRE programs, or the RTEREUS run-time option.

**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. In Language Environment, refers to either a procedure, function, or subroutine.

\* **routine-name.** A user-defined word that identifies a procedure written in a language other than COBOL.

\* **run time.** The time at which an object program is executed. Synonymous with *object time*.

**run-time environment.** The environment in which a COBOL program executes.

\* **run unit.** A stand-alone object program, or several object programs, that interact via COBOL CALL statements, which 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.** The name of an operation and its parameters.

\* **simple condition.** Any single condition chosen from the 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 *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.

**SOM.** See *System Object Model (SOM)*.

\* **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 header, if specified, or with the absence of additional source program lines.

\* **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-character word.** A reserved word that is an arithmetic operator or a relation character.

**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 data format.** The concept used in describing the characteristics of data in a COBOL DATA DIVISION under which the characteristics or properties of the data are expressed in a form oriented to the appearance of the data on a printed page of infinite length and breadth, rather than a form oriented to the manner in which the data is stored internally in the computer, or on a particular external medium.

\* **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 ANSI 85 COBOL standard input and output language and all of the extensions described in *IBM COBOL 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.

\* **superclass.** A class that is inherited by another class. See also *subclass*.

**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.

**System Object Model (SOM).** IBM's object-oriented programming technology for building, packaging, and manipulating class libraries. SOM conforms to the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) standards.

## 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 deck.** Synonym for *object deck* or *object module*.

\* **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 pseudotext that is any of the following characters:

- A separator, except for space; a pseudotext 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 pseudotext, 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.

**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.

\* **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.

**VM/SP (Virtual Machine/System Product).** An IBM-licensed program that manages the resources of a single computer so that multiple computing systems appear to exist. Each virtual machine is the functional equivalent of a "real" machine.

**VSAM file system.** A file system that supports COBOL sequential, relative, and indexed organizations. This file system is available as part of IBM VisualAge COBOL and enables you to read and write files on remote systems such as OS/390.

**volume.** A module of external storage. For tape devices it is a reel; for direct-access devices it is a unit.

**volume switch procedures.** System-specific procedures that are executed automatically when the end of a unit or reel has been reached before end-of-file has been reached.

## W

**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, 05 could be interpreted as 2005. 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.

## 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 item.** See *external decimal item*.

---

## List of resources

---

### VisualAge COBOL

*Fact Sheet*, GC26-9052

*Getting Started on Windows*, GC26-8944

*Language Reference*, SC26-9046

*Programming Guide*, SC27-0812

*Visual Builder User's Guide*, SC26-9053

---

### Related publications

#### COBOL for OS/390 & VM

*Compiler and Run-Time Migration Guide*, GC26-4764

*Debug Tool User's Guide and Reference*, SC09-2137

*Diagnosis Guide*, GC26-9047

*Fact Sheet*, GC26-9048

*Installation and Customization under OS/390*,  
GC26-9045

*Language Reference*, SC26-9046

*Licensed Program Specifications*, GC26-9044

*Programming Guide*, SC26-9049

#### COBOL Set for AIX

*Fact Sheet*, GC26-8484

*Getting Started*, GC26-8425

*Language Reference*, SC26-9046

*LPEX User's Guide and Reference*, SC09-2202

*Program Builder User's Guide*, SC09-2201

*Programming Guide*, SC26-8423

#### VisualAge CICS Enterprise Application Development

*Installation*, GC34-5356

*Customization*, SC34-5357

*Operation*, SC34-5358

*Reference Summary*, SX33-6109

*Intercommunication*, SC34-5359

*Problem Determination*, GC34-5360

*Performance*, SC34-5363

*Application Programming*, SC34-5361

*Messages and Codes*, GC34-5362

#### CICS for Windows NT

*Application Programming Guide*, SC33-1888

*Installation Guide*, GC33-1880

*Intercommunication Guide*, SC33-1882

*Messages & Codes*, SC33-1886

*Problem Determination Guide*, SC33-1883

#### DB2

*Application Programming Guide*, S20H-4643

*DATABASE 2 Command Reference for Common Servers*, S20H-4645

*SQL Reference*, S20H-4665

#### SMARTdata Utilities for Windows

*Data Description and Conversion A Data Language Reference*, SC26-7092

*Data Description and Conversion*, SC26-7091

*User's Guide*, SC26-7134

*VSAM in a Distributed Environment*, SC26-7133

#### SOMobjects Developer's Toolkit

*SOMobjects Developer's Toolkit Programmer's  
Reference*

*SOMobjects Developer's Toolkit Programming Guide*

*SOMobjects Developer's Toolkit User's Guide*

**Other**

*Btrieve Programmer's Guide*

---

# Index

## Special Characters

- # cob2 option 147
- .adt file 160
- .asm file 179
- b cob2 option 145
- c cob2 command 145
- .CBL file extension 147
- \*CBL statement 198
- cmain cob2 option 145
- comprc\_ok cob2 option 146
- \*CONTROL statement 198
- .DEF as linker parameter 147
- .DLL as linker parameter 147
- dll cob2 option 146
- .EXE as linker parameter 147
- .EXP as linker parameter 147
- .exp file 146
- g cob2 option
  - for debugging 146
- >>CALLINT statement 198
- host cob2 option
  - for host data format 146
- I cob2 option
  - searching COPY files 146
- .IMP as linker parameter 147
- .LIB as linker parameter 147
- .lib file 146
- main cob2 option
  - specifying main program 146
- .MAP as linker parameter 147
- .OBJ as linker parameter 147
- p cob2 option
  - analyzing performance 147
- q cob2 option 147
- v cob2 option 147
- .wlist file 179

## A

- abbreviations,
  - compiler options 159
- abends
  - using ERRCOUNT run-time option to induce 218
- ACCEPT statement 26
  - using in GUI applications 142
- ACCEPT statement, environment variables used on 142
- accessing files using environment variables 140
- accessing local files 96
- accessing remote files 96
- ADATA compiler option 161
- adding external program reference to object module 390
- adding records
  - to files 110
- adding stub files 401
- ADDRESS special register, CALL statement 374

- addresses
  - incrementing 377
  - NULL value 377
  - passing between programs 377
  - passing entry point addresses 380
- ADEXIT suboption of EXIT compiler option 173
- AIX, porting to 357
- ALL subscript 43
- ALPHABET clause, establishing collating sequence 8
- alphanumeric date fields,
  - contracting 446
- alternate collating sequence 121
- alternate file system
  - file system ID 140
  - using environment variables 140
- ANALYZE compiler option 161
- ANNUITY intrinsic function 44
- APOST compiler option 184
- applications, porting
  - architectural differences between platforms 351
  - language differences between PC and mainframe 351
  - mainframe to PC
    - choosing compiler options 351
  - mainframe to workstation
    - running mainframe applications on the workstation 353
  - PC to AIX 357
  - PC to mainframe
    - PC-only language features 356
    - PC-only names 357
  - using COPY to isolate platform-specific code 352
- arguments
  - describing in calling program 374
  - IDL passing conventions 327
  - passing BY VALUE 367
  - to main program 386
- arithmetic
  - calculation on dates
    - convert date to COBOL integer format (CEECLDY) 498
    - convert date to Lilian format (CEEDAYS) 509
    - convert timestamp to number of seconds (CEESECS) 529
    - get current Greenwich Mean Time (CEEGMT) 515
  - COMPUTE statement simpler to code 41
  - error handling 126
  - with intrinsic functions 42
- arithmetic comparisons 46
- arithmetic evaluation
  - data format conversion 38
  - examples 45
  - fixed-point versus floating-point 45
  - intermediate results 481

- arithmetic evaluation (*continued*)
  - performance tips 453
  - precedence 482
  - precision 481
- arithmetic expression
  - as reference modifier 86
  - description of 41
  - in nonarithmetic statement 488
  - in parentheses 41
  - with MLE 441
- arithmetic operation
  - with MLE 441
- arithmetic operations
  - with MLE 437
- ASCII
  - DBCS portability 355
  - portability considerations 353
- asm file 179
- assembler language
  - LIST option 458
- assembler language programs
  - debugging 241
- assembler programs
  - listing of 458
- assigning values 23
- ASSIGNment name, environment variable 140
- assumed century window for nondates 436
- AT END (end-of-file) 128
- autoexe.bat, defining environment variables 137
- avoiding coding errors 451

## B

- backward branches 452
- base locator 236
- BASE statement 396
- BASIS statement 198
- Batch compilation 185
- batch debugging, activating 218
- big-endian 36
  - format for data representation 161
  - representation of integers 354
- BINARY
  - general description 35
  - synonyms 33
  - using efficiently 35
- BINARY compiler option 161
- binary data, data representation 161
- binary data item
  - byte reversal 36
  - general description 35
  - intermediate results 485
  - using efficiently 35
- binary search of a table 64
- branch, implicit 75
- Btrieve files
  - processing files 97
- BY VALUE, valid data types 367

byte-reversed integers 354

## C

### C/C/C

- and COBOL 364
- called from COBOL, linkage conventions 365
- communicating with COBOL 364
- data types, correspondence with COBOL 367
- functions called from COBOL, example 367
- functions calling COBOL, example 368
- multiple calls to a COBOL program 365
- variable parameter list 365

call conventions 168

call interface conventions

- CDECL 162
- FAR16 162
- OPTLINK 162
- PASCAL16 162
- STDCALL 162
- SYSTEM 162
- with ODBC 267

CALL statement

- ... USING 374
- BY CONTENT 373
- BY REFERENCE 374
- BY VALUE 373
- CALL identifier 364
- CALL literal 364
- effect of CALLINT option 162
- effect of DYNAM compiler option 168
- exception condition 133
- for error handling 133
- handling of programs name in 183
- overflow condition 133
- to invoke date and time services 465
- with ON EXCEPTION 133
- with ON OVERFLOW 17

callable services

- CEECBLDY—convert date to COBOL integer format 498
- CEEDATE—convert Lilian date to character format 502
- CEEDATM—convert seconds to character timestamp 505
- CEEDAYS—convert date to Lilian format 509
- CEEDYWK—calculate day of week from Lilian date 513
- CEEGMT—get current Greenwich Mean Time 515
- CEEGMTO—get offset from Greenwich Mean Time to local time 517
- CEEISEC—convert integers to seconds 519
- CEELOCT—get current local time 521
- CEEQCEN—query the century window 523

callable services (*continued*)

- CEESCEN—set the century window 525
- CEESECI—convert seconds to integers 526
- CEESECS—convert timestamp to number of seconds 529
- CEEUTC—get Coordinated Universal Time 533
- IGZEDT4—get current date with four-digit year 533
- CALLINT compiler option 162
- CALLINT statement 198
- calls
  - dynamic 364
  - exception condition 133
  - Linkage Section 375
  - overflow condition 133
  - passing arguments 374
  - passing data 373
  - receiving parameters 375
  - recursive 370
  - static 364
  - to date and time services 465
- CANCEL statement
  - handling of programs name in 183
- case structure 69
- CBL file extension 147
- CBL statement 198
- CDECL suboption of CALLINT compiler option 162
- CEECBLDY—convert date to COBOL integer format
  - example 498
  - syntax 498
- CEEDATE—convert Lilian date to character format
  - example 503
  - syntax 502
  - table of sample output 504
- CEEDATM—convert seconds to character timestamp
  - CEESECI callable service 526
  - example 506
  - syntax 505
  - table of sample output 508
- CEEDAYS—convert date to Lilian format
  - example 511
  - syntax 509
- CEEDYWK—calculate day of week from Lilian date
  - example 513
  - syntax 513
- CEEGMT—get current Greenwich Mean Time
  - example 516
  - syntax 515
- CEEGMTO—get offset from Greenwich Mean Time to local time
  - example 518
  - syntax 517
- CEEISEC—convert integers to seconds
  - example 520
  - syntax 519
- CEELOCT—get current local time
  - example 522
  - syntax 521

- CEEQCEN—query the century window
  - example 524
  - syntax 523

- CEESCEN—set the century window
  - example 525
  - syntax 525

- CEESECI—convert seconds to integers
  - example 527
  - syntax 526

- CEESECS—convert timestamp to number of seconds
  - example 531
  - syntax 529

century window

- assumed for nondates 436
- CEECBLDY callable service 500
- CEEDAYS callable service 511
- CEESCEN callable service 525
- CEESECS callable service 531
- concept 471
- fixed 428
- sliding 428

chained list processing 377

changing

- characters to numbers 89
- file-name 10
- title on source listing 6

CHAR compiler option 163

CHAR intrinsic function 90

character strings 170

character timestamp

- converting Lilian seconds to (CEEDATM) 505
- example 506
- converting to COBOL integer format (CEECBLDY) 498
- example 500
- converting to Lilian seconds (CEESECS) 529
- example 529

CHECK(OFF) run-time option 460

CHECK run-time option 217

checking errors, flagging at run time 217

checking for valid data 71

- numeric 40

CICS

- ASCII-EBCDIC differences 251
- coding COBOL applications to run under CICS 250
- coding restrictions 250
- commands relevant to COBOL 249
- compiler options, selecting 253
- compiler restrictions 250
- debugging programs 254
- effect of TRAP run-time option 219
- performance considerations 451
- preparing COBOL applications to run under CICS 253
- programming considerations 249
- run-time, selecting 254
- system date 251
- using dynamic calls 251

CICSENV command 249

CICSMAP command 249

CICSRUN command 249

CICSTCL command 249

- class condition 71
  - numeric 40
- class definition 274
- class programs
  - use as dynamic link libraries (DLLs) 393
- class test 223
  - numeric 40
- client definition 285
- cob2 command
  - description 144
  - examples using, for compiling 145
  - examples using, for linking 154
  - file extensions supported 147
  - options 145
- cob2\_r command
  - threading example 408
- COBCPYEXT environment variable 138
- COBLSTDIR environment variable 138
- COBMSGSG environment variable 140
- COBOL 85 Standard
  - definition xi
- COBOL language usage with SQL
  - statements 246
- COBOL terms 21
- COBOPT environment variable 139
- COBPATH environment variable 139
- COBRTOPT environment variable 140
- code
  - copy 463
  - optimized 458
- code pages
  - national language support 417
- coding
  - class definition 274
  - client definition 285
  - condition tests 72
  - DATA DIVISION 11
  - decisions 67
  - efficient 451
  - ENVIRONMENT DIVISION 7
  - EVALUATE statement 69
  - file input/output overview 100
  - for files 105
  - IDENTIFICATION DIVISION 5
  - IF statement 67
  - input/output overview 105
  - loops 71
  - metaclass definition 301
  - method definition 277
  - OO programs 271
  - PROCEDURE DIVISION 14
  - restrictions for programs for CICS 250
  - subclass definition 289
  - tables 51
  - techniques 451
  - test conditions 72
- collating sequence
  - alternate 8
  - ASCII 8
  - EBCDIC 8
  - HIGH-VALUE 8
  - ISO 7-bit code 8
  - LOW-VALUE 8
  - MERGE 8
  - NATIVE 8
- collating sequence (*continued*)
  - nonnumeric comparisons 8
  - SEARCH ALL 8
  - SORT 8
  - specifying 8
  - symbolic character in the 9
  - the ordinal position of a character 90
- COLLSEQ compiler option 165
- columns in tables 51
- command line arguments 386
- command prompt, defining environment variables 137
- COMMON attribute 6
- COMP (COMPUTATIONAL) 35
- COMP-1 (COMPUTATIONAL-1) 36
- COMP-2 (COMPUTATIONAL-2) 36
- COMP-3 (COMPUTATIONAL-3) 36
- COMP-4 (COMPUTATIONAL-4) 35
- COMP-5 (COMPUTATIONAL-5) 35
- comparison
  - of date fields 432
- compatible dates
  - in comparisons 432
  - with MLE 433
- compilation
  - statistics 234
- COMPILE compiler option 166
  - use NOCOMPILE to find syntax errors 226
- compile-time considerations
  - cob2 command options 145
  - compiler directed errors 150
  - compiling programs 146
  - compiling programs without linking 145
  - display compile and link steps 147
  - error message severity 149
  - executing compile and link steps after display 147
- compile-time error messages
  - choosing severity to be flagged 228
  - determining what severity level to produce 174
  - embedding in source listing 228
- compiler
  - calculation of intermediate results 481
  - generating list of error messages 150
  - invoking 144
  - limits 11
- compiler-directing statements 198
  - list 17
  - overview 17
- compiler listings
  - getting 231
  - specifying output directory 138
- compiler messages
  - analyzing 444
- compiler options
  - abbreviations 159
  - ADATA 161
  - ANALYZE 161
  - APOST 184
  - BINARY 161
  - CALLINT 162
  - CHAR 163
  - COLLSEQ 165
- compiler options (*continued*)
  - COMPILE 166
  - CURRENCY 166
  - DATEPROC 167
  - DYNAM 460
  - ENTRYINT 168
  - EXIT 169
  - FLAG 228
  - FLAGSTD 175
  - FLOAT 176
  - for application portability 351
  - for debugging 225
  - IDLGEN 177
  - LIB 178
  - LINECOUNT 179
  - LIST 231
  - MAP 230
  - NOCOMPILE 226
  - NUMBER 232
  - on compiler invocation 233
  - OPTIMIZE 460
  - PGMNAME 182
  - PROBE 183
  - PROFILE 184
  - QUOTE 184
  - selecting for CICS 253
  - SEPOBJ 185
  - SEQUENCE 227
  - SIZE 187
  - SOSI 187
  - SOURCE 231
  - SPACE 189
  - specifying
    - cob2 command 144
    - environment variable 139
    - PROCESS (CBL) statement 148
    - using COBOPT 139
  - SQL 247
  - SSRANGE 460
  - status 234
  - TERMINAL 190
  - TEST 460
  - THREAD 191
  - TRUNC 192
  - TRUNC(STD|OPT|BIN) 460
  - TYPECHK 194
  - VBREF 231
  - WSCLEAR 195
  - XREF 230
  - YEARWINDOW 197
  - ZWB 197
- compiling a DLL 393
- completion code, sort 122
- complex OCCURS DEPENDING ON
  - basic forms of 491
  - complex ODO item 491
  - variably located data item 492
  - variably located group 492
- computation
  - constant data items 452
  - duplicate 453
  - subscript 456
- COMPUTATIONAL (COMP) 35
- COMPUTATIONAL-1 (COMP-1) 36
- COMPUTATIONAL-2 (COMP-2) 36
- COMPUTATIONAL-3 (COMP-3) 36

- COMPUTATIONAL-3 date fields,
  - potential problems 446
- COMPUTATIONAL-4 (COMP-4) 35
- COMPUTATIONAL-5 (COMP-5) 35
- computer, describing 7
- concatenating data items 79
- condition handling 218
  - date and time services and 465
- condition-name 435
- condition testing 72
- conditional expression
  - EVALUATE statement 68
  - IF statement 67
  - PERFORM statement 76
- conditional statement
  - in EVALUATE statement 68
  - list of 16
  - overview 16
  - with NOT phrase 17
- config.sys, defining environment
  - variables 137
- CONFIGURATION SECTION 7
- considerations
  - System/390 host data type 479
- constant
  - computations 452
  - data items 452
  - figurative 22
- constructor method 303
- contained program integration 459
- continuation
  - of program 127
  - syntax checking 166
- CONTINUE statement 68
- contracting alphanumeric dates 446
- control
  - in nested programs 360
  - program flow 67
  - transfer 359
- CONTROL statement 198
- conversion of data formats 38
- convert character format to Lilian date (CEEDAYS) 509
- convert Lilian date to character format (CEEDATE) 502
- converting data items
  - characters to numbers 89
  - INSPECT statement 87
  - reversing order of characters 89
  - to integers 87
  - to uppercase or lowercase 88
  - with intrinsic functions 88
- converting files to expanded date form,
  - example 431
- copy
  - libraries 464
- copy code, obtaining from user-supplied
  - module 170
- COPY files
  - searching for 146
  - specifying search paths with
    - SYSLIB 139
  - with ODBC 260
- COPY name
  - file extensions searched 138
- COPY statement 198
  - example 464

- COPY statement (*continued*)
  - form and definition 200
  - nested 464
  - uses for portability 352
- copybook 139
  - ODBC3D.CPY sample 265
  - ODBC3EG.CPY sample 261
  - ODBC3P.CPY 262
  - search rules 200
  - supplied ODBC3EG.CPY 261
  - with ODBC 260
- copying, code 463
- counting data items 87
- cross-reference
  - data and procedure names 230
  - embedded 231
  - program-name 238
  - special definition symbols 239
  - verbs 231
- cross-reference list 195
- CURRENCY compiler option 166
- currency signs
  - euro 47
  - hex literals 47
  - multiple-character 47
  - using 47
- CURRENT-DATE intrinsic function 43
- customizing
  - setting environment variables 137

**D**

- data
  - concatenating 79
  - efficient execution 451
  - format, numeric types 33
  - format conversion 38
  - grouping 376
  - incompatible 40
  - joining 79
  - numeric 31
  - passing 373
  - splitting 81
  - validation 40
- data and procedure name cross-reference,
  - description 230
- data areas, dynamic 168
- data definition 235
- data definition attribute codes 235
- data description entry 11
- DATA DIVISION
  - client 285
  - coding 11
  - description 11
  - FD entry 11
  - FILE SECTION 11
  - limits 11
  - LINKAGE SECTION 13
  - listing 231
  - mapping of items 231
  - method 278
  - OCCURS clause 51
  - restrictions 11
  - WORKING-STORAGE SECTION 11
- DATA DIVISION items, mapping 180
- data item
  - common, in subprogram linkage 375

- data item (*continued*)
  - concatenating 79
  - converting characters to numbers 89
  - converting to
    - uppercase/lowercase 88
  - converting with INSPECT 87
  - converting with intrinsic
    - functions 88
  - counting 87
  - evaluating with intrinsic functions 90
  - finding the smallest/largest in
    - group 90
  - index 54
  - numeric 31
  - reference modification 84
  - referencing substrings 84
  - replacing 87
  - reversing characters 89
  - splitting 81
  - unused 181
  - variably located 492
- data-manipulation
  - nonnumeric data 79
- data-name
  - cross-reference 237
  - in MAP listing 235
- data representation
  - compiler option affecting 161
  - portability 353
- date and time
  - format
    - converting from character format
      - to COBOL integer format (CEECLDY) 498
    - converting from character format
      - to Lilian format (CEEDAYS) 509
    - converting from integers to
      - seconds (CEEISEC) 519
    - converting from Lilian format to
      - character format (CEEDATE) 502
    - converting from seconds to
      - character timestamp (CEEDATM) 505
    - converting from seconds to
      - integers (CEESECI) 526
    - converting from timestamp to
      - number of seconds (CEESECS) 529
  - getting date and time (CEELOCT) 521
- services
  - CEECLDY—convert date to
    - COBOL integer format 498
  - CEEDATE—convert Lilian date to
    - character format 502
  - CEEDATM—convert seconds to
    - character timestamp 505
  - CEEDAYS—convert date to Lilian
    - format 509
  - CEEDYWK—calculate day of week
    - from Lilian date 513
  - CEEGMT—get current Greenwich
    - Mean Time 515
  - CEEGMTO—get offset from
    - Greenwich Mean Time to local
      - time 517

- date and time *(continued)*
    - services *(continued)*
      - CEEISEC—convert integers to seconds 519
      - CEELOCT—get current local time 521
      - CEEQCEN—query the century window 524
      - CEESCEN—set the century window 525
      - CEESECI—convert seconds to integers 526
      - CEESECS—convert timestamp to number of seconds 529
      - CEEUTC—get Coordinated Universal Time 533
      - condition feedback 467
      - condition handling 465
      - examples of using 466
      - feedback code 465
      - invoking with a CALL statement 465
      - list of 497
      - overview 497
      - performing calculations with 466
      - picture strings 468
      - return code 465
      - RETURN-CODE special register 465
      - syntax 529
  - date arithmetic 441
  - date comparisons 432
  - Date-Compiled paragraph 5
  - date field expansion 430
    - advantages 428
  - date fields
    - potential problems 446
  - date information, formatting 141
  - DATE-OF-INTEGGER intrinsic function 44
  - date operations
    - intrinsic functions 28
  - date processing with internal bridges
    - advantages 428
  - date windowing
    - advantages 428
    - example 434
    - how to control 443
    - MLE approach 428
    - when not supported 434
  - DATEPROC compiler option 167
    - analyzing warning-level diagnostic messages 444
  - DATEVAL intrinsic function 443
  - day of week, calculating with CEEDYWK 513
  - DB2
    - bind file name 248
    - bind file name, default 247
    - COBOL language usage with SQL statements 246
    - coding considerations 245
    - coprocessor 245
    - ignored options 248
    - in a multithreading environment 407
    - options 247
    - package name 247
  - DB2 *(continued)*
    - package name, default 247
    - return codes 246
    - SQL INCLUDE statement 246
    - SQL statements 245
  - DB2DBDFT environment variable 139
  - DB2INCLUDE environment variable 246
  - DBCS user-defined words, listed in XREF output 230
  - DEBUG run-time option 218
  - Debug Tool
    - compiler options for maximum support 231
  - debugging
    - activating batch features 218
    - assembler 241
    - CICS programs 254
    - producing symbolic information 146
    - useful compiler options 225
    - user exits 240
    - using COBOL language features 221
  - debugging, language features
    - class test 223
    - debugging declaratives 223
    - file status keys 223
    - INITIALIZE statements 223
    - scope terminators 222
    - SET statements 223
  - declarative procedures
    - USE FOR DEBUGGING 223
  - DEF extension as linker parameter 147
  - defining files
    - in COBOL programs 105
    - files, overview 100
  - DELETE statement 198
  - deleting records from file 112
  - delimited scope statement
    - description of 16
    - nested 18
  - depth in tables 52
  - DESC suboption of CALLINT compiler option 163
  - DESCRIPTION statement 396
  - DESCRIPTOR suboption of CALLINT compiler option 163
  - DGROUP 398
  - diagnostic messages
    - from millennium language extensions 444
  - diagnostics, program 234
  - differences with host COBOL 475
  - direct-access
    - direct indexing 55
  - directories
    - adding a path to 146
    - specifying path 140
    - where error listing file is written 150
  - directories, for linker search 154
  - DISPLAY (USAGE IS) 34
  - DISPLAY statement
    - displaying data values 27
    - using in debugging 222
    - using in GUI applications 142
  - DISPLAY statement, environment variables used on 142
  - DLL extension as linker parameter 147
  - DLL files
    - setting directory path 140
  - do loop 76
  - do-until 76
  - do-while 76
  - documentation of program 7
  - DOS, running under 401
  - dumps, when not produced 219
  - duplicate computations 453
  - DYNAM compiler option 168
    - description 168
    - performance considerations 460
  - dynamic calls
    - using in a CICS environment 251
  - dynamic link libraries
    - dll cob2 option 146
    - building 389
    - CALL identifier example 392
    - CALL literal overview 391
    - CICS considerations 252
    - compiling 393
    - creating a module definition file 390
      - example of 392
    - creating DLL source files example of 390
    - creating under Windows 146
    - export file, creating with cob2 146
    - files, setting directory path 140
    - function 389
    - import libraries, creating with cob2 146
    - linker resolving references 390
    - linking 393
    - overview of dynamic linking 390
    - purpose 390
    - subprograms and outermost programs 389
    - use of class programs as DLLs 393
  - dynamic linking 364
    - advantages 390
    - linker resolving DLL references 390
    - overview 390
  - dynamic loading, requirements 140
- ## E
- E-level error message 149
  - EBCDIC
    - DBCS portability 355
    - portability considerations 353
  - EBCDIC\_CODEPAGE environment variable 141
  - efficiency of coding 451
  - EJECT statement 198
  - embedded cross-reference 231
    - example 239
  - embedded error messages 228
  - embedded MAP summary 230
  - embedded SQL
    - advantages 255
  - enclave 359
  - end-of-file phrase (AT END) 128
  - ENTER statement 198
  - entry point
    - ENTRY label 359

- entry point (*continued*)
    - multiple, Windows restriction 381
    - passing entry addresses of 380
    - procedure-pointer data item 380
  - entry points 389
    - call convention 168
  - ENTRY statement
    - handling of programs name in 183
  - ENTRYINT compiler option 168
  - environment, preinitializing 365
  - environment differences, System/390 and the workstation 355
  - ENVIRONMENT DIVISION
    - class 275
    - client 285
    - collating sequence coding 8
    - CONFIGURATION SECTION 7
    - description 7
    - INPUT-OUTPUT SECTION 7
    - method 278
    - subclass 290
  - environment-name 7
  - environment variables
    - accessing files with 140
    - ASSIGNMENT name 140
    - COBMSG 140
    - COBOPT 139
    - COBPAT 139
    - COBRTOPT 140
    - DB2DBDFT 139
    - EBCDIC\_CODEPAGE 141
    - LANG 141
    - LC\_TIME 141
    - library-name 139
    - LOCPATH 141
    - NLSPATH 142
    - run time 140
    - search order precedence 138
    - setting 137
    - SYSIN, SYSIPT, SYSOUT, SYSLIST, SYSLST, CONSOLE, SYSPUNCH, SYSPCH 142
    - SYSLIB 139
    - System Object Model (SOM) 312
    - TEMP 142
    - TEMPMEM 139
    - text-name 139
    - TZ 142
    - used by the compiler 138
  - ERRCOUNT run-time options 218
  - ERRMSG, for generating list of error messages 150
  - error
    - arithmetic 126
    - example of message table 58
    - handling 125
    - messages, compiler
      - choosing severity to be flagged 228
      - embedding in source listing 228
  - error messages
    - appearing in abbreviated form 142
    - compiler-directed 150
    - correcting 149
    - determining what severity level to produce 174
    - format 151
  - error messages (*continued*)
    - generating a list of 150
    - location in listing 151
    - setting national language 141
    - severity levels 149
  - errors
    - compiler-directed 150
    - flagging at run time 217
  - ESTAE exits, affected by TRAP run-time option 219
  - euro currency sign 47
  - EVALUATE statement
    - case structure 69
    - structured programming 451
  - evaluating data item contents
    - class test 40
    - INSPECT statement 87
    - intrinsic functions 90
  - examples
    - CEECLDY—convert date to COBOL integer format 500
    - CEEDATE—convert Lilian date to character format 503
    - CEEDATM—convert seconds to character format 506
    - CEEDAYS—convert date to Lilian format 511
    - CEEDYWK—calculate day of week from Lilian date 513
    - CEEGMT—get current GMT 516
    - CEEGMTO—get offset from Greenwich Mean Time to local time 518
    - CEEISEC—convert integers to seconds 520
    - CEELOCT—get current local time 522
    - CEEQCEN—query century window 524
    - CEESCEN—set century window 525
    - CEESECI—convert seconds to integers 527
    - CEESECS—convert timestamp to number of seconds 531
    - IGZEDT4—get current date with four-digit year 534
  - exception condition 133
  - EXCEPTION/ERROR declarative
    - description 129
    - file status key 130
  - exceptions, intercepting 219
  - EXE extension as linker parameter 147
  - EXIT compiler option 169
  - exit modules
    - called for SYSADATA data set 173
    - debugging 240
    - loading and invoking 171
    - when used in place of library-name 172
    - when used in place of SYSLIB 172
    - when used in place of SYSPRINT 173
  - EXIT PROGRAM statement
    - in main program 360
    - in subprogram 360
  - EXP extension as linker parameter 147
  - exp file 146
  - expanded IF statement 67
  - explicit scope terminator 17
  - exponentiation
    - evaluated in fixed-point arithmetic 483
    - evaluated in floating-point arithmetic 487
    - performance tips 454
  - EXPORTS statement 397
    - listing callable subprograms 390
  - EXTERNAL clause
    - example for files 384
    - for data items 383
    - used for input/output 383
  - external data
    - sharing 383
  - external decimal data item 34
  - external file 11
  - external floating-point data item 34
  - external program reference
    - added to module 390
- ## F
- factoring expressions 452
  - FAR16 suboption of CALLINT compiler option 162
  - feedback token
    - date and time services and 467
  - figurative constant 22
  - file access mode
    - dynamic 103
    - for indexed files 101
    - for relative files 102
    - for sequential files 101
    - random 102
    - sequential 102
    - summary table of 100
  - file conversion
    - with millennium language extensions 431
  - file extensions
    - as linker parameters 147
    - for error messages listing 150
  - file name
    - change 10
  - file organization
    - indexed 101
    - line-sequential 101
    - overview 100
    - relative 102
    - sequential 101
  - file position indicator 106
  - FILE SECTION
    - description 11
    - EXTERNAL clause 11
    - GLOBAL clause 11
    - record description 11
  - FILE STATUS clause
    - file loading 110
    - using 129
    - with VSAM return code 131
  - file status key
    - checking for successful OPEN 129
    - set for error handling 223
    - to check for I/O errors 129
    - used with VSAM return code 131

- file system support
  - Btrieve 218
  - Encina SFS 218
  - STL 218
  - using FILESYS run-time option to
    - access 218
  - VSAM 218
- files
  - accessing local files 96
  - accessing remote files 96
  - accessing using environment
    - variables 140
  - adding records to 110
  - affected by TRAP run-time
    - option 219
  - associating program files to external
    - files 7
  - Btrieve 95
  - COBOL coding
    - overview 105
  - comparison of file organizations 100
  - deleting records from 112
  - describing 11
  - extensions supported by cob2 147
  - file position indicator 106
  - multiple, compiling 144
  - opening 107
  - passed to compiler or linker 147
  - processing
    - Btrieve files 97
    - STL files 97
    - VSAM files 97
  - reading records from 109
  - replacing records in 111
  - STL 95
  - updating records 112
  - usage explanation 10
  - VSAM 95
- FILESYS run-time options 218
- finding the largest or smallest data
  - item 90
- finding the length of data items 92
- fixed century window 428
- fixed-point arithmetic
  - comparisons 46
  - evaluation 45
  - example evaluations 46
  - exponentiation 483
- fixed-point data
  - binary 35
  - conversions between fixed- and
    - floating-point data 39
  - external decimal 34
  - intermediate results 482
  - packed-decimal 36
  - planning use of 453
- FLAG compiler option 174
  - compiler output 228
  - description 228
- flags 72
- FLAGSTD compiler option 175
- FLOAT compiler option 176
- floating-point arithmetic
  - comparisons 46
  - evaluation 45
  - example evaluations 46
  - exponentiation 487

- floating-point data 354
  - conversions between fixed- and
    - floating-point data 39
  - external floating point 34
  - intermediate results 487
  - internal 36
  - planning use of 453
- four-digit years 468
- full date field expansion
  - advantages 428
- functions
  - storing those frequently used 389

## G

- GETMAIN, saving address of 170
- GLOBAL clause for files 14
- global names 363
- GOBACK statement
  - in main program 360
  - in subprogram 360
- Greenwich Mean Time (GMT)
  - getting offset to local time from
    - (CEEGMT0) 517
  - return Lilian date and Lilian seconds
    - (CEEGMT) 515
- Gregorian character string
  - returning local time as a
    - (CEELOCT) 521
  - example 522
- group item
  - variably located 492
- grouping data 376

## H

- header on listing 6
- heap, defining size of 398
- HEAPSIZ statement 398
- help files
  - setting national language 141
  - specifying path name 142
- hex literal as currency sign 47
- HEXADECIMAL
  - portability considerations 354

## I

- I-level error message 149
- IDENTIFICATION DIVISION
  - class 274
  - client 285
  - coding 5
  - Date-Compiled paragraph 5
  - listing header example 6
  - method 277
  - Program-ID paragraph 5
  - required paragraphs 5
  - TITLE statement 6
- IDL 317
  - access intent specifiers 327
  - attributes 320
  - common types 321
  - complex types 324
  - files 338
  - identifiers 319

- IDL (*continued*)
  - literal arguments 327
  - mapping to COBOL 319
  - operations 319
  - parameter-passing conventions 327
  - passing complex types 327
- IDL type
  - any 324
  - array 325
  - enum 322
  - interface 322
  - long 322
  - sequence 325
  - short 323
  - string 323
  - struct 326
  - union 326
- IDLGEN compiler option 177
- IDLStringFromCOBOL 342
- IDLStringToCOBOL 342
- IEEE
  - portability considerations 354
- IF statement
  - coding 67
  - nested 68
  - with null branch 67
- IGZEDT4—get current date with
  - four-digit year 533
- IMP extension as linker parameter 147
- imperative statement, list 16
- implicit scope terminator 17
- IMPORTS statement
  - listing DLL program locations 390
- incompatible data 40
- incrementing addresses 377
- index
  - data item 55
- index, table 54
- index key, detecting faulty 132
- index-name subscripting 55
- index range checking 227
- indexed file organization 101
- Indexed files
  - file access mode 101
- indexing
  - example 59
  - preferred to subscripting 455
  - tables 55
- INITIAL attribute 6
- INITIALIZE statement
  - examples 23
  - loading table values 56
  - using for debugging 223
- initializing
  - a table 56
  - the run-time environment 419
- INITINSTANCE initialization DLL 399
- inline PERFORM 75
- input
  - overview 100
- input/output
  - checking for errors 129
  - coding overview 105
  - GUI applications 142
  - introduction 100
  - logic flow after error 127
  - processing errors for QSAM files 127

- input/output (*continued*)
  - processing errors for VSAM files 127
- input/output coding
  - AT END (end-of-file) phrase 128
  - checking for successful operation 129
  - checking VSAM return codes 131
  - detecting faulty index key 132
  - error handling techniques 127
  - EXCEPTION/ERROR
    - declaratives 129
- INPUT-OUTPUT SECTION 7
- input procedure
  - requires RELEASE or RELEASE FROM 118
  - restrictions 119
  - using 118
- INSERT statement 198
- INSPECT statement 87
- inspecting data 87
- INTEGER intrinsic function 87
- INTEGER-OF-DATE intrinsic function 43
- integers
  - converting Lilian seconds to (CEESECI) 526
  - converting to Lilian seconds (CEEISEC) 519
- Interface Repository (IR)
  - accessing 311
  - definition 311
  - populating 312
- intermediate results 481
- internal bridges
  - advantages 428
  - example 430
  - for date processing 429
- internal floating-point data
  - bytes required 36
  - defining 36
  - uses for 36
- intrinsic functions
  - as reference modifier 86
  - compatibility with CEELOCT callable service 521
  - converting character data items 88
  - DATEVAL 443
  - evaluating data items 90
  - example of
    - ANNUITY 44
    - CHAR 90
    - CURRENT-DATE 43
    - INTEGER 87
    - INTEGER-OF-DATE 43
    - LENGTH 43
    - LOG 44
    - LOWER-CASE 88
    - MAX 43
    - MEAN 45
    - MEDIAN 45
    - MIN 86
    - NUMVAL 89
    - NUMVAL-C 43
    - ORD 90
    - ORD-MAX 91
    - PRESENT-VALUE 44
    - RANGE 45
    - REM 44

- intrinsic functions (*continued*)
  - example of (*continued*)
    - REVERSE 89
    - SQRT 44
    - SUM 65
    - UPPER-CASE 88
    - WHEN-COMPILED 93
  - intermediate results 485
  - introduction to 27
  - nesting 28
  - numeric functions
    - examples of 42
    - nested 43
    - special registers as arguments 43
    - table elements as arguments 43
    - type of—integer, floating-point, mixed 42
    - uses for 42
  - processing table elements 65
  - simplifying coding 463
  - UNDATE 443
- INVALID KEY phrase 132
- INVOKE statement
  - use with PROCEDURE DIVISION RETURNING 382
- invoking
  - date and time services 465
- invoking the compiler and linker 144

## J

- job stream 359

## L

- LABEL declarative 198
- LANG environment variable 141
- language features for debugging
  - DISPLAY statements 222
- last-used state 360
- LC\_COLLATE environment variable 141
- LC\_MESSAGES environment variable 141
- LC\_TIME environment variable 141
- LENGTH intrinsic function
  - example 43
  - variable length results 91
  - versus LENGTH OF special register 92
- length of data items, finding 92
- LENGTH OF special register 374
- level
  - 88 item 71
- level-88 item
  - for windowed date fields 435
  - restriction 435
  - switches and flags 72
- level definition 235
- LIB compiler option
  - description and syntax 178
- LIB extension as linker parameter 147
- lib file 146
- LIBEXIT suboption of EXIT option 172
- library-name
  - alternative if not specified 146
  - specifying path for library text 139

- library-name, when not used 172
- LIBRARY statement 398
- library text
  - specifying path for 139
- Lilian date
  - calculate day of week from (CEEDYWK) 513
  - convert date to (CEEDAYS) 509
  - convert date to COBOL integer format (CEECLDY) 498
  - convert output\_seconds to (CEEISEC) 519
  - convert to character format (CEEDATE) 502
  - get current local date or time as a (CEELOCT) 521
  - get GMT as a (CEEGMT) 515
  - using as input to CEESECI callable service 527
- limits of the compiler 11
- line number 234
- line-sequential file organization 101
- line-sequential files
  - file access mode 101
- LINECOUNT compiler option 179
- LINKAGE SECTION
  - description 375
  - GLOBAL clause 14
  - run unit 13
  - with recursive calls 13
  - with the THREAD option 13
- linkages, data 367
- linker
  - errors 156
  - errors in program names 157
  - files passed to 147
  - invoking 144
  - parameters 147
  - passing information to 145
  - resolving references to DLLs 390
- linker options
  - /? 204
  - /ALIGNADDR 204
  - /ALIGNFILE 204
  - /BASE 205
  - /CODE 205
  - /DATA 206
  - /DBGPACK 206
  - /DEBUG 206
  - /DEFAULTLIBRARYSEARCH 207
  - /DLL 207
  - /ENTRY 208
  - /EXECUTABLE 208
  - /EXTDICTIONARY 208
  - /FIXED 209
  - /FORCE 209
  - /HEAP 209
  - /HELP 210
  - /INCLUDE 210
  - /INFORMATION 210
  - /LINENUMBERS 210
  - /LOGO 211
  - /MAP 211
  - /OUT 212
  - /PMTYP 212
  - /SECTION 212
  - /SEGMENTS 213

- linker options (*continued*)
  - /STACK 214
  - /STUB 214
  - /SUBSYSTEM 214
  - /VERBOSE 215
  - /VERSION 215
- linker response file, echoing
  - contents 211
- linking
  - dynamic 390
  - programs 152
  - static 389
- linking a DLL 393
- LIST compiler option 179
  - conflict with OFFSET option 231
  - getting output 231
  - terms used in output 236
- listings
  - assembler expansion of procedure division 231
  - data- and procedure-name cross reference 230
  - embedded cross-reference 231
  - embedded MAP summary 231
  - including your source code 231
  - line numbers, user-supplied 232
  - mapping DATA DIVISION items 231
  - sorted cross reference of program names 239
  - terms used in MAP output 236
  - verb cross-reference 231
  - with error messages embedded 228
- little-endian 36
  - format for data representation 161
  - representation of integers 354
- load address 396
- load segment 396
- loading a table dynamically 56
- local names 363
- LOCAL-STORAGE section 12
- local time
  - getting (CEELOCT) 521
- locale 411
- locale information database
  - specifying search path name 141
- LOCPATH environment variable 141
- LOG intrinsic function 44
- loops
  - coding 74
  - conditional 76
  - do 76
  - in a table 76
  - performed a definite number of times 76
- LOWER-CASE intrinsic function 88
- lowercase 88
- LST file extension 150

## M

- main program
  - and subprograms 359
  - arguments to 386
  - specifying with cob2 146
- MAP compiler option 230
  - embedded MAP summary 231
  - example 235

- MAP compiler option (*continued*)
  - nested program map 231
  - nested program map, example 237
  - terms used in output 236
- MAP extension as linker parameter 147
- mapping of DATA DIVISION items 231
- mathematics
  - intrinsic functions 42
- MAX intrinsic function 91
  - example 43
- MAXVAL attribute 398
- MEAN intrinsic function 45
- MEDIAN intrinsic function 45
- memory-protection attributes of segments 212
- merge
  - concepts 115
  - description 115
  - files, describing 116
  - successful 122
- MERGE statement
  - description 116
- MERGE work files 142
- message catalogs
  - specifying path name 142
- messages
  - appearing in abbreviated form 142
  - compile-time error
    - choosing severity to be flagged 228
    - embedding in source listing 228
  - compiler-directed 150
  - determining what severity level to produce 174
  - run-time 535
  - setting national language 141
  - severity levels 149
  - when not produced 219
- messages, error
  - generating a list of 150
- messages, run-time 535
  - incomplete abbreviated 158
- metaclass definition 301
- method definition 277
- METHOD-ID paragraph 277
- methods 287
  - PROCEDURE DIVISION RETURNING 382
- millennium language extensions 426
  - assumed century window 436
  - compatible dates 433
  - compiler options affecting 159
  - date windowing 425
  - DATEPROC compiler option 167
  - nondates 437
  - objectives 427
  - principles 426
  - YEARWINDOW compiler option 197
- MIN intrinsic function 86
- MIXED suboption of PGMNAME 183
- MLE 426
- mnemonic-name
  - SPECIAL-NAMES paragraph 7
- module definition files
  - contents 390
  - creating 390
  - module statements 395

- module definition files (*continued*)
  - reserved words 395
  - rules 394
  - when to use 394
- module export files
  - creating 146
- module statements 395
- modules, exit
  - loading and invoking 171
- MOVE statement 25
- MQSeries Three Tier applications 403
- multiple-character currency signs 47
- multiple currency signs 48
- multiple thread environment, running in 191
- multitasking 404
- multithread environment
  - requirements 183
- multithreading
  - control transfer issues 406
  - example 408
  - limitations on COBOL 407
  - overview 403
  - preparing COBOL programs for 403
  - recursion 406
  - scope of language elements
    - program invocation instance
    - scoped elements 405
    - run-unit scoped elements 405
  - synchronizing access to resources 407
  - terminology 403
  - THREAD compiler option
    - restrictions under 191
    - when to choose 406

## N

- name declaration
  - searching for 363
- name decoration 157
- NAME statement 399
- naming
  - programs 5
- national data
  - joining 79
  - splitting 81
  - tallying and replacing 87
- national language setting 141
- National Language Support (NLS)
  - code pages 417
  - considerations 411
  - locale-sensitive collating 417
  - locale sensitivity 411
- NATIVE
  - portability considerations 354
- nested COPY statement 463
- nested delimited scope statements 18
- nested IF statement
  - CONTINUE statement 68
  - description 68
  - EVALUATE statement preferred 68
  - with null branches 68
- nested intrinsic functions 43
- nested program integration 459
- nested program map
  - about 231

- nested program map (*continued*)
  - example 237
- nested programs
  - calling 360
  - conventions for using 360
  - description 361
  - map 231
  - scope of names 363
  - transfer of control 360
- nesting level
  - program 234
  - statement 234
- NLSPATH environment variable 142
- NOCOMPILE compiler option
  - use of to find syntax errors 226
- NODESC suboption of CALLINT
  - compiler option 163
- NODESCRIPTOR suboption of CALLINT
  - compiler option 163
- NONAME attribute 397
- nondates
  - with MLE 437
- NOSSRANGE compiler option
  - affect on checking errors 217
- Notices 575
- null branch 68
- null-terminated strings 83
- NUMBER compiler option 232
  - syntax and description 181
- numeric arguments for the linker 153
- numeric class test 40
- numeric condition 71
- numeric data
  - binary
    - byte reversal of 36
    - USAGE IS BINARY 35
    - USAGE IS COMPUTATIONAL (COMP) 35
    - USAGE IS COMPUTATIONAL-4 (COMP-4) 35
    - USAGE IS COMPUTATIONAL-5 (COMP-5) 35
  - conversions between fixed- and floating-point data 39
  - editing symbols 32
  - external decimal
    - USAGE IS DISPLAY 34
  - external floating-point
    - USAGE IS DISPLAY 34
  - format conversions between fixed- and floating-point 38
  - internal floating-point
    - USAGE IS COMPUTATIONAL-1 (COMP-1) 36
    - USAGE IS COMPUTATIONAL-2 (COMP-2) 36
  - overview 31
  - packed-decimal
    - USAGE IS COMPUTATIONAL-3 (COMP-3) 36
    - USAGE IS PACKED-DECIMAL 36
  - PICTURE clause 31
  - storage formats 33
  - zoned decimal
    - USAGE IS DISPLAY 34
- numeric-edited data item 32

- numeric editing symbol 32
- numeric intrinsic functions
  - example of
    - ANNUITY 44
    - CURRENT-DATE 43
    - INTEGER 87
    - INTEGER-OF-DATE 43
    - LENGTH 43
    - LOG 44
    - MAX 43
    - MEAN 45
    - MEDIAN 45
    - MIN 86
    - NUMVAL 89
    - NUMVAL-C 43
    - ORD 90
    - ORD-MAX 65
    - PRESENT-VALUE 44
    - RANGE 45
    - REM 44
    - SQRT 44
    - SUM 65
  - nested 43
  - special registers as arguments 43
  - table elements as arguments 43
  - types of—integer, floating-point, mixed 42
  - uses for 42
- NUMVAL-C intrinsic function 89
  - example 43
- NUMVAL intrinsic function 89

## O

- OBJ extension as linker parameter 147
- object code
  - controlling 159
  - generating 166
- OBJECT-COMPUTER paragraph 7
- object deck generation 159
- object module
  - adding external program reference 390
- object-oriented COBOL
  - generating IDL definitions 177
  - restrictions for DYNAM compiler option 168
- object references 286
- objectives of millennium language extensions 427
- OCCURS clause 455
- OCCURS DEPENDING ON (ODO) clause
  - complex 491
  - initializing ODO elements 62
  - optimization 455
  - simple 60
  - variable-length tables 60
- ODBC 255
  - accessing return values 259
  - advantages 255
  - background 256
  - CALL interface convention 267
  - driver manager 256
  - embedded SQL 255
  - error messages 267
- ODBC (*continued*)
  - installing and configuring the drivers 256
  - mapping of C data types 256
  - passing a COBOL pointer to 257
  - supplied copybooks 260
    - ODBC3D.CPY example 265
    - ODBC3P.CPY example 262
    - sample 261
  - using APIs from COBOL 256
- OMITTED parameters 465
- OMMAlloc 342
- OMMFree 342
- ON SIZE ERROR
  - with windowed date fields 441
- OO COBOL
  - generating IDL definitions 177
- OPEN operation code 171
- OPEN statement
  - file availability 107
  - file status key 129
- opening files 107
  - using environment variables 140
- optimization
  - avoid ALTER statement 452
  - avoid backward branches 452
  - consistent data 454
  - constant computations 452
  - constant data items 452
  - contained program integration 459
  - duplicate computations 453
  - effect of compiler options on 459
  - effect on performance 451
  - factor expressions 452
  - index computations 456
  - indexing 455
  - nested program integration 459
  - OCCURS DEPENDING ON 455
  - out-of-line PERFORM 452
  - PACKED-DECIMAL data items 454
  - performance implications 455
  - structured programming 451
  - subscript computations 456
  - subscripting 455
  - table elements 455
  - top-down programming 452
  - unused data items 181
- OPTIMIZE compiler option 181
  - description 458
  - effect on performance 458
  - performance considerations 460
- optimizer 458
- OPTLINK suboption of CALLINT
  - compiler option 162
- ORD intrinsic function 90
- ORD-MAX intrinsic function 91
- ORD-MIN intrinsic function 91
- order of evaluation
  - arithmetic operators 482
- ordinal position of data construct 397
- ordinal position of function 397
- out-of-line PERFORM 75
- outermost programs 389
- output
  - overview 100

- output procedure
  - requires RETURN or RETURN INTO statement 119
  - restrictions 119
  - using 119
- overflow condition 125
- overriding linker options, example 155

## P

- PACKED-DECIMAL
  - general description 36
  - synonym 33
  - using efficiently 36
- packed-decimal data item
  - date fields, potential problems 446
  - general description 36
  - using efficiently 36
- page header 234
- paragraph
  - grouping 77
  - introduction 15
- parameter
  - describing in called program 375
  - in main program 386
- parameter list
  - address of with INEXIT 171
  - for ADEXIT 173
  - for PRTEXT 173
- PASCAL16 suboption of CALLINT
  - compiler option 162
- passing addresses between programs 377
- passing data between programs
  - BY CONTENT 373
  - BY REFERENCE 373
  - BY VALUE 373
  - called program 374
  - calling program 374
  - EXTERNAL data 383
  - language used 374
- path name
  - for COPY files search 146
  - library text 139
  - multiple, specifying 139
  - search order precedence 138
  - specifying for catalogs and help files 142
  - specifying for locale information database 141
  - specifying with LIB compiler option 178
- PERFORM statement
  - ...THRU 77
  - coding loops 74
  - for a table 58
  - indexing 55
  - inline 75
  - out-of-line 75
  - performed a definite number of times 76
  - TEST AFTER 76
  - TEST BEFORE 76
  - TIMES 76
  - UNTIL 76
  - VARYING 76
  - VARYING WITH TEST AFTER 76

- PERFORM statement (*continued*)
  - WITH TEST AFTER ... UNTIL 76
  - WITH TEST BEFORE ... UNTIL 76
- performance
  - coding 451
  - coding tables 455
  - data usage 453
  - DYNAM compiler option 460
  - effect of compiler options on 459
  - in a CICS environment 451
  - OCCURS DEPENDING ON 455
  - OPTIMIZE compiler option 460
  - optimizer 458
  - planning arithmetic evaluations 453
  - programming style 451
  - run-time considerations 451
  - SSRANGE compiler option 460
  - table handling 456
  - TEST compiler option 460
  - TRUNC compiler option 192
  - TRUNC(STD|OPT|BIN) compiler option 460
  - use of arithmetic expressions 454
  - using the TEMPMEM environment variable 139
  - variable subscript data format 54
  - worksheet 461
- Performance Analyzer support, PROFILE
  - option for 184
- performance considerations
  - creating a trace file 147
- performing calculations
  - date and time services and 466
- period, as scope terminator 17
- PGMNAME compiler option 182
- PICTURE clause
  - determining symbol used 166
  - numeric data 31
- picture strings
  - date and time services and 468
- platform differences 355
- pointer data item
  - incrementing addresses with 377
  - NULL value 377
  - used to pass addresses 377
  - used to process chained list 377
- portability 351
  - environment differences 355
  - run-time differences between mainframe and the workstation 356
- porting applications
  - architectural differences between platforms 351
  - language differences between PC and mainframe 351
  - mainframe to PC
    - choosing compiler options 351
  - mainframe to workstation
    - running mainframe applications on the workstation 353
  - multitasking 357
  - PC to AIX 357
  - PC to mainframe
    - PC-only language features 356
  - using COPY to isolate platform-specific code 352

- porting applications (*continued*)
  - workstation to mainframe
    - workstation-only compiler options 356
    - workstation-only names 357
- porting your program 32
- potential problems with date fields 446
- precedence
  - arithmetic operators 482
- preinitializing the COBOL
  - environment 419
- PRESENT-VALUE intrinsic function 44
- printer files 101
- PROBE compiler option 183
- procedure and data name cross-reference, description 230
- PROCEDURE DIVISION
  - description 14
  - in subprograms 375
  - method 279
  - RETURNING 15
  - statements
    - compiler-directing 17
    - conditional 16
    - delimited scope 16
    - imperative 16
    - terminology 14
    - USING 15
- PROCEDURE DIVISION RETURNING
  - methods, use of 382
- procedure-pointer data item
  - entry address for entry point 380
- passing parameters to callable services 380
- rules for using 380
- SET statement and 380
- SYSTEM interface convention 380
- Windows restriction 380
- PROCESS statement 148
- processes 403
- processing
  - chained list 377
  - tables 58
  - using indexing 59
  - using subscripting 58
- PROFILE compiler option 184
- profiling support, PROFILE option 184
- program
  - attribute codes 237
  - decisions
    - EVALUATE statement 68
    - IF statement 67
    - loops 76
    - PERFORM statement 76
    - switches and flags 72
  - diagnostics 234
  - limitations 451
  - main 359
  - nesting level 234
  - source code samples
    - definition file 392
    - dynamic link library 390
  - statistics 234
  - structure 5
  - sub 359
- PROGRAM COLLATING SEQUENCE
  - clause 8

- program entry points, call convention 168
- Program-ID paragraph
  - COMMON attribute 6
  - description 5
  - INITIAL attribute 6
- program-name cross-reference 238
- program names, handling of case 183
- programs, running 158
- PRTEXIT suboption of EXIT option 173

## Q

- QSAM files
  - input/output error processing 127
- QUOTE compiler option 184

## R

- RANGE intrinsic function 45
- reading records from files
  - dynamically 109
  - randomly 109
  - sequentially 109
- receiving field 81
- record
  - description 11
  - format 100
- records, affected by TRAP run-time option 219
- recursive calls 6
  - and the LINKAGE SECTION 13
- reentrant code 475
- reference modification
  - example 85
  - out-of-range values 85
  - tables 85
- reference modifier
  - arithmetic expression as 86
  - intrinsic function as 86
  - variables as 85
- relate items to system-names 7
- relation condition 71
- relative file organization 102
- Relative files
  - file access mode 102
- RELEASE FROM statement
  - compared to RELEASE 118
  - example 118
- RELEASE statement
  - compared to RELEASE FROM 118
  - with SORT 118
- REM intrinsic function 44
- REPLACE statement 198
- replacing
  - data items 87
  - records in file 111
- REPOSITORY paragraph 275
- representation
  - data 40
  - sign 40
- RESIDENTNAME attribute 397
- resolving references to DLLs 390
- restrictions
  - input/output procedures 119
  - subscripting 54

- return code
  - compiler 149
  - feedback code from date and time services 465
  - from DB2 246
  - RETURN-CODE special register 465
  - VSAM files 131
- RETURN-CODE special register
  - considerations for DB2 246
  - passing data between programs 382
  - value after call to date and time service 465
- return codes, linker 157
- RETURN INTO statement 119
- RETURN statement 119
- RETURNING phrase
  - methods, use of 382
- REVERSE intrinsic function 89
- reversing characters 89
- rows in tables 52
- run time
  - changing file-name 10
  - differences between platforms 353
  - performance considerations 451
- run-time
  - arguments 386
  - messages 535
- run-time environment,
  - preinitializing 419
- run-time environment variables
  - LC\_COLLATE 141
  - LC\_MESSAGES 141
- run-time error messages
  - setting national language 141
- run-time messages 535
  - appearing in abbreviated form 142
  - incomplete or abbreviated 158
- run-time options
  - CHECK 217
  - CHECK(OFF) 460
  - DEBUG 224
  - ERRCOUNT 218
  - FILESYS 218
  - selecting for CICS 254
  - specifying 141
  - TRAP 219
  - ON SIZE ERROR 126
  - UPSI 219
- run unit 359
  - role in multithreading 403
- running programs 158
- running under DOS 401

## S

- S-level error message 149
- scope of names 363
- scope terminator
  - aids in debugging 222
  - explicit 16
  - implicit 17
- SEARCH ALL statement
  - binary search 64
  - indexing 55
  - ordered table 64
- search rules for linker 155
  - example 156

- SEARCH statement
  - examples 63
  - indexing 55
  - nesting 63
  - serial search 63
- searching a table 62
- searching for name declarations 363
- section
  - declarative 19
  - description of 15
  - grouping 77
- SELECT clause
  - vary input-output file 10
- SELECT OPTIONAL 107
- sending field 81
- sentence 15
- separate digit sign 32
- SEPOBJ compiler option 185
- SEQUENCE compiler option 227
- sequential file organization 101
- sequential files
  - file access mode 101
- serial search 63
- SERVICE LABEL statement 198
- SET command, defining environment variables 137
- SET condition-name TO TRUE statement
  - description 73
  - example 75
- SET statement
  - for procedure-pointer data items 380
  - handling of programs name in 183
  - using for debugging 223
- SET statement, path search order 138
- setting
  - linker options 152
  - switches and flags 73
- sharing
  - data 363
  - files 363
- short listing, example 232
- sign condition 71
- sign representation 40
- SIZE compiler option 187
- SKIP1/2/3 statement 198
- sliding century window 428
- SMARTdata Utilities 96
- SOM 317
  - CORBA-style exceptions 334
  - environment arguments 334
  - errors and exceptions 334
  - COBOL example 335
  - initializers 337
  - memory management with 339
  - SOMerror-style exceptions 334
- sort
  - alternate collating sequence 121
  - concepts 115
  - criteria 120
  - description 115
  - files, describing 116
  - more than one 115
  - restrictions on input/output procedures 119
  - successful 122
  - terminating 123
  - using input procedures 118

- sort (*continued*)
  - using output procedures 119
- Sort File Description (SD) entry
  - example 117
- SORT-RETURN special register 123
- SORT statement
  - description 120
- SORT work files 142
- SOSI compiler option 187
- SOURCE and NUMBER output,
  - example 234
- source code
  - line number 235
  - listing, description 231
- SOURCE compiler option 231
- SOURCE-COMPUTER paragraph 7
- SPACE compiler option 189
- special feature specification 7
- SPECIAL-NAMES paragraph 7
- special register
  - ADDRESS 374
  - arguments in intrinsic functions 43
  - LENGTH OF 374
  - SORT-RETURN 122
  - WHEN-COMPILED 93
- splitting data items 81
- SQL compiler option 247
- SQL INCLUDE statement 246
- SQL statements 245
- SQLCA 245
- SQLCODE 246
- SQLSTATE 246
- SQRT intrinsic function 44
- SSRANGE compiler option 189
  - CHECK(OFF) run-time option 460
  - description 227
  - performance considerations 460
- stack probes, generating 183
- STACKSIZE statement 400
- statement
  - compiler-directing 17
  - conditional 16
  - definition 16
  - delimited scope 16
  - explicit scope terminator 17
  - imperative 16
  - implicit scope terminator 17
- statement nesting level 234
- static linking 364
  - advantages 389
  - disadvantages 389
  - overview 389
- statistics
  - intrinsic functions 45
- STL file system 97
- STOP RUN statement
  - in main program 360
  - in subprogram 360
- storage
  - mapping 231
  - stack 183
- storing frequently used functions 389
- STRING statement
  - description 79
  - example of 79
  - overflow condition 125

- strings
  - null-terminated 376
- structured programming 452
- STUB statement 401
- subclass definition 289
- subprogram
  - and main program 359
  - definition of 359
  - linkage 359
  - common data items 375
  - Procedure Division in 375
- subprograms
  - in DLLs 389
- subscript
  - computations 456
  - range checking 227
- subscripting
  - example of processing a table 58
  - index-names 55
  - literal 52
  - reference modification 54
  - relative 54
  - restrictions 54
  - variable 53
- substrings
  - reference modification 84
  - referencing table items 85
- SUM intrinsic function 65
- switch-status condition 71
- switches 72
- switches and flags
  - about 72
  - defining 72
  - resetting 73
- SYMBOLIC CHARACTER clause 9
- symbolic constant 452
- syntax errors
  - finding with NOCOMPILE compiler option 226
- SYSADATA file 161
- SYSADATA records
  - exit module called 173
  - supplying modules 169
- SYSIN
  - supplying alternative modules 169
- SYSIN, SYSIPT, SYSOUT, SYSLIST, SYSLST, CONSOLE, SYSPUNCH, SYSPCH environment variables 142
- SYSLIB
  - supplying alternative modules 169
  - when not used 172
- SYSLIB environment variable 139
- SYSPRINT
  - supplying alternative modules 169
  - when not used 173
- System/390 host data type
  - considerations 479
- SYSTEM convention restriction (Windows) 381
- system date
  - under CICS 251
- System dialog, defining environment variables 137
- system-name 7
- System Object Model 317
  - environment variables 312
  - Interface Repository (IR) 311

- System Object Model (*continued*)
  - methods and functions 313
  - services 313
  - somFree 286
  - somNew 286
- SYSTEM suboption of CALLINT compiler option 162
- SYSTEM data set
  - sending messages to 190

## T

- table
  - assigning values 57
  - columns 51
  - defining 51
  - depth 52
  - dynamically loading 56
  - efficient coding 456
  - handling 51
  - identical element specifications 455
  - index 54
  - initialize 56
  - intrinsic functions 65
  - loading values in 56
  - looping through 76
  - making reference 53
  - one-dimensional 51
  - reference modification 54
  - referencing table entry substrings 85
  - rows 52
  - searching 62
  - subscripts 52
  - three-dimensional 52
  - two-dimensional 52
  - variable-length 60
- TALLYING option 87
- TEMP environment variable 142
- TEMPMEM environment variable 139
- temporary work file location
  - specifying with TEMP 142
- TERMGLOBAL termination of DLL 399
- terminal, sending messages to 190
- TERMINAL compiler option 190
- TERMINSTANCE termination of DLL 399
- terms used in MAP output 236
- test
  - conditions 76
  - data 71
  - numeric operand 71
  - UPSI switch 71
- TEST AFTER 76
- TEST BEFORE 76
- TEST compiler option 190
  - for full advantage of Debug Tool 231
  - performance considerations 460
- THREAD compiler option 191
  - and the LINKAGE SECTION 13
- thread environment requirements 183
- three-digit years 468
- time, getting local (CEELOCT) 521
- time information, formatting 141
- time zone information
  - specifying with TZ 142
- timestamp 505
- TITLE statement 198

TITLE statement (*continued*)  
     controlling header on listing 6  
 top-down programming  
     constructs to avoid 452  
 transferring control  
     between COBOL programs 360  
     called program 359  
     calling program 359  
     main and subprograms 359  
     nested programs 361  
 translating CICS into COBOL 249  
 TRAP run-time option 219  
     ON SIZE ERROR 126  
 TRUNC compiler option 192  
 TRUNC(STD|OPT|BIN) compiler  
     option 460  
 tuning considerations, performance 460  
 two-digit years  
     querying within 100-year range  
       (CEEQCEN) 523  
     example 524  
     setting within 100-year range  
       (CEESCEEN) 525  
     example 525  
     valid values for 468  
 TYPECHK compiler option 194  
 TZ environment variable 142

## U

U-level error message 149  
 ull-terminated strings 376  
 UNDATE intrinsic function 443  
 UNSTRING statement  
     description 81  
     example 81  
     overflow condition 125  
 UPPER-CASE intrinsic function 88  
 UPPER suboption of PGMNAME 183  
 uppercase 88  
 UPSI run-time options 219  
 UPSI switches, setting 219  
 USAGE clause  
     incompatible data 40  
     IS INDEX 55  
 USE FOR DEBUGGING declarative 218  
 USE FOR DEBUGGING declaratives 223  
 USE statement 198  
 user-defined condition 71  
 user-exit work area 170

## V

valid data  
     numeric 40  
 VALUE clause  
     assigning table values 57  
     Data Description entry 57  
 VALUE IS NULL 377  
 variable  
     as reference modifier 85  
     COBOL term for 21  
 variable-length records  
     OCCURS DEPENDING ON (ODO)  
     clause 455  
 variable-length table 60

variables, environment  
     ASSIGNment name 140  
     COBCPYEXT 138  
     COBLSTDIR 138  
     COBMSG 140  
     COBOPT 139  
     COBPATH 139  
     COBRTOPT 140  
     EBCDIC\_CODEPAGE 141  
     LANG 141  
     LC\_TIME 141  
     library-name 139  
     LOCPATH 141  
     NLSPATH 142  
     run time 140  
     setting 137  
     SYSIN, SYSIPT, SYSOUT, SYSLIST,  
       SYSLST, CONSOLE, SYSPUNCH,  
       SYSPCH 142  
     SYSLIB 139  
     TEMP 142  
     TEMPMEM 139  
     text-name 139  
     TZ 142  
     used by the compiler 138  
 variably located data item 492  
 variably located group 492  
 VBREF compiler option 231  
 VBREF compiler output, example 240  
 verb cross-reference listing  
     description 231  
 verbs used in program 231  
 VERSION statement 401  
 VisualAge COBOL  
     run-time messages 535  
 VSAM  
     accessing remote files 96  
     files  
       processing 97  
       return codes 131  
 VSAM files  
     error processing 127  
     in a multithreading environment 407

## W

W-level error message 149  
 WHEN-COMPILED intrinsic function  
     example 93  
     versus WHEN-COMPILED special  
       register 93  
 WHEN-COMPILED special register 93  
 WHEN phrase  
     EVALUATE statement 69  
     SEARCH statement 63  
 WITH DEBUGGING MODE clause 218  
 wlist file 179  
 WORKING-STORAGE  
     initializing 195  
 WORKING-STORAGE SECTION  
     class 275  
     description 12  
     method 278  
 workstation and workstation COBOL  
     differences from host 475  
 writing compatible code 356  
 WSCLEAR compiler option 195

## X

XREF compiler option 230  
 XREF output  
     data-name cross-references 237  
     program-name cross-references 238

## Y

year field expansion 430  
 year-last date fields 432  
 year windowing  
     advantages 428  
     how to control 443  
     MLE approach 428  
     when not supported 434  
 YEARWINDOW compiler option 197

## Z

zero comparison 439  
 zoned decimal 34  
 ZWB compiler option 197

---

## Readers' Comments — We'd Like to Hear from You

VisualAge COBOL  
Programming Guide  
Version 3.0.2

Publication No. SC27-0812-01

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Complete	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Easy to find	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Easy to understand	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Well organized	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Applicable to your tasks	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

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.

---

Name

---

Address

---

Company or Organization

---

Phone No.



Cut or Fold  
Along Line

Fold and Tape

Please do not staple

Fold and Tape



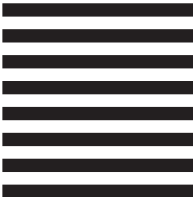
NO POSTAGE  
NECESSARY  
IF MAILED IN THE  
UNITED STATES

**BUSINESS REPLY MAIL**

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation  
Department HHX/H3  
P.O. Box 49023  
San Jose, CA  
United States of America  
95161-9023



Fold and Tape

Please do not staple

Fold and Tape

Cut or Fold  
Along Line





Printed in the United States of America  
on recycled paper containing 10%  
recovered post-consumer fiber.

SC27-0812-01

