

COBOL for OS/390 & VM



# Programming Guide

*Version 2 Release 2*



COBOL for OS/390 & VM



# Programming Guide

*Version 2 Release 2*

**Note!**

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 609.

**Sixth Edition (September 2000)**

This edition applies to Version 2 Release 2 of IBM COBOL for OS/390 & VM (program number 5648-A25) 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.ibm.com/software/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 1991, 2000. All rights reserved.

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



---

# Contents

<b>About this book</b> . . . . .	<b>xi</b>
How this book will help you. . . . .	xi
Abbreviated terms . . . . .	xi
How to read syntax diagrams . . . . .	xii
How examples are shown . . . . .	xiii
Comparison of commonly-used terms . . . . .	xiii
Summary of changes . . . . .	xiv
Major changes . . . . .	xiv

---

## 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 entries . . . . .	7
Specifying the collating sequence . . . . .	8
Defining symbolic characters . . . . .	9
Defining a user-defined class . . . . .	9
Defining files to the operating system . . . . .	10
Describing the data. . . . .	12
Using data in input and output operations . . . . .	13
Comparison of WORKING-STORAGE and LOCAL-STORAGE . . . . .	15
Using data from another program . . . . .	16
Processing the data. . . . .	17
How logic is divided in the PROCEDURE DIVISION . . . . .	17
Declaratives . . . . .	21

### Chapter 2. Using data . . . . . 23

Using variables, structures, literals, and constants . . . . .	23
Variables . . . . .	23
Data structure: data items and group items. . . . .	23
Literals . . . . .	24
Constants . . . . .	24
Figurative constants . . . . .	24
Assigning values to data items . . . . .	25
Examples: initializing variables . . . . .	25
Initializing a structure (INITIALIZE) . . . . .	26
Assigning values to variables or structures (MOVE) . . . . .	27
Assigning arithmetic results (MOVE or COMPUTE) . . . . .	27
Assigning input from a screen or file (ACCEPT) . . . . .	28
Displaying values on a screen or in a file (DISPLAY) . . . . .	29
Displaying data on the system logical output device . . . . .	29
Using WITH NO ADVANCING . . . . .	30
Using intrinsic functions (built-in functions) . . . . .	31

Types of intrinsic functions . . . . .	31
Nesting functions . . . . .	32
Using tables (arrays) and pointers . . . . .	32

### Chapter 3. Working with numbers and arithmetic . . . . . 33

Defining numeric data. . . . .	33
Displaying numeric data . . . . .	34
Controlling how numeric data is stored . . . . .	35
Formats for numeric data. . . . .	36
External decimal (DISPLAY) items. . . . .	36
External floating-point (DISPLAY) items. . . . .	36
Binary (COMP) items . . . . .	37
Native binary (COMP-5) items . . . . .	37
Packed-decimal (COMP-3) items . . . . .	38
Floating-point (COMP-1 and COMP-2) items . . . . .	38
Examples: numeric data and internal representation . . . . .	38
Data format conversions . . . . .	39
Conversions and precision . . . . .	40
Sign representation and processing . . . . .	41
NUMPROC(PFD) . . . . .	41
NUMPROC(NOPFD) . . . . .	41
NUMPROC(MIG) . . . . .	41
Checking for incompatible data (numeric class test) . . . . .	42
Performing arithmetic . . . . .	43
COMPUTE and other arithmetic statements . . . . .	43
Arithmetic expressions . . . . .	43
Numeric intrinsic functions . . . . .	44
Nesting functions and arithmetic expressions . . . . .	45
ALL subscripting and special registers . . . . .	45
Math and date Language Environment services . . . . .	45
Examples: numeric intrinsic functions . . . . .	47
General number handling . . . . .	47
Date and time . . . . .	47
Finance. . . . .	47
Mathematics . . . . .	48
Statistics . . . . .	48
Fixed-point versus floating-point arithmetic . . . . .	49
Floating-point evaluations . . . . .	49
Fixed-point evaluations . . . . .	49
Arithmetic comparisons (relation conditions) . . . . .	50
Examples: fixed-point and floating-point evaluations . . . . .	50
Using currency signs . . . . .	51
Example: multiple currency signs . . . . .	52

### Chapter 4. Handling tables . . . . . 53

Defining a table (OCCURS) . . . . .	53
Nesting tables . . . . .	54
Subscripting . . . . .	54
Indexing . . . . .	55
Referring to an item in a table . . . . .	55
Subscripting . . . . .	56
Indexing . . . . .	57
Putting values into a table . . . . .	58

Loading a table dynamically. . . . .	58
Initializing a table (INITIALIZE) . . . . .	58
Assigning values when you define a table (VALUE) . . . . .	58
Example: PERFORM and subscripting . . . . .	60
Example: PERFORM and indexing. . . . .	61
Creating variable-length tables (DEPENDING ON) . . . . .	62
Loading a variable-length table. . . . .	63
Assigning values to a variable-length table . . . . .	64
Searching a table . . . . .	65
Doing a serial search (SEARCH) . . . . .	65
Doing a binary search (SEARCH ALL) . . . . .	66
Processing table items using intrinsic functions . . . . .	67
Example: intrinsic functions . . . . .	68

## Chapter 5. Selecting and repeating program actions . . . . . 69

Selecting program actions . . . . .	69
Coding a choice of actions . . . . .	69
Coding conditional expressions. . . . .	73
Repeating program actions . . . . .	76
Choosing inline or out-of-line PERFORM . . . . .	76
Coding a loop . . . . .	77
Coding a loop through a table . . . . .	78
Executing multiple paragraphs or sections . . . . .	79

## Chapter 6. Handling strings. . . . . 81

Joining data items (STRING) . . . . .	81
Example: STRING statement. . . . .	81
Splitting data items (UNSTRING) . . . . .	83
Example: UNSTRING statement . . . . .	83
Manipulating null-terminated strings. . . . .	85
Example: null-terminated strings . . . . .	86
Referring to substrings of data items . . . . .	86
Reference modifiers. . . . .	87
Example: arithmetic expressions as reference modifiers . . . . .	88
Example: intrinsic functions as reference modifiers . . . . .	88
Tallying and replacing data items (INSPECT) . . . . .	89
Examples: INSPECT statement . . . . .	89
Converting double-byte character set (DBCS) data . . . . .	90
DBCS notation . . . . .	90
Nonnumeric to DBCS data conversion (IGZCA2D) . . . . .	90
DBCS to nonnumeric data conversion (IGZCD2A) . . . . .	92
Converting data items (intrinsic functions) . . . . .	94
Converting to uppercase or lowercase (UPPER-CASE, LOWER-CASE) . . . . .	94
Converting to reverse order (REVERSE) . . . . .	95
Converting to numbers (NUMVAL, NUMVAL-C) . . . . .	95
Evaluating data items (intrinsic functions) . . . . .	96
Evaluating single characters for collating sequence . . . . .	96
Finding the largest or smallest data item . . . . .	97
Finding the length of data items . . . . .	98
Finding the date of compilation . . . . .	99

## Chapter 7. Processing files . . . . . 101

File organization and input-output devices . . . . .	101
Choosing file organization and access mode . . . . .	103
Format for coding input and output. . . . .	103
Allocating files . . . . .	105
Checking for input or output errors . . . . .	106

## Chapter 8. Processing QSAM files . . . 107

Defining QSAM files and records in COBOL . . . . .	107
Establishing record formats. . . . .	108
Setting block sizes . . . . .	115
Coding input and output statements for QSAM files . . . . .	117
Opening QSAM files . . . . .	117
Dynamically creating QSAM files with CBLQDA . . . . .	118
Adding records to QSAM files. . . . .	119
Updating QSAM files. . . . .	119
Writing QSAM files to a printer or spooled data set . . . . .	119
Closing QSAM files . . . . .	120
Handling errors in QSAM files . . . . .	120
Working with QSAM files under OS/390 . . . . .	121
Defining and allocating QSAM files . . . . .	121
Retrieving QSAM files . . . . .	123
Ensuring file attributes match your program . . . . .	124
Using striped extended-format QSAM data sets . . . . .	126
Accessing HFS files using QSAM. . . . .	127
Restrictions and usage . . . . .	127
Identifying QSAM files to CMS . . . . .	128
Using FILEDEF. . . . .	128
Using LABELDEF . . . . .	129
Labels for QSAM files . . . . .	129
Using trailer and header labels . . . . .	130
Format of standard labels . . . . .	131
Processing QSAM ASCII files on tape . . . . .	132
Requesting the ASCII alphabet . . . . .	132
Defining the record formats . . . . .	132
Defining the ddname. . . . .	132
Processing ASCII file labels. . . . .	133

## Chapter 9. Processing VSAM files . . . 135

VSAM files . . . . .	136
Defining VSAM file organization and records . . . . .	137
Specifying sequential organization for VSAM files . . . . .	138
Specifying indexed organization for VSAM files . . . . .	138
Specifying relative organization for VSAM files . . . . .	139
Specifying access modes for VSAM files . . . . .	141
Defining record lengths for VSAM files. . . . .	141
Coding input and output statements for VSAM files . . . . .	143
File position indicator . . . . .	144
Opening a file (ESDS, KSDS, or RRDS) . . . . .	145
Reading records from a VSAM file . . . . .	147
Updating records in a VSAM file. . . . .	148
Adding records to a VSAM file . . . . .	149
Replacing records in a VSAM file. . . . .	149
Deleting records from a VSAM file . . . . .	150
Closing VSAM files . . . . .	150
Handling errors in VSAM files . . . . .	151
Protecting VSAM files with a password . . . . .	151

Example: password protection for a VSAM indexed file . . . . .	152
Working with VSAM data sets under OS/390 and OS/390 UNIX . . . . .	152
Defining VSAM files under OS/390 . . . . .	153
Creating alternate indexes . . . . .	154
Allocating VSAM files . . . . .	156
Sharing VSAM files through RLS . . . . .	157
Defining VSAM data sets under CMS . . . . .	158
Improving VSAM performance . . . . .	159

**Chapter 10. Processing line-sequential files . . . . . 163**

Defining line-sequential files and records in COBOL . . . . .	163
Allowable control characters . . . . .	164
Describing the structure of a line-sequential file . . . . .	164
Defining and allocating line-sequential files . . . . .	165
Coding input-output statements for line-sequential files . . . . .	165
Opening line-sequential files . . . . .	166
Reading records from line-sequential files . . . . .	166
Adding records to line-sequential files . . . . .	167
Closing line-sequential files . . . . .	167
Handling errors in line-sequential files . . . . .	168

**Chapter 11. Sorting and merging files 169**

Sort and merge process . . . . .	170
Describing the sort or merge file . . . . .	170
Describing the input to sorting or merging . . . . .	171
Example: describing sort and input files for SORT . . . . .	171
Coding the input procedure . . . . .	172
Describing the output from sorting or merging . . . . .	173
Coding the output procedure . . . . .	173
Coding considerations when using DFSORT on OS/390 . . . . .	174
Example: coding the output procedure when using DFSORT . . . . .	174
Restrictions on input and output procedures . . . . .	175
Defining sort and merge data sets under OS/390 . . . . .	175
Defining sort and merge files under CMS . . . . .	176
Sorting variable-length records . . . . .	176
Requesting the sort or merge . . . . .	177
Setting sort or merge criteria . . . . .	177
Example: sorting with input and output procedures . . . . .	178
Choosing alternate collating sequences . . . . .	179
Sorting on windowed date fields . . . . .	179
Preserving the original sequence of records with equal keys . . . . .	180
Determining whether the sort or merge was successful . . . . .	180
Stopping a sort or merge operation prematurely . . . . .	181
Improving sort performance with FASTSRT . . . . .	181
FASTSRT requirements for JCL (OS/390 only) . . . . .	181
FASTSRT requirements for sort input and output files . . . . .	181
Checking for sort errors with NOFASTSRT . . . . .	183
Controlling sort behavior . . . . .	184

Sort special registers . . . . .	184
Changing DFSORT defaults with control statements . . . . .	185
Allocating storage for sort or merge operations . . . . .	186
Using checkpoint/restart with DFSORT under OS/390 . . . . .	186
Sorting under CICS . . . . .	187
CICS SORT application restrictions . . . . .	187

**Chapter 12. Handling errors . . . . . 189**

Requesting dumps . . . . .	189
Creating a formatted dump . . . . .	189
Creating a system dump . . . . .	190
Handling errors in joining and splitting strings . . . . .	190
Handling errors in arithmetic operations . . . . .	191
Example: checking for division by zero . . . . .	191
Handling errors in input and output operations . . . . .	191
Using the end-of-file condition (AT END) . . . . .	193
Coding ERROR declaratives . . . . .	194
Using file status keys . . . . .	194
Example: file status key . . . . .	195
Using VSAM return codes (VSAM files only) . . . . .	196
Example: checking VSAM status codes . . . . .	196
Coding INVALID KEY phrases . . . . .	198
Example: FILE STATUS and INVALID KEY . . . . .	198
Handling errors when calling programs . . . . .	199
Writing routines for handling errors . . . . .	199

**Part 2. Compiling and debugging your program . . . . . 201**

**Chapter 13. Compiling under OS/390 203**

Compiling with JCL . . . . .	203
Using a cataloged procedure . . . . .	204
Writing JCL to compile programs . . . . .	213
Compiling under TSO . . . . .	215
Example: ALLOCATE and CALL for compiling under TSO . . . . .	215
Example: CLIST for compiling under TSO . . . . .	216
Starting the compiler from an assembler program . . . . .	216
Defining compiler input and output . . . . .	218
Data sets used by the compiler under OS/390 . . . . .	218
Defining the source code data set (SYSIN) . . . . .	220
Specifying source libraries (SYSLIB) . . . . .	220
Defining the output data set (SYSPRINT) . . . . .	221
Directing compiler messages to your terminal (SYSTEM) . . . . .	221
Creating object code (SYSLIN or SYSPUNCH) . . . . .	221
Creating an associated data file (SYSADATA) . . . . .	222
Defining the output IDL data set (SYSIDL) . . . . .	222
Defining the debug data set (SYSDEBUG) . . . . .	222
Specifying compiler options under OS/390 . . . . .	223
Specifying compiler options with the PROCESS (CBL) statement . . . . .	223
Example: specifying compiler options using JCL . . . . .	224
Example: specifying compiler options under TSO . . . . .	224
Compiler options and compiler output under OS/390 . . . . .	224
Compiling multiple programs (batch compilation) . . . . .	226

Example: batch compilation . . . . .	226
Specifying compiler options in a batch compilation . . . . .	227
Example: precedence of options in a batch compilation . . . . .	228
Example: LANGUAGE option in a batch compilation . . . . .	229
Correcting errors in your source program . . . . .	230
Generating a list of compiler error messages . . . . .	230
Messages and listings for compiler-detected errors . . . . .	231
Format of compiler error messages . . . . .	231
Severity codes for compiler error messages . . . . .	232

**Chapter 14. Compiling under OS/390 UNIX . . . . . 235**

Setting environment variables under OS/390 UNIX	235
Specifying compiler options under OS/390 UNIX	236
Compiling and linking with the cob2 command	237
Defining input and output . . . . .	237
Creating a DLL. . . . .	238
Example: using cob2 to compile under OS/390 UNIX . . . . .	238
cob2 . . . . .	239
cob2 input and output files. . . . .	240
Compiling using scripts . . . . .	241

**Chapter 15. Compiling under CMS 243**

Accessing the compiler (CP LINK and ACCESS)	243
Specifying a source program to compile . . . . .	244
COBOL2 . . . . .	244
Using compiler-directing statements . . . . .	245
Specifying compiler options under CMS . . . . .	246
Using parentheses in COBOL2 . . . . .	246
Abbreviating compiler options . . . . .	246
Using logical line-editing characters . . . . .	247
Additional compiler options under CMS . . . . .	247
Differences in compiler options under CMS . . . . .	248
Precedence of compiler options under CMS . . . . .	248
Files used by the compiler under CMS . . . . .	248
Naming generated files under CMS . . . . .	251
Overriding FILEDEF . . . . .	251
Using system-generated names . . . . .	251
Correcting errors . . . . .	252
Error messages from COBOL2. . . . .	252

**Chapter 16. Compiler options . . . . . 257**

Option settings for COBOL 85 Standard conformance. . . . .	259
Conflicting compiler options . . . . .	259
ADATA . . . . .	261
ANALYZE . . . . .	261
ADV . . . . .	262
ARITH . . . . .	262
AWO . . . . .	263
BUFSIZE . . . . .	263
CMPR2 . . . . .	264
COMPILE . . . . .	265
CURRENCY. . . . .	265
DATA . . . . .	266

External data . . . . .	267
QSAM input/output buffers . . . . .	267
DATEPROC . . . . .	267
DBCS . . . . .	269
DECK . . . . .	269
DIAGTRUNC . . . . .	269
DISK/PRINT . . . . .	270
DLL . . . . .	270
DUMP . . . . .	271
DYNAM . . . . .	272
EXIT . . . . .	272
EXPORTALL . . . . .	272
FASTSRT . . . . .	273
FLAG . . . . .	274
FLAGMIG . . . . .	275
FLAGSTD . . . . .	275
IDLGEN . . . . .	277
INTDATE . . . . .	278
LANGUAGE . . . . .	279
LIB . . . . .	280
LINECOUNT . . . . .	280
LIST . . . . .	281
MAP . . . . .	281
NAME . . . . .	282
NUMBER . . . . .	283
NUMPROC . . . . .	283
OBJECT . . . . .	284
OFFSET . . . . .	285
OPTIMIZE . . . . .	285
Unused data items: . . . . .	286
OUTDD . . . . .	286
PGMNAME . . . . .	287
PGMNAME(COMPAT) . . . . .	287
PGMNAME(LONGUPPER). . . . .	288
PGMNAME(LONGMIXED) . . . . .	288
PGMNAME usage notes. . . . .	288
QUOTE/APOST . . . . .	289
RENT . . . . .	289
RMODE . . . . .	290
SEQUENCE . . . . .	291
SIZE . . . . .	291
SOURCE . . . . .	292
SPACE . . . . .	293
SQL . . . . .	293
SSRANGE . . . . .	294
TERMINAL . . . . .	295
TEST . . . . .	295
TRUNC . . . . .	297
TRUNC example 1 . . . . .	298
TRUNC example 2 . . . . .	299
TYPECHK . . . . .	300
VBREF . . . . .	301
WORD . . . . .	301
XREF . . . . .	302
YEARWINDOW . . . . .	303
ZWB . . . . .	304
Compiler-directing statements . . . . .	304

**Chapter 17. Debugging . . . . . 309**

Debugging with source language. . . . .	310
Tracing program logic . . . . .	310

Finding and handling input-output errors . . . . .	311
Validating data . . . . .	311
Finding uninitialized data . . . . .	311
Generating information about procedures . . . . .	312
Debugging using compiler options . . . . .	313
Finding coding errors . . . . .	314
Finding line sequence problems . . . . .	315
Checking for valid ranges . . . . .	315
Selecting the level of error to be diagnosed . . . . .	316
Finding program entity definitions and references . . . . .	317
Listing data items . . . . .	318
Getting listings . . . . .	319
Example: short listing . . . . .	320
Example: SOURCE and NUMBER output . . . . .	323
Example: embedded map summary . . . . .	325
Terms used in MAP output . . . . .	326
Symbols used in LIST and MAP output . . . . .	326
Example: nested program map . . . . .	328
Reading LIST output . . . . .	328
Example: XREF output - data-name cross-references . . . . .	340
Example: XREF output - program-name cross-references . . . . .	341
Example: embedded cross-reference . . . . .	341
Example: OFFSET compiler output . . . . .	342
Example: VBREF compiler output . . . . .	343
Preparing to use the debugger . . . . .	343

---

## Part 3. Targeting COBOL programs for certain environments . . . . . 345

### Chapter 18. Developing COBOL programs for CICS . . . . . 347

Coding COBOL programs to run under CICS . . . . .	347
Coding file input and output . . . . .	348
Retrieving the system date and time . . . . .	348
Displaying the contents of data items . . . . .	348
Calling to or from COBOL programs . . . . .	348
Coding a COBOL program to run above the 16-MB line . . . . .	349
Determining the success of ECI calls . . . . .	349
Preparing COBOL programs to run under CICS . . . . .	350
Using the CICS translator . . . . .	350
Compiling your CICS program . . . . .	350
CICS reserved-word table . . . . .	351
Handling errors by using CICS HANDLE . . . . .	352
Example: handling errors by using CICS HANDLE . . . . .	352

### Chapter 19. Programming for a DB2 environment . . . . . 355

Coding SQL statements . . . . .	355
Using SQL INCLUDE . . . . .	355
Using binary items . . . . .	355
Determining the success of SQL statements . . . . .	356
Compiling with the SQL option . . . . .	356
Compiling in batch . . . . .	357
Separating DB2 suboptions . . . . .	357

DB2 coprocessor . . . . .	358
---------------------------	-----

### Chapter 20. Running COBOL programs under IMS . . . . . 359

Compiling and linking COBOL programs for running under IMS . . . . .	359
--	-----

### Chapter 21. Running COBOL programs under OS/390 UNIX . . . . . 361

Running in OS/390 UNIX environments . . . . .	361
Setting and accessing environment variables . . . . .	362
Setting environment variables that affect execution . . . . .	362
Resetting environment variables . . . . .	363
Accessing environment variables . . . . .	363
Example: accessing environment variables . . . . .	363
Calling UNIX/POSIX APIs . . . . .	364
fork, exec, and spawn . . . . .	364
Samples . . . . .	365
Accessing main program parameters . . . . .	365
Example: accessing main program parameters . . . . .	366

### Chapter 22. Running COBOL programs under CMS . . . . . 369

Run-time restrictions under CMS . . . . .	369
Handling QSAM files under CMS . . . . .	369
Run-time message IGZ0002S . . . . .	370

### Chapter 23. Accessing COBOL programs interactively with ISPF . . . . . 371

---

## Part 4. Developing object-oriented programs . . . . . 373

### Chapter 24. Writing object-oriented programs . . . . . 375

Example: mail-order catalog . . . . .	375
Subclasses . . . . .	377
Defining a class . . . . .	378
CLASS-ID paragraph for defining a class . . . . .	378
REPOSITORY paragraph for defining a class . . . . .	379
WORKING-STORAGE SECTION for defining a class . . . . .	379
Example: defining a class . . . . .	380
Defining a class method . . . . .	381
METHOD-ID paragraph for defining a class method . . . . .	381
Overriding a method . . . . .	381
INPUT-OUTPUT SECTION for defining a method . . . . .	382
DATA DIVISION for defining a method . . . . .	382
PROCEDURE DIVISION for defining a method . . . . .	383
Coding special methods . . . . .	383
Example: defining a method . . . . .	384
Defining a client program . . . . .	389
REPOSITORY paragraph for defining a client . . . . .	389
WORKING-STORAGE SECTION for defining a client . . . . .	390

Creating and freeing instances of classes . . . . .	390
Manipulating object references . . . . .	391
Invoking methods . . . . .	391
Example: defining a client . . . . .	392
Defining a subclass . . . . .	393
CLASS-ID paragraph for defining a subclass	394
REPOSITORY paragraph for defining a subclass	394
WORKING-STORAGE SECTION for defining a subclass . . . . .	395
Defining a subclass method . . . . .	395
Example: defining a subclass (with methods)	396
Defining a metaclass . . . . .	405
CLASS-ID paragraph for defining a metaclass	406
REPOSITORY paragraph for defining a metaclass . . . . .	406
WORKING-STORAGE SECTION for defining a metaclass . . . . .	406
Defining a metaclass method . . . . .	406
Changing the definition of a class or subclass	407
Changing a client program . . . . .	407
Example: defining a metaclass (with methods)	408

**Chapter 25. System Object Model. . . . 415**

SOM Interface Repository . . . . .	415
Accessing the SOM Interface Repository . . . . .	415
Populating the SOM Interface Repository . . . . .	416
SOM environment variables . . . . .	417
Example: sample JCL for an object-oriented application . . . . .	418
SOM services . . . . .	420
SOM methods and functions . . . . .	420
Class initialization . . . . .	421
Changing SOM class interfaces . . . . .	422

**Chapter 26. Using SOM IDL-based class libraries . . . . . 425**

SOM objects . . . . .	425
SOM IDL . . . . .	426
Mapping IDL to COBOL . . . . .	427
Using IDL operations . . . . .	427
Expressing IDL attributes . . . . .	428
Common IDL types . . . . .	429
Complex IDL types . . . . .	432
Passing COBOL arguments and return values	435
Example: using a SOM IDL-based class library	440
Handling errors and exceptions . . . . .	442
Passing environment variables . . . . .	443
Checking the exception type field . . . . .	443
Handling exceptions . . . . .	443
Example: checking SOM exceptions . . . . .	443
Creating and initializing object instances . . . . .	445
Looking at the IDL file . . . . .	446
Avoiding memory leaks . . . . .	447
Example: COBOL variable-length string class	448
Source code for helper routines . . . . .	450

**Chapter 27. Wrapping or converting procedure-oriented programs . . . . 451**

OO view of COBOL programs . . . . .	451
Wrapping procedure-oriented programs . . . . .	452

Coordinating procedural code with interface actions . . . . .	452
Integrating procedural code into OO systems	452
Changing procedural code . . . . .	453
Converting from procedure-oriented to OO programs . . . . .	453
Identifying objects . . . . .	454
Analyzing data flow and usage . . . . .	454
Reallocating code to objects . . . . .	454
Writing the object-oriented code . . . . .	455

**Part 5. Working with more complex applications . . . . . 457**

**Chapter 28. Using subprograms . . . . 459**

Main programs, subprograms, and calls . . . . .	459
Ending and reentering main programs or subprograms . . . . .	460
Transferring control to another program . . . . .	461
Making static calls . . . . .	462
Making dynamic calls . . . . .	462
Performance considerations of static and dynamic calls . . . . .	464
Making both static and dynamic calls . . . . .	465
Example: static and dynamic CALL statements	465
NODYNAM restriction (CMS only) . . . . .	467
Calling nested COBOL programs . . . . .	468
Making recursive calls . . . . .	472
Calling to and from object-oriented programs	472
Using procedure pointers . . . . .	472
Calling a C function-pointer . . . . .	473
Calling to alternate entry points . . . . .	473
Making programs reentrant . . . . .	474

**Chapter 29. Sharing data . . . . . 475**

Passing data . . . . .	475
Describing arguments in the calling program	477
Describing parameters in the called program	477
Coding the LINKAGE SECTION . . . . .	477
Coding the PROCEDURE DIVISION for passing arguments . . . . .	478
Grouping data to be passed . . . . .	478
Handling null-terminated strings . . . . .	478
Using pointers to process a chained list . . . . .	479
Passing return code information . . . . .	482
Understanding the RETURN-CODE special register . . . . .	482
Using PROCEDURE DIVISION RETURNING . . . . .	482
Specifying CALL . . . . . RETURNING . . . . .	483
Sharing data by using the EXTERNAL clause . . . . .	483
Sharing files between programs (external files) . . . . .	483
Example: using external files . . . . .	484

**Chapter 30. Creating a DLL or a DLL application . . . . . 489**

Dynamic link libraries (DLLs) . . . . .	489
Compiling programs to create DLLs . . . . .	490
Linking DLLs . . . . .	491

Prelinking certain DLLs . . . . .	492
Example: sample JCL for a procedural DLL application . . . . .	492
Using CALL identifier with DLLs . . . . .	493
Search order for DLLs in HFS . . . . .	494
Using DLL linkage and dynamic calls together . . . . .	494
Using procedure-pointers with DLLs . . . . .	495
Calling DLLs from non-DLLs . . . . .	496
Example: calling DLLs from non-DLLs . . . . .	496
Using COBOL DLLs with C/C++ programs . . . . .	498
Using DLLs in OO COBOL applications . . . . .	498

**Chapter 31. Interrupts and checkpoint/restart . . . . . 501**

Setting checkpoints . . . . .	501
Designing checkpoints . . . . .	502
Testing for a successful checkpoint . . . . .	502
DD statements for defining checkpoint data sets	503
Messages generated during checkpoint . . . . .	504
Restarting programs . . . . .	504
Requesting automatic restart . . . . .	505
Requesting deferred restart . . . . .	505
Formats for requesting deferred restart . . . . .	506
Resubmitting jobs for restart . . . . .	507
Example: restarting a job at a specific checkpoint step . . . . .	507
Example: requesting a step restart . . . . .	507
Example: resubmitting a job for a step restart	507
Example: resubmitting a job for a checkpoint restart . . . . .	508

**Chapter 32. Processing two-digit-year dates . . . . . 509**

Millennium language extensions (MLE) . . . . .	510
Principles and objectives of these extensions . . . . .	510
Resolving date-related logic problems . . . . .	511
Using a century window . . . . .	512
Using internal bridging . . . . .	513
Moving to full field expansion . . . . .	514
Using year-first, year-only, and year-last date fields	516
Compatible dates . . . . .	517
Example: comparing year-first date fields . . . . .	518
Using other date formats . . . . .	518
Example: isolating the year . . . . .	518
Manipulating literals as dates . . . . .	519
Assumed century window . . . . .	520
Treatment of nondates . . . . .	521
Setting triggers and limits . . . . .	521
Example: using limits . . . . .	522
Using sign conditions . . . . .	523
Sorting and merging by date . . . . .	523
Example: sorting by date and time . . . . .	524
Performing arithmetic on date fields . . . . .	525
Allowing for overflow from windowed date fields . . . . .	525
Specifying the order of evaluation . . . . .	526
Controlling date processing explicitly . . . . .	527
Using DATEVAL . . . . .	527
Using UNDATE . . . . .	527

Analyzing and avoiding date-related diagnostic messages . . . . .	528
Avoiding problems in processing dates . . . . .	530
Avoiding problems with packed-decimal fields	530
Moving from expanded to windowed date fields	530

**Part 6. Improving performance and productivity . . . . . 533**

**Chapter 33. Tuning your program . . . . . 535**

Using an optimal programming style . . . . .	535
Using structured programming . . . . .	536
Factoring expressions . . . . .	536
Using symbolic constants . . . . .	536
Grouping constant computations . . . . .	536
Grouping duplicate computations . . . . .	537
Choosing efficient data types . . . . .	537
Computational data items . . . . .	537
Consistent data types . . . . .	538
Arithmetic expressions . . . . .	538
Exponentiations . . . . .	538
Handling tables efficiently . . . . .	539
Optimization of table references . . . . .	540
Optimizing your code . . . . .	542
Optimization . . . . .	542
Example: PERFORM procedure integration . . . . .	544
Choosing compiler features to enhance performance . . . . .	544
Performance-related compiler options . . . . .	545
Evaluating performance . . . . .	548
Running efficiently with CICS, IMS, or VSAM . . . . .	548
CICS . . . . .	548
IMS . . . . .	549
VSAM . . . . .	549

**Chapter 34. Simplifying coding . . . . . 551**

Eliminating repetitive coding . . . . .	551
Example: using the COPY statement . . . . .	552
Using Language Environment callable services . . . . .	553
Sample list of Language Environment callable services . . . . .	554
Calling Language Environment services . . . . .	555
Example: Language Environment callable services . . . . .	556

**Part 7. Appendixes . . . . . 557**

**Appendix A. Intermediate results and arithmetic precision . . . . . 559**

Terminology used for intermediate results . . . . .	560
Example: calculation of intermediate results . . . . .	561
Fixed-point data and intermediate results . . . . .	561
Addition, subtraction, multiplication, and division . . . . .	561
Exponentiation . . . . .	562
Example: exponentiation in fixed-point arithmetic . . . . .	563
Truncated intermediate results . . . . .	564
Binary data and intermediate results . . . . .	564

Intrinsic functions evaluated in fixed-point arithmetic . . . . .	565
Integer functions . . . . .	565
Mixed functions . . . . .	565
Floating-point data and intermediate results . . . . .	566
Exponentiations evaluated in floating-point arithmetic . . . . .	567
Intrinsic functions evaluated in floating-point arithmetic . . . . .	567
ON SIZE ERROR and intermediate results. . . . .	567
Arithmetic expressions in nonarithmetic statements	568

**Appendix B. Complex OCCURS  
DEPENDING ON . . . . . 569**

Example: complex ODO . . . . .	569
How length is calculated . . . . .	570
Setting values of ODO objects . . . . .	570
Effects of change in ODO object value . . . . .	570
Preventing index errors when changing ODO object value . . . . .	571
Preventing overlay when adding elements to a variable table . . . . .	571

**Appendix C. EXIT compiler option . . . 575**

Using the user-exit work area . . . . .	576
Calling from exit modules . . . . .	576
Processing of INEXIT . . . . .	577
Parameter list for INEXIT . . . . .	577
Processing of LIBEXIT . . . . .	578
Processing of LIBEXIT with nested COPY statements . . . . .	578
Parameter list for LIBEXIT . . . . .	579

Processing of PRTEXT . . . . .	580
Parameter list for PRTEXT . . . . .	581
Processing of ADEXIT . . . . .	582
Parameter list for ADEXIT . . . . .	582
Error handling for exit modules . . . . .	583
Example: SYSIN user exit . . . . .	583

**Appendix D. Sample programs . . . . 587**

IGYTCARA: batch application. . . . .	587
Input data for IGYTCARA . . . . .	588
Report produced by IGYTCARA . . . . .	589
Running IGYTCARA . . . . .	590
IGYTCARB: interactive program . . . . .	592
Running IGYTCARB . . . . .	592
IGYTSALE: nested program application . . . . .	595
Input data for IGYTSALE . . . . .	597
Reports produced by IGYTSALE . . . . .	598
Running IGYTSALE . . . . .	602
Language elements and concepts that are illustrated . . . . .	604

**Notices . . . . . 609**

Trademarks . . . . .	610
----------------------	-----

**Glossary . . . . . 611**

**List of resources . . . . . 633**

COBOL for OS/390 & VM . . . . .	633
Related publications . . . . .	633

**Index . . . . . 635**



---

## About this book

Welcome to IBM COBOL for OS/390 & VM, IBM's latest host COBOL compiler!

This version of IBM COBOL for OS/390 & VM is fully source and object compatible with COBOL for MVS & VM Version 1, with the exception of programs containing object-oriented language constructs, which must be recompiled to enable support for OS/390 SOMobjects.

IBM COBOL for OS/390 & VM is referred to as "COBOL for OS/390 & VM" throughout this publication.

---

## How this book will help you

This book will help you write and compile COBOL for OS/390 & VM programs. It will also help you define object-oriented classes and methods, invoke methods, and refer to objects in your programs. A companion volume, *IBM Language Environment Programming Guide*, provides instructions on link-editing and running your programs.

This book assumes experience in developing application programs and some knowledge of COBOL. It focuses on using COBOL for OS/390 & VM 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*.

For information on migrating OS/VS COBOL and VS COBOL II programs to COBOL for OS/390 & VM, see *IBM COBOL for OS/390 & VM Compiler and Run-Time Migration Guide*.

IBM Language Environment for OS/390 & VM provides the run-time environment and run-time services required to run your COBOL for OS/390 & VM programs. You will find information on link-editing and running programs in the *IBM Language Environment Programming Guide* and *IBM Language Environment Programming Reference*.

For a comparison of commonly used COBOL for OS/390 & VM and IBM Language Environment for OS/390 & VM terms, see "Comparison of commonly-used terms" on page xiii.

---

## Abbreviated terms

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 following table. Abbreviations for other terms, if not commonly understood, are shown in *italics* the first time they appear, and are listed in the glossary in the back of this book.

Term used	Long form
CICS	CICS/ESA
COBOL for OS/390 & VM	IBM COBOL for OS/390 & VM
CMS	CMS component of VM/ESA

Term used	Long form
IMS	IMS/VS or IMS/ESA
Language Environment	IBM Language Environment for OS/390 & VM
MVS	MVS/ESA
SOM	System Object Model
VM	VM/ESA

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
- ISO 1989/Amendment 1, Programming Languages - COBOL - Amendment 1: Intrinsic function module
- X3.23-1985, American National Standard for Information Systems - Programming Language - COBOL
- X3.23a-1989, American National Standard for Information Systems - Programming Language - Intrinsic Function Module for COBOL

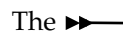
The two ISO standards are identical to the American National standards.

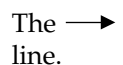
---

## How to read syntax diagrams

The following rules apply to syntax diagrams:

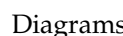
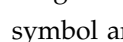
- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

The  symbol indicates the beginning of a statement.

The  symbol indicates that the statement syntax is continued on the next line.

The  symbol indicates that a statement is continued from the previous line.

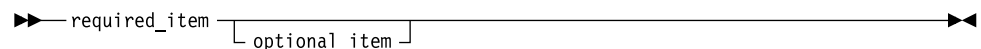
The  symbol indicates the end of a statement.

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

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



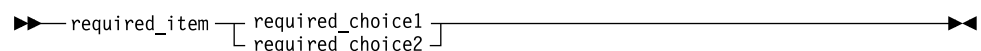
- Optional items appear below the main path.



If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.



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



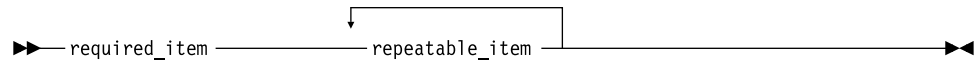
If choosing one of the items is optional, the entire stack appears below the main path.



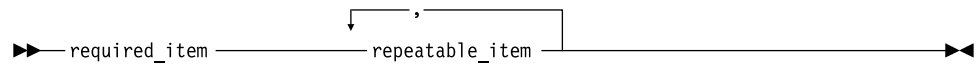
If one of the items is the default, it appears above the main path and the remaining choices are shown below.



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



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



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.

---

## How examples are shown

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

Where it helps to more clearly separate the examples from the explanatory text, they are presented in a different font style.

COBOL keywords and compiler options appearing in text are generally shown in SMALL UPPER CASE. Other terms such as program variable names or method names are sometimes shown in a different font style for clarity.

---

## Comparison of commonly-used terms

In order to better understand the various terms used throughout the IBM Language Environment for OS/390 & VM and IBM COBOL for OS/390 & VM publications and what terms are meant to be equivalent, see the following table.

Language Environment term	COBOL for OS/390 & VM equivalent
Aggregate	Group item
Array	A table created using the OCCURS clause
Array element	Table element

Language Environment term	COBOL for OS/390 & VM equivalent
Enclave	Run unit
External data	WORKING-STORAGE data defined with EXTERNAL clause
Local data	Any non-EXTERNAL data item
Pass parameters directly, by value	BY VALUE
Pass parameters indirectly, by reference	BY REFERENCE
Pass parameters indirectly, by value	BY CONTENT
Routine	Program
Scalar	Elementary item

---

## Summary of changes

This section lists the key changes that have been made to IBM COBOL for OS/390 & VM. Those documented in this publication have an associated page reference for your convenience.

The latest technical changes are marked by a change bar in the left margin. Many corrections have been made without comment.

Chapters have been extensively reorganized, and artwork improved. Cross-references (except for those to examples) have been grouped together at the ends of topics so as not to interrupt the flow of the text.

## Major changes

- Enhanced support for decimal data, raising the maximum number of decimal digits from 18 to 31 and providing an extended-precision mode for arithmetic calculations (“ARITH” on page 262)
- Enhanced production debugging using overlay hooks rather than compiled-in hooks, with symbolic debugging information optionally in a separate file (“TEST” on page 295)
- Support for compiling, linking, and executing in the OS/390 UNIX System Services environment, with COBOL files able to reside in the HFS (hierarchical file system) (“Chapter 14. Compiling under OS/390 UNIX” on page 235)
- Toleration of fork(), exec(), and spawn(); and the ability to call UNIX/POSIX functions (“Calling UNIX/POSIX APIs” on page 364)
- Enhanced input-output function, permitting dynamic file allocation by means of an environment variable named in SELECT . . . ASSIGN, and the accessing of sequentially organized HFS files including by means of ACCEPT and DISPLAY (“Allocating files” on page 105)
- Support for line-sequential file organization for accessing HFS files that contain text data, with records delimited by the new-line character (“Chapter 10. Processing line-sequential files” on page 163)
- COMP-5 data type, new to host COBOL, allowing values of magnitude up to the capacity of the native binary representation (“Formats for numeric data” on page 36)
- Significant performance improvement in processing binary data with the TRUNC(BIN) compiler option (“TRUNC” on page 297)

- Support for linking of COBOL applications using the OS/390 DFSMS binder alone, with the prelinker required only in exceptional cases under CICS (“Compiling programs to create DLLs” on page 490)
- Diagnosis of moves (implicit or explicit) that result in numeric truncation enabled via compiler option (“DIAGTRUNC” on page 269)
- System-determined block size for the listing data set available by specifying BLKSIZE=0 (“Logical record length and block size” on page 219)
- Limit on block size of QSAM tape files raised to 2 GB (“Setting block sizes” on page 115)
- Support under CICS for DISPLAY to the system logical output device and ACCEPT for obtaining date and time (“Coding COBOL programs to run under CICS” on page 347)
- Support for the DB2 coprocessor enabled through a new compiler option, eliminating the need for a separate precompile step and permitting SQL statements in nested programs and copy books (“SQL” on page 293)
- Support for the millennium language extensions now included in the base COBOL product (“Millennium language extensions (MLE)” on page 510)

### Changes in the fifth edition

- New compiler option ANALYZE to check the syntax of embedded SQL and CICS statements (“ANALYZE” on page 261).
- Extension of the ACCEPT statement to cover the recommendation in the *Working Draft for Proposed Revision of ISO 1989:1985 Programming Language COBOL*.
- New intrinsic date functions to convert to dates with a four-digit year.
- The millennium language extensions, enabling compiler-assisted date processing for dates containing two-digit and four-digit years (“Millennium language extensions (MLE)” on page 510).  
Requires IBM VisualAge millennium language extensions for OS/390 & VM (program number 5648-MLE) to be installed with your compiler.

### Changes in the fourth edition

- Extensions to currency support for displaying financial data, including:
  - Support for currency signs of more than one character
  - Support for more than one type of currency sign in the same program
  - Support for the euro currency sign, as defined by the Economic and Monetary Union (EMU)

### Changes from COBOL for MVS & VM

- Support has been added for dynamic link libraries (DLLs). A DLL is a load module containing programs and data that can be accessed from other load modules. The DLL mechanism is the primary means used for packaging SOM class libraries. With this new support, object-oriented COBOL applications can be packaged using separate DLL load modules for the client programs and class definitions. Two new compiler options, DLL|NODLL and EXPORTALL|NOEXPORTALL, are used to control the creation of DLLs. For details, see “Chapter 30. Creating a DLL or a DLL application” on page 489.
- Due to changes in the SOMobjects product that is delivered with OS/390 Release 3, changes in the JCL for building object-oriented COBOL applications are required:
  - The SOMobjects kernel and the SOMobjects class libraries *must* be accessed from the SOMobjects product DLLs, rather than being linked together with the COBOL application load module as was required in the prior version.

This requires changes to the JCL to specify appropriate COBOL compiler options for DLL support; the prelink step must specify (at least) the DLL definition side decks for the SOM kernel and class libraries, and the SOM SGOSPLKD data set is no longer used in the SYSLIB concatenation of the prelink step.

- The format of the SOM profile data set has changed, and the profile is now specified via the SOMENV DD statement rather than the SOMPROF DD statement.

For further details, see the *OS/390 SOMobjects Programmer's Guide*, and "Chapter 25. System Object Model" on page 415.

- The INTDATE compiler option is no longer an installation option only—it can now be specified as an option when invoking the compiler. See "INTDATE" on page 278.

---

# Part 1. Coding your program

<b>Chapter 1. Structuring your program</b> . . . . .	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 entries . . . . .	7
Specifying the collating sequence . . . . .	8
Example: specifying the collating sequence . . . . .	8
Defining symbolic characters . . . . .	9
Defining a user-defined class . . . . .	9
Defining files to the operating system . . . . .	10
Varying the input or output file at run time . . . . .	11
Optimizing buffer and device space . . . . .	12
Describing the data. . . . .	12
Using data in input and output operations . . . . .	13
FILE SECTION entries. . . . .	14
Comparison of WORKING-STORAGE and LOCAL-STORAGE . . . . .	15
Example: storage sections. . . . .	15
Using data from another program . . . . .	16
Sharing data in separately compiled programs . . . . .	16
Sharing data in nested programs . . . . .	17
Processing the data. . . . .	17
How logic is divided in the PROCEDURE DIVISION . . . . .	17
Imperative statements . . . . .	18
Conditional statements . . . . .	19
Compiler-directing statements . . . . .	20
Scope terminators . . . . .	20
Declaratives . . . . .	21
<b>Chapter 2. Using data</b> . . . . .	23
Using variables, structures, literals, and constants . . . . .	23
Variables . . . . .	23
Data structure: data items and group items. . . . .	23
Literals . . . . .	24
Constants . . . . .	24
Figurative constants . . . . .	24
Assigning values to data items . . . . .	25
Examples: initializing variables . . . . .	25
Initializing a structure (INITIALIZE) . . . . .	26
Assigning values to variables or structures (MOVE) . . . . .	27
Assigning arithmetic results (MOVE or COMPUTE) . . . . .	27
Assigning input from a screen or file (ACCEPT) . . . . .	28
Displaying values on a screen or in a file (DISPLAY) . . . . .	29
Displaying data on the system logical output device . . . . .	29
Using WITH NO ADVANCING . . . . .	30
Using intrinsic functions (built-in functions) . . . . .	31
Types of intrinsic functions . . . . .	31
Nesting functions . . . . .	32
Using tables (arrays) and pointers . . . . .	32
<b>Chapter 3. Working with numbers and arithmetic</b> . . . . .	33
Defining numeric data. . . . .	33
Displaying numeric data . . . . .	34
Controlling how numeric data is stored . . . . .	35
Formats for numeric data. . . . .	36
External decimal (DISPLAY) items. . . . .	36
External floating-point (DISPLAY) items. . . . .	36
Binary (COMP) items . . . . .	37
Native binary (COMP-5) items . . . . .	37
Packed-decimal (COMP-3) items . . . . .	38
Floating-point (COMP-1 and COMP-2) items . . . . .	38
Examples: numeric data and internal representation . . . . .	38
Data format conversions . . . . .	39
Conversions and precision . . . . .	40
Conversions that preserve precision . . . . .	40
Conversions that result in rounding . . . . .	40
Sign representation and processing . . . . .	41
NUMPROC(PFD) . . . . .	41
NUMPROC(NOPFD) . . . . .	41
NUMPROC(MIG) . . . . .	41
Checking for incompatible data (numeric class test) . . . . .	42
Performing arithmetic . . . . .	43
COMPUTE and other arithmetic statements . . . . .	43
Arithmetic expressions . . . . .	43
Numeric intrinsic functions . . . . .	44
Nesting functions and arithmetic expressions . . . . .	45
ALL subscripting and special registers . . . . .	45
Math and date Language Environment services . . . . .	45
Math-oriented callable services . . . . .	45
Date callable services . . . . .	46
Examples: numeric intrinsic functions . . . . .	47
General number handling . . . . .	47
Date and time . . . . .	47
Finance. . . . .	47
Mathematics . . . . .	48
Statistics . . . . .	48
Fixed-point versus floating-point arithmetic . . . . .	49
Floating-point evaluations . . . . .	49
Fixed-point evaluations . . . . .	49
Arithmetic comparisons (relation conditions) . . . . .	50
Examples: fixed-point and floating-point evaluations . . . . .	50
Using currency signs . . . . .	51
Example: multiple currency signs . . . . .	52
<b>Chapter 4. Handling tables.</b> . . . . .	53
Defining a table (OCCURS) . . . . .	53
Nesting tables . . . . .	54
Subscripting . . . . .	54
Indexing . . . . .	55
Referring to an item in a table . . . . .	55
Subscripting . . . . .	56
Indexing . . . . .	57
Putting values into a table . . . . .	58

Loading a table dynamically. . . . .	58
Initializing a table (INITIALIZE) . . . . .	58
Assigning values when you define a table (VALUE) . . . . .	58
Initializing each table item individually . . . . .	59
Initializing a table at the 01 level . . . . .	59
Initializing all occurrences of a table element	59
Example: PERFORM and subscripting . . . . .	60
Example: PERFORM and indexing. . . . .	61
Creating variable-length tables (DEPENDING ON)	62
Loading a variable-length table. . . . .	63
Assigning values to a variable-length table . . . . .	64
Searching a table . . . . .	65
Doing a serial search (SEARCH) . . . . .	65
Example: serial search . . . . .	66
Doing a binary search (SEARCH ALL) . . . . .	66
Example: binary search . . . . .	67
Processing table items using intrinsic functions . . . . .	67
Example: intrinsic functions . . . . .	68
<b>Chapter 5. Selecting and repeating program actions . . . . .</b>	<b>69</b>
Selecting program actions . . . . .	69
Coding a choice of actions . . . . .	69
Using nested IF statements . . . . .	70
Using the EVALUATE statement . . . . .	71
Coding conditional expressions. . . . .	73
Switches and flags . . . . .	74
Defining switches and flags . . . . .	74
Example: switches . . . . .	74
Example: flags . . . . .	75
Resetting switches and flags. . . . .	75
Example: set switch on . . . . .	75
Example: set switch off . . . . .	76
Repeating program actions . . . . .	76
Choosing inline or out-of-line PERFORM . . . . .	76
Example: inline PERFORM statement. . . . .	77
Coding a loop . . . . .	77
Coding a loop through a table . . . . .	78
Executing multiple paragraphs or sections . . . . .	79
<b>Chapter 6. Handling strings . . . . .</b>	<b>81</b>
Joining data items (STRING) . . . . .	81
Example: STRING statement. . . . .	81
STRING program results . . . . .	82
Splitting data items (UNSTRING) . . . . .	83
Example: UNSTRING statement . . . . .	83
UNSTRING program results. . . . .	84
Manipulating null-terminated strings. . . . .	85
Example: null-terminated strings . . . . .	86
Referring to substrings of data items . . . . .	86
Reference modifiers. . . . .	87
Example: arithmetic expressions as reference modifiers . . . . .	88
Example: intrinsic functions as reference modifiers . . . . .	88
Tallying and replacing data items (INSPECT) . . . . .	89
Examples: INSPECT statement . . . . .	89
Converting double-byte character set (DBCS) data	90
DBCS notation . . . . .	90

Nonnumeric to DBCS data conversion (IGZCA2D) . . . . .	90
IGZCA2D syntax . . . . .	91
IGZCA2D return codes . . . . .	91
Example: IGZCA2D . . . . .	92
DBCS to nonnumeric data conversion (IGZCD2A) . . . . .	92
IGZCD2A syntax . . . . .	93
IGZCD2A return codes . . . . .	93
Example: IGZCD2A . . . . .	94
Converting data items (intrinsic functions) . . . . .	94
Converting to uppercase or lowercase (UPPER-CASE, LOWER-CASE). . . . .	94
Converting to reverse order (REVERSE) . . . . .	95
Converting to numbers (NUMVAL, NUMVAL-C)	95
Evaluating data items (intrinsic functions) . . . . .	96
Evaluating single characters for collating sequence . . . . .	96
Finding the largest or smallest data item . . . . .	97
MAX and MIN . . . . .	97
ORD-MAX and ORD-MIN . . . . .	97
Returning variable-length results with alphanumeric functions . . . . .	97
Finding the length of data items . . . . .	98
Finding the date of compilation . . . . .	99
<b>Chapter 7. Processing files . . . . .</b>	<b>101</b>
File organization and input-output devices . . . . .	101
Choosing file organization and access mode . . . . .	103
Format for coding input and output. . . . .	103
Allocating files . . . . .	105
Checking for input or output errors . . . . .	106
<b>Chapter 8. Processing QSAM files . . . . .</b>	<b>107</b>
Defining QSAM files and records in COBOL . . . . .	107
Establishing record formats. . . . .	108
Logical records . . . . .	108
Requesting fixed-length format . . . . .	109
Requesting variable-length format . . . . .	110
Requesting spanned format. . . . .	112
Requesting undefined format . . . . .	114
Setting block sizes. . . . .	115
Letting OS/390 determine block size . . . . .	115
Setting block size explicitly. . . . .	115
Taking advantage of LBI. . . . .	116
Block size and the DCB RECFM subparameter . . . . .	116
Coding input and output statements for QSAM files . . . . .	117
Opening QSAM files . . . . .	117
Dynamically creating QSAM files with CBLQDA	118
Adding records to QSAM files. . . . .	119
Updating QSAM files. . . . .	119
Writing QSAM files to a printer or spooled data set . . . . .	119
Controlling the page size . . . . .	119
Controlling the vertical positioning of records	119
Closing QSAM files . . . . .	120
Handling errors in QSAM files . . . . .	120
Working with QSAM files under OS/390 . . . . .	121
Defining and allocating QSAM files . . . . .	121



Parameters for creating QSAM files . . . . .	123
Retrieving QSAM files . . . . .	123
Parameters for retrieving QSAM files . . . . .	124
Ensuring file attributes match your program . . . . .	124
Processing existing files . . . . .	125
Defining variable-length (format-V) records . . . . .	125
Defining format-U records . . . . .	125
Defining fixed-length records . . . . .	125
Processing new files . . . . .	125
Processing files dynamically created by COBOL . . . . .	126
Using striped extended-format QSAM data sets . . . . .	126
Allocation of buffers for QSAM files . . . . .	127
Accessing HFS files using QSAM . . . . .	127
Restrictions and usage . . . . .	127
Identifying QSAM files to CMS . . . . .	128
Using FILEDEF . . . . .	128
Using LABELDEF . . . . .	129
Labels for QSAM files . . . . .	129
Using trailer and header labels . . . . .	130
Getting a user-label track . . . . .	130
Handling user labels . . . . .	130
Format of standard labels . . . . .	131
Standard user labels . . . . .	131
Processing QSAM ASCII files on tape . . . . .	132
Requesting the ASCII alphabet . . . . .	132
Defining the record formats . . . . .	132
Defining the ddname . . . . .	132
Processing ASCII file labels . . . . .	133
<b>Chapter 9. Processing VSAM files.</b> . . . . .	<b>135</b>
VSAM files . . . . .	136
Defining VSAM file organization and records . . . . .	137
Specifying sequential organization for VSAM files . . . . .	138
Specifying indexed organization for VSAM files . . . . .	138
Alternate keys . . . . .	139
Alternate index . . . . .	139
Specifying relative organization for VSAM files . . . . .	139
Fixed-length and variable-length RRDS . . . . .	140
Simulating variable-length RRDS . . . . .	140
Specifying access modes for VSAM files . . . . .	141
Example: using dynamic access with VSAM files . . . . .	141
Defining record lengths for VSAM files . . . . .	141
Defining fixed-length records . . . . .	142
Defining variable-length records . . . . .	142
Coding input and output statements for VSAM files . . . . .	143
File position indicator . . . . .	144
Opening a file (ESDS, KSDS, or RRDS) . . . . .	145
Opening an empty file . . . . .	145
Statements to load records into a VSAM file . . . . .	147
Opening a loaded file (a file with records) . . . . .	147
Reading records from a VSAM file . . . . .	147
Updating records in a VSAM file . . . . .	148
Adding records to a VSAM file . . . . .	149
Adding records sequentially . . . . .	149
Adding records randomly or dynamically . . . . .	149
Replacing records in a VSAM file . . . . .	149
Deleting records from a VSAM file . . . . .	150

Closing VSAM files . . . . .	150
Handling errors in VSAM files . . . . .	151
Protecting VSAM files with a password . . . . .	151
Example: password protection for a VSAM indexed file . . . . .	152
Working with VSAM data sets under OS/390 and OS/390 UNIX . . . . .	152
Defining VSAM files under OS/390 . . . . .	153
Creating alternate indexes . . . . .	154
Example: entries for alternate indexes . . . . .	155
Allocating VSAM files . . . . .	156
Sharing VSAM files through RLS . . . . .	157
Preventing update problems with VSAM files in RLS mode . . . . .	157
Restrictions when using RLS . . . . .	158
Handling errors in VSAM files in RLS mode . . . . .	158
Defining VSAM data sets under CMS . . . . .	158
Improving VSAM performance . . . . .	159

<b>Chapter 10. Processing line-sequential files</b> . . . . .	<b>163</b>
Defining line-sequential files and records in COBOL . . . . .	163
Allowable control characters . . . . .	164
Describing the structure of a line-sequential file . . . . .	164
Defining and allocating line-sequential files . . . . .	165
Coding input-output statements for line-sequential files . . . . .	165
Opening line-sequential files . . . . .	166
Reading records from line-sequential files . . . . .	166
Adding records to line-sequential files . . . . .	167
Closing line-sequential files . . . . .	167
Handling errors in line-sequential files . . . . .	168

<b>Chapter 11. Sorting and merging files</b> . . . . .	<b>169</b>
Sort and merge process . . . . .	170
Describing the sort or merge file . . . . .	170
Describing the input to sorting or merging . . . . .	171
Example: describing sort and input files for SORT . . . . .	171
Coding the input procedure . . . . .	172
Describing the output from sorting or merging . . . . .	173
Coding the output procedure . . . . .	173
Coding considerations when using DFSORT on OS/390 . . . . .	174
Example: coding the output procedure when using DFSORT . . . . .	174
Restrictions on input and output procedures . . . . .	175
Defining sort and merge data sets under OS/390 . . . . .	175
Defining sort and merge files under CMS . . . . .	176
Sorting variable-length records . . . . .	176
Requesting the sort or merge . . . . .	177
Setting sort or merge criteria . . . . .	177
Example: sorting with input and output procedures . . . . .	178
Choosing alternate collating sequences . . . . .	179
Sorting on windowed date fields . . . . .	179
Preserving the original sequence of records with equal keys . . . . .	180
Determining whether the sort or merge was successful . . . . .	180
Stopping a sort or merge operation prematurely . . . . .	181

Improving sort performance with FASTSRT . . . . .	181
FASTSRT requirements for JCL (OS/390 only)	181
FASTSRT requirements for sort input and output files . . . . .	181
QSAM requirements . . . . .	182
VSAM requirements . . . . .	183
Checking for sort errors with NOFASTSRT . . . . .	183
Controlling sort behavior . . . . .	184
Sort special registers . . . . .	184
Changing DFSORT defaults with control statements . . . . .	185
Default characteristics of the IGZSRTCD data set . . . . .	185
Allocating storage for sort or merge operations	186
Using checkpoint/restart with DFSORT under OS/390 . . . . .	186
Sorting under CICS . . . . .	187
CICS SORT application restrictions . . . . .	187
<b>Chapter 12. Handling errors . . . . .</b>	<b>189</b>
Requesting dumps . . . . .	189
Creating a formatted dump. . . . .	189
Creating a system dump . . . . .	190
Handling errors in joining and splitting strings . . . . .	190
Handling errors in arithmetic operations . . . . .	191
Example: checking for division by zero. . . . .	191
Handling errors in input and output operations	191
Using the end-of-file condition (AT END) . . . . .	193
Coding ERROR declaratives . . . . .	194
Using file status keys. . . . .	194
Example: file status key . . . . .	195
Using VSAM return codes (VSAM files only)	196
Example: checking VSAM status codes . . . . .	196
Coding INVALID KEY phrases . . . . .	198
INVALID KEY and ERROR declaratives . . . . .	198
NOT INVALID KEY . . . . .	198
Example: FILE STATUS and INVALID KEY . . . . .	198
Handling errors when calling programs . . . . .	199
Writing routines for handling errors . . . . .	199

---

## 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 12

“Processing the data” on page 17

“Defining a class” on page 378

“Defining a class method” on page 381

---

## 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. 03/30/2000.
```

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.
- If you use the NAME compiler option, the name is placed on the NAME linkage editor control statement to identify the object module created by the compilation.

**Tip:** Do not use program names that start with prefixes used by IBM products. If you try to use programs whose names start with any of the following, your CALL statements might resolve to IBM library or compiler routines rather than to your intended program:

AFB	AFH	CBC	CEE	EDC
IBM	IFY	IGY	IGZ	ILB

**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”  
“Identifying a program as recursive”  
“Marking a program as callable by containing programs”  
“Setting a program to an initial state”

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

“Making recursive calls” on page 472

## 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 469

## 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. For example:

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 external QSAM, VSAM, or HFS (hierarchical file system) data sets where they physically reside.

The terms *file*, in COBOL terminology, and *data set* or *HFS file*, in operating system terminology, have essentially the same meaning and are used interchangeably in this documentation.

For Customer Information Control System (CICS) and online Information Management System (IMS) message processing programs (MPP), code only the ENVIRONMENT DIVISION header and, optionally, the CONFIGURATION SECTION. CICS does not allow COBOL definition of files. IMS allows COBOL definition of files only for batch programs.

- Provide information to control efficient transmission of the data records between your program and the external medium.

“Example: FILE-CONTROL entries”

RELATED TASKS

“Specifying the collating sequence” on page 8

“Defining symbolic characters” on page 9

“Defining a user-defined class” on page 9

“Defining files to the operating system” on page 10

RELATED REFERENCES

Sections and paragraphs (*IBM COBOL Language Reference*)

### Example: FILE-CONTROL entries

The following table shows FILE-CONTROL entries for a QSAM sequential file, a VSAM indexed file, and a line-sequential file.

QSAM file	VSAM file	Line-sequential file
SELECT PRINTFILE <sup>1</sup> ASSIGN TO UPDPRINT <sup>2</sup> ORGANIZATION IS SEQUENTIAL <sup>3</sup> ACCESS IS SEQUENTIAL. <sup>4</sup>	SELECT COMMUTER-FILE <sup>1</sup> ASSIGN TO COMMUTER <sup>2</sup> ORGANIZATION IS INDEXED <sup>3</sup> ACCESS IS RANDOM <sup>4</sup> RECORD KEY IS COMMUTER-KEY <sup>5</sup> FILE STATUS IS <sup>5</sup> COMMUTER-FILE-STATUS COMMUTER-VSAM-STATUS.	SELECT PRINTFILE <sup>1</sup> ASSIGN TO UPDPRINT <sup>2</sup> ORGANIZATION IS LINE SEQUENTIAL <sup>3</sup> ACCESS IS SEQUENTIAL. <sup>4</sup>

1. The SELECT clause chooses a file in the COBOL program to be associated with an external data set.
2. The ASSIGN clause associates the program's name for the file with the external name for the actual data file. You can define the external name with a DD statement or an environment variable.
3. The ORGANIZATION clause describes the file's organization. For QSAM files, the ORGANIZATION clause is optional.
4. The ACCESS MODE clause defines the manner in which the records are made available for processing—sequential, random, or dynamic. For QSAM and line-sequential files, the ACCESS MODE clause is optional. These files always have sequential organization.
5. For VSAM files, you might have additional statements in the FILE-CONTROL paragraph depending on the type of VSAM file you use.

#### RELATED TASKS

- “Chapter 8. Processing QSAM files” on page 107
- “Chapter 9. Processing VSAM files” on page 135
- “Chapter 10. Processing line-sequential files” on page 163
- “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 COLLATINGSEQUENCE 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:

- EBCDIC (use NATIVE if the native character set is EBCDIC), the default if you omit the ALPHABET clause
- ASCII (use STANDARD-1)
- ISO 7-bit code, International Reference Version (use STANDARD-2)
- An alteration of the EBCDIC sequence that you define in the SPECIAL-NAMES paragraph

#### RELATED TASKS

- “Choosing alternate collating sequences” on page 179

### 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 EBCDIC 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.

Source-Computer. IBM-390.

Object-Computer. IBM-390.

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 backspace character (X'16' in the EBCDIC alphabet), code:

```
SYMBOLIC CHARACTERS BACKSPACE IS 23
```

## 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.

## Defining files to the operating system

For all files that you process in your COBOL program, you need to define the files to the operating system with an appropriate system data definition:

- DD statement for OS/390 JCL
- ALLOCATE command under TSO
- FILEDEF command for CMS
- Environment variable for OS/390 or OS/390 UNIX. The contents can define either an MVS data set or a file in the HFS (hierarchical file system).

The following shows the relationship of a FILE-CONTROL entry to the system data definition and to the FD entry in the FILE SECTION.

- JCL DD statement:

```
      (1)
//OUTFILE DD  DSNAME=MY.OUT171,UNIT=SYSDA,SPACE=(TRK,(50,5)),
//           DCB=(BLKSIZE=400)
/*
. . .
```

- FILEDEF command:

```
      (1)
FILEDEF  OUTFILE DISK MY OUTPUT A1 (BLKSIZE 400)
. . .
```

- Environment variable (export command):

```
      (1)
export  OUTFILE=DSNAME(MY.OUT171),UNIT(SYSDA),SPACE(TRK,(50,5))
. . .
```

- COBOL code:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT CARPOOL
        ASSIGN TO OUTFILE (1)
        ORGANIZATION IS SEQUENTIAL.
. . .
DATA DIVISION.
FILE SECTION.
FD CARPOOL      (2)
    LABEL RECORD STANDARD
    BLOCK CONTAINS 0 CHARACTERS
    RECORD CONTAINS 80 CHARACTERS
```

- (1) The *ddname* in the DD statement or the FILEDEF command, or the environment variable name in the export command, corresponds to the *assignment-name* in the ASSIGN clause:

- //OUTFILE DD DSNAME=OUT171 . . ., or
- FILEDEF OUTFILE DISK . . ., or
- export OUTFILE= . . .

This *assignment-name* points to the *ddname* OUTFILE in the DD statement or the FILEDEF command, or the environment variable name OUTFILE in the export command:

```
ASSIGN TO OUTFILE
```

- (2) When you specify a file in a COBOL FILE-CONTROL entry, you must describe the file in an FD entry for *file-name*.

```
SELECT CARPOOL
. . .
FD CARPOOL
```



#### RELATED TASKS

“Optimizing buffer and device space” on page 12

### 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 on the DD statement, export command, or FILEDEF 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 *dsname* in the DD statement, the *dsname* or *path-name* in your export command, or *fn*, *ft*, and *fm* parameters in the FILEDEF command.

**Rules for using different files:** The name you use in the *assignment-name* of the ASSIGN clause must be the same as the *ddname* in the DD statement, the environment variable in the export command, or the *ddname* in the FILEDEF command. You can change the actual file by using the DSNAMES in the DD statement, the DSNAMES or path name in the environment variable, or the *fn*, *ft*, and *fm* in the FILEDEF.

The *file-name* that you use with the SELECT clause (such as SELECT MASTER) must be the same as in the FD *file-name* entry.

Two files should not use the same *ddname* or environment variable name in their SELECT clauses; otherwise, results could be unpredictable. For example, if DISPLAY is directed to SYSOUT, do not use SYSOUT as the *ddname* or environment variable name on the SELECT clause for a file.

“Example: using different input files on OS/390”

“Example: using different input files on VM/CMS” on page 12

**Example: using different input files on OS/390:** 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 DA-3330-S-MASTERA
```

Under OS/390, assume the three possible input files are MASTER1, MASTER2, and MASTER3. Then you must code one of the following DD statements in the job step that calls for program execution, or issue one of the following export commands from the same shell from which you run the program, prior to running the program:

```
//MASTERA DD DSN=MY.MASTER1, . . .
export MASTERA=DSN(MY.MASTER1),...
```

```
//MASTERA DD DSN=MY.MASTER2, . . .
export MASTERA=DSN(MY.MASTER2),...
```

```
//MASTERA DD DSN=MY.MASTER3, . . .
export MASTERA=DSN(MY.MASTER3),...
```

Any reference in the program to MASTERA will therefore be a reference to the file currently assigned to *ddname* or environment variable name MASTERA.

Notice that in this example, you cannot use the PATH(*path*) form of the export command to reference a line-sequential file in the HFS, because you cannot specify an organization field (S- or AS-) with a line-sequential file.

**Example: using different input files on VM/CMS:** 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 DA-3330-S-MASTERA
```

Under VM, assume the three possible input files on your A-disk are MASTER1 INPUT, MASTER2 INPUT, and MASTER3 INPUT. Then you need to issue one of the following FILEDEF commands before running the program:

```
FILEDEF MASTERA DISK MASTER1 INPUT A1 . . .
FILEDEF MASTERA DISK MASTER2 INPUT A1 . . .
FILEDEF MASTERA DISK MASTER3 INPUT A1 . . .
```

Any reference in the program to MASTERA will therefore be a reference to the file named in the FILEDEF command.

### Optimizing buffer and device space

Use the APPLY WRITE-ONLY clause to make optimum use of buffer and device space when creating a sequential file with blocked variable-length records. With APPLY WRITE-ONLY specified, a buffer is truncated only when the next record does not fit in the unused portion of the buffer. Without APPLY WRITE-ONLY specified, a buffer is truncated when it does not have enough space for a maximum-size record.

The APPLY WRITE-ONLY clause has meaning only for sequential files that have variable-length records and are blocked.

The AWO compiler option applies an implicit APPLY WRITE-ONLY clause to all eligible files. The NOAWO compiler option has no effect on files that have the APPLY WRITE-ONLY clause specified. The APPLY WRITE-ONLY clause takes precedence over the NOAWO compiler option.

The APPLY-WRITE ONLY clause can cause input files to use a record area rather than process the data in the buffer. This use might affect the processing of both input files and output files.

RELATED REFERENCES  
"AWO" on page 263

---

## 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 IBM COBOL for OS/390 & VM compiler limits the maximum size of DATA DIVISION elements.

#### RELATED CONCEPTS

“Comparison of WORKING-STORAGE and LOCAL-STORAGE” on page 15

#### RELATED TASKS

“Using data in input and output operations”

“Using data from another program” on page 16

#### 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 fields of the records in the file:
  - You can code a level-01 description of the entire record, and then in the WORKING-STORAGE SECTION code a working copy that describes the fields of the record in more detail. Use the READ INTO statement to bring the records into WORKING-STORAGE. Processing occurs on the copy of data in WORKING-STORAGE. A WRITE FROM statement then writes processed data into the record area defined in the FILE SECTION.
  - The record-name established is the object of WRITE and REWRITE statements.
  - For QSAM files only, you can set the record format in the RECORDING MODE clause. If you omit the RECORDING MODE clause, the compiler determines the record format based on the RECORD clause and on the level-01 record descriptions.
  - For QSAM files, you can set a blocking factor for the file in the BLOCK CONTAINS clause. If you omit the BLOCK CONTAINS clause, the file defaults to unblocked. However, you can override this with OS/390 data management facilities (including a DD file job control statement) or with CMS file services under VM.
  - For line-sequential files, you can set a blocking factor for the file in the BLOCK CONTAINS clause. When you code BLOCK CONTAINS 1 RECORDS, or BLOCK CONTAINS *n* CHARACTERS, where *n* is the length of one logical record, WRITE statements result in the record being transferred immediately to the file, rather than being buffered. This technique is useful when you want each record written immediately, such as to an error log.

Programs in the same run unit can share, or have access to, common files. The method for doing this depends on whether the programs are part of a nested (contained) structure or are separately compiled (including programs compiled as part of a batch sequence).

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.

**RELATED CONCEPTS**

“Nested programs” on page 469

**RELATED TASKS**

“Sharing files between programs (external files)” on page 483

**RELATED REFERENCES**

“FILE SECTION entries”

**FILE SECTION entries**

Clause	To define	Notes
FD	The <i>file-name</i> to be referred to in PROCEDURE DIVISION input-output statements (OPEN, CLOSE, READ, also START and DELETE for VSAM)	Must match <i>file-name</i> in the SELECT clause. <i>file-name</i> is associated with a <i>ddname</i> through the <i>assignment-name</i> .
BLOCK CONTAINS	Size of physical record	QSAM: If provided, must match information on JCL or data set label. If not provided, the system determines the optimal block size for you.  Line sequential: can be specified to control buffering for WRITE statements.  VSAM: Handled as comments.
RECORD CONTAINS <i>n</i>	Size of logical records (fixed length)	If provided, must match information on JCL or data set label. If <i>n</i> is equal to 0, LRECL must be coded on JCL or data set label.
RECORD IS VARYING	Size of logical records (variable length)	If provided, must match information on JCL or data set label; compiler checks match with record descriptions.
RECORD CONTAINS <i>n</i> TO <i>m</i>	Size of logical records (variable length)	If provided, must match information on JCL or data set label; compiler checks match with record descriptions.
LABEL RECORDS	See reference below.	VSAM: Handled as comments.
STANDARD	Labels exist	QSAM: Handled as comments.
OMITTED	Labels do not exist	QSAM: Handled as comments.
<i>data-name</i>	Labels defined by the user	QSAM: Allowed for (optional) tape or disk.
VALUE OF	An item in the label records associated with file	Comments only.
DATA RECORDS	Names of records associated with file	Comments only.
LINAGE	Depth of logical page	QSAM only.

Clause	To define	Notes
CODE-SET	ASCII or EBCDIC files	<p>QSAM only.</p> <p>When an ASCII file is identified with the CODE-SET clause, the corresponding DD statement might need to have DCB=(OPTCD=Q. . .) or DCB=(RECFM=D. . .) coded if the file was not created using VS COBOL II or COBOL for OS/390 &amp; VM.</p>
RECORDING MODE	Physical record description	QSAM only.

RELATED CONCEPTS

“Labels for QSAM files” on page 129

## Comparison of WORKING-STORAGE and LOCAL-STORAGE

WORKING-STORAGE is allocated at the start of the run unit. 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 dynamically called and then canceled  
In this case, WORKING-STORAGE data items are reinitialized on the first reentry into the program following the CANCEL.

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

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”

RELATED TASKS

“Ending and reentering main programs or subprograms” on page 460

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 nocmpr2,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
W-S numb	5	4	3	2	1	0
fact	0	0	0	0	0	0

Recursive GOBACKS:						
	5	4	3	2	1	Main
L-S num	0	1	2	3	4	5
W-S numb	0	0	0	0	0	0
fact	1	1	2	6	24	120

#### RELATED CONCEPTS

“Comparison of WORKING-STORAGE and LOCAL-STORAGE” on page 15

## 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” on page 17

### 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 475

## 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 469

---

## 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 551

## 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 20

“Scope terminators” on page 20

“Imperative statements”

“Conditional statements” on page 19

“Declaratives” on page 21

**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” on page 19

“Scope terminators” on page 20



## 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 18

“Scope terminators” on page 20

#### RELATED TASKS

“Selecting program actions” on page 69

#### 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, or control flow.

#### RELATED REFERENCES

“Compiler-directing statements” on page 304

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
```

#### RELATED CONCEPTS

“Conditional statements” on page 19

“Imperative statements” on page 18

## Declaratives

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

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

#### RELATED TASKS

“Finding and handling input-output errors” on page 311

#### 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 25

“Using intrinsic functions (built-in functions)” on page 31

“Using tables (arrays) and pointers” on page 32

---

## 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 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
To establish a constant  <b>Note:</b> Not a language feature; for optimized code only: the optimizer recognizes an invariant VALUE item and treats it as a constant.	Use the VALUE clause in the definition of the data item.
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 26

“Assigning values to variables or structures (MOVE)” on page 27

“Assigning arithmetic results (MOVE or COMPUTE)” on page 27

“Assigning input from a screen or file (ACCEPT)” on page 28

“Displaying values on a screen or in a file (DISPLAY)” on page 29

## 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	□□□□□ <sup>1</sup>
99XX9	12AB3	□□□□□ <sup>1</sup>
XXBX/XX	ABbC/DE	□□□□/□□ <sup>1</sup>
**99.9CR	1234.5CR	**□□.□□□ <sup>1</sup>
A(5)	ABCDE	□□□□□ <sup>1</sup>
+99.99E+99	+12.34E+02	+□□.□□E+□□

1. The symbol □ 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	□□□□□□□ <sup>1</sup>	□□□□ABCD <sup>1</sup>

1. The symbol □ 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/□□□".
. . .
  INITIALIZE ALPHANUM-EDIT-1
    REPLACING ALPHANUMERIC-EDITED DATA BY ALPHANUM-EDIT-3
```

ALPHANUM-EDIT-3	ALPHANUM-EDIT-1 before	ALPHANUM-EDIT-1 after
M/□□ <sup>1</sup>	AB□C/DEF <sup>1</sup>	M/□□/□□□ <sup>1</sup>

1. The symbol □ 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	R0013830002400000000000000000	␣0000000000000000000000000000 <sup>1</sup>
2	R0013900004800000000000000000	␣0000000000000000000000000000 <sup>1</sup>
3	S0014100001200000000000000000	␣0000000000000000000000000000 <sup>1</sup>
4	C001383000000000425000000000	␣0000000000000000000000000000 <sup>1</sup>
5	C0020100000000000000100000000	␣0000000000000000000000000000 <sup>1</sup>
1. The symbol ␣ 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.

## 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. You can, however, specify the DIAGTRUNC compiler option to request that the compiler issue a warning diagnostic for MOVE statements that might truncate numeric receivers.

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

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)
```

Results of date, time, and mathematical calculations, as well as other operations, can be assigned to data items using Language Environment callable services. These Language Environment services are available via a standard COBOL CALL statement, and the values they return are passed in the parameters in the CALL statement. For example, you can call the Language Environment service CEESIABS to find the absolute value of a variable with the statement:

```
Call 'CEESIABS' Using Arg, Feedback-code, Result.
```

As a result of this call, the variable Result is assigned to be the absolute value of the value that is in the variable Arg; the variable Feedback-code contains the return code indicating whether the service completed successfully. You have to define all the variables in the Data Division using the correct descriptions, according to the requirements of the particular callable service you are using. For the example above, the variables could be defined like this:

```
77 Arg                Pic s9(9)  Binary.
77 Feedback-code     Pic x(12)  Display.
77 Result            Pic s9(9)  Binary.
```

#### RELATED REFERENCES

“DIAGTRUNC” on page 269

Intrinsic functions (*IBM COBOL Language Reference*)

Callable services (*Language Environment Programming 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 the following change:

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

Note that *device* can be a ddname that references a hierarchical file system (HFS) path. If this DD is not defined and your program is running in an OS/390 UNIX environment, stdin is the input source. If this DD is not defined and your program is not running in an OS/390 UNIX environment, the ACCEPT statement fails.

#### 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 system logical output device:

```
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 with a destination other than SYSOUT. For example, the following statement writes to the file specified in the SYSPUNCH DD statement:

```
Display "Hello" UPON SYSPUNCH.
```

Note that you can specify a file in the hierarchical file system (HFS) with this ddname. For example, with the following definition, your DISPLAY output is written to the HFS file /u/userid/cobol/demo.lst:

```
//SYSPUNCH DD PATH='/u/userid/cobol/demo.lst',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),PATHMODE=SIRWXU,
// FILEDATA=TEXT
```

The following statement writes to the job log or console:

```
Display "Hello" UPON CONSOLE.
```

## Displaying data on the system logical output device

To write data to the system logical output device, either omit the UPON clause or use the UPON clause with destination SYSOUT. For example:

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

The output is directed to the ddname that you specify in the OUTDD compiler option. Note that you can specify a file in the hierarchical file system (HFS) with this ddname.

If the OUTDD ddname is not allocated and you are not running in an OS/390 UNIX environment, a default DD of SYSOUT=\* is allocated.

If the OUTDD ddname is not allocated and you are running in an OS/390 UNIX environment, the \_IGZ\_SYSOUT environment variable is used as follows:

#### Undefined or set to stdout

Output is routed to stdout (file descriptor 1).

### Set to stderr

Output is routed to stderr (file descriptor 2).

### Otherwise (set to something other than stdout or stderr)

The DISPLAY statement fails; a severity-3 Language Environment condition is raised.

When DISPLAY output is routed to stdout or stderr, the output is not subdivided into records; rather the output is written as a single stream of characters without line breaks.

If OUTDD and the Language Environment run-time option MSGFILE both specify the same ddname, DISPLAY output and Language Environment run-time diagnostics are both routed to the Language Environment message file.

## Using WITH NO ADVANCING

If you specify the WITH NO ADVANCING phrase and the output is going to a ddname, the printer control character + (plus) is placed into the first output position from the *next* DISPLAY statement. + is the ANSI-defined printer control character that suppresses line spacing before a record is printed.

If you specify the WITH NO ADVANCING phrase and the output is going to stdout or stderr, a new-line character is not appended to the end of the stream. A subsequent DISPLAY statement might add additional characters to the end of the stream.

If you do not specify WITH NO ADVANCING and the output is going to a ddname, the printer control character ' ' (space) is placed into the first output position from the next DISPLAY statement, indicating single-spaced output.

For example:

```
DISPLAY "ABC"  
DISPLAY "CDEF" WITH NO ADVANCING  
DISPLAY "GHIJK" WITH NO ADVANCING  
DISPLAY "LMNOPQ"  
DISPLAY "RSTVMX"
```

If you use the statements above, the result sent to the output device is:

```
ABC  
CDEF  
+GHIJK  
+LMNOPQ  
RSTVMX
```

The output printed depends on how the output device interprets printer control characters.

If you do not specify the WITH NO ADVANCING phrase and the output is going to stdout or stderr, a new-line character is appended to the end of the stream.

### RELATED TASKS

“Setting and accessing environment variables” on page 362  
“Coding COBOL programs to run under CICS” on page 347

### RELATED REFERENCES

DISPLAY statement (*IBM COBOL Language Reference*)  
“OUTDD” on page 286

---

## 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 67

“Converting data items (intrinsic functions)” on page 94

“Evaluating data items (intrinsic functions)” on page 96

---

## 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 53

---

## 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 34

"Controlling how numeric data is stored" on page 35

"Checking for incompatible data (numeric class test)" on page 42

"Performing arithmetic" on page 43

"Using currency signs" on page 51

---

## 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".
```

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

Other characters of special significance that you can code are:

- P** Indicates leading or trailing zeroes
- S** Indicates a sign, positive or negative
- V** Implies a decimal point

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

```
05 Price           Pic s99v99.
```

The field can therefore hold a positive or a negative value. The v indicates the position of an implied decimal point, but does not contribute to the size of the

item because it does not require a storage position. An s usually does not contribute to the size of a numeric item, because by default 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 38

#### RELATED CONCEPTS

“Appendix A. Intermediate results and arithmetic precision” on page 559

#### RELATED TASKS

“Displaying numeric data”

“Controlling how numeric data is stored” on page 35

“Performing arithmetic” on page 43

#### RELATED REFERENCES

“Sign representation and processing” on page 41

“ARITH” on page 262

“NUMPROC” on page 283

---

## 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 $zz,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 38

#### RELATED TASKS

“Controlling how numeric data is stored”

“Defining numeric data” on page 33

“Performing arithmetic” on page 43

#### 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 38

#### RELATED CONCEPTS

“Formats for numeric data”

“Data format conversions” on page 39

“Appendix A. Intermediate results and arithmetic precision” on page 559

#### RELATED TASKS

“Defining numeric data” on page 33

“Displaying numeric data” on page 34

“Performing arithmetic” on page 43

#### RELATED REFERENCES

“Conversions and precision” on page 40

“Sign representation and processing” on page 41

---

## 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. If you compile with the default option ARITH (COMPAT), external floating-point numbers are converted to long (64-bit) floating-point format. If you compile with ARITH (EXTEND), they are instead converted to extended-precision (128-bit) floating-point format.

## Binary (COMP) items

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

Binary format numbers occupy 2, 4, or 8 bytes of storage. If the picture clause specifies that the item is signed, the leftmost bit is used as the operational sign.

A binary number with a PICTURE description of four or fewer decimal digits occupies 2 bytes; five to nine decimal digits, 4 bytes; and 10 to 18 decimal digits, 8 bytes. Binary items with nine or more digits require more handling by the compiler. Testing them for the SIZE ERROR condition and rounding is more cumbersome than with other types.

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. However, unlike USAGE COMP items, they can contain values of magnitude up to the capacity of the native binary representation (2, 4, or 8 bytes) rather than being limited to the value implied by the number of 9s in the PICTURE clause. When you move or store numeric data into a COMP-5 item, truncation occurs at the binary field size rather than at the COBOL PICTURE size limit. When you reference a COMP-5 item, the full binary field size is used in the operation.

COMP-5 is thus particularly useful for binary data items 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).

## 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. The leftmost bit contains the sign and the next 7 bits contain the exponent; the remaining 3 or 7 bytes contain the mantissa.

On OS/390 & VM, COMP-1 and COMP-2 data items are stored in System/390 hexadecimal format.

### RELATED CONCEPTS

“Appendix A. Intermediate results and arithmetic precision” on page 559

### RELATED REFERENCE

“TRUNC” on page 297

---

## Examples: numeric data and internal representation

This table shows the internal representation of numeric items.

Numeric type	PICTURE and USAGE and optional SIGN clause	Value	Internal representation
External decimal	PIC S9999 DISPLAY	+ 1234	F1 F2 F3 C4
		- 1234	F1 F2 F3 D4
		1234	F1 F2 F3 C4
	PIC 9999 DISPLAY	1234	F1 F2 F3 F4
	PIC S9999 DISPLAY SIGN LEADING	+ 1234	C1 F2 F3 F4
		- 1234	D1 F2 F3 F4
	PIC S9999 DISPLAY SIGN LEADING SEPARATE PIC S9999 DISPLAY SIGN TRAILING SEPARATE	+ 1234	4E F1 F2 F3 F4
	- 1234	60 F1 F2 F3 F4	
	+ 1234	F1 F2 F3 F4 4E	
	- 1234	F1 F2 F3 F4 60	

Numeric type	PICTURE and USAGE and optional SIGN clause	Value	Internal representation
Binary	PIC S9999 BINARY COMP COMP-4	+ 1234 - 1234	04 D2 FB 2E
	COMP-5	+ 12345 <sup>1</sup> - 12345 <sup>1</sup>	30 39 CF C7
	PIC 9999 BINARY COMP COMP-4	1234	04 D2
	COMP-5	60000 <sup>1</sup>	EA 60
Internal decimal	PIC S9999 PACKED-DECIMAL COMP-3	+ 1234 - 1234	01 23 4C 01 23 4D
	PIC 9999 PACKED-DECIMAL COMP-3	+ 1234 - 1234	01 23 4F 01 23 4F
Internal floating point	COMP-1	+ 1234	43 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
<p>1. The example demonstrates that COMP-5 data items can contain values of magnitude up to the capacity of the native binary representation (2, 4, or 8 bytes), rather than being limited to the value implied by the number of 9s in the PICTURE clause.</p>			

## 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 or COMPUTE statement

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

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 41

## **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 nine digits, the fixed-point data item will receive only nine significant digits, and the remaining digits will be zero.

When a USAGE COMP-2 data item is moved to a fixed-point data item with more than 18 digits, the fixed-point data item will receive only 18 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 nine 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:**

C, A, E, and F

**Negative:**

D and B

The COBOL NUMPROC compiler option affects sign processing for external decimal and internal decimal data. NUMPROC has no effect on binary data or floating-point data.

### NUMPROC(PFD)

Given  $X'sd'$ , where  $s$  is the sign representation and  $d$  represents the digit, when you use NUMPROC(PFD), the compiler assumes that the sign in your data is one of three preferred signs:

**Signed positive or 0:**

$X'C'$

**Signed negative:**

$X'D'$

**Unsigned or alphanumeric:**

$X'F'$

Based on this assumption, the compiler uses whatever sign it is given to process data. The preferred sign is generated only where necessary (for example, when unsigned data is moved to signed data). Using the NUMPROC(PFD) option can save processing time, but you must use preferred signs with your data for correct processing.

### NUMPROC(NOPFD)

When the NUMPROC(NOPFD) compiler option is in effect, the compiler accepts any valid sign configuration. The preferred sign is always generated in the receiver. NUMPROC(NOPFD) is less efficient than NUMPROC(PFD), but you should use it whenever data that does not use preferred signs might exist.

If an unsigned, external-decimal sender is moved to an alphanumeric receiver, the sign is unchanged (even with NUMPROC(NOPFD)).

### NUMPROC(MIG)

When NUMPROC(MIG) is in effect, the compiler generates code that is similar to that produced by OS/VS COBOL. This option can be especially useful if you migrate OS/VS COBOL programs to COBOL for OS/390 & VM.

---

## 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. For example, a packed-decimal item is checked for hexadecimal values X'0' through X'9' in the digit positions, and for a valid sign value in the sign position (whether separate or nonseparate).

For external decimal, external floating-point, and packed-decimal items, the numeric class test is affected by the NUMPROC compiler option and the NUMCLS option (which is set at installation time). To determine the NUMCLS setting used at your installation, consult your system programmer.

If NUMCLS(PRIM) is in effect at your installation, use the following table to find the values that the compiler considers valid for the sign.

	NUMPROC (NOPFD)	NUMPROC (PFD)	NUMPROC (MIG)
<b>Signed</b>	C, D, F	C, D, +0 (positive zero)	C, D, F
<b>Unsigned</b>	F	F	F
<b>Separate sign</b>	+, -	+, -, +0 (positive zero)	+, -

If NUMCLS(ALT) is in effect at your installation, use the following table to find the values that the compiler considers valid for the sign.

	NUMPROC (NOPFD)	NUMPROC (PFD)	NUMPROC (MIG)
<b>Signed</b>	A to F	C, D, +0 (positive zero)	A to F
<b>Unsigned</b>	F	F	F
<b>Separate sign</b>	+, -	+, -, +0 (positive zero)	+, -

### RELATED REFERENCES

"NUMPROC" on page 283



---

## 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
- Language Environment callable services

### 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
**	Exponentiation	Second

Operator	Meaning	Order of evaluation
/ 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.

Many of the capabilities of numeric intrinsic functions are also provided by Language Environment callable services.

Numeric functions are classified into these categories:

### Integer

Those that return an integer

### Floating point

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

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

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

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

"Examples: numeric intrinsic functions" on page 47

## 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  $x = \text{Function Sum}(a \ b \ (c \ / \ d))$

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.

## Math and date Language Environment services

Many of the capabilities of COBOL intrinsic functions are also provided by Language Environment callable services. Language Environment callable services provide a means of assigning arithmetic results to data items. They include mathematical functions, and date and time operations.

### Math-oriented callable services

For most COBOL intrinsic functions there are corresponding math-oriented callable services you can use that produce the same results, as shown in the following table. When you compile with the default option ARITH(COMPAT), COBOL floating-point intrinsic functions return long (64-bit) results. When you compile with option ARITH(EXTEND), COBOL floating-point intrinsic functions (with the exception of RANDOM) return extended-precision (128-bit) results.

So for example (considering the first row of the table), if you compile using ARITH(COMPAT), CEESDACS returns the same result as ACOS. If you compile using ARITH(EXTEND), CEESQACS returns the same result as ACOS.

COBOL intrinsic function	Corresponding Language Environment callable services		Results same for intrinsic function and callable service?
	Long precision	Extended precision	
ACOS	CEESDACS	CEESQACS	Yes
ASIN	CEESDASN	CEESQASN	Yes
ATAN	CEESDATN	CEESQATN	Yes
COS	CEESDCOS	CEESQCOS	Yes
LOG	CEESDLOG	CEESQLOG	Yes
LOG10	CEESDLG1	CEESQLG1	Yes
RANDOM <sup>1</sup>	CEERAN0	none	No
REM	CEESDMOD	CEESQMOD	Yes
SIN	CEESDSIN	CEESQSIN	Yes
SQRT	CEESDSQT	CEESQSQT	Yes

TAN	CEESDTAN	CEESQTAN	Yes
1. RANDOM returns a long (64-bit) floating-point result even if you pass it a 31-digit argument and compile using option ARITH(EXTEND).			

Both the RANDOM intrinsic function and CEERAN0 service generate random numbers between zero and one. However, because each uses its own algorithm, RANDOM and CEERAN0 produce different random numbers from the same seed.

Even for functions that produce the same results, how you use intrinsic functions and Language Environment callable services differs. The rules for the data types required for intrinsic function arguments are less restrictive. For numeric intrinsic functions, you can use arguments that are of any numeric data type. When you invoke a Language Environment callable service with a CALL statement, however, you must ensure that the parameters match the numeric data types required by that service (generally COMP-1 or COMP-2).

The error handling of intrinsic functions and Language Environment callable services sometimes differs. If you pass an explicit feedback token when calling the Language Environment math services, you must check the feedback code after each call and take explicit action to deal with errors. However, if you call with the feedback token explicitly OMITTED, you do not need to check the token; Language Environment automatically signals any errors.

### Date callable services

Both the COBOL date intrinsic functions and the Language Environment date callable services are based on the Gregorian calendar. However, the starting dates can differ depending on the setting of the INTDATE compiler option. When the default setting of INTDATE(ANSI) is in effect, COBOL uses January 1, 1601 as day 1. When INTDATE(LILIAN) is in effect, COBOL uses October 15, 1582 as day 1. Language Environment always uses October 15, 1582 as day 1.

This means that if you use INTDATE(LILIAN), you get equivalent results from COBOL intrinsic functions and Language Environment callable date services.

The following table shows the results when INTDATE(ANSI) is in effect.

COBOL intrinsic function	Language Environment callable service	Results
INTEGER-OF-DATE	CEECBLDY	Compatible
DATE-OF-INTEGERS	CEEDATE with picture string YYYYMMDD	Incompatible
DAY-OF-INTEGERS	CEEDATE with picture string YYYYDDD	Incompatible
INTEGER-OF-DATE	CEEDAYS	Incompatible

The following table shows the results when INTDATE(LILIAN) is in effect.

COBOL intrinsic function	Language Environment callable service	Results
DATE-OF-INTEGERS	CEEDATE with picture string YYYYMMDD	Compatible
DAY-OF-INTEGERS	CEEDATE with picture string YYYYDDD	Compatible
INTEGER-OF-DATE	CEEDAYS	Compatible
INTEGER-OF-DATE	CEECBLDY	Incompatible

#### RELATED CONCEPTS

“Fixed-point versus floating-point arithmetic” on page 49

“Appendix A. Intermediate results and arithmetic precision” on page 559

#### RELATED TASKS

“Using Language Environment callable services” on page 553

#### RELATED REFERENCES

“ARITH” on page 262

---

## 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 568

## 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'9F' used as the currency sign:

```

    Currency Sign X'9F' 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'9F' or X'5A' depending on the code page in use, as shown in the following table.

Code page	Applicable countries	Modified from	Euro sign
IBM-1140	USA, Canada, Netherlands, Portugal, Australia, New Zealand	IBM-037	X'9F'
IBM-1141	Austria, Germany	IBM-273	X'9F'
IBM-1142	Denmark, Norway	IBM-277	X'5A'

Code page	Applicable countries	Modified from	Euro sign
IBM-1143	Finland, Sweden	IBM-278	X'5A'
IBM-1144	Italy	IBM-280	X'9F'
IBM-1145	Spain, Latin America - Spanish	IBM-284	X'9F'
IBM-1146	UK	IBM-285	X'9F'
IBM-1147	France	IBM-297	X'9F'
IBM-1148	Belgium, Canada, Switzerland	IBM-500	X'9F'
IBM-1149	Iceland	IBM-871	X'9F'

“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:

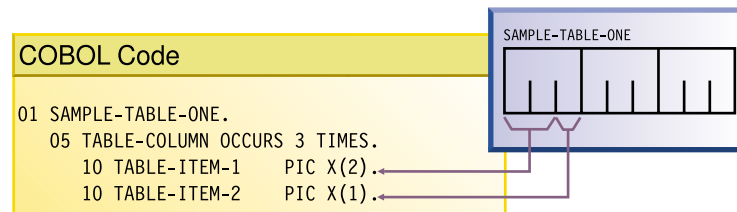
```
Deposit in Euro = EUR 8000.00
Deposit in Franc = FRF 54312.08
```

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 55

“Putting values into a table” on page 58

“Nesting tables” on page 54

“Creating variable-length tables (`DEPENDING ON`)” on page 62

“Searching a table” on page 65

“Processing table items using intrinsic functions” on page 67

“Handling tables efficiently” on page 539

---

## 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 (DEPENDING ON)” on page 62
- “Nesting tables”
- “Putting values into a table” on page 58
- “Referring to an item in a table” on page 55
- “Searching a table” on page 65

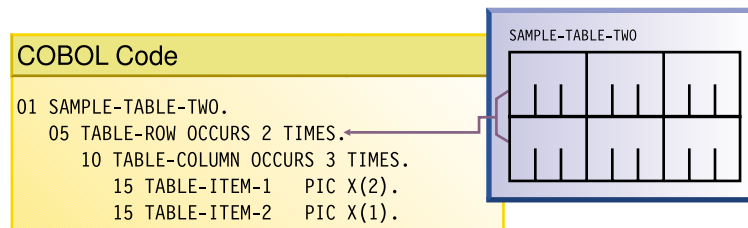
RELATED REFERENCES

OCCURS clause (*IBM COBOL Language Reference*)

---

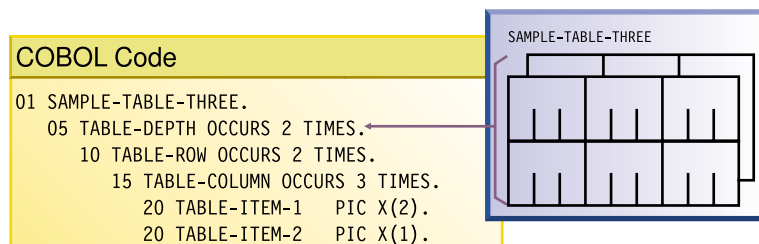
## Nesting tables

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 53

“Referring to an item in a table”

“Putting values into a table” on page 58

“Creating variable-length tables (DEPENDING ON)” on page 62

“Searching a table” on page 65

“Processing table items using intrinsic functions” on page 67

“Handling tables efficiently” on page 539

### 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 indenames.
- Use both subscripts and indexes together.

**RELATED TASKS**

“Indexing” on page 57  
 “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.

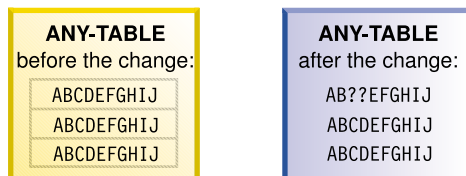
You can increment or decrement a literal or variable subscript by a specified integer amount. For example:

```
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:



**RELATED TASKS**

“Indexing” on page 57

“Putting values into a table” on page 58  
“Searching a table” on page 65  
“Handling tables efficiently” on page 539

## 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 56  
“Putting values into a table” on page 58

“Searching a table” on page 65  
“Processing table items using intrinsic functions” on page 67  
“Handling tables efficiently” on page 539

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 63  
“Initializing a table (INITIALIZE)”  
“Assigning values when you define a table (VALUE)”  
“Assigning values to a variable-length table” on page 64

## 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 60  
“Example: PERFORM and indexing” on page 61

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-OBJ 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-On
      Move Space To Error-Flag (Flag-Index)
      Set Message-Index To Flag-Index
      Move Error-Message (Message-Index) To
        Print-Message
      Perform 260-Print-Report
    End-Search
  End-Perform
. . .
```

---

## Creating variable-length tables (DEPENDING ON)

If you do not know before run time how many times a table element occurs, 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.

This behavior is different if the CMPR2 compiler option is in effect.

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
   RECORDING MODE F
   BLOCK 0 RECORDS
   RECORD 80 CHARACTERS
   LABEL RECORD STANDARD.
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 B. Complex OCCURS DEPENDING ON" on page 569

#### RELATED TASKS

"Assigning values to a variable-length table" on page 64

"Loading a variable-length table"

*COBOL for OS/390 & VM Compiler and Run-Time Migration Guide*

#### RELATED REFERENCES

OCCURS DEPENDING ON clause (*IBM COBOL Language Reference*)

"CMR2" on page 264

## 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
   RECORDING MODE F
   BLOCK 0 RECORDS
   RECORD 80 CHARACTERS
   LABEL RECORD STANDARD.
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).

This behavior is different if the CMPR2 compiler option is in effect.

### RELATED TASKS

*COBOL for OS/390 & VM Compiler and Run-Time Migration Guide*

RELATED REFERENCES

OCCURS DEPENDING ON clause (*IBM COBOL Language Reference*)  
“CMPR2” on page 264

---

## 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 66

## 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” on page 66

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” on page 67

#### 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” on page 68

#### RELATED TASKS

“Using intrinsic functions (built-in functions)” on page 31

#### 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.
. . .
Procedure Division.
   Compute Max-Salary = Function Max(Emp-Salary(ALL))
   Compute I = Function Ord-Max(Emp-Salary(ALL))
   Compute Avg-Salary = Function Mean(Emp-Salary(ALL))
   Compute Salary-Range = Function Range(Emp-Salary(ALL))
   Compute Total-Payroll = Function Sum(Emp-Salary(ALL))
```

---

## Chapter 5. Selecting and repeating program actions

Use COBOL control language to choose program actions based on the outcome of logical tests, to iterate over selected parts of your program and data, and to identify statements to be performed as a group. These controls include:

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

### RELATED TASKS

“Selecting program actions”

“Repeating program actions” on page 76

---

### 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 73

### 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 72

“Example: EVALUATE using multiple WHEN statements” on page 72

“Example: EVALUATE testing several conditions” on page 72

#### RELATED TASKS

“Coding conditional expressions” on page 73

“Using the EVALUATE statement” on page 71

“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.

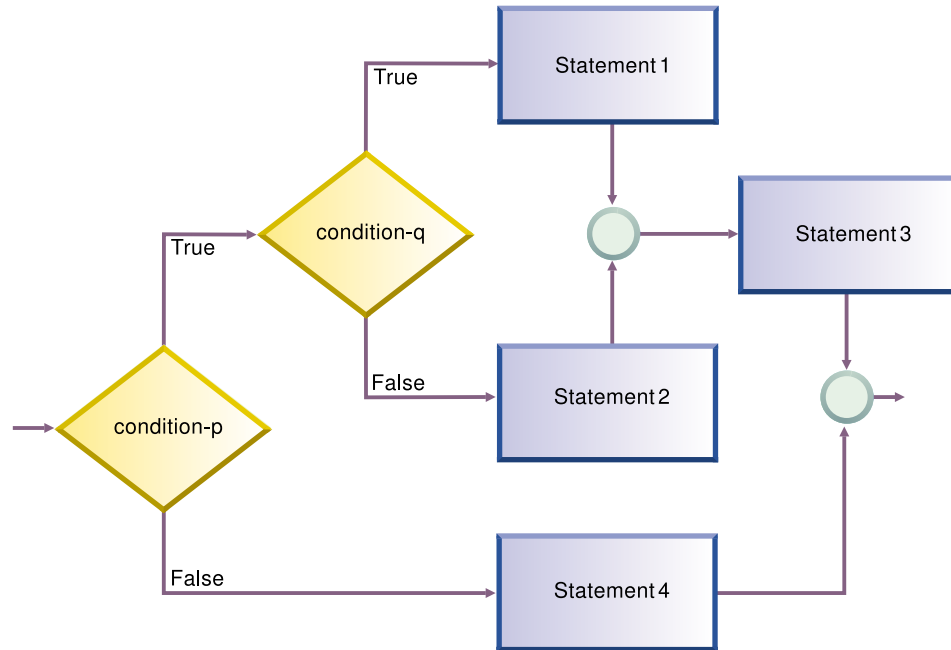
Use nested IF statements sparingly. The logic can be difficult to follow, although explicit scope terminators and proper indentation help.

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 69

**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 69

**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.
Source-Computer. IBM-390.
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. COBOL lets you 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 74

### RELATED TASKS

“Defining switches and flags” on page 74

“Resetting switches and flags” on page 75

“Checking for incompatible data (numeric class test)” on page 42

#### 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*)

### 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 75

### 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 75

### 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 76

### 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”

“Coding a loop” on page 77

“Coding a loop through a table” on page 78

“Executing multiple paragraphs or sections” on page 79

#### 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:

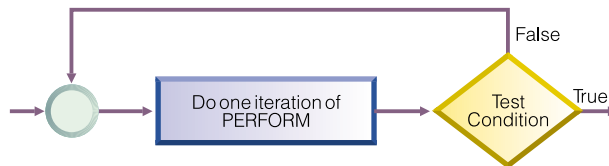
```
PERFORM 010-PROCESS-ONE-MONTH 12 TIMES
INSPECT . . .
```

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 . . .
```

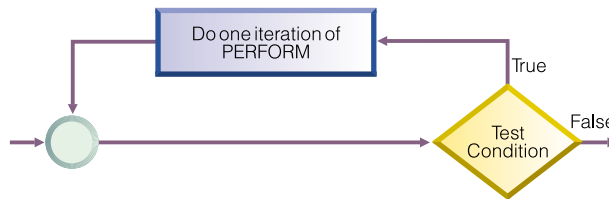
Use the PERFORM . . . WITH TEST AFTER . . . UNTIL if you want to execute the paragraph at least once and then test before any subsequent execution. This statement is equivalent to the do-until structure:



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 67



---

## 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 83

“Manipulating null-terminated strings” on page 85

“Referring to substrings of data items” on page 86

“Tallying and replacing data items (INSPECT)” on page 89

“Converting double-byte character set (DBCS) data” on page 90

“Converting data items (intrinsic functions)” on page 94

“Evaluating data items (intrinsic functions)” on page 96

---

### 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)

“Example: STRING statement”

### RELATED TASKS

“Handling errors in joining and splitting strings” on page 190

### 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 ◊ indicates a blank space):

```

J.B.◊SMITH◊◊◊◊◊◊
444◊SPRING◊ST.,◊CHICAGO,◊ILL.◊◊◊◊◊◊
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

**CMPR2 consideration:** The UNSTRING statement works differently when the CMPR2 compiler option is in effect. For VS COBOL II Release 2 compatibility and migration details, see the references indicated below.

“Example: UNSTRING statement”

### RELATED TASKS

“Handling errors in joining and splitting strings” on page 190  
*COBOL for OS/390 & VM Compiler and Run-Time Migration Guide*

### RELATED REFERENCES

“CMPR2” on page 264  
 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.

---

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

---

### 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-DOLS. 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, in parentheses immediately following the data item you code two values separated by a colon:

- Ordinal position (from the left) of the character 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 55

**RELATED REFERENCES**

“SSRANGE” on page 294

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 86

## 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
- Replace portions of a data item with a corresponding portion of another data item
- Count the number of times a specific character (zero, space, or asterisk, for example) occurs in a data item

“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 double-byte character set (DBCS) data

If you use byte-oriented nonnumeric operations (for example, STRING and UNSTRING) on nonnumeric data items containing double-byte characters, results are unpredictable. You should instead convert such items to pure DBCS data using the Language Environment IGZCA2D service routine.

After you have processed the DBCS data, use the Language Environment IGZCD2A service routine to convert the DBCS data items back to nonnumeric data items containing double-byte characters.

The DBCS compiler option does not affect the operation of these service routines.

### RELATED REFERENCES

"DBCS notation"

"Nonnumeric to DBCS data conversion (IGZCA2D)"

"DBCS to nonnumeric data conversion (IGZCD2A)" on page 92

"DBCS" on page 269

## DBCS notation

These symbols describe DBCS items:

Symbols	Meaning
< and >	Shift-out (SO) and Shift-in (SI), respectively
D0, D1, D2, . . . , Dn	Any DBCS character except for double-byte EBCDIC characters
.A, .B, .C, . . .	Any double-byte EBCDIC character; the period (.) represents the value X'42'
A single letter, such as A, B, or s	Any single-byte EBCDIC character

## Nonnumeric to DBCS data conversion (IGZCA2D)

The Language Environment IGZCA2D service routine converts nonnumeric data that contains double-byte characters to pure DBCS data.



## IGZCA2D syntax

To use the IGZCA2D service routine, pass the following four parameters to the routine using the CALL statement:

### parameter-1

The sending field for the conversion, handled as a nonnumeric data item.

### parameter-2

The receiving field for the conversion, handled as a DBCS data item.

You cannot use reference modification with *parameter-2*.

### parameter-3

The number of bytes in *parameter-1* to be converted.

It can be the LENGTH OF special register of *parameter-1*, or a 4-byte USAGE IS BINARY data item containing the number of bytes of *parameter-1* to be converted. Shift codes count as 1 byte each.

### parameter-4

The number of bytes in *parameter-2* that will receive the converted data.

It can be the LENGTH OF special register of *parameter-2*, or a 4-byte USAGE IS BINARY data item containing the number of bytes of *parameter-2* to receive the converted data.

## Usage notes

- You can pass *parameter-1*, *parameter-3*, and *parameter-4* to the routine BY REFERENCE or BY CONTENT, but you must pass *parameter-2* BY REFERENCE.
- The compiler does not perform syntax checking on these parameters. Ensure that the parameters are correctly set and passed to the conversion routine using the CALL statement. Otherwise, results are unpredictable.
- When creating *parameter-2* from *parameter-1*, IGZCA2D makes these changes:
  - Removes the shift codes, leaving the DBCS data unchanged
  - Converts the single-byte EBCDIC data to double-byte EBCDIC characters
  - Converts EBCDIC space (X'40') to DBCS space (X'4040'), instead of X'4240'
- IGZCA2D does not change the contents of *parameter-1*, *parameter-3*, or *parameter-4*.
- The valid range for the contents of *parameter-3* and *parameter-4* is 1 to 16,777,215.

"Example: IGZCA2D" on page 92

## RELATED REFERENCES

"IGZCA2D return codes"

## IGZCA2D return codes

IGZCA2D sets the RETURN-CODE special register to reflect the status of the conversion, as shown in the table below.

Return code	Explanation
0	Parameter-1 was converted and the results were placed in parameter-2.
2	Parameter-1 was converted and the results were placed in parameter-2. Parameter-2 was padded on the right with DBCS spaces.
4	Parameter-1 was converted and the results were placed in parameter-2. The DBCS data placed in parameter-2 was truncated on the right.

Return code	Explanation
6	Parameter-1 was converted and the results were placed in parameter-2. An EBCDIC character in the range X'00' to X'3F' or X'FF' was encountered. The valid EBCDIC character has been converted into an out-of-range DBCS character.
8	Parameter-1 was converted and the results were placed in parameter-2. An EBCDIC character in the range X'00' to X'3F' or X'FF' was encountered. The valid EBCDIC character has been converted into an out-of-range DBCS character.  Parameter-2 was padded on the right with DBCS spaces.
10	Parameter-1 was converted and the results were placed in parameter-2. An EBCDIC character in the range X'00' to X'3F' or X'FF' was encountered. The valid EBCDIC character has been converted into an out-of-range DBCS character.  The DBCS data in parameter-2 was truncated on the right.
12	An odd number of bytes was found between paired shift codes in parameter-1. No conversion occurred.
13	Unpaired or nested shift codes were found in parameter-1. No conversion occurred.
14	Parameter-1 and parameter-2 were overlapping. No conversion occurred.
15	The value provided for parameter-3 or parameter-4 was out of range. No conversion occurred.
16	An odd number of bytes was coded in parameter-4. No conversion occurred.

### Example: IGZCA2D

The following CALL statement converts the nonnumeric data in alpha-item to DBCS data. The results of the conversion are placed in dbc-item.

```
CALL "IGZCA2D" USING BY REFERENCE alpha-item dbc-item
      BY CONTENT LENGTH OF alpha-item LENGTH OF dbc-item
```

Suppose the contents of alpha-item and dbc-item and the lengths before the conversion are:

```
alpha-item = AB<D1D2D3>CD
dbc-item   = D4D5D6D7D8D9D0
LENGTH OF alpha-item = 12
LENGTH OF dbc-item   = 14
```

Then after the conversion, alpha-item and dbc-item will contain:

```
alpha-item = AB<D1D2D3>CD
dbc-item   = .A.BD1D2D3.C.D
```

The content of the RETURN-CODE register is 0.

#### RELATED REFERENCES

"DBCS notation" on page 90

## DBCS to nonnumeric data conversion (IGZCD2A)

The Language Environment IGZCD2A routine converts pure DBCS data to nonnumeric data that can contain double-byte characters.

## IGZCD2A syntax

To use the IGZCD2A service routine, pass the following four parameters to the routine using the CALL statement:

### parameter-1

The sending field for the conversion, handled as a DBCS data item.

### parameter-2

The receiving field for the conversion, handled as a nonnumeric data item.

### parameter-3

The number of bytes in *parameter-1* to be converted.

It can be the LENGTH OF special register of *parameter-1*, or a 4-byte USAGE IS BINARY data item containing the number of bytes of *parameter-1* to be converted.

### parameter-4

The number of bytes in *parameter-2* that will receive the converted data.

It can be the LENGTH OF special register of *parameter-2*, or a 4-byte USAGE IS BINARY data item containing the number of bytes of *parameter-2* to receive the converted data. Shift codes count as 1 byte each.

## Usage notes

- You can pass *parameter-1*, *parameter-3*, and *parameter-4* to the routine BY REFERENCE or BY CONTENT, but you must pass *parameter-2* BY REFERENCE.
- The compiler does not perform syntax checking on these parameters. Ensure that the parameters are correctly set and passed to the conversion routine. Otherwise, results are unpredictable.
- When creating *parameter-2* from *parameter-1*, IGZCD2A makes these changes:
  - Inserts shift codes around DBCS characters that are not double-byte EBCDIC characters
  - Converts double-byte EBCDIC characters to single-byte EBCDIC characters
  - Converts the DBCS space (X'4040') to an EBCDIC space (X'40')
- IGZCD2A does not change the contents of *parameter-1*, *parameter-3*, or *parameter-4*.
- If the converted data contains double-byte characters, shift codes are counted in the length of *parameter-2*.
- The valid range for the contents of *parameter-3* and *parameter-4* is 1 to 16,777,215.

“Example: IGZCD2A” on page 94

## RELATED REFERENCES

“IGZCD2A return codes”

## IGZCD2A return codes

IGZCD2A sets the RETURN-CODE special register to reflect the status of the conversion, as shown in the table below.

Return code	Explanation
0	Parameter-1 was converted and the results were placed in parameter-2.
2	Parameter-1 was converted and the results were placed in parameter-2. Parameter-2 was padded on the right with EBCDIC spaces.
4	Parameter-1 was converted and the results were placed in parameter-2. Parameter-2 was truncated on the right. <sup>1</sup>

Return code	Explanation
14	Parameter-1 and parameter-2 were overlapping. No conversion occurred.
15	The value of parameter-3 or parameter-4 was out of range. No conversion occurred.
16	An odd number of bytes was coded in parameter-3. No conversion occurred.
1. If a truncation occurs within the DBCS characters, the truncation is on an even-byte boundary and a shift-in (SI) is inserted. If necessary, the nonnumeric data is padded with an EBCDIC space after the shift-in.	

### Example: IGZCD2A

The following CALL statement converts the DBCS data in `dbcs-item` to nonnumeric data with double-byte characters. The results of the conversion are placed in `alpha-item`.

```
CALL "IGZCD2A" USING BY REFERENCE dbc-item alpha-item
BY CONTENT LENGTH OF dbc-item LENGTH OF alpha-item
```

Suppose the contents of `dbcs-item` and `alpha-item` and the lengths before the conversion are:

```
dbcs-item = .A.BD1D2D3.C.D
alpha-item = ssssssssssss
LENGTH OF dbc-item = 14
LENGTH OF alpha-item = 12
```

Then after the conversion, `dbcs-item` and `alpha-item` will contain:

```
dbcs-item = .A.BD1D2D3.C.D
alpha-item = AB<D1D2D3>CD
```

The content of the RETURN-CODE register is 0.

#### RELATED REFERENCES

"DBCS notation" on page 90

---

## Converting data items (intrinsic functions)

Intrinsic functions are available 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 89

## 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.

## 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, the value after the statement is performed is NOSNH0J, where 0 represents a blank space.

## 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 when you compile with the default option ARITH(COMPAT) (*compatibility mode*) nor 31 digits when you compile with ARITH(EXTEND) (*extended mode*), not including the editing symbols.

NUMVAL and NUMVAL-C return long (64-bit) floating-point values in compatibility mode, and return extended-precision (128-bit) floating-point values in extended mode. 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 $999V99.
. . .
  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 36

#### RELATED TASKS

“Assigning input from a screen or file (ACCEPT)” on page 28

“Displaying values on a screen or in a file (DISPLAY)” on page 29

#### RELATED REFERENCES

NUMVAL (*IBM COBOL Language Reference*)

NUMVAL-C (*IBM COBOL Language Reference*)

“ARITH” on page 262

## 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” on page 97

“Finding the length of data items” on page 98

“Finding the date of compilation” on page 99

## 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 VALLEJ0 to the first 10 character positions of Customer-record, where 0 represents a blank space:

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

If you use MIN instead, then THOMAS is assigned.

### 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).

### 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 0 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 g○○○○○○○○ 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 43

“Processing table items using intrinsic functions” on page 67

#### 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 43

“Processing table items using intrinsic functions” on page 67

#### 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 and writing data is an essential part of every program. Your program retrieves information, processes it as you request, and then produces the results.

The source of the information and the target for the results can be one or more of the following:

- Another program
- Direct-access storage device
- Magnetic tape
- Printer
- Terminal
- Card reader or punch

The information as it exists on an external device is in a physical record or block, a collection of information that is handled as a unit by the system during input or output operations.

Your COBOL program does not directly handle physical records. It processes logical records. A logical record can correspond to a complete physical record, part of a physical record, or to parts or all of one or more physical records. Your COBOL program handles logical records exactly as you have defined them.

In COBOL, a collection of logical records is a file, a sequence of pieces of information that your program can process.

### RELATED CONCEPTS

“File organization and input-output devices”

### RELATED TASKS

“Choosing file organization and access mode” on page 103

“Allocating files” on page 105

“Checking for input or output errors” on page 106

---

## File organization and input-output devices

Depending on the input-output devices, your file organization can be sequential, line sequential, indexed, or relative. Decide on the file types and devices to be used when you design your program.

You have the following choices of file organization:

### Sequential file organization

The chronological order in which records are entered when a file is created establishes the arrangement of the records. Each record except the first has a unique predecessor record, and each record except the last has a unique successor record. Once established, these relationships do not change.

The access (record transmission) mode allowed for sequential files is sequential only.

### **Line-sequential file organization**

Line-sequential files are sequential files that reside on the hierarchical file system (HFS) and that contain only characters as data. Each record ends with a new-line character.

The only access (record transmission) mode allowed for line-sequential files is sequential.

### **Indexed file organization**

Each record in the file contains a special field whose contents form the record key. The position of the key is the same in each record. The index component of the file establishes the logical arrangement of the file, an ordering by record key. The actual physical arrangement of the records in the file is not significant to your COBOL program.

An indexed file can also use alternate indexes in addition to the record key. These keys let you access the file using a different logical ordering of the records.

The access (record transmission) modes allowed for indexed files are sequential, random, or dynamic. When you read or write indexed files sequentially, the sequence is that of the key values.

### **Relative file organization**

Records in the file are identified by their location relative to the beginning of the file. 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 on.

The access (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.

With COBOL for OS/390 & VM, requests to the operating system for the storage and retrieval of records from input-output devices are handled by the two access methods QSAM and VSAM, and the OS/390 UNIX file system.

The device type upon which you elect to store your data could affect the choices of file organization available to you. Direct-access storage devices provide greater flexibility in the file organization options. Sequential-only devices limit organization options but have other characteristics, such as the portability of tapes, that might be useful.

### **Sequential-only devices**

Terminals, printers, card readers, and punches are called *unit-record devices* because they process one line at a time. Therefore, you must also process records one at a time sequentially in your program when it reads from or writes to unit-record devices.

On tape, records are ordered sequentially, so your program must process them sequentially. Use QSAM physical sequential files when processing tape files. The records on tape can be fixed length or variable length. The rate of data transfer is faster than it is for cards.

### **Direct-access storage devices**

Direct-access storage devices hold many records. The record arrangement of files stored on these devices determines the ways that your program can process the data. When using direct-access devices, you have greater flexibility within your program, because you can use several types of file organization:

- Sequential (VSAM or QSAM)

- Line sequential (OS/390 UNIX)
- Indexed (VSAM)
- Relative (VSAM)

**RELATED TASKS**

- “Allocating files” on page 105
- “Chapter 8. Processing QSAM files” on page 107
- “Chapter 9. Processing VSAM files” on page 135
- “Chapter 10. Processing line-sequential files” on page 163
- “Choosing file organization and access mode”

---

## Choosing file organization and access mode

Consider the following guidelines when determining which file organization and access mode to use in your application:

- If your application accesses records (fixed-length or variable-length records) only sequentially and does not insert records between existing ones, a QSAM or VSAM sequential file is the simplest type.
- If you are developing an application for OS/390 UNIX that sequentially accesses records that contain only printable characters and certain control characters, line-sequential files work the best.
- If your application requires both sequential and random access (fixed-length or variable-length records), a VSAM indexed file is the most flexible type.
- If your application randomly inserts and deletes records, a relative file works well.

If a large percentage of a file is referenced or updated in your application, sequential processing is faster than random or dynamic access. If a small percentage of records is processed during each run of your application, use random or dynamic access.

The following table shows the possible file organizations, access modes, and record formats for COBOL files.

File organization	Sequential access	Random access	Dynamic access	Fixed-length	Variable-length
QSAM (physical sequential)	X			X	X
Line sequential	X			X <sup>1</sup>	X
VSAM sequential (ESDS)	X			X	X
VSAM indexed (KSDS)	X	X	X	X	X
VSAM relative (RRDS)	X	X	X	X	X
1. The data itself is in variable format but can be read into and written from COBOL fixed-length records.					

**RELATED REFERENCES**

- “Format for coding input and output”
- “Allowable control characters” on page 164

## Format for coding input and output

The following code shows the general format of input-output coding. Explanations of the user-supplied information 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 PICTURE 99.
. . .
PROCEDURE DIVISION.
. . .
    OPEN iomode filename (9)
. . .
    READ filename
. . .
    WRITE recordname
. . .
    CLOSE filename
. . .
STOP RUN.

```

The user-supplied information in the code above is as follows:

**(1) *filename***

Any legal COBOL name. You must use the same file name on the SELECT and the FD statements, and on the READ, OPEN, and CLOSE statements. In addition, the file name is required if you use the START or DELETE statements. This name is not necessarily the actual name of the data set as known to the system. Each file requires its own SELECT, FD, and input-output statements.

**(2) *assignment-name***

Any name you choose, provided that it follows COBOL and system naming rules. The name can be 1-30 characters long if it is a user-defined word, or 1-160 characters long if it is a literal. You code the *name* part of the *assignment-name* on a DD statement, in an ALLOCATE command (TSO), in a FILEDEF command (CMS), or as an environment variable (for example, in an export command) (OS/390 UNIX).

**(3) *org*** The organization can be SEQUENTIAL, LINE SEQUENTIAL, INDEXED, or RELATIVE. This clause is optional for QSAM files.

**(4) *access***

The access mode can be SEQUENTIAL, RANDOM, or DYNAMIC. For sequential file processing, including line-sequential, you can omit this clause.

**(5) *file-status***

The two-character COBOL FILE STATUS key.

**(6) *recordname***

The name of the record used in the WRITE and REWRITE statements.

**(7) *fieldlength***

The logical length of the field.

(8) *type*

The record format of the file. If you break the record entry beyond the level-01 description, each element should map accurately against the fields in the record.

(9) *iomode*

The INPUT or OUTPUT mode. If you are only reading from a file, code INPUT. If you are only writing to it, code OUTPUT or EXTEND. If you are both reading and writing, code I-O, except for organization LINE SEQUENTIAL.

RELATED TASKS

“Chapter 8. Processing QSAM files” on page 107

“Chapter 9. Processing VSAM files” on page 135

“Chapter 10. Processing line-sequential files” on page 163

---

## Allocating files

For any type of file (sequential, line sequential, indexed, or relative) in your OS/390 or OS/390 UNIX applications, you can define the external name with either a ddname or an environment variable name. The external name is the name in the *assignment-name* of the ASSIGN clause.

If the file is in the HFS, you can use either a DD definition or an environment variable to define the file by specifying its path name with the PATH keyword.

The environment variable name must be uppercase. The allowable attributes for its value depend on the organization of the file being defined.

Because you can define the external name in either of two ways, the COBOL run time goes through the following steps to find the definition of the file:

1. If the ddname is explicitly allocated, it is used. The definition can be from a DD statement in JCL, an ALLOCATE command from TSO/E, or a user-initiated dynamic allocation.
2. If the ddname is not explicitly allocated and an environment variable of the same name is set, the value of the environment variable is used.

The file is dynamically allocated using the attributes specified by the environment variable. At a minimum, you must specify either the PATH() or DSN() option. All options and attributes must be in uppercase, except for the *path-name* suboption of the PATH option, which is case sensitive. You cannot specify a temporary data set name in the DSN() option.

File status code 98 results from any of the following:

- The contents (including a value of null or all blanks) of the environment variable are not valid.
- The dynamic allocation of the file fails.
- The dynamic deallocation of the file fails.

The COBOL run time checks the contents of the environment variable at each OPEN statement. If a file with the same external name was dynamically allocated by a previous OPEN statement and the contents of the environment variable have changed since that OPEN, the run time dynamically deallocates the previous allocation and reallocates the file using the options currently set in the environment variable. If the contents of the environment variable have not changed, the run time uses the current allocation.

3. If neither a ddname nor an environment variable is defined, the following occurs:
  - If the allocation is for a QSAM file and the CBLQDA run-time option is in effect, CBLQDA dynamic allocation processing takes place for those eligible files. This type of “implicit” dynamic allocation persists for the life of the run unit and cannot be reallocated.
  - Otherwise the allocation fails.

The COBOL run time follows these steps for both OS/390 and OS/390 UNIX applications that were compiled with the NOCMR2 compiler option. Dynamic allocation of files using an environment variable is not available to programs compiled with the CMR2 compiler option or programs run under CMS.

The COBOL run time deallocates all dynamic allocations at run-unit termination, except “implicit” CBLQDA allocations.

#### RELATED TASKS

“Setting and accessing environment variables” on page 362  
“Defining and allocating QSAM files” on page 121  
“Dynamically creating QSAM files with CBLQDA” on page 118  
“Allocating VSAM files” on page 156

#### RELATED REFERENCES

“CMR2” on page 264

---

## Checking for input or output errors

After each input or output statement is performed, the FILE STATUS key is updated with a value that indicates the success or failure of the operation. Using a FILE STATUS clause, test the FILE STATUS key after each input or output statement, and call an error-handling procedure if a nonzero file status code is returned.

With VSAM files, you can use a second *data-name* in the FILE STATUS clause to get additional VSAM return code information.

Another way of handling errors in input and output operations is to code ERROR (synonymous with EXCEPTION) declaratives, as explained in the references below.

#### RELATED TASKS

“Handling errors in input and output operations” on page 191  
“Coding ERROR declaratives” on page 194  
“Using file status keys” on page 194



---

## Chapter 8. Processing QSAM files

Queued sequential access method (QSAM) files are unkeyed files in which the records are placed one after another, according to entry order. Your program can process these files only sequentially, retrieving (with the READ statement) records in the same order as they are in the file. Each record is placed after the preceding record.

To process QSAM files in your program, use COBOL language statements that:

- Identify and describe the QSAM files in the ENVIRONMENT DIVISION and the DATA DIVISION.
- Process the records in these files in the PROCEDURE DIVISION.

After you have created a record, you cannot change its length or its position in the file, and you cannot delete it. You can, however, update QSAM files on direct-access storage devices (using REWRITE), though not in the HFS.

QSAM files can be on tape, direct-access storage devices (DASDs), unit-record devices, and terminals. QSAM processing is best for tables and intermediate storage.

You can also access byte-stream files in the HFS using QSAM. These files are binary byte-oriented sequential files with no record structure. The record definitions that you code in your COBOL program and the length of the variables that you read into and write from determine the amount of data transferred.

### RELATED CONCEPTS

“Labels for QSAM files” on page 129

*OS/390 DFSMS: Using Data Sets* (how QSAM files are organized, and how the system processes them)

### RELATED TASKS

“Defining QSAM files and records in COBOL”

“Coding input and output statements for QSAM files” on page 117

“Handling errors in QSAM files” on page 120

“Working with QSAM files under OS/390” on page 121

“Identifying QSAM files to CMS” on page 128

“Processing QSAM ASCII files on tape” on page 132

“Processing ASCII file labels” on page 133

---

## Defining QSAM files and records in COBOL

Use the FILE-CONTROL entry to:

- Define the files in your COBOL program as QSAM files.
- Associate the files with the external file names (ddnames or environment variable names). (An *external file name* is the name by which a file is known to the operating system.)

In the following example, COMMUTER-FILE-MST is your program’s name for the file; COMMUTR is the external name.

```

FILE-CONTROL.
  SELECT COMMUTER-FILE-MST
  ASSIGN TO S-COMMUTR
  ORGANIZATION IS SEQUENTIAL
  ACCESS MODE IS SEQUENTIAL.

```

Your ASSIGN clause *name* can include an S- before the external name (ddname or environment variable name) to document that the file is a QSAM file.

Both the ORGANIZATION and ACCESS MODE clauses are optional.

#### RELATED TASKS

“Establishing record formats”

“Setting block sizes” on page 115

## Establishing record formats

In the FD entry in the DATA DIVISION, code the record format and whether the records are blocked. In the associated record description entry or entries, define the *record-name* and record length.

You can code a record format of F, V, S, or U in the RECORDING MODE clause. COBOL determines the record format from the RECORD clause or from the record descriptions associated with your FD entry for the file. If you want the records to be blocked, code the BLOCK CONTAINS clause in your FD entry.

The following example shows how the FD entry might look for a file with fixed-length records:

```

FILE SECTION.
FD  COMMUTER-FILE-MST
   RECORDING MODE IS F
   BLOCK CONTAINS 0 RECORDS
   RECORD CONTAINS 80 CHARACTERS.
01  COMMUTER-RECORD-MST.
   05  COMMUTER-NUMBER          PIC  X(16).
   05  COMMUTER-DESCRIPTION    PIC  X(64).

```

A recording mode of S is not supported for files in the HFS. The above example is appropriate for such a file.

#### RELATED CONCEPTS

“Logical records”

#### RELATED TASKS

“Requesting fixed-length format” on page 109

“Requesting variable-length format” on page 110

“Requesting spanned format” on page 112

“Requesting undefined format” on page 114

“Defining QSAM files and records in COBOL” on page 107

## Logical records

The term *logical record* is used in a slightly different way in the COBOL language and in OS/390 QSAM. For format-V and format-S files, the QSAM logical record includes a 4-byte prefix in front of the user data portion of the record that is not included in the definition of a COBOL logical record. For format-F and format-U files, and for HFS byte-stream files, the definitions of QSAM logical record and COBOL logical record are identical.

In this information, *QSAM logical record* refers to the QSAM definition, and *logical record* refers to the COBOL definition.

#### RELATED REFERENCES

- “Layout of format-F records”
- “Layout of format-V records” on page 111
- “Layout of format-S records” on page 113
- “Layout of format-U records” on page 114

### Requesting fixed-length format

Fixed-length records are in format F. Use RECORDING MODE F to explicitly request this format.

You can omit the RECORDING MODE clause. The compiler determines the recording mode to be F if the length of the largest level-01 record associated with the file is not greater than the block size coded in the BLOCK CONTAINS clause, and you take one of the following actions:

- Use the RECORD CONTAINS *integer* clause (RECORD clause format 1).  
When you use this clause, the file is always fixed format with record length *integer*, even if there are multiple level-01 record description entries with different lengths associated with the file.
- Omit the RECORD CONTAINS *integer* clause, but code the same fixed size and no OCCURS DEPENDING ON clause for all level-01 record description entries associated with the file. This fixed size is the record length.

In an unblocked format-F file, the logical record is the same as the block.

In a blocked format-F file, the number of logical records in a block (the blocking factor) is constant for every block in the file, except the last block, which might be shorter. Files in the HFS are never blocked.

#### RELATED CONCEPTS

“Logical records” on page 108

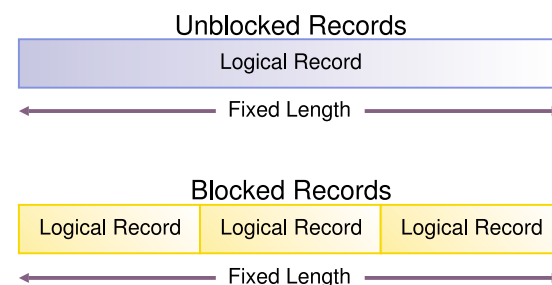
#### RELATED TASKS

- “Requesting variable-length format” on page 110
- “Requesting spanned format” on page 112
- “Requesting undefined format” on page 114
- “Establishing record formats” on page 108

#### RELATED REFERENCES

“Layout of format-F records”

**Layout of format-F records:** The layout of format-F QSAM records is shown below.



#### RELATED CONCEPTS

“Logical records” on page 108

#### RELATED TASKS

“Requesting fixed-length format” on page 109

Fixed-length record formats (*OS/390 DFSMS: Using Data Sets*)

#### RELATED REFERENCES

“Layout of format-V records” on page 111

“Layout of format-S records” on page 113

“Layout of format-U records” on page 114

### Requesting variable-length format

Variable-length records can be in format V or format D. Format-D records are variable-length records on ASCII tape files. Format-D records are processed in the same way as format-V records. Use RECORDING MODE V for both.

You can omit the RECORDING MODE clause. The compiler determines the recording mode to be V if the largest level-01 record associated with the file is not greater than the block size set in the BLOCK CONTAINS clause, and you take one of the following actions:

- Use the RECORD IS VARYING clause (RECORD clause format 3).  
If you provide values for *integer-1* and *integer-2* (RECORD IS VARYING FROM *integer-1* TO *integer-2*), the maximum record length is the value coded for *integer-2*, regardless of the lengths coded in the level-01 record description entries associated with the file.  
If you omit *integer-1* and *integer-2*, the maximum record length is determined to be the size of the largest level-01 record description entry associated with the file.
- Use the RECORD CONTAINS *integer-1* TO *integer-2* clause (RECORD clause format 2). Make *integer-1* and *integer-2* match the minimum length and the maximum length of the level-01 record description entries associated with the file. The maximum record length is the *integer-2* value.
- Omit the RECORD clause, but code multiple level-01 records (associated with the file) that are of different sizes or contain an OCCURS DEPENDING ON clause.  
The maximum record length is determined to be the size of the largest level-01 record description entry associated with the file.

When you specify a READ INTO statement for a format-V file, the record size read for that file is used in the MOVE statement generated by the compiler. Consequently, you might not get the result you expect if the record just read does not correspond to the level-01 record description. All other rules of the MOVE statement apply. For example, when you specify a MOVE statement for a format-V record read in by the READ statement, the size of the record moved corresponds to its level-01 record description.

When you specify a READ statement for a format-V file followed by a MOVE of the level-01 record, the actual record length is not used. The program will attempt to move the number of bytes described by the level-01 record description. If this number exceeds the actual record length and extends outside the area addressable by the program, results are unpredictable. If the number of bytes described by the level-01 record description is shorter than the physical record read, truncation of bytes beyond the 01-level description occurs. To find the actual length of a variable-length record, specify data-name-1 in format 3 of the RECORD clause of the File Definition (FD).

The RECORD clause is sensitive to the CMPR2 compiler option.

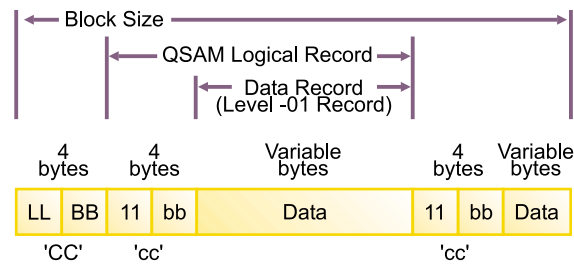
**RELATED TASKS**

- “Requesting fixed-length format” on page 109
- “Requesting spanned format” on page 112
- “Requesting undefined format” on page 114
- “Establishing record formats” on page 108

**RELATED REFERENCES**

- “Layout of format-V records”
- “CMPR2” on page 264
- COBOL for OS/390 & VM Compiler and Run-Time Migration Guide (VS COBOL II Release 2 compatibility and migration)*

**Layout of format-V records:** Format-V QSAM records have control fields (shown below) preceding the data. The QSAM logical record length is determined by adding 4 bytes (for the control fields) to the record length defined in your program, but you must not include these 4 bytes in the description of the record and record length.



- CC** The first 4 bytes of each block contain control information.
  - LL** Represents 2 bytes designating the length of the block (including the 'CC' field).
  - BB** Represents 2 bytes reserved for system use.
- cc** The first 4 bytes of each logical record contain control information.
  - 11** Represents 2 bytes designating the logical record length (including the 'cc' field).
  - bb** Represents 2 bytes reserved for system use.

The block length is determined as follows:

- Unblocked format-V records: CC + cc + the data portion
- Blocked format-V records: CC + the cc of each record + the data portion of each record

The operating system provides the control bytes when the file is written; the control byte fields do not appear in your description of the logical record in the DATA DIVISION of your program. COBOL allocates input and output buffers large enough to accommodate the control bytes. These control fields in the buffer are not available for you to use in your program. When variable-length records are written on unit record devices, control bytes are neither printed nor punched. They appear, however, on other external storage devices, as well as in buffer areas of storage. If you move V-mode records from an input buffer to a WORKING-STORAGE area, they'll be moved without the control bytes.

Files in the HFS are never blocked.

**RELATED CONCEPTS**

“Logical records” on page 108

**RELATED TASKS**

“Requesting variable-length format” on page 110

**RELATED REFERENCES**

“Layout of format-F records” on page 109

“Layout of format-S records” on page 113

“Layout of format-U records” on page 114

### **Requesting spanned format**

Spanned records are in format S. A spanned record is a QSAM logical record that can be contained in one or more physical blocks. You can code RECORDING MODE S for spanned records in QSAM files assigned to magnetic tape or to direct access devices. Do not request spanned records for files in the HFS.

You can omit the RECORDING MODE clause. The compiler determines the recording mode to be S if the maximum record length plus 4 is greater than the block size set in the BLOCK CONTAINS clause.

For files with format S in your program, the compiler determines the maximum record length with the same rules used for format V. The length is based on your usage of the RECORD clause.

When creating files containing format-S records, and a record is larger than the remaining space in a block, COBOL writes a segment of the record to fill the block. The rest of the record is stored in the next block or blocks, depending on its length. COBOL supports QSAM Spanned records up to 32760 bytes long.

When retrieving files with format-S records, your program can retrieve only complete records.

**Benefits of format-S files:** You can efficiently use external storage and still organize your files with logical record lengths by defining files with format-S records:

- You can set block lengths to efficiently use track capacities on direct access devices.
- You are not required to adjust the logical record lengths to device-dependent physical block lengths. One logical record can span two or more physical blocks.
- You have greater flexibility when you want to transfer logical records between direct access storage types.

You will, however, have additional overhead in processing format-S files.

**Format-S files and READ INTO:** By specifying a READ INTO statement for a format-S file, the record size just read for that file is used in the MOVE statement generated by the compiler. Consequently, you might not get the result you expect if the record just read does not correspond to the level-01 record description. All other rules of the MOVE statement apply.

**RELATED CONCEPTS**

“Logical records” on page 108

“Spanned blocked and unblocked files” on page 113

RELATED TASKS

- “Requesting fixed-length format” on page 109
- “Requesting variable-length format” on page 110
- “Requesting undefined format” on page 114
- “Establishing record formats” on page 108

RELATED REFERENCES

- “Layout of format-S records”

**Spanned blocked and unblocked files:** A spanned blocked QSAM file is made up of blocks, each containing one or more logical records or segments of logical records. The logical records can be either fixed or variable in length and their size can be smaller than, equal to, or larger than the physical block size. There are no required relationships between logical records and physical block sizes.

A spanned unblocked file is made up of physical blocks, each containing one logical record or one segment of a logical record. The logical records can be either fixed or variable in length. When the physical block contains one logical record, the block length is determined by the logical record size. When a logical record has to be segmented, the system always writes the largest physical block possible. The system segments the logical record when the entire logical record cannot fit on a track.

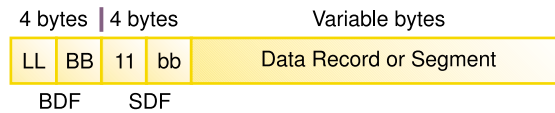
RELATED CONCEPTS

- “Logical records” on page 108

RELATED TASKS

- “Requesting spanned format” on page 112

**Layout of format-S records:** Spanned records are preceded by control fields, as shown below.



Each block is preceded by a block descriptor field. There is only one block descriptor field at the beginning of each physical block.

Each segment of a record in a block, even if the segment is the entire record, is preceded by a segment descriptor field. There is one segment descriptor field for each record segment in the block. The segment descriptor field also indicates whether the segment is the first, the last, or an intermediate segment.

You do not describe these fields in the DATA DIVISION of your COBOL program, and the fields are not available for you to use in your program.

RELATED TASKS

- “Requesting spanned format” on page 112

RELATED REFERENCES

- “Layout of format-F records” on page 109
- “Layout of format-V records” on page 111
- “Layout of format-U records” on page 114

## Requesting undefined format

Format-U records have undefined or unspecified characteristics. With format U, you can process blocks that do not meet format-F or format-V specifications.

The compiler determines the recording mode to be U only if you code RECORDING MODE U.

The record length is determined in your program based on how you use the RECORD clause:

- If you use the RECORD CONTAINS *integer* clause (RECORD clause format 1), the record length is the *integer* value, regardless of the lengths of the level-01 record description entries associated with the file.
- If you use the RECORD IS VARYING clause (RECORD clause format 3), the record length is determined based on whether you code *integer-1* and *integer-2*.  
If you code *integer-1* and *integer-2* (RECORD IS VARYING FROM *integer-1* TO *integer-2*), the maximum record length is the *integer-2* value, regardless of the lengths of the level-01 record description entries associated with the file.  
If you omit *integer-1* and *integer-2*, the maximum record length is determined to be the size of the largest level-01 record description entry associated with the file.
- If you use the RECORD CONTAINS *integer-1* TO *integer-2* clause (RECORD clause format 2), with *integer-1* and *integer-2* matching the minimum length and the maximum length of the level-01 record description entries associated with the file, the maximum record length is the *integer-2* value.
- If you omit the RECORD clause, the maximum record length is determined to be the size of the largest level-01 record description entry associated with the file.

**Format-U files and READ INTO:** When you specify a READ INTO statement for a format-U file, the size of the record just read for that file is used in the MOVE statement generated by the compiler. Consequently, you might not get the result you expect if the record just read does not correspond to the level-01 record description. All other rules of the MOVE statement apply.

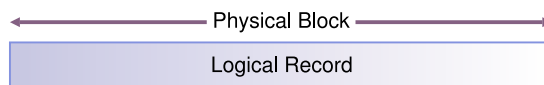
### RELATED TASKS

- “Requesting fixed-length format” on page 109
- “Requesting variable-length format” on page 110
- “Requesting spanned format” on page 112
- “Establishing record formats” on page 108

### RELATED REFERENCES

- “Layout of format-U records”

**Layout of format-U records:** Each block of external storage is handled as a logical record. There are no record-length or block-length fields.



### RELATED CONCEPTS

- “Logical records” on page 108

### RELATED TASKS

- “Requesting undefined format”



#### RELATED REFERENCES

“Layout of format-F records” on page 109

“Layout of format-V records” on page 111

“Layout of format-S records” on page 113

## Setting block sizes

In COBOL, you establish the size of a physical record with the `BLOCK CONTAINS` clause. If you omit this clause, the compiler assumes that the records are not blocked. Blocking QSAM files on tape and disk can enhance processing speed and minimize storage requirements. You can block UNIX system services files (including those in the HFS), PDSE members, and spooled data sets, but doing so has no effect on how the system stores the data.

If you set the block size explicitly in the `BLOCK CONTAINS` clause, it must not be greater than the maximum block size for the device. The block size set for a format-F file must be an integral multiple of the record length.

If your program uses QSAM files on tape, use a physical block size of at least 12 to 18 bytes. Otherwise, the block will be skipped over when a parity check occurs while doing one of the following:

- Reading a block of records of fewer than 12 bytes
- Writing a block of records of fewer than 18 bytes

### Letting OS/390 determine block size

We recommend that to maximize performance, you not explicitly set the block size for a blocked file in your COBOL source program. For new blocked data sets on OS/390, it is simpler to allow OS/390 to supply a system-determined block size. To use this feature:

- Code `BLOCK CONTAINS 0` in your source program.
- Do not code `RECORD CONTAINS 0` in your source program.
- Do not code a `BLKSIZE` value in the JCL DD statement.

### Setting block size explicitly

If you prefer to set a block size explicitly (in either an OS/390 or CMS environment), your program will be most flexible if you:

- Code `BLOCK CONTAINS 0` in your source program.
- Code a `BLKSIZE` value in the ddname definition (the JCL DD statement or the FILEDEF command).

For extended-format QSAM data sets on OS/390, DFSMS adds a 32-byte block suffix to the physical record. If you specify a block size explicitly (using JCL or ISPF), do not include the size of this block suffix in the block size. This block suffix is not available for you to use in your program. DFSMS allocates the space used to read in the block suffix. However, when you calculate how many blocks of a QSAM striped data set will fit on a track of a direct access device, you need to include the size of the block suffix in the block size.

If you specify a block size larger than 32760 directly on your `BLOCK CONTAINS` clause or indirectly with the use of `BLOCK CONTAINS n RECORDS`, and you do not meet both the following conditions, the `OPEN` of the data set fails with file status code 90:

- You use OS/390 V2R10.0 DFSMS or later.
- You define the data set to be tape.

For existing blocked data sets, it is simplest to:

- Code BLOCK CONTAINS 0 in your source program.
- Code no BLKSIZE value in the ddname definition.

When you omit the BLKSIZE from the ddname definition, the block size is automatically obtained by the system from the data set label.

### Taking advantage of LBI

You can improve the performance of tape data sets by using OS/390 V2R10.0 DFSMS, which provides the large block interface (LBI) for large block sizes. When it is available, the COBOL run time automatically uses this facility for those tape files for which you use system-determined block size. LBI is also used for those files for which you explicitly define a block size in JCL or a BLOCK CONTAINS clause. Use of the LBI allows block sizes to exceed 32760 if the tape device supports it.

The LBI is not used in all cases. An attempt to use a block size greater than 32760 in the following cases is diagnosed at compile time or results in a failure at OPEN:

- Spanned records
- OPEN I-0

Using a block size that exceeds 32760 might result in your not being able to read the tape on another system. A tape that you create with a block size greater than 32760 can be read only on an MVS system that uses OS/390 V2R10.0 DFSMS or later and has a tape device that supports block sizes greater than 32760. If you specify a block size that is too large for the file, the device, or the operating system level, a run-time message is issued.

To limit a system-determined block size to 32760, do not specify BLKSIZE anywhere, and do one of the following:

- Specify the BLKSZLIM keyword on your DD statement for the data set.
- Have your systems programmer set BLKSZLIM for the data class using the BLKSZLIM keyword.
- Have your systems programmer set a block-size limit for the system in the DEVSUPxx member of SYS1.PARMLIB using the keyword TAPEBLKSZLIM.

If no BLKSIZE or BLKSZLIM value is available from any source, the system limits BLKSIZE to 32760. You can then enable LBI selectively one of two ways:

- Specify a BLKSZLIM value greater than 32760 in the DD statement for the file.
- Specify a value greater than 32760 for the BLKSIZE in the DD statement or in the BLOCK CONTAINS clause in your COBOL source.

BLKSZLIM is device-independent.

### Block size and the DCB RECFM subparameter

Under OS/390, you can code the S or T option in the DCB RECFM subparameter:

- Use the S (standard) option in the DCB RECFM subparameter for a format-F record with only standard blocks (ones that have no truncated blocks or unfilled tracks in the file, except for the last block of the file). S is also supported for records on tape. It is ignored if the records are not on DASD or tape.

Using this standard block option might improve input-output performance, especially for direct access devices.

- The T (track overflow) option for QSAM files is no longer useful.

#### RELATED TASKS

“Defining QSAM files and records in COBOL” on page 107  
*OS/390 DFSMS: Using Data Sets*

---

## Coding input and output statements for QSAM files

Code the following input and output statements to process a QSAM file or a byte-stream file in the HFS using QSAM:

**OPEN** Makes the file available to your program.

You can open all QSAM files as INPUT, OUTPUT, or EXTEND (depending on device capabilities).

You can also open QSAM files on direct access storage devices as I-0. You cannot open HFS files as I-0; you will receive a file status of 37 if you attempt to do so.

**READ** Reads a record from the file.

With sequential processing, your program reads one record after another in the same order in which they were entered when the file was created.

**WRITE** Creates a record in the file.

Your program writes new records to the end of the file.

**REWRITE**

Updates a record. You cannot update a file in the HFS using REWRITE.

**CLOSE** Releases the connection between the file and your program.

### RELATED TASKS

“Opening QSAM files”

“Adding records to QSAM files” on page 119

“Updating QSAM files” on page 119

“Writing QSAM files to a printer or spooled data set” on page 119

“Closing QSAM files” on page 120

### RELATED REFERENCES

OPEN statement (*IBM COBOL Language Reference*)

READ statement (*IBM COBOL Language Reference*)

WRITE statement (*IBM COBOL Language Reference*)

REWRITE statement (*IBM COBOL Language Reference*)

CLOSE statement (*IBM COBOL Language Reference*)

## Opening QSAM files

Before your program can use any READ, WRITE, or REWRITE statements to process records in a file, it must first open the file with an OPEN statement.

An OPEN statement works if both of the following are true:

- The file is available or has been dynamically allocated.
- The *fixed file attributes* coded in the ddname definition or the data set label for a file match the attributes coded for that file in the SELECT and FD statements of your COBOL program.

Mismatches in the file organization attributes, code set, maximum record size, or record type (fixed or variable) result in a file status code 39, and the OPEN statement fails. Mismatches in maximum record size and record type are not considered errors when opening files in the HFS.

For fixed-length QSAM files under OS/390, when you code RECORD CONTAINS 0 in the FD, the record size attributes are not in conflict. The record size is taken from the DD statement or the data set label, and the OPEN statement is successful.

Code CLOSE WITH LOCK so that the file cannot be opened again while the program is running.

Use the REVERSED option of the OPEN statement to process tape files in reverse order. Execution of the OPEN statement will then position the file at its end. Subsequent READ statements read the data records in reverse order, starting with the last record. The REVERSED option is supported only for files with fixed-length records.

#### RELATED TASKS

“Dynamically creating QSAM files with CBLQDA”

#### RELATED REFERENCES

OPEN statement (*IBM COBOL Language Reference*)

## Dynamically creating QSAM files with CBLQDA

A file is considered to be available on OS/390 when it has been identified to the operating system using a DD statement, an export command (an environment variable), or a TSO ALLOCATE command. (For availability, a DD statement with a misspelled ddname is equivalent to a missing DD statement; an environment variable with a value that is not valid is equivalent to an unset variable.)

A file is available on CMS when it has been identified using a CMS FILEDEF command or if the file FILE *ddname* exists on your minidisk.

Sometimes a QSAM file is unavailable on OS/390 or CMS, but the COBOL language defines that the file be created. The file is implicitly created for you if you use the run-time option CBLQDA and one of the following circumstances exists:

- The file is being opened for OUTPUT, regardless of the OPTIONAL phrase. An OPTIONAL file is being opened as EXTEND or I-0.

Optional files are files that are not necessarily present each time the program is run. You can define files opened in INPUT, I-0, or EXTEND mode as optional by using the SELECT OPTIONAL phrase in the FILE-CONTROL paragraph.

The file is allocated with the system default attributes established at your installation and the attributes coded in the SELECT and FD statements in your program.

Do not confuse this implicit allocation mechanism with the dynamic allocation of files through the use of environment variables. That explicit dynamic allocation requires a valid environment variable to be set. This CBLQDA support is used only when the QSAM file is unavailable as defined above, which includes no valid environment variable being set.

Under OS/390, files created using the CBLQDA option are temporary data sets and do not exist after the program has run. Under CMS, they are named FILE *ddname* A4 and do exist after the program has run.

#### RELATED TASKS

“Opening QSAM files” on page 117

## Adding records to QSAM files

To add to a QSAM file, open the file as EXTEND and use the WRITE statement to add records immediately after the last record in the file.

To add records to a file opened as I-0, you must first close the file and open it as EXTEND.

### RELATED REFERENCES

READ statement (*IBM COBOL Language Reference*)

WRITE statement (*IBM COBOL Language Reference*)

## Updating QSAM files

You can update QSAM files that reside on direct access storage devices only. You cannot update files in the HFS.

Replace an existing record with another record of the same length by doing the following:

1. Open the file as I-0.
2. Use REWRITE to update an existing record in the file. (The last file processing statement before REWRITE must have been a successful READ statement.)

### RELATED REFERENCES

REWRITE statement (*IBM COBOL Language Reference*)

## Writing QSAM files to a printer or spooled data set

COBOL provides language statements to control the size of a printed page and control the vertical positioning of records.

### Controlling the page size

Use the LINAGE clause of the FD entry to control the size of your printed page: the number of lines in the top and bottom margins and in the footing area of the page. When you use the LINAGE clause, COBOL handles the file as if you had also requested the ADV compiler option.

If you use the LINAGE clause in combination with WRITE BEFORE/AFTER ADVANCING *nn* LINES, be careful about the values you set. With the ADVANCING *nn* LINES clause, COBOL first calculates the sum of LINAGE-COUNTER plus *nn*. Subsequent actions depend on the size of *nn*. The END-OF-PAGE imperative statement is performed after the LINAGE-COUNTER is increased. Consequently, the LINAGE-COUNTER could be pointing to the next logical page instead of to the current footing area when the END-OF-PAGE statement is performed.

AT END-OF-PAGE or NOT AT END-OF-PAGE imperative statements are performed only if the write operation completes successfully. If the write operation is unsuccessful, control is passed to the end of the WRITE statement, omitting all conditional phrases.

### Controlling the vertical positioning of records

Use the WRITE ADVANCING statement to control the vertical positioning of each record you write on a printed page.

- . . . BEFORE ADVANCING prints the record before the page is advanced.
- . . . AFTER ADVANCING prints the record after the page is advanced.

Specify the number of lines the page is advanced with an integer (or an *identifier* with a *mnemonic-name*) following ADVANCING. If you omit the ADVANCING option from your WRITE statement, you get the equivalent of:

```
AFTER ADVANCING 1 LINE
```

#### RELATED REFERENCES

WRITE statement (*IBM COBOL Language Reference*)

## Closing QSAM files

Use the CLOSE statement to disconnect your program from the QSAM file. If you try to close a file that is already closed, you will get a logic error.

If you do not close a QSAM file, the file is automatically closed for you under the following conditions:

- When the run unit ends normally, the run time closes all open files that are defined in any of the following types of programs in the run unit: COBOL for OS/390 & VM, COBOL for MVS & VM, and VS COBOL II.
- If the run unit ends abnormally and you have set the TRAP(ON) run-time option, the run time closes all open files that are defined in any of the following types of programs in the run unit: COBOL for OS/390 & VM, COBOL for MVS & VM, and VS COBOL II.
- When Language Environment condition handling is completed and the application resumes in a routine other than where the condition occurred, the run time closes all open files that are defined in any of the following types of programs in the run unit that might be called again and reentered: COBOL for OS/390 & VM, COBOL for MVS & VM, and VS COBOL II.

You can change the location of where the program resumes running (after a condition is handled) by moving the resume cursor with the Language Environment CEEMRCR callable service or by using HLL language constructs such as a C longjmp.

- When you use CANCEL for a COBOL for OS/390 & VM, COBOL for MVS & VM, or VS COBOL II subprogram, the run time closes any open nonexternal files that are defined in that program.
- When a COBOL for OS/390 & VM, COBOL for MVS & VM, or VS COBOL II subprogram with the INITIAL attribute returns control, the run time closes any open nonexternal files that are defined in that program.

File status codes are set when these implicit CLOSE operations are performed, but EXCEPTION/ERROR and LABEL declaratives are not invoked.

#### RELATED REFERENCES

CLOSE statement (*IBM COBOL Language Reference*)

---

## Handling errors in QSAM files

When an input-statement or output-statement operation fails, COBOL does not take corrective action for you. You choose whether or not your program will continue running after a less-than-severe input or output error occurs.

COBOL provides these ways for you to intercept and handle certain QSAM input and output errors:

- End of file phrase (AT END)
- EXCEPTION/ERROR declarative

- FILE STATUS clause
- INVALID KEY phrase

If you do not code a FILE STATUS key or a declarative, serious QSAM processing errors will cause a message to be issued and a Language Environment condition to be signaled, which will cause an abend if you specify the run-time option ABTERMENC (ABEND).

If you use the FILE STATUS clause or the EXCEPTION/ERROR declarative, code EROPT=ACC in the DCB of the DD statement for that file. Otherwise, your COBOL program will not be able to continue processing after some error conditions.

If you use the FILE STATUS clause, be sure to check the key and take appropriate action based on its value. If you do not check the key, your program might continue, but the results will probably not be what you expected.

#### RELATED TASKS

“Handling errors in input and output operations” on page 191

---

## Working with QSAM files under OS/390

This section describes:

- “Defining and allocating QSAM files”
- “Retrieving QSAM files” on page 123
- “Ensuring file attributes match your program” on page 124
- “Using striped extended-format QSAM data sets” on page 126

#### RELATED REFERENCES

“Allocation of buffers for QSAM files” on page 127

### Defining and allocating QSAM files

You can define a QSAM file or a byte-stream file in the HFS using either a DD statement or an environment variable. When you use an environment variable, the name must be in uppercase. Allocation of these files follows the general rules for the allocation of COBOL files. When you use an environment variable to define a QSAM file, specify the MVS data set as follows:

DSN(*dataset-name*) or DSN(*dataset-name(member-name)*). *dataset-name* must be fully qualified and cannot be a temporary data set (must not start with &).

You can optionally specify the following attributes in any order following the DSN:

- A disposition value, one of: NEW, OLD, SHR, or MOD
- TRACKS or CYL
- SPACE(*nnn,mmm*)
- VOL(*volume-serial*)
- UNIT(*type*)
- KEEP, DELETE, CATALOG, or UNCATALOG
- STORCLAS(*storage-class*)
- MGMTCLAS(*management-class*)
- DATACLAS(*data-class*)

You can use either an environment variable or a DD definition to define a file in the HFS. To do this, define one of the following with a name that matches the external name on your ASSIGN clause:

- A DD allocation that uses `PATH='absolute-path-name'` and `FILEDATA=BINARY`
- An environment variable with a value `PATH(pathname)`, where *pathname* is an absolute path name (starting with /).

For compatibility with earlier releases of COBOL, you can also specify `FILEDATA=TEXT` when using a DD allocation for HFS files, but this use is not recommended. To process text files in the HFS, use `LINE SEQUENTIAL` organization. If you do use `QSAM` to process text files in the HFS, you cannot use environment variables to define the files.

When you define a `QSAM` file, use the specified parameters to do the following:

What you want to do	DD parameter to use	EV keyword to use
Name the file	DSNAME (data set name)	DSN
Select the type and quantity of input-output devices to be allocated for the file.	UNIT	UNIT for type only
Give instructions for the volume in which the file will reside and for volume mounting.	VOLUME, or let the system choose an output volume.	VOL
Allocate the type and amount of space the file needs. (For direct access storage devices only.)	SPACE	SPACE for the amount of space (primary and secondary only); TRACKS or CYL for the type of space
Specify the type (and some of the contents of) the label associated with the file.	LABEL	N/A
Indicate whether you want to catalog, pass, or keep the file after the job step is completed.	DISP	NEW, OLD, SHR, MOD plus KEEP, DELETE, CATALOG, or UNCATALOG
Complete any data control block information you want to add.	DCB subparameters	N/A

Some of the information about the `QSAM` file must always be coded in the `FILE-CONTROL` entry, the `FD` entry, and other COBOL clauses. Other information must be coded in the DD statement or environment variable for output files. For input files, the system can obtain information from the file label (for standard label files). If DCB information is provided in the DD statement for input files, it overrides information on the data set label. For example, the amount of space allocated for a new direct-access device file can be set in the DD statement by the `SPACE` parameter.

You cannot express certain characteristics of `QSAM` files in the COBOL language, but you can code them in the DD statement for the file using the DCB parameter. Use the subparameters of the DCB parameter to provide information that the system needs for completing the data set definition, including the following:

- Block size (`BLKSIZE=`), if `BLOCK CONTAINS 0 RECORDS` was coded at compile time (which is recommended)
- Options to be executed if an error occurs in reading or writing a record
- `TRACK OVERFLOW` or standard blocks



- Mode of operation for a card reader or punch

DCB attributes coded for a DD DUMMY do not override those coded in the FD entry of your COBOL program.

#### RELATED TASKS

“Setting block sizes” on page 115

“Defining QSAM files and records in COBOL” on page 107

“Allocating files” on page 105

#### RELATED REFERENCES

“Parameters for creating QSAM files”

*OS/390 MVS JCL Reference*

## Parameters for creating QSAM files

The following DD statement parameters are frequently used to create QSAM files.

---

```

DSNAME= [ dataset-name
DSN=     dataset-name(member-name)
         &&name
         &&name(member-name) ]

UNIT= ( name[,unitcount] )

VOLUME= ( [PRIVATE] [,RETAIN] [,vol-sequence-num] [,volume-count] ...
VOL=     ... [ ,SER=(volume-serial[,volume-serial]...) ]
         [ ,REF=[ dsname
                  *.ddname
                  *.stepname.ddname
                  *.stepname.procstep.ddname ] ] )

SPACE= ( [TRK
         CYL
         average-record-length] [,primary-quantity[,secondary-quantity][,directory-quantity]])

LABEL= ( [data-set-sequene-number,] [ NL
                                     SL
                                     SUL ] [ ,EXPDT= [yyddd
                                                         ,yyy/ddd] ] [ ,RETPD=xxxx ] )

DISP= ( [NEW] [ ,DELETE ] [ ,DELETE ]
        [MOD] [ ,KEEP   ] [ ,KEEP   ]
        [PASS] [ ,CATLG ] [ ,CATLG ] )

DCB= ( subparameter-list )

```

---

#### RELATED TASKS

“Defining and allocating QSAM files” on page 121

## Retrieving QSAM files

You retrieve QSAM files, cataloged or not, by using job control statements or environment variables.

### Cataloged files

All data set information, such as volume and space, is stored in the catalog and file label. All you have to code are the data set name and a disposition. When you use a DD statement, this is the DSNAME parameter and

the DISP parameter. When you use an environment variable, this is the DSN parameter and one of the parameters OLD, SHR, or MOD.

### Noncataloged files

Some information is stored in the file label, but you must code the unit and volume information as well as the *dsname* and disposition.

If you are using JCL, and you created the file in the current job step or in a previous job step in the current job, you can refer to the previous DD statement for most of the data set information. You do, however, need to code DSNNAME and DISP.

#### RELATED REFERENCES

“Parameters for retrieving QSAM files”

### Parameters for retrieving QSAM files

The following DD statement parameters are used to retrieve previously created files.

---

```
DSNAME= [ dataset-name
         dataset-name(member-name)
DSN=    [ *.ddname
         *.stepname.ddname
         &&name
         &&name(member-name) ]

UNIT=   ( name[,unitcount] )

VOLUME= ( subparameter-list )
VOL=

LABEL=  ( subparameter-list )

DISP=   ( [ OLD ] [ ,DELETE ] [ ,DELETE ] )
         [ SHR ] [ ,KEEP ] [ ,KEEP ]
         [ MOD ] [ ,PASS ] [ ,CATLG ]
                [ ,UNCATLG ]

DCB=    ( subparameter-list )
```

---

#### RELATED TASKS

“Retrieving QSAM files” on page 123

## Ensuring file attributes match your program

When the fixed file attributes coded in the DD statement or the data set label for a file and the attributes coded for that file in the SELECT and FD statements of your COBOL program are not consistent, an OPEN statement in your program might not work. Mismatches in the attributes for file organization, record format (fixed or variable), record length, or the code set result in a file status code 39, and the OPEN statement fails. An exception exists for files in the HFS: mismatches in record format and record length do not cause an error.

To prevent common file status 39 problems, follow the guidelines listed below for processing files that are existing, new, or dynamically created by COBOL.

Remember that information in the JCL or environment variable overrides information in the data set label.

### Processing existing files

When your program processes an existing file, code the description of the file in your COBOL program to be consistent with the file attributes of the data set. Use these guidelines to define the maximum record length.

For this format	Specify this
V or S	Exactly 4 bytes smaller than the length attribute of the data set
F	Same as the length attribute of the data set
U	Same as the length attribute of the data set

### Defining variable-length (format-V) records

The easiest way to define variable-length records in your program is to use RECORD IS VARYING FROM *integer-1* TO *integer-2* in the FD entry and set an appropriate value for *integer-2*. For example, assume that you have determined the length attribute of the data set to be 104 (LRECL=104). Remembering that the maximum record length is determined from the RECORD IS VARYING clause (in which values are set) and not from the level-01 record descriptions, you could define a format-V file in your program with this code:

```
FILE SECTION.  
FD COMMUTER-FILE-MST  
   RECORDING MODE IS V  
   RECORD IS VARYING FROM 4 TO 100 CHARACTERS.  
01 COMMUTER-RECORD-A      PIC X(4).  
01 COMMUTER-RECORD-B      PIC X(75).
```

### Defining format-U records

Assume that the existing file in the previous example was format-U instead of format-V. If the 104 bytes are all user data, you could define the file in your program with this code:

```
FILE SECTION.  
FD COMMUTER-FILE-MST  
   RECORDING MODE IS U  
   RECORD IS VARYING FROM 4 TO 104 CHARACTERS.  
01 COMMUTER-RECORD-A      PIC X(4).  
01 COMMUTER-RECORD-B      PIC X(75).
```

### Defining fixed-length records

To define fixed-length records in your program, use either the RECORD CONTAINS *integer* clause, or omit this clause and code all level-01 record descriptions to be the same fixed size. In either case, use a value that equals the value of the length attribute of the data set. When you intend to use the same program to process different files at run time and the files have differing fixed-length record lengths, the recommended way to avoid record-length conflicts is to code RECORD CONTAINS 0.

If the existing file is an ASCII data set (DCB=(OPTCD=Q)), you must use the CODE-SET clause in the program's FD entry for the file.

### Processing new files

When your COBOL program will write records to a new file that is made available before the program is run, ensure that the file attributes you code in the DD statement, the environment variable, or the allocation do not conflict with the attributes you have coded in your program. Usually, you need to code only a minimum of parameters when predefining your files.

When you do need to explicitly set a length attribute for the data set, (for example, you are using an ISPF allocation panel or if your DD statement is for a batch job in which the program uses RECORD CONTAINS 0):

- For format-V and format-S files, set a length attribute that is 4 bytes larger than that defined in the program.
- For format-F and format-U files, set a length attribute that is the same as that defined in the program.
- If you open your file as OUTPUT and write it to a printer, the compiler might add 1 byte to the record length to account for the carriage control character, depending on the ADV compiler option and the COBOL language used in your program. In such a case, take the added byte into account when coding the LRECL.

For example, suppose your program contains the following code for a file with variable-length records:

```
FILE SECTION.  
FD  COMMUTER-FILE-MST  
   RECORDING MODE IS V  
   RECORD CONTAINS 10 TO 50 CHARACTERS.  
01  COMMUTER-RECORD-A          PIC X(10).  
01  COMMUTER-RECORD-B          PIC X(50).
```

The LRECL in your DD statement or allocation should be 54.

### **Processing files dynamically created by COBOL**

When you have not made a file available with a DD statement or a TSO ALLOCATE command and your COBOL program defines that the file be created, COBOL for OS/390 & VM dynamically allocates the file. When the file is opened, the file attributes coded in your program are used. You do not have to worry about file attribute conflicts.

#### **RELATED TASKS**

- “Requesting fixed-length format” on page 109
- “Requesting variable-length format” on page 110
- “Requesting undefined format” on page 114
- “Dynamically creating QSAM files with CBLQDA” on page 118

## **Using striped extended-format QSAM data sets**

A striped extended-format QSAM data set is an extended-format QSAM data set that is spread over multiple volumes, allowing parallel data access.

Striped extended-format QSAM data sets can benefit an application with these characteristics:

- The application processes files that contain large volumes of data.
- The time for the input and output operations to the files significantly affects overall performance.

For you to gain the maximum benefit from using QSAM striped data sets, DFSMS needs to be able to allocate the required number of buffers above the 16-MB line.

When you develop applications that contain files allocated to QSAM striped data sets, follow these guidelines:

- Avoid using a QSAM striped data set for a file that cannot have buffers allocated above the 16-MB line.

- Omit the RESERVE clause in the FILE-CONTROL paragraph entry for the file. Omitting the RESERVE clause allows DFSMS to determine the optimum number of buffers for the data set.
- Compile your program with the DATA(31) and RENT compiler options, and make the load module AMODE 31.
- Specify the ALL31(ON) run-time option if the file is an EXTERNAL file with format-F, format-V, or format-U records.

RELATED TASKS

*OS/390 DFSMS: Using Data Sets* (performance considerations)

RELATED REFERENCES

“Allocation of buffers for QSAM files”

### Allocation of buffers for QSAM files

DFSMS automatically allocates buffers for storing input and output for QSAM files above or below the 16-MB line as appropriate for the file being used. Most QSAM files have buffers allocated above the 16-MB line. Exceptions are:

- Programs running in AMODE 24.
- Programs compiled with the DATA(24) and RENT options.
- Programs compiled with the NORENT and RMODE(24) options.
- Programs compiled with the NORENT and RMODE(AUTO) options.
- EXTERNAL files, when the ALL31(OFF) run-time option is being specified. To specify the ALL31(ON) run-time option, all programs in the run unit must be capable of running in 31-bit addressing mode.
- Files allocated to the TSO terminal.
- A file with format-S (spanned) records, if the file is any of the following:
  - An EXTERNAL file (even if the ALL31(ON) option is specified).
  - A file specified in a SAME RECORD AREA clause of the I-O-CONTROL paragraph.
  - A blocked file that is opened I-O and updated using the REWRITE statement.

RELATED TASKS

“Using striped extended-format QSAM data sets” on page 126

## Accessing HFS files using QSAM

You can process byte-stream files in the hierarchical file system (HFS) as COBOL ORGANIZATION SEQUENTIAL files using QSAM. To do this, specify as the *assignment-name* on the ASSIGN clause one of the following:

**ddname**

A DD allocation that identifies the file with the keywords PATH= and FILEDATA=BINARY

**Environment variable name**

An environment variable with the run-time value of the HFS path for the file

## Restrictions and usage

Note the following restrictions:

- File status 39 (fixed file attribute conflict) is not enforced for either of the following:
  - Record-length conflict
  - Record-type conflict (fixed vs. variable)

- Spanned record format is not supported.
- A READ returns the number of bytes equal to that of the maximum logical record size for the file except for the last record, which might be shorter.  
For example, suppose your file definition has 01 record descriptions of 3, 5, and 10 bytes long, and you write the following three records: 'abc', 'defgh', and 'ijklmnopqr', in that order. Your first READ of this file returns 'abcdefghij', your second READ returns 'klmnopqr ', and your third READ results in the AT END condition.
- OPEN I-0 and REWRITE are not supported. If you attempt one of these operations, you will get the following file status conditions:
  - 37 from OPEN I-0
  - 47 from REWRITE (because you could not have opened the file I-0 successfully)

For compatibility with earlier releases of COBOL, you can also specify FILEDATA=TEXT when using a DD allocation for HFS files, but this use is not recommended. To process text files in the HFS, use the LINE SEQUENTIAL organization. If you do use QSAM to process text files in the HFS, you cannot use environment variables to define the files.

#### RELATED TASKS

“Allocating files” on page 105

“Defining and allocating QSAM files” on page 121

Accessing HFS files via BSAM and QSAM (*OS/390 DFSMS: Using Data Sets*)

---

## Identifying QSAM files to CMS

Under CMS, when you run a COBOL for OS/390 & VM program that has input or output files or both, or you need to read an OS data set onto a CMS disk, you must first identify the files to CMS with the FILEDEF command. The FILEDEF command in CMS performs the same functions as the data definition (DD) statement in OS/390 job control language (JCL): it describes the input and output files.

For certain files, you also need to use the LABELDEF command.

### Using FILEDEF

The FILEDEF command relates a ddname (file name) in your program with an input or output device. If the FILEDEF command is issued for a program input or output file, the ddname must be the same as the ddname or file name coded for the file in the source program. Code the ddname in the ASSIGN clause of the SELECT statement in the FILE CONTROL paragraph.

For example, suppose your program contains this code:

```
FILE-CONTROL.
  SELECT INFILE
    ASSIGN TO UR-2540R-S-CARDIN
  SELECT OUTFILE
    ASSIGN TO DA-3330-S-OUTDD.
. . .
FD INFILE
. . .
FD OUTFILE
. . .
```

The ddname for INFILE is CARDIN, and the ddname for OUTFILE is OUTDD. These are the names you use in the ddname portion of the FILEDEF command. If the input file is to be read from your virtual card reader, your FILEDEF command might be:

```
FILEDEF CARDIN READER
```

Although you can read data sets from operating system (OS) disks when you run COBOL programs in CMS, you cannot write files on the disks unless they are VSAM files. You can, however, write OS simulated data sets on CMS disks, which retain file characteristics of OS sequential data sets. To do this, use file-mode 4 when creating the files. You should use OS simulated data sets for files defined in the COBOL program as blocked and with variable-length records.

When a program abends, or an HX Immediate command is issued during execution, CMS automatically clears all FILEDEFs even if PERM has been specified.

## Using LABELDEF

The LABELDEF command is required in addition to FILEDEF for certain files.

CMS multivolume tape files must be standard labeled tape files (SL or AL). Also, the LABELDEF command must be used to sequentially identify the tape volumes that compose the logical file.

If you are using OPEN EXTEND and standard labeled tapes, issue the LABELDEF command with the FID option. For example:

```
FILEDEF OUTFILE TAP3 SL  
LABELDEF OUTFILE FID EXTR4C
```

### RELATED REFERENCES

The FILEDEF command (*VM/ESA CMS Command Reference*)

---

## Labels for QSAM files

You can use labels to identify magnetic tape and direct access volumes and data sets. The operating system uses label processing routines to identify and verify labels and locate volumes and data sets.

There are two kinds of labels: standard and nonstandard. COBOL for OS/390 & VM does not support nonstandard user labels. In addition, standard user labels contain user-specified information about the associated data set.

Standard labels consist of volume labels and groups of data set labels. Volume labels precede or follow data on the volume, and identify and describe the volume. The data set labels precede or follow each data set on the volume, and identify and describe the data set.

- The data set labels that precede the data set are called *header labels*.
- The data set labels that follow the data set are called *trailer labels*. They are similar to the header labels, except that they also contain a count of blocks in the data set.
- The data set label groups can optionally include standard user labels.
- The volume label groups can optionally include standard user labels.

### RELATED TASKS

“Using trailer and header labels” on page 130

## Using trailer and header labels

You can create, examine, or update user labels when the beginning or end of a data set or volume (reel) is reached. End-of-volume or beginning-of-volume exits are allowed. You can also create or examine intermediate trailers and headers.

You can create, examine, or update up to eight header labels and eight trailer labels on each volume of the data set. (QSAM EXTEND works in a manner identical to OUTPUT except that the beginning-of-file label is not processed.) Labels reside on the initial volume of a multivolume data set. This volume must be mounted as CLOSE if trailer labels are to be created, examined, or updated. Trailer labels for files opened as INPUT or I-0 are processed when a CLOSE statement is performed for the file that has reached an AT END condition.

If you code a header or trailer with the wrong position number, the result is unpredictable. (Data management might force the label to the correct relative position.)

When you use standard label processing, code the label type of the standard and user labels (SUL) on the DD statement that describes the data set.

### Getting a user-label track

If you use a LABEL subparameter of SUL for direct access volumes, a separate user-label track will be allocated when the data set is created. This additional track is allocated at initial allocation and for sequential data sets at end-of-volume (volume switch). The user-label track (one per volume of a sequential data set) will contain both user header and user trailer labels. If a LABEL name is referenced outside the user LABEL declarative, results are unpredictable.

### Handling user labels

The USE AFTER LABEL declarative provides procedures for handling user labels on supported files. The AFTER option indicates processing of standard user labels.

List the labels as *data-names* in the LABEL RECORDS clause in the FD entry for the file.

When the file is opened as:	And:	Result:
INPUT	USE . . . LABEL declarative is coded for the OPEN option or for the file.	The label is read and control is passed to the LABEL declarative.
OUTPUT	USE . . . LABEL declarative is coded for the OPEN option or for the file.	A buffer area for the label is provided and control is passed to the LABEL declarative.
INPUT or I-0	CLOSE statement is performed for the file that has reached the AT END condition.	Control is passed to the LABEL declarative for processing trailer labels.

You can specify a special exit by using the statement GO TO MORE-LABELS. When this statement results in an exit from a label DECLARATIVE SECTION, the system does one of the following:

- Writes the current beginning or ending label and then reenters the USE section at its beginning to create more labels. After creating the last label, the system exits by performing the last statement of the section.



- Reads an additional beginning or ending label, and then reenters the USE section at its beginning to check more labels. When processing user labels, the system reenters the section only if there is another user label to check. Hence, a program path that flows through the last statement in the section is not needed.

If a GO TO MORE-LABELS statement is not performed for a user label, the DECLARATIVE SECTION is not reentered to check or create any immediately succeeding user labels.

**RELATED CONCEPTS**

“Labels for QSAM files” on page 129

## Format of standard labels

Standard labels are 80-character records that are recorded in EBCDIC or ASCII. The first four characters are always used to identify the labels. The figure below shows these *identifiers* for tape.

Identifier	Description
VOL1	Volume label
HDR1 or HDR2	Data set header labels
EOV1 or EOV2	Data set trailer labels (end-of-volume)
EOF1 or EOF2	Data set trailer labels (end-of-data-set)
UHL1 to UHL8	User header labels
UTL1 to UTL8	User trailer labels

The format of the label for a direct-access volume is the almost the same as the format of the label group for a tape volume label group. The difference is that a data set label of the initial DASTO volume label consists of the data set control block (DSCB). The DSCB appears in the volume table of contents (VTOC) and contains the equivalent of the tape data set header and trailer, in addition to control information such as space allocation.

### Standard user labels

User labels are optional within the standard label groups.

The format used for user header labels (UHL1-8) and user trailer labels (UTL1-8) consists of a label 80 characters in length recorded in either:

- EBCDIC on DASD or on IBM standard labeled tapes, or
- ASCII on ANSI/ISO/FIPS labeled tapes

The first 3 bytes consist of the characters that identify the label as either:

- UHL for a user header label (at the beginning of a data set), or
- UTL for a user trailer label (at the end-of-volume or end-of-data set)

The next byte contains the relative position of this label within a set of labels of the same type. One through eight labels are permitted.

The remaining 76 bytes consist of user-specified information.

Standard user labels are not supported for QSAM striped data sets.

**RELATED CONCEPTS**

“Labels for QSAM files” on page 129

---

## Processing QSAM ASCII files on tape

If your program processes an QSAM ASCII file, do the following:

1. Request the ASCII alphabet.
2. Define the record formats.
3. Define the ddname (JCL for OS/390 or FILEDEF for CMS).

In addition, if your program processes numeric data items from ASCII files, use the separately signed numeric data type (SIGN IS LEADING SEPARATE).

### Requesting the ASCII alphabet

In the SPECIAL-NAMES paragraph, code STANDARD-1 for ASCII:

```
ALPHABET-NAME IS STANDARD-1
```

In the FD statement for the file, code:

```
CODE-SET IS ALPHABET-NAME
```

### Defining the record formats

Process QSAM ASCII tape files with any of these record formats:

- Fixed length (format F)
- Undefined (format U)
- Variable length (format V)

If you are using variable-length records, you cannot explicitly code format D; instead, code RECORDING MODE V. The format information is internally converted to D mode. D-mode records have a 4-byte record descriptor for each record.

### Defining the ddname

Under OS/390, processing ASCII files requires special JCL coding. Code these subparameters of the DCB parameter in the DD statement:

```
BUFOFF=[L | n]
```

*L* A 4-byte block prefix that contains the block length (including the block prefix).

*n* The length of the block prefix:

- For input, from 0 through 99
- For output, either 0 or 4

Use this value if you coded BLOCK CONTAINS 0.

```
BLKSIZE=n
```

*n* The size of the block, including the length of the block prefix.

```
LABEL=[AL | AUL | NL]
```

AL American National Standard (ANS) labels.

AUL ANS and user labels.

NL No labels.

OPTCD=Q

- Q This value is required for ASCII files, and is the default if the file was created using COBOL for OS/390 & VM.

RELATED TASKS

“Processing ASCII file labels”

---

## Processing ASCII file labels

Standard label processing for ASCII files is the same as standard label processing for EBCDIC files. The system translates ASCII code into EBCDIC before processing.

All ANS user labels are optional. ASCII files can have user header labels (UHL*n*) and user trailer labels (UTL*n*). There is no limit to the number of user labels at the beginning and the end of a file; you can write as many labels as you need. All user labels must be 80 bytes in length.

To create or verify user labels (user label exit), code a USE AFTER STANDARD LABEL procedure. You cannot use USE BEFORE STANDARD LABEL procedures. Under CMS, code the OPTCD=Q option on the FILEDEF command for ASCII files.

ASCII files on tape can have:

- ANS labels
- ANS and user labels
- No labels

Any labels on an ASCII tape must be in ASCII code only. Tapes containing a combination of ASCII and EBCDIC cannot be read.

RELATED TASKS

“Processing QSAM ASCII files on tape” on page 132



---

## Chapter 9. Processing VSAM files

Virtual storage access method (VSAM) is an access method for files on direct-access storage devices. With VSAM you can:

- Load a file
- Retrieve records from a file
- Update a file
- Add, replace, and delete records in a file

VSAM processing has these advantages over QSAM:

- Protection of data against unauthorized access
- Compatibility across systems
- Independence of devices (no need to be concerned with block size and other control information)
- Simpler JCL (information needed by the system is provided in integrated catalogs)
- Ability to use indexed file organization or relative file organization

The lists below show how VSAM terms differ from COBOL terms and other terms that you might be familiar with.

<b>VSAM term</b>	<b>COBOL term</b>	<b>Similar non-VSAM term</b>
Data set	File	Data set
Entry-sequenced data set (ESDS)	Sequential file	QSAM data set
Key-sequenced data set (KSDS)	Indexed file	ISAM data set
Relative-record data set (RRDS)	Relative file	BDAM data set
Control interval size (CISZ)		Block size
Buffers (BUFNI/BUFND)		BUFNO
Access method control block (ACB)		Data control block (DCB)
Cluster (CL)		Data set
Cluster definition		Data set allocation
AMP parameter of JCL DD statement		DCB parameter of JCL DD statement
Record size		Record length

The term *file* in this VSAM information refers to either a COBOL file or a VSAM data set.

If you have complex requirements or frequently use VSAM, review the VSAM publications for your operating system.

### RELATED TASKS

“Defining VSAM file organization and records” on page 137

“Coding input and output statements for VSAM files” on page 143

“Protecting VSAM files with a password” on page 151

“Handling errors in VSAM files” on page 151

“Working with VSAM data sets under OS/390 and OS/390 UNIX” on page 152

“Defining VSAM data sets under CMS” on page 158

“Improving VSAM performance” on page 159

#### RELATED REFERENCES

*OS/390 DFSMS: Using Data Sets*  
*OS/390 DFSMS: Macro Instructions for Data Sets*  
*OS/390 DFSMS: Access Method Services for Catalogs*  
*VSAM Administration: Macro Instruction Reference*

---

## VSAM files

The physical organization of VSAM data sets differs considerably from those used by other access methods. VSAM data sets are held in control intervals and control areas (CA). The size of these is normally determined by the access method, and the way in which they are used is not visible to you.

You can use three types of file organization with VSAM.

### **VSAM sequential file organization**

(Also referred to as VSAM ESDS (entry-sequenced data set) organization.)

In VSAM sequential file organization, the records are stored in the order in which they were entered. VSAM entry-sequenced data sets are equivalent to QSAM sequential files. The order of the records is fixed.

### **VSAM indexed file organization**

(Also referred to as VSAM KSDS (key-sequenced data set) organization.) In

a VSAM indexed file (KSDS), the records are ordered according to the collating sequence of an embedded prime key field, which you define. The prime key consists of one or more consecutive characters in the records.

The prime key uniquely identifies the record and determines the sequence in which it is accessed with respect to other records. A prime key for a record might be, for example, an employee number or an invoice number.

### **VSAM relative file organization**

(Also referred to as VSAM fixed-length or variable-length RRDS (relative-record data set) organization.) A VSAM relative record data set (RRDS) contains records ordered by their relative key. The relative key is the relative record number that represents the location of the record relative to where the file begins. The relative record number identifies the fixed- or variable-length record.

In a VSAM fixed-length RRDS, records are placed in a series of fixed-length slots in storage. Each slot is associated with a relative record number. For example, in a fixed-length RRDS containing 10 slots, the first slot has a relative record number of 1, and the tenth slot has a relative record number of 10.

In a VSAM variable-length RRDS, the records are ordered according to their relative record number. Records are stored and retrieved according to the relative record number that you set.

Throughout this documentation, the term *VSAM relative-record data set* (or *RRDS*) is used to mean both relative-record data sets with fixed-length records and with variable-length records, unless they need to be differentiated.

The following table compares the different types of VSAM data sets in terms of several characteristics.

Characteristic	Entry-sequenced data set (ESDS)	Key-sequenced data set (KSDS)	Relative-record data set (RRDS)
Order of records	Order in which they are written	Collating sequence by key field	Order of relative record number
Access	Sequential	By key through an index	By relative record number, which is handled like a key
Alternate indexes	Can have one or more alternate indexes, although not supported in COBOL	Can have one or more alternate indexes	Cannot have alternate indexes
Relative byte address (RBA) and relative record number (RRN) of a record	RBA cannot change.	RBA can change.	RRN cannot change.
Space for adding records	Uses space at the end of the data set	Uses distributed free space for inserting records and changing their lengths in place	For fixed-length RRDS, uses empty slots in the data set  For variable-length RRDS, uses distributed free space and changes the lengths of added records in place
Space from deleting records	You cannot delete a record, but you can reuse its space for a record of the same length.	Space from a deleted or shortened record is automatically reclaimed in a control interval.	Space from a deleted record can be reused.
Spanned records	Can have spanned records	Can have spanned records	Cannot have spanned records
Reuse as work file	Can be reused unless it has an alternate index, is associated with key ranges, or exceeds 123 extents per volume	Can be reused unless it has an alternate index, is associated with key ranges, or exceeds 123 extents per volume	Can be reused

#### RELATED TASKS

“Specifying sequential organization for VSAM files” on page 138

“Specifying indexed organization for VSAM files” on page 138

“Specifying relative organization for VSAM files” on page 139

“Defining VSAM files under OS/390” on page 153

“Defining VSAM data sets under CMS” on page 158

---

## Defining VSAM file organization and records

Use the FILE-CONTROL entry in the ENVIRONMENT DIVISION to define the VSAM file organization and access modes for the files in your COBOL program.

In the FILE SECTION of the DATA DIVISION, code a file description (FD) entry for the file. In the associated record description entry or entries, define the *record-name* and record length. Code the logical size of the records with the RECORD clause.

**Important:** You can process VSAM data sets in COBOL for OS/390 & VM programs only after you define them with access method services.

The following table summarizes VSAM file organization, access modes, and record formats (fixed or variable length).

File organization	Sequential access	Random access	Dynamic access	Fixed length	Variable length
VSAM sequential (ESDS)	Yes	No	No	Yes	Yes
VSAM indexed (KSDS)	Yes	Yes	Yes	Yes	Yes
VSAM relative (RRDS)	Yes	Yes	Yes	Yes	Yes

#### RELATED TASKS

“Specifying sequential organization for VSAM files”

“Specifying indexed organization for VSAM files”

“Specifying relative organization for VSAM files” on page 139

“Using file status keys” on page 194

“Using VSAM return codes (VSAM files only)” on page 196

“Defining VSAM files under OS/390” on page 153

“Defining VSAM data sets under CMS” on page 158

“Specifying access modes for VSAM files” on page 141

## Specifying sequential organization for VSAM files

Identify VSAM ESDS files in your COBOL program with the ORGANIZATION IS SEQUENTIAL clause.

You can access (read or write) records in sequential files only sequentially.

After you place a record in the 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.

The following example shows typical FILE-CONTROL entries for a VSAM sequential file (ESDS):

```
SELECT S-FILE
  ASSIGN TO SEQUENTIAL-AS-FILE
  ORGANIZATION IS SEQUENTIAL
  ACCESS IS SEQUENTIAL
  FILE STATUS IS FSTAT-CODE VSAM-CODE.
```

#### RELATED CONCEPTS

“VSAM files” on page 136

## Specifying indexed organization for VSAM files

Identify VSAM KSDS in your COBOL program with the ORGANIZATION IS INDEXED clause.

Code a prime key for the record by using the clause:

```
RECORD KEY IS data-name
```

where *data-name* is the name of the key field as you defined it in the record description entry in the DATA DIVISION.

The following example shows the statements for a VSAM indexed file (KSDS) that is accessed dynamically. In addition to the primary key, COMMUTER-NO, there is an alternate key, LOCATION-NO:



```
SELECT I-FILE
  ASSIGN TO INDEXED-FILE
  ORGANIZATION IS INDEXED
  ACCESS IS DYNAMIC
  RECORD KEY IS IFILE-RECORD-KEY
  ALTERNATE RECORD KEY IS IFILE-ALTREC-KEY
  FILE STATUS IS FSTAT-CODE VSAM-CODE.
```

### Alternate keys

In addition to the primary key, you can also code one or more alternate keys to use for retrieving records. Using alternate keys, you can access the indexed file to read records in some sequence other than the prime key sequence. For example, you could access the file through employee department rather than through employee number. Alternate keys need not be unique. More than one record will be accessed, given a department number as a key. This is permitted if alternate keys are coded to allow duplicates.

You define the alternate key in your COBOL program with the ALTERNATE RECORD KEY clause:

```
ALTERNATE RECORD KEY IS data-name
```

Here *data-name* is the name of the key field as you defined it in the record description entry in the DATA DIVISION.

### Alternate index

To use an alternate index, you need to define a data set (using access method services) called the alternate index (AIX). The AIX contains one record for each value of a given alternate key; the records are in sequential order by alternate key value. Each record contains the corresponding primary keys of all records in the associated indexed files that contain the alternate key value.

#### RELATED CONCEPTS

“VSAM files” on page 136

#### RELATED TASKS

“Creating alternate indexes” on page 154

## Specifying relative organization for VSAM files

Identify VSAM RRDS files in your COBOL program with the ORGANIZATION IS RELATIVE clause.

Use the RELATIVE KEY IS clause to associate each logical record with its relative record number.

The following example shows a relative-record data set (RRDS) that is accessed randomly by the value in the relative key ITEM-NO:

```
SELECT R-FILE
  ASSIGN TO RELATIVE-FILE
  ORGANIZATION IS RELATIVE
  ACCESS IS RANDOM
  RELATIVE KEY IS RFILE-RELATIVE-KEY
  FILE STATUS IS FSTAT-CODE VSAM-CODE.
```

You can use a randomizing routine to associate a key value in each record with the relative record number for that record. Although there are many techniques to convert a record key to a relative record number, the most commonly used randomizing algorithm is the division/remainder technique. With this technique,

you divide the key by a value equal to the number of slots in the data set to produce a quotient and remainder. When you add one to the remainder, the result will be a valid relative record number.

Alternate indexes are not supported for VSAM RRDS.

### Fixed-length and variable-length RRDS

In an RRDS with fixed-length records, each record occupies one slot, and you store and retrieve records according to the relative record number of that slot. When you load the file, you have the option of skipping over slots and leaving them empty.

When you load an RRDS with variable-length records, you can skip over relative record numbers. Unlike fixed-length RRDS, a variable-length RRDS does not have slots. Instead, the free space that you define allows for more efficient record insertions.

VSAM variable-length RRDS is supported on OS/390 and VM/ESA.

### Simulating variable-length RRDS

If you cannot use VSAM variable-length RRDS or prefer not to, COBOL for OS/390 & VM provides another way for you to have relative-record data sets with variable-length records. This support, called *COBOL simulated variable-length RRDS*, is provided by the SIMVRD|NOSIMVRD run-time option. When you use the SIMVRD option, COBOL for OS/390 & VM simulates variable-length RRDS using a VSAM KSDS.

The coding you use in your COBOL program to identify and describe VSAM variable-length RRDS and COBOL simulated variable-length RRDS is similar. How you use the SIMVRD run-time option and whether you define the VSAM file as a RRDS or KSDS differs, however.

To use a variable-length RRDS, do the following steps, depending on the whether you want to simulate an RRDS:

Step	VSAM variable-length RRDS	COBOL simulated variable-length RRDS
1	Define the file in your COBOL program with the ORGANIZATION IS RELATIVE clause.	Same
2	Use FD statements in your COBOL program to describe the records with variable-length sizes.	Same
3	Use the NOSIMVRD run-time option.	Use the SIMVRD run-time option.
4	Define the VSAM file through access method services as an RRDS.	Define the VSAM file through access method services as a KSDS with a 4-byte key.

#### RELATED CONCEPTS

“VSAM files” on page 136

#### RELATED TASKS

“Defining VSAM files under OS/390” on page 153

## Specifying access modes for VSAM files

You can access records in VSAM sequential files only sequentially. You can access records in VSAM indexed and relative files in three ways: sequentially, randomly, or dynamically.

For sequential access, code `ACCESS IS SEQUENTIAL` in the `FILE-CONTROL` entry. Records in indexed files are then accessed in the order of the key field selected (either primary or alternate). Records in relative files are accessed in the order of the relative record numbers.

For random access, code `ACCESS IS RANDOM` in the `FILE-CONTROL` entry. Records in indexed files are then accessed according to the value you place in a key field. Records in relative files are accessed according to the value you place in the relative key.

For dynamic access, code `ACCESS IS DYNAMIC` in the `FILE-CONTROL` entry. Dynamic access is a mixed sequential-random access in the same program. Using dynamic access, you can write one program to perform both sequential and random processing, accessing some records in sequential order and others by their keys.

“Example: using dynamic access with VSAM files”

### RELATED TASKS

“Reading records from a VSAM file” on page 147

### Example: using dynamic access with VSAM files

Suppose that you have an indexed file of employee records and the employee’s hourly wage forms the record key. Your program is processing those employees who earn between \$10.00 and \$12.00 per hour and those who earn \$20.00 per hour and above.

Using dynamic access of VSAM files, the program would do as follows:

1. Retrieve the first record randomly (with a random-retrieval `READ`) based on the key of 1000.
2. Read sequentially (using `READ NEXT`) until the salary field exceeds 1200.
3. Retrieve the next record randomly, this time based on a key of 2000.
4. Read sequentially until the end of the file.

### RELATED TASKS

“Reading records from a VSAM file” on page 147

## Defining record lengths for VSAM files

VSAM records can be fixed or variable in length. COBOL determines the record format from the `RECORD` clause and the record descriptions associated with your `FD` entry for the file.

Because the concept of blocking has no meaning for VSAM files, you can omit the `BLOCK CONTAINS` clause. The clause is syntax-checked, but it has no effect on how the program runs.

## Defining fixed-length records

To define the records to be fixed length, use one of the following coding options:

RECORD clause	Clause format	Record length	Comments
Code RECORD CONTAINS <i>integer</i>	1	Fixed in size with a length of <i>integer-2</i>	The lengths of the level-01 record description entries associated with the file do not matter.
Omit the RECORD clause, but code all level-01 records (associated with the file) as the same size and none with an OCCURS DEPENDING ON clause.		The fixed size that you coded	

## Defining variable-length records

To define the records to be variable-length, use one of the following coding options:

RECORD clause	Clause format	Maximum record length	Comments
Code RECORD IS VARYING FROM <i>integer-1</i> TO <i>integer-2</i>	3	<i>integer-2</i> value	The lengths of the level-01 record description entries associated with the file do not matter.
Code RECORD IS VARYING	3	Size of the largest level-01 record description entry associated with the file	The compiler determines the maximum record length.
Code RECORD CONTAINS <i>integer-1</i> TO <i>integer-2</i>	2	<i>integer-2</i> value	The minimum record length is the <i>integer-2</i> value.
Omit the RECORD clause, but code multiple level-01 records (associated with the file) that are of different sizes or that contain an OCCURS DEPENDING ON clause.		Size of the largest level-01 record description entry associated with the file	The compiler determines the maximum record length.

The RECORD clause is sensitive to the CMPR2 compiler option.

When you specify a READ INTO statement for a format-V file, the record size read for that file is used in the MOVE statement generated by the compiler. Consequently, you might not get the result you expect if the record read in does not correspond to the level-01 record description. All other rules of the MOVE statement apply. For example, when you specify a MOVE statement for a format-V record read in by the READ statement, the size of the record corresponds to its level-01 record description.

### RELATED TASKS

*COBOL for OS/390 & VM Compiler and Run-Time Migration Guide*

---

## Coding input and output statements for VSAM files

Use these COBOL statements for processing VSAM files:

**OPEN** To connect the VSAM data set to your COBOL program for processing.

**WRITE** To add records to a file or load a file.

**START** To establish the current location in the cluster for a READ NEXT statement.

START does not retrieve a record; it only sets the current record pointer.

**READ and READ NEXT**

To retrieve records from a file.

**REWRITE**

To update records.

**DELETE** To logically remove records from indexed and relative files only.

**CLOSE** To disconnect the VSAM data set from your program.

All of the following factors determine which input and output statements you can use for a given VSAM data set:

- Access mode (sequential, random, or dynamic)
- File organization (ESDS, KSDS, or RRDS)
- Mode of OPEN statement (INPUT, OUTPUT, I-0, or EXTEND)

The following table shows the possible combinations with sequential files (ESDS). The X indicates that you can use the statement with the open mode at the top of the column.

Access mode	COBOL statement	OPEN INPUT	OPEN OUTPUT	OPEN I-0	OPEN EXTEND
Sequential	OPEN	X	X	X	X
	WRITE		X		X
	START				
	READ	X		X	
	REWRITE			X	
	DELETE				
	CLOSE	X	X	X	X

The following table shows the possible combinations you can use with indexed (KSDS) files and relative (RRDS) files. The X indicates that you can use the statement with the open mode 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	
Dynamic	OPEN	X	X	X	
	WRITE		X	X	
	START	X		X	
	READ	X		X	
	REWRITE			X	
	DELETE			X	
	CLOSE	X	X	X	

The fields you code in the FILE STATUS clause are updated by VSAM after each input-output statement to indicate the success or failure of the operation.

**RELATED CONCEPTS**

“File position indicator”

**RELATED TASKS**

“Opening a file (ESDS, KSDS, or RRDS)” on page 145

“Reading records from a VSAM file” on page 147

“Updating records in a VSAM file” on page 148

“Adding records to a VSAM file” on page 149

“Replacing records in a VSAM file” on page 149

“Deleting records from a VSAM file” on page 150

“Closing VSAM files” on page 150

## 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, and READ NEXT statements. Subsequent READ or READ NEXT 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.

#### RELATED TASKS

“Reading records from a VSAM file” on page 147

## Opening a file (ESDS, KSDS, or RRDS)

Before you can use any WRITE, START, READ, REWRITE, or DELETE statements to process records in a file, you must first open the file with an OPEN statement. File availability and creation affect OPEN processing, optional files, and file status codes 05 and 35.

For example, if you OPEN EXTEND, OPEN I-0, or OPEN INPUT a file that is neither optional nor available, you get file status 35 and the OPEN statement fails. If the file is OPTIONAL, the OPEN EXTEND, OPEN I-0, or OPEN INPUT creates the file and returns file status 05.

An OPEN operation works successfully only when you set fixed file attributes in the DD statement or data set label for a file and specify consistent attributes for that file in the SELECT and FD statements of your COBOL program. Mismatches in the following items result in a file status code 39, and the OPEN statement fails:

- Attributes for file organization (sequential, relative, or indexed)
- Prime record key
- Alternate record keys
- Maximum record size
- Record type (fixed or variable)

How you code the OPEN statement in your COBOL program for a VSAM file depends on whether the file is empty (a file that has never contained records) or loaded. For either type of file, your program should check the file status key after each OPEN statement.

#### RELATED TASKS

“Opening an empty file”

“Opening a loaded file (a file with records)” on page 147

#### RELATED REFERENCES

“Statements to load records into a VSAM file” on page 147

### Opening an empty file

To open a file that has never contained records (an empty file), use the following statements depending on the type of file:

- OPEN OUTPUT for ESDS files.
- OPEN OUTPUT or OPEN EXTEND for KSDS and RRDS files. (Either coding has the same effect.) If you have coded the file for random or dynamic access and the file is optional, you can use OPEN I-0.

Optional files are files that are not necessarily present each time the program is run. You can define files opened in INPUT, I-0, or OUTPUT mode as optional by defining them with the SELECT OPTIONAL phrase in the FILE-CONTROL section of your program.

**Initially loading records sequentially into a file:** Initially loading a file means writing records into the file for the first time. This is not the same as writing records into a file from which all previous records have been deleted.

To initially load a VSAM file:

1. Open the file.
2. Use sequential processing (ACCESS IS SEQUENTIAL) because it is faster.
3. Use WRITE to add a record to the file.

Using OPEN OUTPUT to load a VSAM file significantly improves the performance of your program. Using OPEN I-O or OPEN EXTEND has a negative impact on the performance of your program.

When you load VSAM indexed files sequentially, you optimize both loading performance and subsequent processing performance, because sequential processing maintains user-defined free space. Future insertions will be more efficient.

With ACCESS IS SEQUENTIAL, you must write the records in ascending RECORD KEY order.

When you load VSAM relative files sequentially, the records are placed in the file in the ascending order of relative record numbers.

**Initially loading a file randomly or dynamically:** You can use random or dynamic processing to load a file, but they are not as efficient as sequential processing. Because VSAM does not support random or dynamic processing, COBOL has to perform some extra processing to enable you to use ACCESS IS RANDOM or ACCESS IS DYNAMIC with OPEN OUTPUT or OPEN I-O. These steps prepare the file for use and give it the status of a loaded file, having been used at least once.

In addition to extra overhead for preparing files for use, random processing does not consider any user-defined free space. As a result, any future insertions might be inefficient. Sequential processing maintains user-defined free space.

**Loading a VSAM data set with access method services:** You can load or update a VSAM data set with the access method services REPRO command. Use REPRO whenever possible.

#### RELATED TASKS

“Opening a loaded file (a file with records)” on page 147

#### RELATED REFERENCES

“Statements to load records into a VSAM file” on page 147

REPRO (*OS/390 DFSMS: Access Method Services for Catalogs*)



## Statements to load records into a VSAM file

Division	ESDS	KSDS	RRDS
ENVIRONMENT DIVISION	SELECT ASSIGN FILE STATUS PASSWORD ACCESS MODE	SELECT ASSIGN ORGANIZATION IS INDEXED RECORD KEY ALTERNATE RECORD KEY FILE STATUS PASSWORD ACCESS MODE	SELECT ASSIGN ORGANIZATION IS RELATIVE RELATIVE KEY FILE STATUS PASSWORD ACCESS MODE
DATA DIVISION	FD entry	FD entry	FD entry
PROCEDURE DIVISION	OPEN OUTPUT OPEN EXTEND WRITE CLOSE	OPEN OUTPUT OPEN EXTEND WRITE CLOSE	OPEN OUTPUT OPEN EXTEND WRITE CLOSE

### RELATED TASKS

“Opening an empty file” on page 145

“Updating records in a VSAM file” on page 148

### Opening a loaded file (a file with records)

To open a file that already contains records, use OPEN INPUT, OPEN I-0, or OPEN EXTEND.

If you open a VSAM entry-sequenced or relative-record file as EXTEND, the added records are placed after the last existing records in the file.

If you open a VSAM key-sequenced file as EXTEND, each record you add must have a record key higher than the highest record in the file.

### RELATED TASKS

“Opening an empty file” on page 145

“Working with VSAM data sets under OS/390 and OS/390 UNIX” on page 152

“Defining VSAM data sets under CMS” on page 158

### RELATED REFERENCES

“Statements to load records into a VSAM file”

*Access Method Services Reference*

## Reading records from a VSAM file

Use the READ statement to retrieve (READ) records from a file. To read a record, you must have opened the file INPUT or I-0. Your program should check the file status key after each READ.

You can retrieve records in VSAM sequential files only in the sequence in which they were written.

You can retrieve records in VSAM indexed and relative record files in any of the following ways:

### Sequentially

According to the ascending order of the key you are using, the RECORD KEY

or the ALTERNATE RECORD KEY, beginning at the current position of the file position indicator for indexed files, or according to ascending relative record locations for relative files.

**Randomly**

In any order, depending on how you set the RECORD KEY or ALTERNATE RECORD KEY or the RELATIVE KEY before your READ request.

**Dynamically**

Mixed sequential and random.

With dynamic access, you can switch between reading a specific record directly and reading records sequentially, by using READ NEXT 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 statement to set the file position indicator to point to a particular record. When you code START followed by READ NEXT, the next record is read and the file position indicator is reset to the next record. You can move the file position indicator randomly by using START, but all reading is done sequentially from that point.

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

When a direct READ is performed for a VSAM indexed file, based on an alternate index for which duplicates exist, only the first record in the data set (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 code of 02 is returned if there are more records with the same alternate key value to be read; a code of 00 is returned when the last record with that key value has been read.

RELATED CONCEPTS

“File position indicator” on page 144

RELATED TASKS

“Specifying access modes for VSAM files” on page 141

## Updating records in a VSAM file

To update a VSAM file, use the ENVIRONMENT DIVISION and DATA DIVISION statements to load records into a VSAM file and the following PROCEDURE DIVISION statements to update VSAM files.

Access method	ESDS	KSDS	RRDS
ACCESS IS SEQUENTIAL	OPEN EXTEND WRITE CLOSE <i>or</i> OPEN I-0 READ REWRITE CLOSE	OPEN EXTEND WRITE CLOSE <i>or</i> OPEN I-0 READ REWRITE DELETE CLOSE	OPEN EXTEND WRITE CLOSE <i>or</i> OPEN I-0 READ REWRITE DELETE CLOSE

Access method	ESDS	KSDS	RRDS
ACCESS IS RANDOM	Not applicable	OPEN I-0 READ WRITE REWRITE DELETE CLOSE	OPEN I-0 READ WRITE REWRITE DELETE CLOSE
ACCESS IS DYNAMIC (sequential processing)	Not applicable	OPEN I-0 READ NEXT WRITE REWRITE START DELETE CLOSE	OPEN I-0 READ NEXT WRITE REWRITE START DELETE CLOSE
ACCESS IS DYNAMIC (random processing)	Not applicable	OPEN I-0 READ WRITE REWRITE DELETE CLOSE	OPEN I-0 READ WRITE REWRITE DELETE CLOSE

#### RELATED REFERENCES

“Statements to load records into a VSAM file” on page 147

## Adding records to a VSAM file

Use the COBOL WRITE statement to add a record to a file without replacing any existing records. The record to be added must not be larger than the maximum record size that you set when you defined the file. Your program should 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 VSAM file that has been opened with either OUTPUT or EXTEND.

Sequential files are always written sequentially.

For indexed files, you must write new records in ascending key sequence. If you open the file EXTEND, the record keys of the records to be added must be higher than the highest primary record key on the file when you opened the file.

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 and ACCESS IS RANDOM or ACCESS IS DYNAMIC, you can write the records in any order.

## Replacing records in a VSAM file

To replace records in a VSAM file, use REWRITE on a file that you have opened for I-0. If you try to use REWRITE on a file that is not opened I-0, the record is not rewritten and the status key is set to 49. Your program should check the file status key after each REWRITE statement.

- For sequential files, the length of the record you rewrite must be the same as the length of the original record.
- For indexed files, you can change the length of the record you rewrite.
- For variable-length relative files, you can change the length of the record you rewrite.

To replace records randomly or dynamically, your program need not read the record to be rewritten. Instead, to position the record that you want to update, do 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.

## Deleting records from a VSAM file

Open the file I-0 and use the DELETE statement to remove an existing record from an indexed or relative file. You cannot use DELETE on a sequential file.

When you use ACCESS IS SEQUENTIAL or the file contains spanned records, your program must first read the record to be deleted. The DELETE then removes the record that was read. If the DELETE is not preceded by a successful READ, the deletion is not done and the status key value is set to 92.

When you use ACCESS IS RANDOM or ACCESS IS DYNAMIC, your program need not first read the record to be deleted. To delete a record, move the key of the record to be deleted to the RECORD KEY data item and then issue the DELETE. Your program should check the file status key after each DELETE statement.

## Closing VSAM files

Use the CLOSE statement to disconnect your program from the VSAM file. If you try to close a file that is already closed, you will get a logic error. Check the file status key after each CLOSE statement.

If you do not close a VSAM file, the file is automatically closed for you under the following conditions:

- When the run unit ends normally, all open files defined in any COBOL for OS/390 & VM, COBOL for MVS & VM, or VS COBOL II programs in the run unit are closed.
- When the run unit ends abnormally, if the TRAP(ON) run-time option has been set, all open files defined in any COBOL for OS/390 & VM, COBOL for MVS & VM, or VS COBOL II programs in the run unit are closed.
- When Language Environment condition handling is completed and the application resumes in a routine other than where the condition occurred, open files defined in any COBOL for OS/390 & VM, COBOL for MVS & VM, or VS COBOL II programs in the run unit that might be called again and reentered are closed.

You can change the location where a program resumes after a condition is handled. To make this change, you can, for example, move the resume cursor with the CEEMRCR callable service or use HLL language constructs such as a C longjmp statement.

- When you issue CANCEL for a COBOL for OS/390 & VM, COBOL for MVS & VM, or VS COBOL II subprogram, any open nonexternal files defined in that program are closed.

- When a COBOL for OS/390 & VM, COBOL for MVS & VM, or VS COBOL II subprogram with the INITIAL attribute returns control, any open nonexternal files defined in that program are closed.

File status codes are set when these implicit CLOSE operations are performed, but EXCEPTION/ERROR declaratives are not invoked.

---

## Handling errors in VSAM files

When an input statement or output statement operation fails, COBOL does not perform corrective action for you.

All OPEN and CLOSE errors with a VSAM file, whether logical errors in your program or input/output errors on the external storage media, return control to your COBOL program, even when you have coded no DECLARATIVE and no FILE STATUS clause.

If any other input or output statement operation fails, you choose whether your program will continue running after a less-than-severe input/output error occurs.

COBOL provides these ways for you to intercept and handle certain VSAM input and output errors:

- End-of-file phrase (AT END)
- EXCEPTION/ERROR declarative
- FILE STATUS clause (file status key and VSAM return code)
- INVALID KEY phrase

You should define a status key for each VSAM file that you define in your program. Check the status key value after every input or output request, especially OPEN and CLOSE.

If you do not code a FILE STATUS key or a declarative, serious VSAM processing errors will cause a message to be issued and a Language Environment condition to be signaled, which will cause an abend if you specify the run-time option ABTERMENC (ABEND).

### RELATED TASKS

“Handling errors in input and output operations” on page 191

“Using VSAM return codes (VSAM files only)” on page 196

### RELATED REFERENCES

*OS/390 DFSMS: Macro Instructions for Data Sets*

---

## Protecting VSAM files with a password

Although the preferred security mechanism on an OS/390 system is RACF, COBOL for OS/390 & VM also supports using explicit passwords on VSAM files to prevent unauthorized access and update.

To use explicit passwords, code the PASSWORD clause in the SELECT statement of your program. Use this clause only if the catalog entry for the files includes a read or an update password.

- If the catalog entry includes a read password, you cannot open and access the file in a COBOL program unless you use the password clause in the FILE-CONTROL paragraph and describe it in the DATA DIVISION. The *data-name* referred to must contain a valid password when the file is opened.

- If the catalog entry includes an update password, you can open and access it, but not update it, unless you code the password clause in the FILE-CONTROL paragraph and describe it in the DATA DIVISION.
- If the catalog entry includes both a read password and an update password, specify the update password to both read and update the file in your program.

If your program only retrieves records and does not update them, you need only the read password. If your program loads files or updates them, you need to specify the update password that was cataloged.

For indexed files, the PASSWORD data item for the RECORD KEY must contain the valid password before the file can be successfully opened.

If you password-protect a VSAM indexed file, you must also password-protect every alternate index in order to be fully password-protected. Where you place the PASSWORD clause becomes important because each alternate index has its own password. The PASSWORD clause must directly follow the key clause to which it applies.

“Example: password protection for a VSAM indexed file”

### Example: password protection for a VSAM indexed file

The following example shows the COBOL code used for a VSAM indexed file with password protection.

```

. . .
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT LIBFILE
        ASSIGN TO PAYMAST
        ORGANIZATION IS INDEXED
        RECORD KEY IS EMPL-NUM
        PASSWORD IS BASE-PASS
        ALTERNATE RECORD KEY IS EMPL-PHONE
        PASSWORD IS PATH1-PASS
. . .
WORKING-STORAGE SECTION.
01 BASE-PASS          PIC X(8) VALUE "25BSREAD".
01 PATH1-PASS        PIC X(8) VALUE "25ATREAD".

```

---

## Working with VSAM data sets under OS/390 and OS/390 UNIX

There are some special considerations for VSAM files under OS/390 and OS/390 UNIX in terms of coding access method services (IDCAMS) commands, environment variables, and JCL.

A VSAM file is *available* if all of the following are true:

- You define it using access method services.
- You define it for your program by providing a DD statement, an environment variable, or an ALLOCATE command for it.
- It has previously contained a record.

A VSAM file is *unavailable* if it has never contained a record, even if you have defined it.

You always get a return code of zero on completion of the OPEN statement for a VSAM sequential file.

Use the access method services REPRO command to empty a file. Deleting records in this manner resets the high-use relative byte address (RBA) of the file to zero. The file is effectively empty and appears to COBOL as if it never contained a record.

#### RELATED TASKS

“Defining files to the operating system” on page 10

“Defining VSAM files under OS/390”

“Creating alternate indexes” on page 154

“Allocating VSAM files” on page 156

“Sharing VSAM files through RLS” on page 157

## Defining VSAM files under OS/390

You can process VSAM entry-sequenced, key-sequenced, and relative-record data sets in COBOL for OS/390 & VM only after you define them through access method services (IDCAMS).

A VSAM *cluster* is a logical definition for a VSAM data set and has one or two components:

- The data component of a VSAM cluster contains the data records.
- The index component of a VSAM key-sequenced cluster consists of the index records.

Use the access method services DEFINE CLUSTER command to define your VSAM data sets (clusters). This process includes creating an entry in an integrated catalog without any data transfer.

Define the following information about the cluster:

- Name of the entry
- Name of the catalog to contain this definition and its password (can use default name).
- Organization (sequential, indexed, or relative)
- Device and volumes that the data set will occupy
- Space required for the data set
- Record size and control interval sizes (CISIZE)
- Passwords (if any) required for future access

Depending on what kind of data set is in the cluster, also define the following information for each cluster:

- For VSAM indexed data sets (KSDS), specify length and position of the prime key in the records.
- For VSAM fixed-length relative-record data sets (RRDS), specify the record size as greater than or equal to the maximum size COBOL record:

```
DEFINE CLUSTER NUMBERED  
RECORDSIZE(n,n)
```

When you define a data set in this way, all records will be padded to the fixed slot size *n*. If you use the RECORD IS VARYING ON *data-name* form of the RECORD clause, a WRITE or REWRITE will use the length specified in the DEPENDING ON *data-name* as the length of the record to be transferred by VSAM. This data is then padded to the fixed slot size. READ statements always return the fixed slot size in the DEPENDING ON *data-name*.

- For VSAM variable-length relative-record data sets (RRDS), specify the average size COBOL record expected and the maximum size COBOL record expected:

```
DEFINE CLUSTER NUMBERED
RECORDSIZE(avg,m)
```

The average size COBOL record expected must be less than the maximum size COBOL record expected.

- For COBOL simulated variable-length relative-record data sets, specify the average size of the COBOL records and a size that is greater than or equal to the maximum size COBOL record plus 4:

```
DEFINE CLUSTER INDEXED
KEYS(4,0)
RECORDSIZE(avg,m)
```

The average size COBOL record expected must be less than the maximum size COBOL record expected.

#### RELATED TASKS

“Creating alternate indexes”

“Allocating VSAM files” on page 156

“Specifying relative organization for VSAM files” on page 139

#### RELATED REFERENCES

*OS/390 DFSMS: Access Method Services for Catalogs*

*Access Method Services Reference*

## Creating alternate indexes

An alternate index provides access to the records in a data set using more than one key. It accesses records in the same way as the prime index key of an indexed data set (KSDS).

When planning to use an alternate index, you must know:

- The type of data set (base cluster) with which the index will be associated
- Whether the keys will be unique or not unique
- Whether the index is to be password protected
- Some of the performance aspects of using alternate indexes

Because an alternate index is, in practice, a VSAM data set that contains pointers to the keys of a VSAM data set, you must define the alternate index and the alternate index path (the entity that establishes the relationship between the alternate index and the prime index). After you define an alternate index, make a catalog entry to establish the relationship (or path) between the alternate index and its base cluster. This path allows you to access the records of the base cluster through the alternate keys.

To use an alternate index, you need to follow these steps:

1. Define the alternate index by using the `DEFINE ALTERNATEINDEX` command. In it, define the following:
  - Name of the alternate index
  - Name of its related VSAM indexed data set
  - Location in the record of any alternate indexes and whether they are unique or not
  - Whether or not alternate indexes are to be updated when the data set is changed
  - Name of the catalog to contain this definition and its password (can use default name)



In your COBOL program the alternate index is identified solely by the ALTERNATE RECORD KEY clause of the FILE CONTROL paragraph. The ALTERNATE RECORD KEY definitions must match the definitions that you have made in the catalog entry. Any password entries that you have cataloged should be coded directly after the ALTERNATE RECORD KEY phrase.

2. Relate the alternate index to the base cluster (the data set to which the alternate index gives you access) by using the DEFINE PATH command. In it, define the following:
  - Name of the path
  - Alternate index to which the path is related
  - Name of the catalog that contains the alternate index

The base cluster and alternate index are described by entries in the same catalog.

3. Load the VSAM indexed data set.
4. Build the alternate index by using (typically) the BLDINDEX command. Identify the input file as the indexed data set (base cluster) and the output file as the alternate index or its path. This command BLDINDEX reads all the records in your VSAM indexed data set (or base cluster) and extracts the data needed to build the alternate index.

Alternatively, you can use the run-time option AIXBLD to build the alternate index at run time. However, this option might adversely affect run-time performance.

“Example: entries for alternate indexes”

#### RELATED REFERENCES

AIXBLD (*Language Environment Programming Reference*)

### Example: entries for alternate indexes

The following example maps the relationships between the COBOL FILE-CONTROL entry and the DD statements or environment variables for a VSAM indexed file with two alternate indexes.

Using JCL:

```
//MASTERA DD DSN=clustname,DISP=OLD (1)
//MASTERA1 DD DSN=path1,DISP=OLD (2)
//MASTERA2 DD DSN=path2,DISP=OLD (3)
```

Using environment variables:

```
export MASTERA=DSN(clustname),OLD (1)
export MASTERA1=DSN(path1),OLD (2)
export MASTERA2=DSN(path2),OLD (3)
```

```
. . .
FILE-CONTROL.
  SELECT MASTER-FILE ASSIGN TO MASTERA (4)
  RECORD KEY IS EM-NAME
  PASSWORD IS PW-BASE (5)
  ALTERNATE RECORD KEY IS EM-PHONE (6)
  PASSWORD IS PW-PATH1
  ALTERNATE RECORD KEY IS EM-CITY (7)
  PASSWORD IS PW-PATH2.
```

- (1) The base cluster name is *clustname*.
- (2) The name of the first alternate index path is *path1*.
- (3) The name of the second alternate index path is *path2*.

- (4) The ddname or environment variable name for the base cluster is specified with the ASSIGN clause.
- (5) Passwords immediately follow their indexes.
- (6) The key EM-PHONE relates to the first alternate index.
- (7) The key EM-CITY relates to the second alternate index.

RELATED TASKS

“Creating alternate indexes” on page 154

## Allocating VSAM files

You must predefine and catalog all VSAM data sets through the access method services DEFINE command. Most of the information about a VSAM data set is in the catalog. You need to specify only minimal DD or environment variable information for a VSAM file. When you use an environment variable, the name must be in uppercase. Usually the input and data buffers are the only variables that you are concerned about.

Allocation of VSAM files (indexed, relative, and sequential) follows the general rules for the allocation of COBOL files. If you use an environment variable to allocate a VSAM file, you must specify these options in the order shown, but no others:

- DSN(*dsname*), where *dsname* is the name of the base cluster
- OLD or SHR

The basic DD statement that you need for your VSAM files is:

```
//ddname DD DSN=dsname,DISP=SHR,AMP=AMORG
```

The corresponding export command is:

```
export evname="DSN(dsname),SHR"
```

In either case, *dsname* must be the same as the name used in the access method services DEFINE CLUSTER or DEFINE PATH command. DISP must be OLD or SHR because the data set is already cataloged. If you specify MOD when using JCL, the data set is treated as OLD.

AMP is a VSAM JCL parameter used to supplement the information that the program supplies about the data set. AMP takes effect when your program opens the VSAM file. Any information that you set through the AMP parameter takes precedence over the information that is in the catalog or that the program supplies.

The AMP parameter is not required except under the following circumstances:

- You use a dummy VSAM data set. For example,
 

```
//ddname DD DUMMY,AMP=AMORG
```
- You request additional index or data buffers. For example,
 

```
//ddname DD DSN=VSAM.dsname,DISP=SHR,
// AMP=('BUFNI=4,BUFND=8')
```

You cannot specify AMP if you allocate your VSAM data set with an environment variable.

For a VSAM base cluster, specify the same system name (ddname or or environment variable name) that you specify in the ASSIGN clause of the SELECT statement in your COBOL program.

When you use alternate indexes in your COBOL program, you must specify not only a system name (using a DD statement or environment variable) for the base cluster, but also one for each alternate index path. No language mechanism exists to explicitly declare system names for alternate index paths within the program. Therefore, you must adhere to the following guidelines for forming the system name (ddname or environment variable name) for each alternate index path:

- Concatenate the base cluster name with an integer.
- Begin with 1 for the path associated with the first alternate record defined for the file in your program (ALTERNATE RECORD KEY clause of the SELECT statement).
- Increment by 1 for the path associated with each successive alternate record definition for that file.

For example, if the system name of a base cluster is ABCD, the system name for the first alternate index path defined for the file in your program is ABCD1, the system name for the second alternate index path is ABCD2, and so on.

If the length of the base cluster system name and sequence number exceeds eight characters, the base cluster portion of the system name is truncated on the right to reduce the concatenated result to eight characters. For example, if the system name of a base cluster is ABCDEFGH, the system name of the first alternate index path is ABCDEFG1, the tenth is ABCDEF10, and so on.

#### RELATED TASKS

“Allocating files” on page 105

#### RELATED REFERENCES

*OS/390 MVS JCL Reference*

## Sharing VSAM files through RLS

By using the VSAM JCL parameter RLS, you can specify the use of record level sharing with VSAM. Use RLS=CR when consistent read protocols are required, and RLS=NRI when no read integrity protocols are required. Specifying the RLS parameter is the only way to request the RLS mode when running COBOL programs.

You cannot specify RLS if you allocate your VSAM data set with an environment variable.

### Preventing update problems with VSAM files in RLS mode

When a VSAM data set is opened in RLS mode for I-O (updates), the first READ causes an exclusive lock of the control interval that contains the record, regardless of RLS=CR or RLS=NRI that you specify. The exclusive lock is released after a WRITE or REWRITE statement is issued or another READ statement is issued for another record.

Specifying RLS=CR locks a record and prevents an update to it until another READ is requested for another record. While a lock on the record being read is in effect, other users can request a READ for the same record, but they cannot update the record until the read lock is released. When you specify RLS=NRI, no lock will be in effect when a READ for input is issued and another user might update the record.

The locking rules for RLS=CR can cause the application to wait for availability of a record lock, and this wait might slow down the READ for input.

You might need to modify your application logic to use the RLS=CR capability. Do not use the RLS JCL parameter for batch jobs that update nonrecoverable spheres until you are sure that the application functions correctly in a multiple updater environment.

When you open a VSAM data set in RLS mode for INPUT or I-O processing, it is a good idea to issue an OPEN or START *immediately* before a READ. If there is a delay between the OPEN or START and the actual READ, another user might add records before the record on which the application is positioned after the OPEN or START. The COBOL run time points explicitly to the beginning of the VSAM data set at the time when OPEN was requested, but another user might add records that would alter the true beginning of the VSAM data set if the READ is delayed.

### Restrictions when using RLS

The following restrictions apply to RLS mode:

- The VSAM cluster attributes KEYRANGE and IMBED are not supported when you open a VSAM file in RLS mode.
- The VSAM cluster attribute REPLICATE is not recommended with RLS mode because the benefits are negated by the system-wide buffer pool and potentially large CF cache structure in the storage hierarchy.
- The AIXBLD run-time option is not supported when you open a VSAM file in RLS mode, because VSAM does not allow an empty path to be opened. If you need the AIXBLD run-time option to build the alternate index data set, open the VSAM data set in non-RLS mode.
- Temporary data sets are not allowed in RLS mode.
- The SIMVRD run-time option is not supported for VSAM files opened in RLS mode.

### Handling errors in VSAM files in RLS mode

If your application accesses a VSAM data set in RLS mode, be sure to check the file status and VSAM feedback codes after *each* request.

If your application encounters “SMSVSAM server not available” while processing input or output, explicitly close the VSAM file before you try to open it again. VSAM generates return code 16 for failures like “SMSVSAM server not available,” and there is no feedback code. You can have your COBOL programs check the first two bytes of the second file status area for a VSAM return code 16.

The COBOL run time generates message IGZ0205W and automatically closes the file if the error occurs during OPEN processing.

All other RLS mode errors return a VSAM return code of 4, 8, or 12.

---

## Defining VSAM data sets under CMS

You can run COBOL programs that read and write VSAM files in CMS. The VSAM used is VSE VSAM (not OS/390 VSAM). Nevertheless, the VSAM files written in CMS have the same format as those created under OS/390 and are fully compatible with OS/390 VSAM data sets. VSAM files created in CMS can be read by an OS/390 system, and vice versa.

COBOL coding requirements are the same whether OS/390 or VSE VSAM is used. However, instead of using the FILEDEF command to identify the VSAM files, you must use the DLBL command. It has the same basic format as the FILEDEF command. For example:

```
DLBL INPUT C DSN COBTEST DATA (VSAM
```

This file might be identified in your program as follows:

```
FILE-CONTROL.  
  SELECT INVSAM  
  ASSIGN TO INPUT  
  ORGANIZATION IS INDEXED. . .  
FD INVSAM  
. . .
```

If you are running a program that uses a VSAM file and a non-VSAM file, use the DLBL command to identify the VSAM file and the FILEDEF command to identify your non-VSAM file. You can set additional options if the VSAM data set is a multivolume file or if it is cataloged in a user catalog. If you use any of these special options, you do not need to use the VSAM option.

A special ddname is provided for you to identify the catalog that you use during a terminal session:

```
DLBL IJSYSCT F DSN MASTCAT (PERM
```

Entering this command makes the catalog available to you for the remainder of your terminal session.

When a program abends or an HX Immediate command is issued while the program is running, CMS automatically clears all DLBL and FILEDEF definitions, even if PERM has been issued.

You must use access method services (CMS command AMSERV) to define catalogs, data spaces, and clusters, and to perform functions such as REPRO, EXPORT, IMPORT, and LISTCAT. Notice that the commands for these functions under CMS are VSE/VSAM commands.

If a file with alternate indexes is open for input or output, you must define it with SHAREOPTIONS(4).

#### RELATED REFERENCES

DLBL command (*VM/ESA CMS User's Guide*)  
AMSERV command (*CMS Command Reference*)  
*VSE/VSAM Commands and Macros*

---

## Improving VSAM performance

Most likely, your system programmer is responsible for tuning the performance of COBOL and VSAM. As an application programmer, you can control the aspects of VSAM listed in this table.

Aspect of VSAM	What you can do	Rationale and comments
Invoking access methods service	Build your alternate indexes in advance, using IDCAMS (for OS/390) or AMSERV (for CMS).	

Aspect of VSAM	What you can do	Rationale and comments
Buffering	<p>For sequential access, request more data buffers; for random access, request more index buffers. Specify both BUFND and BUFNI when ACCESS IS DYNAMIC.</p> <p>Avoid coding additional buffers unless your application will run interactively; and then code buffers only when response-time problems arise that might be caused by delays in input and output.</p>	The default is one index (BUFNI) and two data buffers (BUFND).
Loading records, using access methods services	<p>Use the access methods service REPRO command when:</p> <ul style="list-style-type: none"> <li>• The target indexed data set already contains records.</li> <li>• The input sequential data set contains records to be updated or inserted into the indexed data set.</li> </ul> <p>If you use a COBOL program to load the file, use OPEN OUTPUT and ACCESS SEQUENTIAL.</p>	The REPRO command can update an indexed data set as fast or faster than any COBOL program under these conditions.
File access modes	For best performance, access records sequentially.	Dynamic access is less efficient than sequential access, but more efficient than random access. Random access results in increased EXCPs because VSAM must access the index for each request.
Key design	Design the key in the records so that the high-order portion is relatively constant and the low-order portion changes often.	This method compresses the key best.
Multiple alternate indexes	Avoid using multiple alternate indexes.	Updates must be applied through the primary paths and are reflected through multiple alternate paths, perhaps slowing performance,
Relative file organization	Use VSAM fixed-length relative data sets rather than VSAM variable-length relative data sets.	Although not as space efficient, VSAM fixed-length relative data sets are more run-time efficient than VSAM variable-length relative data sets, which have performance characteristics comparable to COBOL simulated relative data sets.

Aspect of VSAM	What you can do	Rationale and comments
Control interval sizes (CISZ)	<p>Provide your system programmer with information about the data access and future growth of your VSAM data sets. From this information, your system programmer can determine the best control interval size (CISZ) and FREESPACE size (FSPC).</p> <p>Choose proper values for CISZ and FSPC to minimize control area (CA) splits. You can diagnose the current number of CA splits by issuing the LISTCAT ALL command on the cluster, and then compress (using EXPORT or IMPORT or REPRO) the cluster to omit all CA splits periodically.</p>	<p>VSAM calculates CISZ to best fit the direct-access storage device (DASD) usage algorithm, which might not, however, be efficient for your application.</p> <p>An average CISZ of 4K is suitable for most applications. A smaller CISZ means faster retrieval for random processing at the expense of inserts (that is, more CISZ splits and therefore more space in the data set). A larger CISZ results in the transfer of more data across the channel for each READ. This is more efficient for sequential processing, similar to a large OS BLKSIZE.</p> <p>Many control area (CA) splits are unfavorable for VSAM performance. The FREESPACE value can affect CA splits, depending on how the file is used.</p>

#### RELATED TASKS

“Specifying access modes for VSAM files” on page 141

Deciding how big a virtual resource pool to provide (*OS/390 DFSMS: Using Data Sets*)

Selecting the optimal percentage of free space (*OS/390 DFSMS: Using Data Sets*)

#### RELATED REFERENCES

AMSERV command (*CMS Command Reference*)

*Access Method Services Reference*





---

## Chapter 10. Processing line-sequential files

Line-sequential files are files that reside in the hierarchical file system (HFS) and that contain only printable characters and certain control characters as data. Each record ends with an EBCDIC new-line character (X'15'), which is not included in the length of the record. Because these are sequential files, records are placed one after another, according to entry order. Your program can process these files only sequentially, retrieving (with the READ statement) records in the same order as they are in the file. A new record is placed after the preceding record.

To process line-sequential files in your program, use COBOL language statements that:

- Identify and describe the files in the ENVIRONMENT DIVISION and the DATA DIVISION.
- Process the records in the files in the PROCEDURE DIVISION.

After you have created a record, you cannot change its length or its position in the file, and you cannot delete it.

### RELATED CONCEPTS

*OS/390 UNIX System Services User's Guide*

### RELATED TASKS

"Defining line-sequential files and records in COBOL"

"Describing the structure of a line-sequential file" on page 164

"Coding input-output statements for line-sequential files" on page 165

"Handling errors in line-sequential files" on page 168

"Defining and allocating line-sequential files" on page 165

### RELATED REFERENCES

"Allowable control characters" on page 164

---

## Defining line-sequential files and records in COBOL

Use the FILE-CONTROL entry in the ENVIRONMENT DIVISION to:

- Define the files in your COBOL program as line-sequential files.
- Associate them with the external file names (*ddnames* or environment variable names). An external file name is the name by which a file is known to the operating system.

In the following example, COMMUTER-FILE is the name that your program uses for the file; COMMUTR is the external name.

```
FILE-CONTROL.  
  SELECT COMMUTER-FILE  
  ASSIGN TO COMMUTR  
  ORGANIZATION IS LINE SEQUENTIAL  
  ACCESS MODE IS SEQUENTIAL  
  FILE STATUS IS ECODE.
```

Your ASSIGN *assignment-name* clause must not include an organization field (S- or AS-) before the external name. The ACCESS phrase and the FILE STATUS phrase are optional.

RELATED TASKS

“Describing the structure of a line-sequential file”

“Coding input-output statements for line-sequential files” on page 165

“Defining and allocating line-sequential files” on page 165

RELATED REFERENCES

“Allowable control characters”

## Allowable control characters

The control characters shown in the table below are the only characters other than printable characters that line-sequential files can contain. The hex values are in EBCDIC.

Hex value	Control character
X'05'	Horizontal tab
X'0B'	Vertical tab
X'0C'	Form feed
X'0D'	Carriage return
X'0E'	DBCS shift-out
X'0F'	DBCS shift-in
X'15'	New-line
X'16'	Backspace
X'2F'	Alarm

The new-line character is treated as a record delimiter. The other control characters are treated as data and are part of the record.

RELATED TASKS

“Defining line-sequential files and records in COBOL” on page 163

---

## Describing the structure of a line-sequential file

In the FILE SECTION of the DATA DIVISION, code a file description (FD) entry for the file. In the associated record description entry or entries, define the *record-name* and record length.

Code the logical size of the records with the RECORD clause. Line-sequential files are stream files. Because of their character-oriented nature, the physical records are of variable length.

The following examples show how the FD entry might look for a line-sequential file:

**With fixed-length records:**

```
FILE SECTION.  
FD COMMUTER-FILE  
  RECORD CONTAINS 80 CHARACTERS.  
01 COMMUTER-RECORD.  
  05 COMMUTER-NUMBER      PIC X(16).  
  05 COMMUTER-DESCRIPTION PIC X(64).
```

**With variable-length records:**

```
FILE SECTION.  
FD COMMUTER-FILE  
  RECORD VARYING FROM 16 TO 80 CHARACTERS.
```

```

01 COMMUTER-RECORD.
   05 COMMUTER-NUMBER      PIC X(16).
   05 COMMUTER-DESCRIPTION PIC X(64).

```

If you code the same fixed size and no OCCURS DEPENDING ON clause for any level-01 record description entries associated with the file, that fixed size is the logical record length. However, because blanks at the end of a record are not written to the file, the physical records might be of varying lengths.

#### RELATED TASKS

“Defining line-sequential files and records in COBOL” on page 163

“Coding input-output statements for line-sequential files”

“Defining and allocating line-sequential files”

#### RELATED REFERENCES

Data Division—file description entries (*IBM COBOL Language Reference*)

---

## Defining and allocating line-sequential files

You can define a line-sequential file in the HFS using either a DD statement or an environment variable. Allocation of these files follows the general rules for allocating COBOL files.

To define a line-sequential file, define one of the following with a name that matches the external name on your ASSIGN clause:

- A DD allocation:
  - A DD statement that specifies `PATH='absolute-path-name'`
  - A TSO allocation that specifies `PATH('absolute-path-name')`

You can optionally also specify these options:

- PATHOPTS
- PATHMODE
- PATHDISP
- An environment variable with a value of `PATH(absolute-path-name)`. No other values can be specified.

For example, to have your COBOL program use HFS file `/u/myfiles/comtfile` for a COBOL file with an *assignment-name* of `COMMUTR`, you would use the following command:

```
export COMMUTR="PATH(/u/myfiles/commuterfile)"
```

#### RELATED TASKS

“Allocating files” on page 105

“Defining line-sequential files and records in COBOL” on page 163

#### RELATED REFERENCES

*OS/390 MVS JCL Reference*

---

## Coding input-output statements for line-sequential files

Code the following input and output statements to process a line-sequential file:

**OPEN** To make the file available to your program.

You can open a line-sequential file as `INPUT`, `OUTPUT`, or `EXTEND`. You cannot open a line-sequential file as `I-0`.

**READ** To read a record from the file.

With sequential processing, your program reads one record after another in the same order in which they were entered when the file was created.

**WRITE** To create a record in the file.

Your program writes new records to the end of the file.

**CLOSE** To release the connection between the file and your program.

#### RELATED TASKS

“Defining line-sequential files and records in COBOL” on page 163

“Describing the structure of a line-sequential file” on page 164

“Opening line-sequential files”

“Reading records from line-sequential files”

“Adding records to line-sequential files” on page 167

“Closing line-sequential files” on page 167

“Handling errors in line-sequential files” on page 168

#### RELATED REFERENCES

OPEN statement (*IBM COBOL Language Reference*)

READ statement (*IBM COBOL Language Reference*)

WRITE statement (*IBM COBOL Language Reference*)

CLOSE statement (*IBM COBOL Language Reference*)

## Opening line-sequential files

Before your program can use any READ or WRITE statements to process records in a file, it must first open the file with an OPEN statement.

An OPEN statement works if the file is available or has been dynamically allocated.

Code CLOSE WITH LOCK so that the file cannot be opened again while the program is running.

#### RELATED TASKS

“Reading records from line-sequential files”

“Adding records to line-sequential files” on page 167

“Closing line-sequential files” on page 167

“Defining and allocating line-sequential files” on page 165

#### RELATED REFERENCES

OPEN statement (*IBM COBOL Language Reference*)

CLOSE statement (*IBM COBOL Language Reference*)

## Reading records from line-sequential files

To read from a line-sequential file, open the file and use the READ statement.

With sequential processing, your program reads one record after another in the same order in which the records were entered when the file was created.

Characters in the file record are read one at a time into the record area until one of the following conditions occurs:

- The record delimiter (the EBCDIC new-line character) is encountered.  
The delimiter is discarded and the remainder of the record area is filled with spaces. (Record area is longer than the file record.)

- The entire record area is filled with characters.  
If the next unread character is the record delimiter, it is discarded. The next READ reads from the first character of the next record. (Record area is the same length as the file record.)  
Otherwise the next unread character is the first character to be read by the next READ. (Record area is shorter than the file record.)

#### RELATED TASKS

- “Opening line-sequential files” on page 166
- “Adding records to line-sequential files”
- “Closing line-sequential files”
- “Defining and allocating line-sequential files” on page 165

#### RELATED REFERENCES

- OPEN statement (*IBM COBOL Language Reference*)
- WRITE statement (*IBM COBOL Language Reference*)

## Adding records to line-sequential files

To add to a line-sequential file, open the file as EXTEND and use the WRITE statement to add records immediately after the last record in the file.

Blanks at the end of the record area are removed and the record delimiter is added. The characters in the record area from the first character up to and including the added record delimiter are written to the file as one record.

Records written to line-sequential files must contain only USAGE DISPLAY and DISPLAY-1 items. External decimal data items must be unsigned or declared with the SEPARATE CHARACTER phrase if signed.

#### RELATED TASKS

- “Opening line-sequential files” on page 166
- “Reading records from line-sequential files” on page 166
- “Closing line-sequential files”
- “Defining and allocating line-sequential files” on page 165

#### RELATED REFERENCES

- OPEN statement (*IBM COBOL Language Reference*)
- WRITE statement (*IBM COBOL Language Reference*)

## Closing line-sequential files

Use the CLOSE statement to disconnect your program from a line-sequential file. If you try to close a file that is already closed, you will get a logic error.

If you do not close a line-sequential file, the file is automatically closed for you under the following conditions:

- When the run unit ends normally.
- When the run unit ends abnormally, if the TRAP(ON) run-time option is set.
- When Language Environment condition handling is completed and the application resumes in a routine other than where the condition occurred, open files defined in any COBOL programs in the run unit that might be called again and reentered are closed.

You can change the location where the program resumes (after a condition is handled) by moving the resume cursor with the Language Environment CEEMRCR callable service or using HLL language constructs such as a C longjmp call.

File status codes are set when these implicit CLOSE operations are performed, but EXCEPTION/ERROR declaratives are not invoked.

#### RELATED TASKS

- “Opening line-sequential files” on page 166
- “Reading records from line-sequential files” on page 166
- “Adding records to line-sequential files” on page 167
- “Defining and allocating line-sequential files” on page 165

#### RELATED REFERENCES

CLOSE statement (*IBM COBOL Language Reference*)

---

## Handling errors in line-sequential files

When an input or output statement operation fails, COBOL does not take corrective action for you. You choose whether or not your program will continue running after an input or output error occurs.

COBOL provides these techniques for intercepting and handling certain line-sequential input and output errors:

- End of file phrase (AT END)
- EXCEPTION/ERROR declarative
- FILE STATUS clause

If you do not use one of these techniques, an error in processing input or output raises a Language Environment condition.

If you use the FILE STATUS clause, be sure to check the key and take appropriate action based on its value. If you do not check the key, it is possible that your program could continue; but the results will probably not be what you expected.

#### RELATED TASKS

- “Coding input-output statements for line-sequential files” on page 165
- “Handling errors in input and output operations” on page 191

---

## Chapter 11. 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.

With COBOL for OS/390 & VM, your IBM licensed program for sorting and merging must be DFSORT or an equivalent. Where DFSORT is mentioned, you can use any equivalent sort or merge product.

COBOL programs containing SORT or MERGE statements can reside above or below the 16M line.

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.

**OS/390 UNIX:** You cannot run a COBOL program containing SORT or MERGE statements under OS/390 UNIX. This restriction includes BPXBATCH. The USING or GIVING files on a SORT or MERGE statement can be sequential files residing in the HFS.

### **RELATED CONCEPTS**

Sort and merge process

### **RELATED TASKS**

“Describing the sort or merge file” on page 170

“Describing the input to sorting or merging” on page 171

“Describing the output from sorting or merging” on page 173

“Requesting the sort or merge” on page 177

“Determining whether the sort or merge was successful” on page 180

“Improving sort performance with FASTSORT” on page 181

“Controlling sort behavior” on page 184

*DFSORT Application Programming Guide*

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. You do not need to provide a ddname definition for the file.

RELATED REFERENCES

“FILE SECTION entries” on page 14

---

## 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  
    LABEL RECORDS ARE STANDARD  
    BLOCK CONTAINS 0 CHARACTERS  
    RECORDING MODE IS F  
    RECORD CONTAINS 100 CHARACTERS.  
01 Input-Record PIC X(100).
```

RELATED TASKS

“Coding the input procedure” on page 172

“Requesting the sort or merge” on page 177

RELATED REFERENCES

“FILE SECTION entries” on page 14

## 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
    Label Records Are Standard
    Block Contains 0 Characters
    Recording Mode is F
    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 177

---

## 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
<pre>MOVE EXT-RECORD   TO SORT-EXT-RECORD PERFORM RELEASE-SORT-RECORD . . . RELEASE-SORT-RECORD. RELEASE SORT-RECORD</pre>	<pre>PERFORM RELEASE-SORT-RECORD . . . RELEASE-SORT-RECORD. RELEASE SORT-RECORD   FROM SORT-EXT-RECORD</pre>

**RELATED REFERENCES**

“Restrictions on input and output procedures” on page 175  
 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
  LABEL RECORDS ARE STANDARD
  BLOCK CONTAINS 0 CHARACTERS
  RECORDING MODE IS F
  RECORD CONTAINS 100 CHARACTERS.
01 Output-Record PIC X(100).
```

**RELATED TASKS**

“Coding the output procedure”  
 “Requesting the sort or merge” on page 177

**RELATED REFERENCES**

“FILE SECTION entries” on page 14

## 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.

## Coding considerations when using DFSORT on OS/390

When a RETURN statement does not encounter an AT END condition before your COBOL program finishes running, the SORT statement could end abnormally with DFSORT message IEC025A. To avoid this situation, take these steps:

1. Code the RETURN statement with the AT END phrase.
2. Ensure that the RETURN statement is executed until the AT END condition is encountered.

The AT END condition occurs after the last record is returned to the program from the sort work file and a subsequent RETURN statement is executed.

“Example: coding the output procedure when using DFSORT”

### RELATED REFERENCES

“Restrictions on input and output procedures” on page 175  
RETURN statement (*IBM COBOL Language Reference*)

## Example: coding the output procedure when using DFSORT

The following example shows a coding technique that ensures that the RETURN statement encounters the AT END condition before the program finishes running. The RETURN statement, coded with the AT END phrase, is executed until the AT END condition occurs.

```
IDENTIFICATION DIVISION.
DATA DIVISION.
FILE SECTION.
SD OUR-FILE.
01 OUR-SORT-REC.
   03 SORT-KEY          PIC X(10).
   03 FILLER            PIC X(70).
. . .
WORKING-STORAGE SECTION.
01 WS-SORT-REC          PIC X(80).
01 END-OF-SORT-FILE-INDICATOR PIC X VALUE 'N'.
   88 NO-MORE-SORT-RECORDS      VALUE 'Y'.
. . .
PROCEDURE DIVISION.
A-CONTROL SECTION.
   SORT OUR-FILE ON ASCENDING KEY SORT-KEY
   INPUT PROCEDURE IS B-INPUT
   OUTPUT PROCEDURE IS C-OUTPUT.
. . .
B-INPUT SECTION.
   MOVE . . . . . TO WS-SORT-REC.
   RELEASE OUR-SORT-REC FROM WS-SORT-REC.
. . .
C-OUTPUT SECTION.
   DISPLAY 'STARTING READS OF SORTED RECORDS: '.
```

```

RETURN OUR-FILE
  AT END
    SET NO-MORE-SORT-RECORDS TO TRUE.
PERFORM WITH TEST BEFORE UNTIL NO-MORE-SORT-RECORDS
  IF SORT-RETURN = 0 THEN
    DISPLAY 'OUR-SORT-REC = ' OUR-SORT-REC
    RETURN OUR-FILE
  AT END
    SET NO-MORE-SORT-RECORDS TO TRUE
  END-IF
END-PERFORM.

```

---

## 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 that follows standard linkage conventions. However, the called program cannot issue a SORT or MERGE statement. For information on linkage convention considerations with Language Environment callable services, see the reference below.
- 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.
- Language Environment condition handling does not allow user-written condition handlers to be established in an input or output procedure.

### RELATED TASKS

“Coding the input procedure” on page 172

“Coding the output procedure” on page 173

*Language Environment Programming Guide*

---

## Defining sort and merge data sets under OS/390

To use DFSORT under OS/390, code DD statements in the run-time JCL to describe the necessary data sets:

### Sort or merge work

Define a minimum of three data sets: SORTWK01, SORTWK02, SORTWK03, . . . , SORTWKnn (where nn is 99 or less). These data sets cannot be in the HFS.

**SYSOUT** Define for sort diagnostic messages, unless you change the data set name. (Change the name using either the MSGDDN keyword of the OPTION control statement in the SORT-CONTROL data set, or using the SORT-MESSAGE special register.)

### SORTCKPT

Define if the sort or merge is to take checkpoints.

### **Input and output**

Define input and output data sets, if any.

### **SORTLIB (DFSORT library)**

Define the library containing the sort modules, for example, SYS1.SORTLIB.

#### RELATED TASKS

“Controlling sort behavior” on page 184

“Using checkpoint/restart with DFSORT under OS/390” on page 186

---

## **Defining sort and merge files under CMS**

You can run programs containing SORT and MERGE statements under CMS using DFSORT/CMS or equivalent. To use DFSORT/CMS, identify the following files and TXTLIB:

### **Sort or merge work**

Sort and merge work files are allocated by DFSORT/CMS.

You can tailor the sort or merge work files in these ways:

- Create your own OPTION CONTROL statement in a SORT-CONTROL data set. Issue the FILEDEF command for the SORT-CONTROL data set.
- Use the DFSORT/CMS parameters WRKADR, WRKDEV, and WRKSIZ.

### **Input and output**

Identify by issuing the FILEDEF command.

### **SORT-CONTROL**

Identify by issuing the FILEDEF command. The file must correspond to the default (IGZSRTECD) or to the name you code in the SORT-CONTROL special register.

### **DFSORT/CMS library**

Before running a program that uses DFSORT/CMS, you must also define the TXTLIB that contains the DFSORT/CMS library. Use the GLOBAL TXTLIB command (discussed further in the reference below) to add the name of the DFSORT/CMS library.

#### RELATED TASKS

“Controlling sort behavior” on page 184

“Identifying QSAM files to CMS” on page 128 (FILEDEF command)

Using DFSORT/CMS (*DFSORT/CMS User's Guide*)

Using the GLOBAL command (*Language Environment Programming Guide*)

---

## **Sorting variable-length records**

Your sort work file will be variable length only if you define it to be variable length, even if the input file to the sort contains variable-length records.

The compiler will determine that the sort work file is variable length if you code one of the following in its SD entry:

- A RECORD IS VARYING clause
- Two or more record descriptions that define records with different sizes, or records that contain an OCCURS DEPENDING ON clause

You cannot code RECORDING MODE V for the sort work file because the SD entry does not allow the RECORDING MODE clause.

To improve sort performance on variable-length files, specify the most frequently occurring record length of the input file (the modal length) on the SMS= control card or in the SORT-MODE-SIZE special register.

**RELATED TASKS**

“Changing DFSORT defaults with control statements” on page 185

“Controlling sort behavior” on page 184

---

## 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 171

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 178

**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.

There is no maximum number of keys, but the keys must be located in the first 4092 bytes of the record description.

The total length of the keys cannot exceed 4092 bytes unless the EQUALS keyword is coded in the DFSORT OPTION control statement, in which case the total length of the keys must not exceed 4088 bytes.

**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 B. Complex OCCURS DEPENDING ON” on page 569 (variably located items)

#### RELATED TASKS

“Defining sort and merge data sets under OS/390” on page 175

“Defining sort and merge files under CMS” on page 176

#### RELATED REFERENCES

SORT control statement (*DFSORT Application Programming Guide*)

SORT statement (*IBM COBOL Language Reference*)

MERGE 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 177

## Choosing alternate collating sequences

You can sort or merge records on the EBCDIC or ASCII collating sequence, or on another collating sequence. The default collating sequence is EBCDIC unless you code the PROGRAM COLLATING SEQUENCE clause in the OBJECT-COMPUTER paragraph. To override the sequence named in the PROGRAM COLLATING SEQUENCE clause, 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.

When you sort or merge an ASCII file, you have to request the ASCII collating sequence. To do so, code the COLLATING SEQUENCE phrase of the SORT or MERGE statement, where you define the *alphabet-name* as STANDARD-1 in the SPECIAL-NAMES paragraph.

#### RELATED TASKS

“Specifying the collating sequence” on page 8

#### RELATED REFERENCES

SORT statement (*IBM COBOL Language Reference*)

## Sorting on windowed date fields

You can specify windowed date fields as sort keys if your version of DFSORT supports the Y2PAST option. If so, DFSORT can sort or merge on the windowed date sequence.

To sort on a windowed date field, use the DATE FORMAT clause to define a windowed date field; then use the field as the sort key. DFSORT will use the same century window as that used by the compilation unit. Specify the century window with the YEARWINDOW compiler option.

DFSORT supports year-last windowed date fields, although the compiler itself does not provide automatic windowing for year-last windowed date fields in statements other than MERGE or SORT.

#### RELATED REFERENCES

“YEARWINDOW” on page 303

DATE FORMAT clause (*IBM COBOL Language Reference*)  
Y2PAST option (*DFSORT Application Programming Guide*)

## Preserving the original sequence of records with equal keys

You can preserve the order of identical collating records from input to output in one of these ways:

- Install DFSORT with the EQUALS option as the default.
- Provide, at run time, an OPTION card with the EQUALS keyword in the IGZSRTECD data set.
- Use the WITH DUPLICATES IN ORDER phrase in the SORT statement. Doing so adds the EQUALS keyword to the OPTION card in the IGZSRTECD data set.

Do not use the NOEQUALS keyword on the OPTION card *and* use the DUPLICATES phrase, or the run unit will be ended.

### RELATED REFERENCES

OPTION control statement (*DFSORT Application Programming Guide*)

---

## Determining whether the sort or merge was successful

The DFSORT program 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.

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.

By default, DFSORT diagnostic messages are sent to the SYSOUT data set. If you want to change this default, use the MSGDDN parameter of the DFSORT OPTION control card or use the SORT-MESSAGE special register.

### RELATED TASKS

“Checking for sort errors with NOFASTSRT” on page 183

“Controlling sort behavior” on page 184

---

## 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 immediately after the next implicit RELEASE or RETURN is performed, which will occur after a record has been read from or written to the USING or GIVING file.

Control then returns to the statement following the SORT or MERGE statement.

---

## Improving sort performance with FASTSRT

Using the FASTSRT compiler option improves the performance of most sort operations. With FASTSRT, the DFSORT product (instead of COBOL for OS/390 & VM) performs the I/O on the input and output files you name in the following statements:

```
SORT . . . USING
SORT . . . GIVING
```

The compiler issues informational messages to point out statements in which FASTSRT can improve performance.

**Restrictions:** You cannot use the DFSORT options SORTIN or SORTOUT if you use FASTSRT. The FASTSRT compiler option does not apply to line-sequential files you use as USING or GIVING files.

If you specify file status and use FASTSRT, file status is ignored during the sort.

## RELATED REFERENCES

“FASTSRT” on page 273

“FASTSRT requirements for JCL (OS/390 only)”

“FASTSRT requirements for sort input and output files”

## FASTSRT requirements for JCL (OS/390 only)

In the run-time JCL, you must assign the sort work files (SORTWKnn) to a direct-access device, not to tape data sets.

For the input and output files, the DCB parameter of the DD statement in the JCL must match the FD description.

## FASTSRT requirements for sort input and output files

If you specify FASTSRT but your code does not meet FASTSRT requirements, the compiler issues a message and the COBOL run time performs the I/O instead. Your program will not experience the performance improvements otherwise possible.

To use FASTSORT, you must describe and process the input files to the sort, and the output files from the sort, in these ways:

- You can mention only one input file in the USING phrase. You can mention only one output file in the GIVING phrase.
- You cannot use an input procedure on an input file, nor an output procedure on an output file.

Instead of using input or output procedures, you might be able to use DFSORT control statements:

- INREC
- OUTREC
- INCLUDE
- OMIT
- STOPAFT
- SKIPREC
- SUM

Many DFSORT functions perform the same operations that are common in input or output procedures. Code the appropriate DFSORT control statements instead, and place them either in the IGZSRTCD data set or the SORTCNTL data set.

- Do not code the LINAGE clause for the output FD entry.
- Do not code any INPUT declarative (for input files), OUTPUT declarative (for output files), or file-specific declaratives (for either input or output files) to apply to any FDs used in the sort.
- Do not use a variable relative file as the input or output file.
- Do not use a line-sequential file as the input or output file.
- For either an input or an output file, the record descriptions of the SD and FD entry must define the same format (fixed or variable), and the largest records of the SD and FD entry must define the same record length.

Note that if you code a RELATIVE KEY clause for an output file, it will not be set by the sort.

**Tip:** If you block your input and output records, the sort performance could be significantly improved.

### QSAM requirements

- QSAM files must have a record format of fixed, variable, or spanned.
- A QSAM input file can be empty.
- To use the same QSAM file for both input and output, you must describe the file using two different DD statements. For example, in the FILE-CONTROL SECTION you might code the following:

```
SELECT FILE-IN ASSIGN INPUTF.  
SELECT FILE-OUT ASSIGN OUTPUTF.
```

In the DATA DIVISION, you would have an FD entry for both FILE-IN and FILE-OUT, where FILE-IN and FILE-OUT are identical except for their names.

In the PROCEDURE DIVISION, your SORT statement could look like this:

```
SORT file-name  
  ASCENDING KEY data-name-1  
  USING FILE-IN GIVING FILE-OUT
```

Then in your JCL, you would code:

```
//INPUTF DD DSN=INOUT,DISP=SHR  
//OUTPUTF DD DSN=INOUT,DISP=SHR
```

where data set INOUT has been cataloged.

On the other hand, if you code the same file name in the USING and GIVING phrases, or assign the input and output files the same ddname, then the file can be accepted for FASTSORT either for input or output, but not both. If no other conditions disqualify the file from being eligible for FASTSORT on input, then the file will be accepted for FASTSORT on input, but not on output. If the file was found to be ineligible for FASTSORT on input, it might be eligible for FASTSORT on output.

A QSAM file that qualifies for FASTSORT can be accessed by the COBOL program while the SORT statement is being performed. For example, if the file is used for FASTSORT on input, you can access it in an output procedure; if it is used for FASTSORT on output, you can access it in an input procedure.

### VSAM requirements

- A VSAM input file must not be empty.
- VSAM files cannot be password protected.
- You cannot name the same VSAM file in both the USING and GIVING phrases.
- A VSAM file that qualifies for FASTSORT cannot be accessed by the COBOL program until the SORT statement processing is completed. For example, if the file qualifies for FASTSORT on input, you cannot access it in an output procedure and vice versa. (If you do so, OPEN will fail.)

#### RELATED TASKS

*DFSORT Application Programming Guide*

---

## Checking for sort errors with NOFASTSORT

When you compile with the NOFASTSORT option, the sort process does not check for errors on open, close, or input or output operations for files you reference in the USING or GIVING phrase of the SORT statement. Therefore, you might need to check whether the SORT statement completed successfully.

The code required depends on whether you code a FILE STATUS clause or an ERROR declarative for the files referenced in the USING and GIVING phrases, as shown in the table below.

FILE STATUS clause?	ERROR declarative?	Then do:
No	No	No special coding. Any failure during the sort process causes the program to end abnormally.
Yes	No	Test the SORT-RETURN special register after the SORT statement, and test the file status key. (Not recommended if you want complete file status checking, because the file status code is set but COBOL cannot check it.)
Maybe	Yes	In the ERROR declarative, set the SORT-RETURN special register to 16 to stop the sort process and indicate that it was not successful. Test the SORT-RETURN special register after the SORT statement.

#### RELATED TASKS

“Determining whether the sort or merge was successful” on page 180

“Using file status keys” on page 194

“Coding ERROR declaratives” on page 194

“Stopping a sort or merge operation prematurely” on page 181

---

## Controlling sort behavior

You can control several aspects of sort behavior by the following means:

- Inserting values in special registers before the sort
- Using compiler options
- Using control statement keywords

You can also verify sort behavior by examining the contents of special registers after the sort.

The table below lists those aspects of sort behavior that you can affect using special registers or compiler options, and the equivalent sort control statement keywords if any. For a full list of sort keywords, see the reference below.

To set or test	Use this special register or compiler option	Or this control statement (and keyword if applicable)
Amount of main storage to be reserved	SORT-CORE-SIZE special register	OPTION (keyword RESINV)
Amount of main storage to be used	SORT-CORE-SIZE special register	OPTION (keywords MAINSIZE or MAINSIZE=MAX)
Modal length of records in a file with variable-length records	SORT-MODE-SIZE special register	SMS= <i>nnnnn</i>
Name of sort control statement data set (default IGZSR TCD)	SORT-CONTROL special register	None
Name of sort message file (default SYSOUT)	SORT-MESSAGE special register	OPTION (keyword MSGDDN)
Number of sort records	SORT-FILE-SIZE special register	OPTION (keyword FILSZ)
Sort completion code	SORT-RETURN special register	None
Century window for sorting or merging on date fields	YEARWINDOW compiler option	OPTION (keyword Y2PAST)
Format of windowed date fields used as sort or merge keys	(Derived from PICTURE, USAGE, and DATE FORMAT clauses)	SORT (keyword FORMAT=Y2x)

## Sort special registers

SORT-CONTROL is an eight-character COBOL special register that contains the ddname of the sort control statement file. If you do not want to use the default ddname IGZSR TCD, assign to SORT-CONTROL the ddname of the data set that contains your sort control statements.

The SORT-CORE-SIZE, SORT-FILE-SIZE, SORT-MESSAGE, and SORT-MODE-SIZE special registers are used in the SORT interface if you assign them nondefault values. At

run time, however, any parameters on control statements in the sort control statement data set override corresponding settings in the special registers, and a message to that effect is issued.

You can use the SORT-RETURN special register to determine whether the sort or merge was successful and to stop a sort or merge operation prematurely, as discussed in the cross-references below.

A compiler warning message (W-level) is issued for each sort special register that you set in a program.

#### RELATED TASKS

“Determining whether the sort or merge was successful” on page 180

“Stopping a sort or merge operation prematurely” on page 181

“Changing DFSORT defaults with control statements”

Using DFSORT program control statements (*DFSORT Application Programming Guide*)

#### RELATED REFERENCES

“Default characteristics of the IGZSRTCD data set”

## Changing DFSORT defaults with control statements

If you want to change DFSORT system defaults to improve sort performance, pass information to DFSORT through control statements in the run-time data set IGZSRTCD.

The control statements you can include in the IGZSRTCD data set (in the order listed) are:

1. SMS=nnnnn, where nnnnn is the length in bytes of the most frequently occurring record size. (Use only if the SD file is variable length.)
2. OPTION (except keywords SORTIN or SORTOUT).
3. Other DFSORT control statements (except SORT, MERGE, RECORD, or END).

Code control statements between columns 2 and 71. You can continue a control statement record by ending the line with a comma and starting the next line with a new keyword. No labels or comments are allowed on a record, and a record itself cannot be a DFSORT comment statement.

#### RELATED TASKS

“Controlling sort behavior” on page 184

Using DFSORT program control statements (*DFSORT Application Programming Guide*)

#### RELATED REFERENCES

“Default characteristics of the IGZSRTCD data set”

### Default characteristics of the IGZSRTCD data set

- LRECL=80
- BLKSIZE=400
- ddname is IGZSRTCD. (You can use a different ddname by coding it in the SORT-CONTROL special register.)

The IGZSRTCD data set is optional. If you defined a ddname for the SORT-CONTROL data set and you receive the message IGZ0027W, an OPEN failure occurred that you should investigate.

#### RELATED TASKS

“Controlling sort behavior” on page 184

## Allocating storage for sort or merge operations

Certain parameters set during the installation of DFSORT determine the amount of storage it uses. In general, the more storage DFSORT has available, the faster the sort or merge operations in your program will be.

DFSORT installation should not allocate all the free space in the region for its COBOL operation, however. When your program is running, storage must be available for the following:

- COBOL programs that are dynamically called from an input or output procedure
- Language Environment run-time library modules
- Data management modules that can be loaded into the region for use by an input or output procedure
- Any storage obtained by these modules

For a specific sort or merge operation, you can override the DFSORT storage values set at installation. To do so, code the MAINSIZE and RESINV keywords on the OPTION control statement in the sort control statement data set, or use the SORT-CORE-SIZE special register.

Be careful not to override the storage allocation to the extent that all the free space in the region is used for sort operations in your COBOL program.

#### RELATED TASKS

“Controlling sort behavior” on page 184

*DFSORT Installation and Customization*

#### RELATED REFERENCES

OPTION control statement (*DFSORT Application Programming Guide*)

---

## Using checkpoint/restart with DFSORT under OS/390

You cannot use checkpoints taken while DFSORT is running under OS/390 to restart, unless the checkpoints are taken by DFSORT. Checkpoints taken by your COBOL program while SORT or MERGE statements execute are invalid; the restarts are detected and canceled.

To take a checkpoint during a sort or merge operation, follow these steps:

1. Add a DD statement for SORTCKPT in the JCL.
2. Code the RERUN clause in the I-0-CONTROL paragraph:  
`RERUN ON assignment-name`
3. Code the CKPT (or CHKPT) keyword on an OPTION control statement in the sort control statement data set (default ddname IGZSRTECD).

#### RELATED CONCEPTS

“Chapter 31. Interrupts and checkpoint/restart” on page 501

#### RELATED TASKS

“Changing DFSORT defaults with control statements” on page 185

“Setting checkpoints” on page 501



---

## Sorting under CICS

There is no IBM sort product that is supported under CICS. However, you can use the SORT statement with a sort program you write that runs under CICS to sort small amounts of data.

You must have both an input and an output procedure for the SORT statement. In the input procedure, use the RELEASE statement to transfer records from the COBOL program to the sort program before the sort is performed. In the output procedure, use the RETURN statement to transfer records from the sort program to the COBOL program after the sort is performed.

### RELATED TASKS

- “Coding the input procedure” on page 172
- “Coding the output procedure” on page 173

### RELATED REFERENCES

- “CICS SORT application restrictions”

## CICS SORT application restrictions

The following restrictions apply to COBOL applications that run under CICS and use the SORT statement:

- SORT statements that include the USING or GIVING phrase are not supported.
- Sort control data sets are not supported. Data in the SORT-CONTROL special register is ignored.
- Using the following CICS commands in the input or output procedures can cause unpredictable results:
  - CICS LINK
  - CICS XCTL
  - CICS RETURN
  - CICS HANDLE
  - CICS IGNORE
  - CICS PUSH
  - CICS POP
- You can use CICS commands other than those in the preceding list provided you use the NOHANDLE or RESP option. Unpredictable results can occur if you do not use NOHANDLE or RESP.



---

## Chapter 12. 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:

- User-requested dumps
- 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
- User-written routines for handling errors

### RELATED TASKS

“Handling errors in joining and splitting strings” on page 190

“Handling errors in arithmetic operations” on page 191

“Handling errors in input and output operations” on page 191

“Handling errors when calling programs” on page 199

“Writing routines for handling errors” on page 199

---

## Requesting dumps

You can obtain a formatted dump of the run-time environment by calling the Language Environment service CEE3DMP. To obtain a system dump, you can request an abend without cleanup by calling the Language Environment service CEE3ABD with a cleanup value of zero.

### Creating a formatted dump

You can cause a dump of the Language Environment run-time environment and the member language libraries at any prespecified point in your program. Simply code a call to the Language Environment callable service CEE3DMP. For example:

```
77 Title-1          Pic x(80)   Display.
77 Options          Pic x(255)  Display.
01 Feedback-code   Pic x(12)   Display.
. . .
    Call "CEE3DMP" Using Title-1, Options, Feedback-code
```

To have symbolic variables included in the formatted dump produced by Language Environment, you must compile with the SYM suboption of the TEST compiler option and use the VARIABLES subparameter of CEE3DMP.

You can also request, through run-time options, that a dump be produced for error conditions of your choosing.

## Creating a system dump

You can cause a system dump at any prespecified point in your program. Simply request an abend without cleanup by calling the Language Environment service CEE3ABD with a cleanup value of zero.

This callable service stops the run unit immediately, and a system dump is requested when the abend is issued.

### RELATED REFERENCES

*Language Environment Debugging Guide and Run-Time Messages*

*Language Environment Programming Reference*

---

## 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)	EEEEAA	EEEEAA
Item-3	X(2)	EA	EA
Item-4	X(8)	□□□□□□□□ <sup>1</sup>	□□□□□□□□ <sup>1</sup>
String-ptr	9(2)	0	0

1. The symbol □ 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

Floating-point exponent overflow occurs when the value of a floating-point computation cannot be represented in the System/390 floating-point operand format. This type of overflow does not cause SIZE ERROR; an abend occurs instead. You could code a user-written condition handler to intercept the abend and provide your own error recovery logic.

“Example: checking for division by zero”

### RELATED REFERENCES

“ON SIZE ERROR and intermediate results” on page 567

### 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:

- Imperative-statement phrases on your READ or WRITE statement

- ERROR declaratives
- FILE STATUS clauses

For VSAM files, if you specify a FILE STATUS clause, you can also test the VSAM return code to direct your program to error-handling logic.

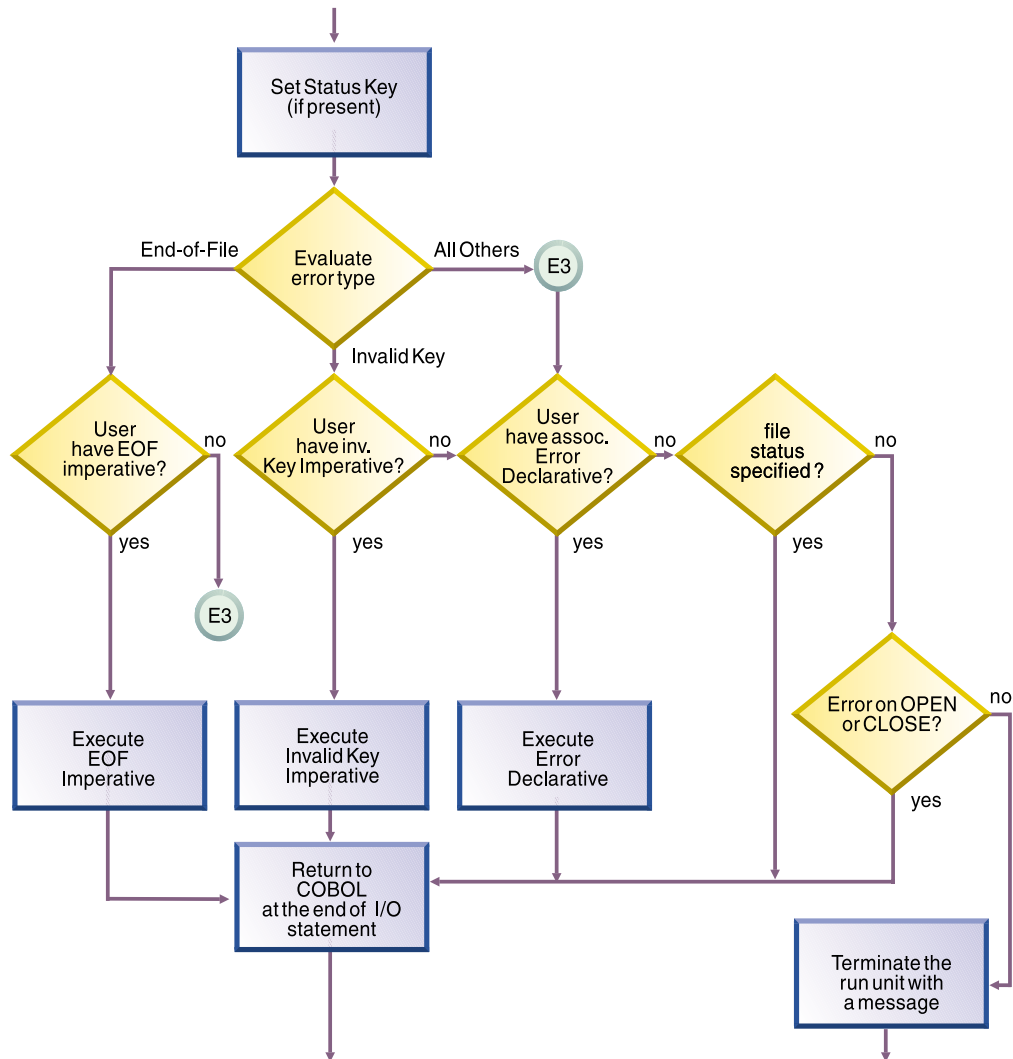
If you choose to have your program continue (by incorporating error-handling code into your design), you must code the appropriate error-recovery procedure. You might code, for example, a procedure to check the value of the file status key.

If you do not handle an input or output error in any of these ways, a severity-3 Language Environment condition is signaled, which causes the run unit to end if the condition is not handled.

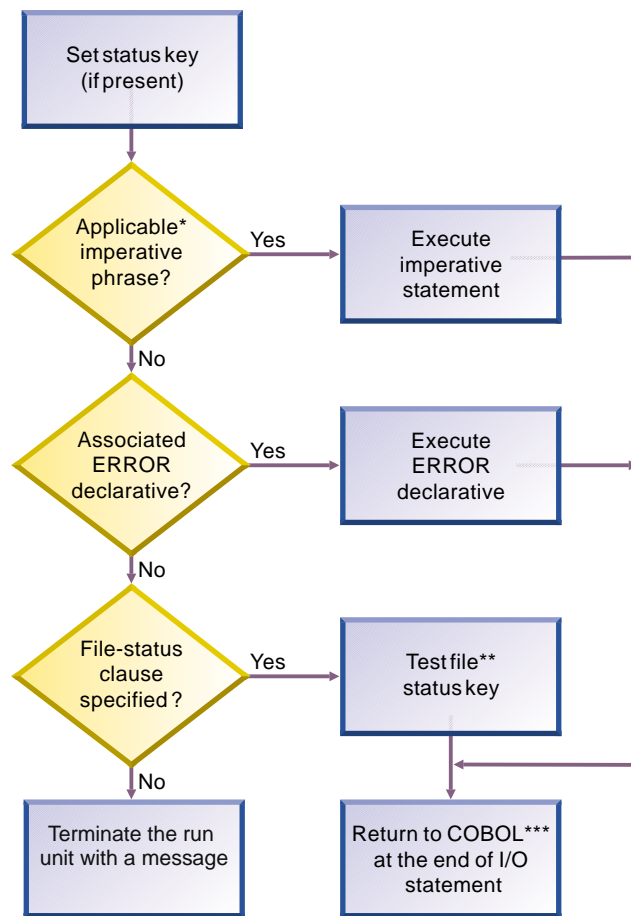
The following figures show the flow of logic after the indicated errors:

- 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:



The following figure shows the flow of logic after an input or output error with QSAM or line-sequential files. The error can be from a READ statement, a WRITE statement, or a CLOSE statement with a REEL/UNIT clause (QSAM only).



\*Possible phrases for QSAM are AT END, AT END-OF-PAGE, and INVALID KEY; for line sequential, AT END.

\*\*You need to write the code to test the file status key.

\*\*\*Execution of your COBOL program continues after the input or output statement that caused the error.

#### RELATED TASKS

“Using the end-of-file condition (AT END)”

“Coding ERROR declaratives” on page 194

“Using file status keys” on page 194

“Handling errors in QSAM files” on page 120

“Using VSAM return codes (VSAM files only)” on page 196

“Coding INVALID KEY phrases” on page 198

## 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-0, 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.

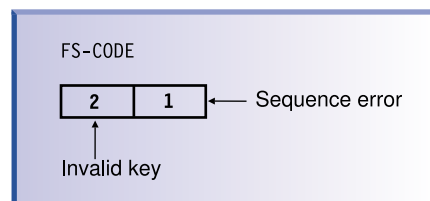
```
FILE STATUS IS data-name-1
```

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 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 VSAM 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: 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 VSAM return codes (VSAM files only)

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 input or output requests by coding a second status area:

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

The variable *data-name-1* specifies the two-character COBOL file status key. The variable *data-name-8* specifies a 6-byte data item that contains the VSAM return code when the COBOL file status key is not 0.

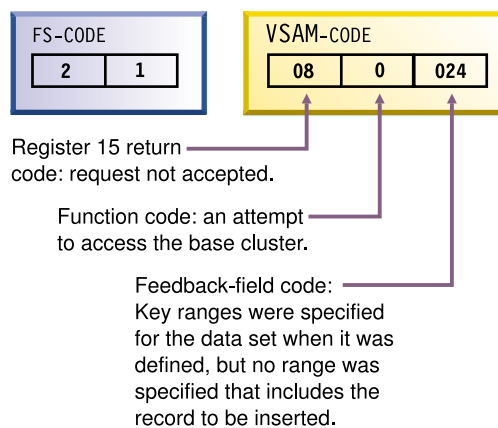
You can define the second status area, *data-name-8*, in the WORKING-STORAGE SECTION as in VSAM-CODE here:

```

01 RETURN-STATUS.
   05 FS-CODE          PIC X(2).
   05 VSAM-CODE.
      10 VSAM-R15-RETURN PIC S9(4) Usage Binary.
      10 VSAM-FUNCTION  PIC S9(4) Usage Binary.
      10 VSAM-FEEDBACK  PIC S9(4) Usage Binary.

```

COBOL for OS/390 & VM uses *data-name-8* to pass information supplied by VSAM. In the following example, FS-CODE corresponds to *data-name-1* and VSAM-CODE corresponds to *data-name-8*:



“Example: checking VSAM status codes”

## Example: checking VSAM status 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 VSAMFILE ASSIGN TO VSAMFILE
    ORGANIZATION IS INDEXED
    ACCESS DYNAMIC
    RECORD KEY IS VSAMFILE-KEY
    FILE STATUS IS FS-CODE VSAM-CODE.
DATA DIVISION.
FILE SECTION.
FD VSAMFILE
   RECORD 30.
01 VSAMFILE-REC.
   10 VSAMFILE-KEY           PIC X(6).
   10 FILLER                 PIC X(24).
WORKING-STORAGE SECTION.
01 RETURN-STATUS.
   05 FS-CODE                PIC XX.
   05 VSAM-CODE.
     10 VSAM-RETURN-CODE     PIC S9(2) Usage Binary.
     10 VSAM-COMPONENT-CODE  PIC S9(1) Usage Binary.
     10 VSAM-REASON-CODE     PIC S9(3) Usage Binary.
PROCEDURE DIVISION.
OPEN INPUT VSAMFILE.
DISPLAY "OPEN INPUT VSAMFILE FS-CODE: " FS-CODE.

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

MOVE "000005" TO VSAMFILE-KEY.
START VSAMFILE KEY IS EQUAL TO VSAMFILE-KEY.
DISPLAY "START VSAMFILE KEY=" VSAMFILE-KEY
        " FS-CODE: " FS-CODE.
IF FS-CODE NOT = "00"
    PERFORM VSAM-CODE-DISPLAY
END-IF.

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

CLOSE VSAMFILE.
STOP RUN.

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

VSAM-CODE-DISPLAY.
DISPLAY "VSAM-CODE ==>"
        " RETURN: " VSAM-RETURN-CODE,
        " COMPONENT: " VSAM-COMPONENT-CODE,
        " REASON: " VSAM-REASON-CODE.
```

Below is a sample of the output from the example program that checks VSAM code information:

```
OPEN INPUT VSAMFILE FS-CODE: 00
START VSAMFILE KEY=000005 FS-CODE: 00
READ NEXT VSAMFILE FS-CODE: 00
000005 THIS IS RECORD NUMBER 5
READ NEXT VSAMFILE FS-CODE: 00
000006 THIS IS RECORD NUMBER 6
READ NEXT VSAMFILE FS-CODE: 10
VSAM-CODE ==> RETURN: 08 COMPONENT: 2 REASON: 004
```

## 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.

You can also include INVALID KEY on WRITE requests for QSAM files, but the INVALID KEY phrase has limited meaning here. It is used only if you try to write to a disk that is full.

### 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 behavior of the `ON EXCEPTION/OVERFLOW` clause is sensitive to the `CMPR2` compiler option.

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.

### RELATED TASKS

*COBOL for OS/390 & VM Compiler and Run-Time Migration Guide*

### RELATED REFERENCES

"`CMPR2`" on page 264

---

## Writing routines for handling errors

You can handle most error conditions that might occur while your program is running by using the `ON EXCEPTION` phrase, the `ON SIZE ERROR` phrase, and other language constructs. But if an extraordinary condition like a machine check occurs, normally your application will not regain control—it will be abnormally terminated.

However, `COBOL for OS/390 & VM` and `Language Environment` provide a way for a user-written program to gain control when such conditions occur. Using

Language Environment condition handling you can write your own error-handling programs in COBOL. They can report, analyze, or even fix up and allow your program to resume running.

To have Language Environment pass control to your user-written error program, you must first identify and register its entry point to Language Environment. PROCEDURE-POINTER data items allow you to pass the entry address of procedure entry points to Language Environment services.

**RELATED TASKS**

“Using procedure pointers” on page 472

---

## Part 2. Compiling and debugging your program

<b>Chapter 13. Compiling under OS/390.</b> . . . . .	203	<b>Chapter 14. Compiling under OS/390 UNIX</b> . . . . .	235
Compiling with JCL . . . . .	203	Setting environment variables under OS/390 UNIX	235
Using a cataloged procedure . . . . .	204	Specifying compiler options under OS/390 UNIX	236
Compile procedure (IGYWC) . . . . .	205	Compiling and linking with the cob2 command	237
Compile and link-edit procedure (IGYWCL)	206	Defining input and output . . . . .	237
Compile, link-edit, and run procedure		Creating a DLL. . . . .	238
(IGYWCLG) . . . . .	207	Example: using cob2 to compile under OS/390	
Compile, load, and run procedure (IGYWCG)	208	UNIX . . . . .	238
Compile, prelink, and link-edit procedure		cob2 . . . . .	239
(IGYWCPL) . . . . .	209	cob2 input and output files. . . . .	240
Compile, prelink, link-edit, and run		Compiling using scripts . . . . .	241
procedure (IGYWCPLG). . . . .	210		
Prelink and link-edit procedure (IGYWPL)	211	<b>Chapter 15. Compiling under CMS</b> . . . . .	243
Compile, prelink, load, and run procedure		Accessing the compiler (CP LINK and ACCESS)	243
(IGYWCPLG) . . . . .	212	Specifying a source program to compile . . . . .	244
Writing JCL to compile programs. . . . .	213	COBOL2 . . . . .	244
Example: user-written JCL for compiling . . . . .	214	Using compiler-directing statements. . . . .	245
Compiling under TSO . . . . .	215	Specifying compiler options under CMS . . . . .	246
Example: ALLOCATE and CALL for compiling		Using parentheses in COBOL2 . . . . .	246
under TSO . . . . .	215	Abbreviating compiler options . . . . .	246
Example: CLIST for compiling under TSO. . . . .	216	Using logical line-editing characters . . . . .	247
Starting the compiler from an assembler program	216	Additional compiler options under CMS . . . . .	247
Defining compiler input and output. . . . .	218	Differences in compiler options under CMS . . . . .	248
Data sets used by the compiler under OS/390	218	Precedence of compiler options under CMS . . . . .	248
Logical record length and block size. . . . .	219	Files used by the compiler under CMS . . . . .	248
Defining the source code data set (SYSIN). . . . .	220	Compiler options and compiler output under	
Specifying source libraries (SYSLIB) . . . . .	220	CMS . . . . .	249
Defining the output data set (SYSPRINT) . . . . .	221	Naming generated files under CMS . . . . .	251
Directing compiler messages to your terminal		Overriding FILEDEF . . . . .	251
(SYSTERM) . . . . .	221	Using system-generated names . . . . .	251
Creating object code (SYSLIN or SYSPUNCH)	221	Correcting errors . . . . .	252
Creating an associated data file (SYSADATA)	222	Error messages from COBOL2. . . . .	252
Defining the output IDL data set (SYSIDL) . . . . .	222		
Defining the debug data set (SYSDEBUG). . . . .	222	<b>Chapter 16. Compiler options</b> . . . . .	257
Specifying compiler options under OS/390 . . . . .	223	Option settings for COBOL 85 Standard	
Specifying compiler options with the PROCESS		conformance. . . . .	259
(CBL) statement . . . . .	223	Conflicting compiler options . . . . .	259
Example: specifying compiler options using JCL	224	ADATA . . . . .	261
Example: specifying compiler options under		ANALYZE . . . . .	261
TSO . . . . .	224	ADV . . . . .	262
Compiler options and compiler output under		ARITH . . . . .	262
OS/390 . . . . .	224	AWO . . . . .	263
Compiling multiple programs (batch compilation)	226	BUFSIZE . . . . .	263
Example: batch compilation . . . . .	226	CMR2 . . . . .	264
Specifying compiler options in a batch		COMPILE . . . . .	265
compilation . . . . .	227	CURRENCY. . . . .	265
Example: precedence of options in a batch		DATA . . . . .	266
compilation . . . . .	228	External data . . . . .	267
Example: LANGUAGE option in a batch		QSAM input/output buffers . . . . .	267
compilation . . . . .	229	DATEPROC . . . . .	267
Correcting errors in your source program . . . . .	230	DBCS . . . . .	269
Generating a list of compiler error messages . . . . .	230	DECK . . . . .	269
Messages and listings for compiler-detected		DIAGTRUNC . . . . .	269
errors . . . . .	231	DISK/PRINT . . . . .	270
Format of compiler error messages . . . . .	231	DLL . . . . .	270
Severity codes for compiler error messages . . . . .	232	DUMP . . . . .	271

DYNAM . . . . .	272	Checking syntax only. . . . .	314
EXIT . . . . .	272	Compiling conditionally. . . . .	314
EXPORTALL . . . . .	272	Finding line sequence problems . . . . .	315
FASTSRT . . . . .	273	Checking for valid ranges . . . . .	315
FLAG . . . . .	274	Selecting the level of error to be diagnosed . . . . .	316
FLAGMIG . . . . .	275	Example: embedded messages. . . . .	316
FLAGSTD . . . . .	275	Finding program entity definitions and	
IDLGEN . . . . .	277	references . . . . .	317
INTDATE . . . . .	278	Listing data items . . . . .	318
LANGUAGE . . . . .	279	Getting listings . . . . .	319
LIB . . . . .	280	Example: short listing . . . . .	320
LINECOUNT . . . . .	280	Example: SOURCE and NUMBER output . . . . .	323
LIST . . . . .	281	Example: MAP output . . . . .	324
MAP . . . . .	281	Example: embedded map summary . . . . .	325
NAME . . . . .	282	Terms used in MAP output. . . . .	326
NUMBER . . . . .	283	Symbols used in LIST and MAP output . . . . .	326
NUMPROC . . . . .	283	Example: nested program map . . . . .	328
OBJECT . . . . .	284	Reading LIST output . . . . .	328
OFFSET . . . . .	285	Example: program initialization code . . . . .	329
OPTIMIZE . . . . .	285	Signature information bytes: compiler	
Unused data items: . . . . .	286	options . . . . .	331
OUTDD . . . . .	286	Signature information bytes: DATA	
PGMNAME . . . . .	287	DIVISION . . . . .	333
PGMNAME(COMPAT) . . . . .	287	Signature information bytes:	
PGMNAME(LONGUPPER). . . . .	288	ENVIRONMENT DIVISION . . . . .	333
PGMNAME(LONGMIXED) . . . . .	288	Signature information bytes: PROCEDURE	
PGMNAME usage notes. . . . .	288	DIVISION verbs . . . . .	333
QUOTE/APOST . . . . .	289	Signature information bytes: more	
RENT . . . . .	289	PROCEDURE DIVISION items . . . . .	335
RMODE . . . . .	290	Example: assembler code generated from	
SEQUENCE . . . . .	291	source code . . . . .	336
SIZE . . . . .	291	Example: TGT memory map . . . . .	338
SOURCE . . . . .	292	Example: location and size of	
SPACE . . . . .	293	WORKING-STORAGE . . . . .	339
SQL . . . . .	293	Example: DSA memory map . . . . .	339
SSRANGE . . . . .	294	Example: XREF output - data-name	
TERMINAL . . . . .	295	cross-references. . . . .	340
TEST . . . . .	295	Example: XREF output - program-name	
TRUNC . . . . .	297	cross-references. . . . .	341
TRUNC example 1 . . . . .	298	Example: embedded cross-reference . . . . .	341
TRUNC example 2 . . . . .	299	Example: OFFSET compiler output . . . . .	342
TYPECHK . . . . .	300	Example: VBREF compiler output . . . . .	343
VBREF . . . . .	301	Preparing to use the debugger. . . . .	343
WORD . . . . .	301		
XREF . . . . .	302		
YEARWINDOW . . . . .	303		
ZWB . . . . .	304		
Compiler-directing statements . . . . .	304		
<b>Chapter 17. Debugging . . . . .</b>	<b>309</b>		
Debugging with source language. . . . .	310		
Tracing program logic . . . . .	310		
Finding and handling input-output errors . . . . .	311		
Validating data . . . . .	311		
Finding uninitialized data . . . . .	311		
Generating information about procedures . . . . .	312		
Debugging lines . . . . .	312		
Debugging statements . . . . .	312		
Example: USE FOR DEBUGGING . . . . .	312		
Debugging using compiler options . . . . .	313		
Finding coding errors . . . . .	314		



---

## Chapter 13. Compiling under OS/390

You can compile COBOL for OS/390 & VM programs under OS/390 using job control language (JCL), TSO commands, CLISTs, or ISPF panels:

- For compiling with JCL, IBM provides a set of cataloged procedures, which can reduce the amount of JCL coding that you need to write. If the cataloged procedures do not meet your needs, you can write your own JCL. Using JCL, you can compile a single program or compile several programs as part of a batch job.
- When compiling under TSO, you can use TSO commands, CLISTs, or ISPF panels.

You might also want to start the COBOL for OS/390 & VM compiler from an assembler program, such as when your shop has developed a tool or interface that calls the COBOL for OS/390 & VM compiler.

Whether you are compiling using JCL or under TSO, as part of the compilation step, you need to define the data sets needed for the compilation and specify any compiler options necessary for your program and the desired output.

The compiler translates your COBOL program into language that the computer can process (object code). The compiler also lists errors in your source statements and provides supplementary information to help you debug and tune your program. Use compiler-directing statements and compiler options to control your compilation.

After compiling your program, you need to review the results of the compilation and correct any compiler-detected errors.

### RELATED TASKS

“Compiling with JCL”

“Compiling under TSO” on page 215

“Starting the compiler from an assembler program” on page 216

“Defining compiler input and output” on page 218

“Specifying compiler options under OS/390” on page 223

“Compiling multiple programs (batch compilation)” on page 226

“Correcting errors in your source program” on page 230

### RELATED REFERENCES

“Compiler-directing statements” on page 304

“Data sets used by the compiler under OS/390” on page 218

“Compiler options and compiler output under OS/390” on page 224

---

## Compiling with JCL

You need to include the following information in the JCL for compilation:

- Job description
- Statement to run the compiler
- Definitions for the data sets needed (including the directory paths if using HFS files)

The simplest way to compile your program under OS/390 is to code JCL that uses a cataloged procedure. A *cataloged procedure* is a set of job control statements placed in a partitioned data set called the procedure library (SYS1.PROCLIB).

The following JCL shows the general format for a cataloged procedure.

```
//jobname JOB parameters
//stepname EXEC [PROC=]procname[, {PARM=|PARM.stepname=} 'options']
//SYSIN DD data set parameters
. . . (source program to be compiled)
/*
//
```

Additional considerations apply when you use cataloged procedures to compile object-oriented programs.

“Example: sample JCL for an object-oriented application” on page 418

“Example: sample JCL for a procedural DLL application” on page 492

#### RELATED TASKS

“Using a cataloged procedure”

“Writing JCL to compile programs” on page 213

“Specifying compiler options under OS/390” on page 223

“Specifying compiler options in a batch compilation” on page 227

“Compiling programs to create DLLs” on page 490

#### RELATED REFERENCES

“Data sets used by the compiler under OS/390” on page 218

## Using a cataloged procedure

Specify a cataloged procedure in an EXEC statement in your JCL.

The following JCL calls the IBM-supplied cataloged procedure (IGYWC) for compiling a COBOL for OS/390 & VM program and defining the required data sets:

---

```
//JOB1 JOB1
//STEPA EXEC PROC=IGYWC ← (name of the cataloged procedure)
//COBOL.SYSIN DD * ← (or appropriate parameters describing the data set)
000100 IDENTIFICATION DIVISION ← (the source code)
.
.
.
/* (optional delimiter statement)
```

---

You can use these procedures with any of the job schedulers that are part of OS/390. When a scheduler encounters parameters that it does not require, the scheduler either ignores them or substitutes alternative parameters.

If the compiler options are not explicitly supplied with the procedure, default options established at the installation apply. You can override these default options by using an EXEC statement that includes the desired options.

You can specify data sets to be in the hierarchical file system by overriding the corresponding DD statement. However, the compiler utility files (SYSUTx) and copy libraries (SYSLIB) you specify must be MVS data sets.

Additional details on invoking cataloged procedures, overriding and adding to EXEC statements, and overriding and adding to DD statements are in the Language Environment information.

#### RELATED TASKS

*Language Environment Programming Guide*

#### RELATED REFERENCES

“Compile procedure (IGYWC)”  
“Compile and link-edit procedure (IGYWCL)” on page 206  
“Compile, link-edit, and run procedure (IGYWCLG)” on page 207  
“Compile, load, and run procedure (IGYWCG)” on page 208  
“Compile, prelink, and link-edit procedure (IGYWCPL)” on page 209  
“Compile, prelink, link-edit, and run procedure (IGYWCPLG)” on page 210  
“Prelink and link-edit procedure (IGYWPL)” on page 211  
“Compile, prelink, load, and run procedure (IGYWCPG)” on page 212

### Compile procedure (IGYWC)

The IGYWC procedure is a single-step procedure for compiling a program. It produces an object module. The compile steps in all other cataloged procedures that invoke the compiler are similar.

You must supply the following DD statement, indicating the location of the source program, in the input stream.

```
//COBOL.SYSIN DD *          (or appropriate parameters)
```

The following statements make up the IGYWC cataloged procedure.

```
//IGYWC PROC LNGPRFX='IGY.V2R2M0',SYSLBLK=3200
//*
//* COMPILE A COBOL PROGRAM
//*
//* PARAMETER  DEFAULT VALUE  USAGE
//* SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//* LNGPRFX   IGY.V2R2M0      PREFIX FOR LANGUAGE DATA SET NAMES
//*
//* CALLER MUST SUPPLY //COBOL.SYSIN DD . . .
//*
//COBOL EXEC PGM=IGYCRCTL,REGION=2048K
//STEPLIB DD DSNAME=&LNGPRFX..SIGYCOMP,          (1)
//          DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN DD DSNAME=&&LOADSET,UNIT=SYSDA,
//          DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
//          DCB=(BLKSIZE=&SYSLBLK)
//SYSUT1 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5 DD UNIT=SYSDA,SPACE=(CYL,(1,1))          (2)
//SYSUT6 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
```

(1) STEPLIB can be installation-dependent.

(2) SYSUT5 is needed only if the LIB option is used.

“Example: JCL for compiling using HFS”

**Example: JCL for compiling using HFS:** The following job uses procedure IGYWC to compile a COBOL program demo.cbl that is located in the hierarchical

file system (HFS). It writes the generated compiler listing demo.lst, object file demo.o, Interface Definition Language file demo.idl, and SYSADATA file demo.adt to the HFS.

```
//HFSDEMO JOB ,
// TIME=(1),MSGLEVEL=(1,1),MSGCLASS=H,CLASS=A,REGION=50M,
// NOTIFY=&SYSUID,USER=&SYSUID
//COMPILE EXEC IGYWC,
// PARM.COBOL='LIST,MAP,RENT,FLAG(I,I),XREF,ADATA,IDLGEN'
//SYSPRINT DD PATH='/u/userid/cobol/demo.lst',      (1)
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),              (2)
// PATHMODE=SIRWXU,                                (3)
// FILEDATA=TEXT                                    (4)
//SYSLIN DD PATH='/u/userid/cobol/demo.o',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=SIRWXU
//SYSIDL DD PATH='/u/userid/cobol/demo.idl',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=SIRWXU,
// FILEDATA=TEXT
//SYSADATA DD PATH='/u/userid/cobol/demo.adt',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=SIRWXU
//SYSIN DD PATH='/u/userid/cobol/demo.cbl',
// PATHOPTS=ORDONLY,
// FILEDATA=TEXT,
// RECFM=F
```

- (1) PATH specifies the path name for an HFS file.
- (2) PATHOPTS indicates the access for the file (such as read or read-write) and sets the status for the file (such as append, create, or truncate).
- (3) PATHMODE indicates the permissions, or file access attributes, to be set when a file is created.
- (4) FILEDATA specifies whether the data is to be treated as text or binary.

You can use a mixture of HFS (*PATH='hfs-directory-path'*) and MVS data sets (*DSN=traditional-data-set-name*) on the compilation DD statements shown in this example as overrides. However, the compiler utility files (DD statements *SYSUTx*) and COPY libraries (DD statements *SYSLIB*) must be MVS data sets.

#### RELATED REFERENCES

*OS/390 UNIX System Services Command Reference*

*OS/390 MVS JCL Reference*

“Data sets used by the compiler under OS/390” on page 218

### Compile and link-edit procedure (IGYWCL)

The IGYWCL procedure is a two-step procedure to compile and link-edit a program. The COBOL job step produces an object module that is input to the linkage editor. Other object modules can be added.

You must supply the following DD statement, indicating the location of the source program, in the input stream.

```
//COBOL.SYSIN DD *      (or appropriate parameters)
```

The following statements make up the IGYWCL cataloged procedure.

```
//IGYWCL PROC  LNGPRFX='IGY.V2R2M0',SYSLBLK=3200,
//              LIBPRFX='CEE',
//              PGMLIB='&&GOSET',GOPGM=GO
//*
//*  COMPILE AND LINK EDIT A COBOL PROGRAM
```

```

/**
/** PARAMETER  DEFAULT VALUE  USAGE
/** LNGPRFX   IGY.V2R2M0      PREFIX FOR LANGUAGE DATA SET NAMES
/** SYSLBLK   3200             BLOCK SIZE FOR OBJECT DATA SET
/** LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
/** PGMLIB    &&GOSET          DATA SET NAME FOR LOAD MODULE
/** GOPGM     GO               MEMBER NAME FOR LOAD MODULE
/**
/** CALLER MUST SUPPLY //COBOL.SYSIN DD . . .
/**
//COBOL EXEC PGM=IGYCRCTL,REGION=2048K
//STEPLIB DD DSNAME=&LNGPRFX..SIGYCOMP,           (1)
//          DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN  DD DSNAME=&&LOADSET,UNIT=SYSDA,
//          DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
//          DCB=(BLKSIZE=&SYSLBLK)
//SYSUT1  DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2  DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3  DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4  DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5  DD UNIT=SYSDA,SPACE=(CYL,(1,1))           (2)
//SYSUT6  DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7  DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//LKED EXEC PGM=HEWL,COND=(8,LT,COBOL),REGION=1024K
//SYSLIB  DD DSNAME=&LIBPRFX..SCEELKED,           (3)
//          DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN  DD DSNAME=&&LOADSET,DISP=(OLD,DELETE)
//          DD DDNAME=SYSIN
//SYSLMOD DD DSNAME=&PGMLIB(&GOPGM),
//          SPACE=(TRK,(10,10,1)),
//          UNIT=SYSDA,DISP=(MOD,PASS)
//SYSUT1  DD UNIT=SYSDA,SPACE=(TRK,(10,10))

```

- (1) STEPLIB can be installation-dependent.
- (2) SYSUT5 is needed only if the LIB option is used.
- (3) SYSLIB can be installation-dependent.

### Compile, link-edit, and run procedure (IGYWCLG)

The IGYWCLG procedure is a three-step procedure to compile, link-edit, and run a program.

The COBOL job step produces an object module that is input to the linkage editor. Other object modules can be added. If the COBOL program refers to any data sets, DD statements that define these data sets must also be supplied.

You must supply the following DD statement, indicating the location of the source program, in the input stream.

```
//COBOL.SYSIN DD *          (or appropriate parameters)
```

The following shows the statements that make up the IGYWCLG cataloged procedure.

```

//IGYWCLG PROC LNGPRFX='IGY.V2R2M0',SYSLBLK=3200,
//             LIBPRFX='CEE',GOPGM=GO
/**
/** COMPILE, LINK EDIT AND RUN A COBOL PROGRAM
/**
/** PARAMETER  DEFAULT VALUE  USAGE
/** LNGPRFX   IGY.V2R2M0      PREFIX FOR LANGUAGE DATA SET NAMES
/** SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
/** LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
/** GOPGM     GO               MEMBER NAME FOR LOAD MODULE

```

```

/**
/** CALLER MUST SUPPLY //COBOL.SYSIN DD . . .
/**
//COBOL EXEC PGM=IGYCRCTL,REGION=2048K
//STEPLIB DD DSNAME=&LNGPRFX..SIGYCOMP,           (1)
//          DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN   DD DSNAME=&&LOADSET,UNIT=SYSDA,
//          DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
//          DCB=(BLKSIZE=&SYSLBLK)
//SYSUT1   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5   DD UNIT=SYSDA,SPACE=(CYL,(1,1))           (2)
//SYSUT6   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//LKED EXEC PGM=HEWL,COND=(8,LT,COBOL),REGION=1024K
//SYSLIB DD DSNAME=&LIBPRFX..SCEELKED,             (3)
//          DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN   DD DSNAME=&&LOADSET,DISP=(OLD,DELETE)
//          DD DDNAME=SYSIN
//SYSLMOD DD DSNAME=&&GOSET(&GOPGM),SPACE=(TRK,(10,10,1)),
//          UNIT=SYSDA,DISP=(MOD,PASS)
//SYSUT1   DD UNIT=SYSDA,SPACE=(TRK,(10,10))
//GO EXEC PGM=*.LKED.SYSLMOD,COND=((8,LT,COBOL),(4,LT,LKED)),
//          REGION=2048K
//STEPLIB DD DSNAME=&LIBPRFX..SCEERUN,             (1)
//          DISP=SHR
//SYSPRINT DD SYSOUT=*
//CEEDUMP DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*

```

- (1) STEPLIB can be installation-dependent.
- (2) SYSUT5 is needed only if the LIB option is used.
- (3) SYSLIB can be installation-dependent.

### Compile, load, and run procedure (IGYWCG)

The IGYWCG procedure is a two-step procedure to compile, load, and run a program. The COBOL job step produces an object module that is input to the loader. If the COBOL program refers to any data sets, the DD statements that define these data sets must also be supplied.

You must supply the following DD statement, indicating the location of the source program, in the input stream.

```
//COBOL.SYSIN DD *      (or appropriate parameters)
```

The following shows the statements that make up the IGYWCG cataloged procedure.

```

//IGYWCG PROC  LNGPRFX='IGY.V2R2M0',SYSLBLK=3200,
//             LIBPRFX='CEE'
/**
/** COMPILE, LOAD AND RUN A COBOL PROGRAM
/**
/** PARAMETER  DEFAULT VALUE  USAGE
/**  LNGPRFX  IGY.V2R2M0      PREFIX FOR LANGUAGE DATA SET NAMES
/**  SYSLBLK  3200            BLKSIZE FOR OBJECT DATA SET
/**  LIBPRFX  CEE             PREFIX FOR LIBRARY DATA SET NAMES
/**
/** CALLER MUST SUPPLY //COBOL.SYSIN DD . . .
/**
//COBOL EXEC PGM=IGYCRCTL,REGION=2048K

```

```

//STEPLIB DD DSNAME=&LNGPRFX..SIGYCOMP,           (1)
//          DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN DD DSNAME=&&LOADSET,UNIT=SYSDA,           (2)
//          DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
//          DCB=(BLKSIZE=&SYSLBLK)
//SYSUT1 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5 DD UNIT=SYSDA,SPACE=(CYL,(1,1))           (3)
//SYSUT6 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//GO EXEC PGM=LOADER,COND=(8,LT,COBOL),REGION=2048K
//SYSLIB DD DSNAME=&LIBPRFX..SCEELKED,           (4)
//          DISP=SHR
//SYSLOUT DD SYSOUT=*
//SYSLIN DD DSNAME=&&LOADSET,DISP=(OLD,DELETE)
//STEPLIB DD DSNAME=&LIBPRFX..SCEERUN,           (1)
//          DISP=SHR
//SYSPRINT DD SYSOUT=*
//CEEDUMP DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*

```

- (1) STEPLIB can be installation-dependent.
- (2) SYSLIN can reside in the HFS.
- (3) SYSUT5 is needed only if the LIB option is used.
- (4) SYSLIB can be installation-dependent.

### Compile, prelink, and link-edit procedure (IGYWCPL)

The IGYWCPL procedure is a three-step procedure for compiling, prelinking, and link-editing a program.

You must supply the following DD statement, indicating the location of the source program, in the input stream.

```
SYSIN DD *          (or appropriate parameters)
```

The following shows the statements that make up the IGYWCPL cataloged procedure.

```

//IGYWCPL PROC LNGPRFX='IGY.V2R2M0',SYSLBLK=3200,
//          LIBPRFX='CEE',PLANG=EDCPMSGE,
//          PGMLIB='&&GOSET',GOPGM=GO
//*
//* COMPILER, PRELINK AND LINK EDIT A COBOL PROGRAM
//*
//* PARAMETER  DEFAULT VALUE  USAGE
//* LNGPRFX   IGY.V2R2M0      PREFIX FOR LANGUAGE DATA SET NAMES
//* SYSLBLK   3200            BLOCK SIZE FOR OBJECT DATA SET
//* LIBPRFX   CEE             PREFIX FOR LIBRARY DATA SET NAMES
//* PLANG     EDCPMSGE        PRELINKER MESSAGES MODULE
//* PGMLIB    &&GOSET         DATA SET NAME FOR LOAD MODULE
//* GOPGM     GO              MEMBER NAME FOR LOAD MODULE
//*
//* CALLER MUST SUPPLY //COBOL.SYSIN DD . . .
//*
//COBOL EXEC PGM=IGYCRCTL,REGION=2048K
//STEPLIB DD DSNAME=&LNGPRFX..SIGYCOMP,           (1)
//          DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN DD DSNAME=&&LOADSET,UNIT=SYSDA,
//          DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
//          DCB=(BLKSIZE=&SYSLBLK)
//SYSUT1 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2 DD UNIT=SYSDA,SPACE=(CYL,(1,1))

```

```

//SYSUT3 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5 DD UNIT=SYSDA,SPACE=(CYL,(1,1)) (2)
//SYSUT6 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//PLKED EXEC PGM=EDCPRLK,PARM=' ',COND=(8,LT,COBOL),
// REGION=2048K
//STEPLIB DD DSNAME=&LIBPRFX..SCEERUN,
// DISP=SHR
//SYSMSGSD DSNAME=&LIBPRFX..SCEEMSGP(&PLANG),
// DISP=SHR
//SYSLIB DD DUMMY
//SYSIN DD DSN=&&LOADSET,DISP=(OLD,DELETE)
//SYSMOD DD DSNAME=&&PLKSET,UNIT=SYSDA,DISP=(NEW,PASS),
// SPACE=(32000,(100,50)),
// DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSDEFSD DD DUMMY
//SYSOUT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//*
//LKED EXEC PGM=HEWL,COND=(8,LT,COBOL),REGION=1024K
//SYSLIB DD DSNAME=&LIBPRFX..SCEELKED, (3)
// DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN DD DSNAME=&&PLKSET,DISP=(OLD,DELETE)
// DD DDNAME=SYSIN
//SYSLMOD DD DSNAME=&PGMLIB(&GOPGM),
// SPACE=(TRK,(10,10,1)),
// UNIT=SYSDA,DISP=(MOD,PASS)
//SYSUT1 DD UNIT=SYSDA,SPACE=(TRK,(10,10))

```

(1) STEPLIB can be installation-dependent.

(2) SYSUT5 is needed only if the LIB option is used.

(3) SYSLIB can be installation-dependent.

### Compile, prelink, link-edit, and run procedure (IGYWCPLG)

The IGYWCPLG procedure is a four-step procedure for compiling, prelinking, link-editing, and running a program.

You must supply the following DD statement, indicating the location of the source program, in the input stream.

```

SYSIN DD * (or appropriate parameters)

```

The following shows the statements that make up the IGYWCPLG cataloged procedure.

```

//IGYWCPLG PROC LNGPRFX='IGY.V2R2M0',SYSLBLK=3200,
// PLANG=EDCPMSGE,
// LIBPRFX='CEE',GOPGM=GO
//*
//* COMPILER, PRELINK, LINK EDIT, AND RUN A COBOL PROGRAM
//*
//* PARAMETER DEFAULT VALUE USAGE
//* LNGPRFX IGY.V2R2M0 PREFIX FOR LANGUAGE DATA SET NAMES
//* SYSLBLK 3200 BLKSIZE FOR OBJECT DATA SET
//* PLANG EDCPMSGE PRELINKER MESSAGES MODULE
//* LIBPRFX CEE PREFIX FOR LIBRARY DATA SET NAMES
//* GOPGM GO MEMBER NAME FOR LOAD MODULE
//*
//* CALLER MUST SUPPLY //COBOL.SYSIN DD . . .
//*
//COBOL EXEC PGM=IGYCRCTL,REGION=2048K
//STEPLIB DD DSNAME=&LNGPRFX..SIGYCOMP, (1)
// DISP=SHR
//SYSPRINT DD SYSOUT=*

```



```

//SYSLIN DD DSNAME=&&LOADSET,UNIT=SYSDA,
// DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
// DCB=(BLKSIZE=&SYSLBLK)
//SYSUT1 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5 DD UNIT=SYSDA,SPACE=(CYL,(1,1)) (2)
//SYSUT6 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//PLKED EXEC PGM=EDCPRLK,PARM=' ',COND=(8,LT,COBOL),
// REGION=2048K
//STEPLIB DD DSNAME=&LIBPRFX..SCEERUN,
// DISP=SHR
//SYSMSG DD DSNAME=&LIBPRFX..SCEEMSGP(&PLANG),
// DISP=SHR
//SYSLIB DD DUMMY
//SYSIN DD DSN=&&LOADSET,DISP=(OLD,DELETE)
//SYSMOD DD DSNAME=&&PLKSET,UNIT=SYSDA,DISP=(NEW,PASS),
// SPACE=(32000,(100,50)),
// DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSDEFSD DD DUMMY
//SYSOUT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
/*
//LKED EXEC PGM=HEWL,COND=(8,LT,COBOL),REGION=1024K
//SYSLIB DD DSNAME=&LIBPRFX..SCEELKED, (3)
// DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN DD DSNAME=&&PLKSET,DISP=(OLD,DELETE)
// DD DDNAME=SYSIN
//SYSMOD DD DSNAME=&&GOSET(&GOPGM),SPACE=(TRK,(10,10,1)),
// UNIT=SYSDA,DISP=(MOD,PASS)
//SYSUT1 DD UNIT=SYSDA,SPACE=(TRK,(10,10))
//GO EXEC PGM=*.LKED.SYSLMOD,COND=((8,LT,COBOL),(4,LT,LKED)),
// REGION=2048K
//STEPLIB DD DSNAME=&LIBPRFX..SCEERUN,
// DISP=SHR
//SYSPRINT DD SYSOUT=*
//CEEDUMP DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*

```

- (1) STEPLIB can be installation-dependent.
- (2) SYSUT5 is needed only if the LIB option is used.
- (3) SYSLIB can be installation-dependent.

### Prelink and link-edit procedure (IGYWPL)

The IGYWPL procedure is a two-step procedure for prelinking and link-editing a program.

The following statements make up the IGYWPL cataloged procedure.

```

//IGYWPL PROC PLANG=EDCPMSGE,SYSLBLK=3200,
// LIBPRFX='CEE',
// PGMLIB='&&GOSET',GOPGM=GO
/*
/* PRELINK AND LINK EDIT A COBOL PROGRAM
/*
/* PARAMETER DEFAULT VALUE USAGE
/* PLANG EDCPMSGE PRELINK MESSAGES MEMBER NAME
/* SYSLBLK 3200 BLKSIZE FOR OBJECT DATA SET
/* LIBPRFX CEE PREFIX FOR LIBRARY DATA SET NAMES
/* PGMLIB &&GOSET DATA SET NAME FOR LOAD MODULE
/* GOPGM GO MEMBER NAME FOR LOAD MODULE
/*
/* CALLER MUST SUPPLY //PLKED.SYSIN DD . . .

```

```

/**
//PLKED EXEC PGM=EDCPRLK,PARM=' ',
// REGION=2048K
//STEPLIB DD DSNNAME=&LIBPRFX..SCEERUN, (1)
// DISP=SHR
//SYMSGS DD DSNNAME=&LIBPRFX..SCEEMSGP(&PLANG),
// DISP=SHR
//SYSLIB DD DUMMY
//SYSMOD DD DSNNAME=&&PLKSET,UNIT=SYSDA,DISP=(NEW,PASS),
// SPACE=(32000,(100,50)),
// DCB=(RECFM=FB,LRECL=80,BLKSIZE=&SYSLBLK)
//SYSDEFSD DD DUMMY
//SYSOUT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
/**
//LKED EXEC PGM=HEWL,COND=(4,LT,PLKED),REGION=1024K
//SYSLIB DD DSNNAME=&LIBPRFX..SCEELKED, (2)
// DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN DD DSNNAME=*.PLKED.SYSMOD,DISP=(OLD,DELETE)
// DD DDNAME=SYSIN
//SYSLMOD DD DSNNAME=&PGMLIB(&GOPGM),SPACE=(TRK,(10,10,1)),
// UNIT=SYSDA,DISP=(MOD,PASS)
//SYSUT1 DD UNIT=SYSDA,SPACE=(TRK,(10,10))
//SYSIN DD DUMMY

```

(1) STEPLIB can be installation-dependent.

(2) SYSLIB can be installation-dependent.

### Compile, prelink, load, and run procedure (IGYWCPG)

The IGYWCPG procedure is a four-step procedure for compiling, prelinking, loading, and running a program.

You must supply the following DD statement, indicating the location of the source program, in the input stream.

```
//COBOL.SYSIN DD * (or appropriate parameters)
```

The following statements make up the IGYWCPG cataloged procedure.

```

//IGYWCPG PROC LNGPRFX='IGY.V2R2M0',SYSLBLK=3200,
// PLANG=EDCPMSGE,
// LIBPRFX='CEE'
/**
/** COMPILE, PRELINK, LOAD, AND RUN A COBOL PROGRAM
/**
/** PARAMETER DEFAULT VALUE USAGE
/** LNGPRFX IGY.V2R2M0 PREFIX FOR LANGUAGE DATA SET NAMES
/** SYSLBLK 3200 BLKSIZE FOR OBJECT DATA SET
/** PLANG EDCPMSGE PRELINKER MESSAGES MODULE
/** LIBPRFX CEE PREFIX FOR LIBRARY DATA SET NAMES
/**
/** CALLER MUST SUPPLY //COBOL.SYSIN DD . . .
/**
//COBOL EXEC PGM=IGYCRCTL,REGION=2048K
//STEPLIB DD DSNNAME=&LNGPRFX..SIGYCOMP, (1)
// DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN DD DSNNAME=&&LOADSET,UNIT=SYSDA,
// DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
// DCB=(BLKSIZE=&SYSLBLK)
//SYSUT1 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5 DD UNIT=SYSDA,SPACE=(CYL,(1,1)) (2)

```

```

//SYSUT6 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//PLKED EXEC PGM=EDCPRLK,PARM=' ',COND=(8,LT,COBOL),
//
//          REGION=2048K
//STEPLIB DD DSNAME=&LIBPRFX..SCEERUN,
//          DISP=SHR
//SYSMSGG DD DSNAME=&LIBPRFX..SCEEMSGP(&PLANG),
//          DISP=SHR
//SYSLIB DD DUMMY
//SYSIN DD DSN=&&LOADSET,DISP=(OLD,DELETE)
//SYSMOD DD DSNAME=&&PLKSET,UNIT=SYSDA,DISP=(NEW,PASS), (3)
//          SPACE=(32000,(100,50)),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSDEFSD DD DUMMY
//SYSOUT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//*
//GO EXEC PGM=LOADER,COND=(8,LT,COBOL),REGION=2048K
//SYSLIB DD DSNAME=&LIBPRFX..SCEELKED, (4)
//          DISP=SHR
//SYSLOUT DD SYSOUT=*
//SYSLIN DD DSNAME=&&PLKSET,DISP=(OLD,DELETE)
//STEPLIB DD DSNAME=&LIBPRFX..SCEERUN,
//          DISP=SHR
//SYSPRINT DD SYSOUT=*
//CEEDUMP DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*

```

- (1) STEPLIB can be installation-dependent.
- (2) SYSUT5 is needed only if the LIB option is used.
- (3) SYSMOD can reside in the HFS.
- (4) SYSLIB can be installation-dependent.

## Writing JCL to compile programs

If the cataloged procedures do not give you the OS/390 programming flexibility you need for more complex programs, write your own job control statements. The following JCL shows the general format used to compile a program:

```

//jobname JOB acctno,name,MSGCLASS=1 (1)
//stepname EXEC PGM=IGYCRCTL,PARM=(options) (2)
//STEPLIB DD DSNAME=IGY.V2R2M0.SIGYCOMP,DISP=SHR (3)
//SYSUT1 DD UNIT=SYSDA,SPACE=(subparms) (4)
//SYSUT2 DD UNIT=SYSDA,SPACE=(subparms)
//SYSUT3 DD UNIT=SYSDA,SPACE=(subparms)
//SYSUT4 DD UNIT=SYSDA,SPACE=(subparms)
//SYSUT5 DD UNIT=SYSDA,SPACE=(subparms)
//SYSUT6 DD UNIT=SYSDA,SPACE=(subparms)
//SYSUT7 DD UNIT=SYSDA,SPACE=(subparms)
//SYSPRINT DD SYSOUT=A (5)
//SYSLIN DD DSNAME=MYPROG,UNIT=SYSDA, (6)
//          DISP=(MOD,PASS),SPACE=(subparms)
//SYSIN DD DSNAME=dsname,UNIT=device, (7)
//          VOLUME=(subparms),DISP=SHR

```

- (1) The JOB statement indicates the beginning of a job.
- (2) The EXEC statement specifies that the COBOL for OS/390 & VM compiler (IGYCRCTL) is to be invoked.
- (3) This DD statement defines the data set where the COBOL for OS/390 & VM compiler resides.
- (4) The SYSUT DD statements define the utility data sets that the compiler will use to process the source program. All SYSUT files must be on direct-access storage devices.

- (5) The SYSPRINT DD statement defines the data set that receives output from options such as LIST and MAP. SYSOUT=A is the standard designation for data sets whose destination is the system output device.
- (6) The SYSLIN DD statement defines the data set that receives output from the OBJECT option (the object module).
- (7) The SYSIN DD statement defines the data set to be used as input to the job step (source code).

You can use a mixture of HFS (PATH=*hfs-directory-path*) and MVS data sets (DSN=*traditional-data-set-name*) on the compilation DD statements for the following data sets:

- Sources files
- Object files
- Listings
- Interface Definition Language files
- ADATA files
- Debug files
- Executable modules

However, the compiler utility files (DD statements SYSUTx) and COPY files (DD statement SYSLIB) must be MVS data sets.

“Example: user-written JCL for compiling”

“Example: sample JCL for a procedural DLL application” on page 492

“Example: sample JCL for an object-oriented application” on page 418

#### RELATED REFERENCES

*OS/390 MVS JCL Reference*

### Example: user-written JCL for compiling

This example shows a few possibilities for adapting the basic JCL.

```
//JOB1      JOB                               (1)
//STEP1    EXEC PGM=IGYCRCTL,PARM='OBJECT'    (2)
//STEPLIB  DD  DSNAME=IGY.V2R2M0.SIGYCOMP,DISP=SHR
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT6   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSPRINT DD  SYSOUT=A
//SYSLIN   DD  DSNAME=MYPROG,UNIT=SYSDA,
//          DISP=(MOD,PASS),SPACE=(TRK,(3,3))
//SYSIN    DD  *                               (3)
000100 IDENTIFICATION DIVISION.
. . .
/*                                             (4)
```

- (1) JOB1 is the name of the job.
- (2) STEP1 is the name of the single job step in the job. The EXEC statement also specifies that the generated object code be placed on disk or tape to be used later as input to the link step.
- (3) The asterisk indicates that the input data set follows in the input stream.

- (4) The delimiter statement `/*` separates data from subsequent control statements in the input stream.

---

## Compiling under TSO

Under TSO, you can use TSO commands, command lists (CLISTs), REXX execs, or ISPF to compile your program using traditional MVS data sets. You can use TSO commands or REXX execs to compile your program using HFS files. With each method, you need to allocate the data sets and request the compilation.

1. Use the `ALLOCATE` command to allocate data sets.

For any compilation, allocate the work data sets (`SYSUTn`), and the `SYSIN` and `SYSPRINT` data sets. If you specify certain compiler options, allocate the following data sets:

- If you specified the `OBJECT` compiler option, allocate the `SYSLIN` data set to produce an object module.
- If you specified the `TERMINAL` option, allocate the `SYSTEM` data set to get compiler messages at your terminal.
- If you specified the `LIB` option and if you have used `COPY` or `REPLACE` statements in your COBOL for OS/390 & VM program, allocate the `SYSLIB` data set. This must be a traditional MVS data set, not an HFS path.
- If you specified the `LIB` option in your program, allocate `SYSUT5`. This utility data set must be a traditional MVS data set, not an HFS path.

You can allocate data sets in any order. However, you must allocate all needed data sets before you start to compile.

2. Use the `CALL` command to request compilation:

```
READY  
CALL 'IGY.V2R2M0.SIGYCOMP(IGYCRCTL)'
```

You can specify the `ALLOCATE` and `CALL` commands on the TSO command line, or, if you are not using HFS files, include them in a CLIST.

You can allocate HFS files for all the compiler data sets except the `SYSUTx` utility data sets and the `SYSLIB` libraries. Your `ALLOCATE` statements have the following form:

```
Allocate File(SYSIN) Path('/u/myu/myap/std/prog2.cb1')  
Pathopts(ORDONLY) Filedata(TEXT)
```

“Example: `ALLOCATE` and `CALL` for compiling under TSO”

“Example: CLIST for compiling under TSO” on page 216

### Example: `ALLOCATE` and `CALL` for compiling under TSO

The following example shows how to specify `ALLOCATE` and `CALL` commands when you are compiling under TSO.

```
READY  
ALLOCATE FILE(SYSUT1) CYLINDERS SPACE(1 1)  
READY  
ALLOCATE FILE(SYSUT2) CYLINDERS SPACE(1 1)  
READY  
ALLOCATE FILE(SYSUT3) CYLINDERS SPACE(1 1)  
READY  
ALLOCATE FILE(SYSUT4) CYLINDERS SPACE(1 1)  
READY  
ALLOCATE FILE(SYSUT5) CYLINDERS SPACE(1 1)  
READY  
ALLOCATE FILE(SYSUT6) CYLINDERS SPACE(1 1)
```

```

READY
ALLOCATE FILE(SYSUT7) CYLINDERS SPACE(1 1)
READY
ALLOCATE FILE(SYSPRINT) SYSOUT
READY
ALLOCATE FILE(SYSTEM) DATASET(*)
READY
ALLOCATE FILE(SYSLIN) DATASET(PROG2.OBJ) NEW TRACKS SPACE(3,3)
READY
ALLOCATE FILE(SYSIN) DATASET(PROG2.COBOL) SHR
READY
CALL 'IGY.V2R2M0.SIGYCOMP(IGYCRCTL)' 'LIST,NOCOMPILE(S),OBJECT,FLAG(E,E),TERMINAL'
.
(COBOL listings and messages)
.
READY
FREE FILE(SYSUT1,SYSUT2,SYSUT3,SYSUT4,SYSUT5,SYSUT6,SYSUT7,SYSPRINT,SYSTEM,SYSIN,SYSLIN)
READY

```

## Example: CLIST for compiling under TSO

In the following sample CLIST for compiling under TSO, the FREE commands are not required. However, good programming practice dictates that you free your files before you allocate them.

```

PROC 1 MEM
CONTROL LIST
FREE (SYSUT1)
FREE (SYSUT2)
FREE (SYSUT3)
FREE (SYSUT4)
FREE (SYSUT5)
FREE (SYSUT6)
FREE (SYSUT7)
FREE (SYSPRINT)
FREE (SYSIN)
FREE (SYSLIN)
ALLOC F(SYSPRINT) SYSOUT
ALLOC F(SYSIN) DA(COBOL.SOURCE(&MEM)) SHR REUSE
ALLOC F(SYSLIN) DA(COBOL.OBJECT(&MEM)) OLD REUSE
ALLOC F(SYSUT1) NEW SPACE(5,5) TRACKS UNIT(SYSDA)
ALLOC F(SYSUT2) NEW SPACE(5,5) TRACKS UNIT(SYSDA)
ALLOC F(SYSUT3) NEW SPACE(5,5) TRACKS UNIT(SYSDA)
ALLOC F(SYSUT4) NEW SPACE(5,5) TRACKS UNIT(SYSDA)
ALLOC F(SYSUT5) NEW SPACE(5,5) TRACKS UNIT(SYSDA)
ALLOC F(SYSUT6) NEW SPACE(5,5) TRACKS UNIT(SYSDA)
ALLOC F(SYSUT7) NEW SPACE(5,5) TRACKS UNIT(SYSDA)
CALL 'IGY.V2R2M0.SIGYCOMP(IGYCRCTL)'

```

---

## Starting the compiler from an assembler program

An assembler program can start the COBOL for OS/390 & VM compiler by using the ATTACH or the LINK macro instruction, by dynamic invocation. Using dynamic invocation, you must supply the following information to the COBOL for OS/390 & VM compiler:

- Options to be specified for the compilation
- ddnames of the data sets to be used during processing by the COBOL for OS/390 & VM compiler

### Format

```

symbol {LINK|ATTACH} EP=IGYCRCTL,
        PARAM=(optionlist[,ddnamelist]),VL=1

```

**EP** Specifies the symbolic name of the COBOL for OS/390 & VM compiler. The control program (from the library directory entry) determines the entry point at which the program should begin running.

**PARAM** Specifies, as a sublist, address parameters to be passed from the assembler program to the COBOL for OS/390 & VM compiler. The first fullword in the address parameter list contains the address of the COBOL *optionlist*. The second fullword contains the address of the *ddnamelist*.

The third and fourth fullwords contain the addresses of null parameters, or zero.

#### *optionlist*

Specifies the address of a variable-length list containing the COBOL options specified for compilation. This address must be written although no list is provided.

The *optionlist* must begin on a halfword boundary. The two high-order bytes contain a count of the number of bytes in the remainder of the list. If no options are specified, the count must be zero. The *optionlist* is free form, with each field separated from the next by a comma. No blanks or zeros should appear in the list. The compiler recognizes only the first 100 characters in the list.

#### *ddnamelist*

Specifies the address of a variable-length list containing alternative ddnames for the data sets used during COBOL compiler processing. If standard ddnames are used, the *ddnamelist* can be omitted.

The *ddnamelist* must begin on a halfword boundary. The two high-order bytes contain a count of the number of bytes in the remainder of the list. Each name of fewer than 8 bytes must be left-justified and padded with blanks. If an alternate ddname is omitted from the list, the standard name will be assumed. If the name is omitted within the list, the 8-byte entry must contain binary zeros. You can omit names from the end by shortening the list.

All SYSUT $n$  data sets specified must be on direct-access storage devices and have physical sequential organization. They must not reside in the HFS.

The following table shows the sequence of the 8-byte entries in the *ddnamelist*.

ddname 8-byte entry	Name for which substituted
1	SYSLIN
2	Not applicable
3	Not applicable
4	SYSLIB
5	SYSIN
6	SYSPRINT
7	SYSPUNCH
8	SYSUT1
9	SYSUT2
10	SYSUT3
11	SYSUT4
12	SYSTEM
13	SYSUT5
14	SYSUT6
15	SYSUT7

<b>ddname</b> 8-byte entry	<b>Name for which substituted</b>
16	SYSADATA
17	SYSIDL
18	SYSDEBUG

**VL** Specifies that the sign bit is to be set to 1 in the last fullword of the address parameter list.

When the COBOL for OS/390 & VM compiler completes processing, it puts a return code in register 15.

---

## Defining compiler input and output

You need to define several kinds of data sets that the compiler uses to do its work. The compiler takes input data sets and libraries and produces various types of output, including object code, listings, and messages. The compiler also uses utility data sets during compilation.

### RELATED TASKS

“Defining the source code data set (SYSIN)” on page 220

“Specifying source libraries (SYSLIB)” on page 220

“Defining the output data set (SYSPRINT)” on page 221

“Directing compiler messages to your terminal (SYSTEM)” on page 221

“Creating object code (SYSLIN or SYSPUNCH)” on page 221

“Creating an associated data file (SYSADATA)” on page 222

“Defining the output IDL data set (SYSIDL)” on page 222

“Defining the debug data set (SYSDEBUG)” on page 222

### RELATED REFERENCES

“Data sets used by the compiler under OS/390”

## Data sets used by the compiler under OS/390

The following table lists the function, device requirements, and allowable device classes for each data set that the compiler uses.

Type	ddname	Function	Required?	Device requirements	Allowable device classes	Reside on HFS?
Input	SYSIN <sup>1</sup>	Reads source program	Yes	Card reader; intermediate storage	Any	Yes
	SYSLIB or other copy libraries <sup>1</sup>	Reads user source libraries (PDSs)	If program has COPY or BASIS statements (LIB is required)	Direct access	SYSDA	No
	SOMENV	Reads SOM profile when OO COBOL programs are compiled	If TYPECHK or IDLGEN is in effect			



Type	ddname	Function	Required?	Device requirements	Allowable device classes	Reside on HFS?
Utility	SYSUT1, SYSUT2, SYSUT3, SYSUT4, SYSUT6 <sup>2</sup>	Work data set used by compiler during compilation	Yes	Direct access	SYSDA	No
	SYSUT5 <sup>2</sup>	Work data set used by compiler during compilation	If program has COPY, REPLACE, or BASIS statements (LIB is required )	Direct access	SYSDA	No
	SYSUT7 <sup>2</sup>	Work data set used by compiler to create listing	Yes	Direct access	SYSDA	No
Output	SYSPRINT <sup>1</sup>	Writes storage map, listings, and messages	Yes	Printer; intermediate storage	SYSSQ, SYSDA, standard output class A	Yes
	SYSTEM	Writes progress and diagnostic messages	If TERM is in effect	Output device; TSO terminal		Yes
	SYSPUNCH	Creates object code	If DECK is in effect	Card punch; direct access	SYSSQ, SYSDA	Yes
	SYSLIN	Creates object module data set as output from compiler and input to linkage editor	If OBJECT is in effect	Direct access	SYSSQ, SYSDA	Yes
	SYSADATA	Writes associated data file records	If ADATA or EVENTS is in effect	Output device		Yes
	SYSIDL	Creates SOM IDL for class definitions	If IDLGEN is in effect	Direct access		Yes
	SYSUDUMP, SYSABEND, or SYSDUMP	Writes dump	If DUMP is in effect (should be rarely used)	Direct access	SYSDA	Yes
	SYSDEBUG	Writes symbolic debug information tables to a data set separate from the object module	If TEST(...,SYM,SEPARATE) is in effect	Direct access	SYSDA	Yes

1. You can use the EXIT option to provide user exits from these data sets.
2. These data sets must be single volume.

#### RELATED REFERENCES

“Logical record length and block size”  
“EXIT” on page 272

### Logical record length and block size

For compiler data sets other than the work data sets (SYSUT*n*) and HFS files, you can set the block size using the BLKSIZE subparameter of the DCB parameter. The value must be permissible for the device on which the data set resides. The values you set depend on whether the data sets are fixed length or variable length.

For fixed-length records (RECFM=F or RECFM=FB), LRECL is the logical record length, and BLKSIZE equals LRECL multiplied by  $n$ , where  $n$  is equal to the blocking factor. The following table shows the defined values for the fixed-length data sets. In general, you should not change these values, but you can change the value for:

- SYSIDL, SYSDEBUG: You can specify any LRECL in the listed range, with 255 recommended.
- SYSPRINT, SYSDEBUG: You can specify BLKSIZE=0, which results in a system-determined block size.

Data set	RECFM	LRECL (bytes)	BLKSIZE <sup>1</sup>
SYSIN	F or FB	80	80 x $n$
SYSLIN	F or FB	80	80 x $n$
SYSPUNCH	F or FB	80	80 x $n$
SYSLIB or other copy libraries	F or FB	80	80 x $n$
SYSPRINT <sup>2</sup>	F or FB	133	133 x $n$
SYSTEM	F or FB	80	80 x $n$
SYSIDL	F or FB	80 to 255	LRECL x $n$
SYSDEBUG <sup>2</sup>	F or FB	80 to 1024	LRECL x $n$

1.  $n$  = blocking factor  
2. If you specify BLKSIZE=0, the system will determine the block size.

For variable-length records (RECFM=V), LRECL is the logical record length, and BLKSIZE equals LRECL plus 4.

Data set	RECFM	LRECL (bytes)	BLKSIZE (bytes) minimum acceptable value
SYSADATA	VB	1020	1024

## Defining the source code data set (SYSIN)

Define the data set that contains your source code with the SYSIN DD statement, as in:

```
//SYSIN DD DSNAME=dsname,UNIT=SYSSQ,
// VOLUME=(subparms),DISP=SHR
```

You can place your source code or BASIS statement directly in the input stream. If you do, use this SYSIN DD statement:

```
//SYSIN DD *
```

When you use the DD \* convention, the source code or BASIS statement must follow the statement. If another job step follows the compilation, the EXEC statement for that step follows the /\* statement or the last source statement.

## Specifying source libraries (SYSLIB)

Add the SYSLIB DD statements if your program contains COPY or BASIS statements. These DD statements define the libraries (partitioned data sets) that contain the data requested by COPY statements (in the source code) or by a BASIS statement in the input stream.

```
//SYSLIB DD DSNAME=copylibname,DISP=SHR
```

Concatenate multiple DD statements if you have multiple copy or basis libraries.

```
//SYSLIB DD DSNAME=PROJECT.USERLIB,DISP=SHR
//      DD DSNAME=SYSTEM.COPYX,DISP=SHR
```

Libraries are on direct-access storage devices. They cannot be in the hierarchical file system when you compile using JCL or under TSO.

You do not need the SYSLIB DD statement if the NOLIB option is in effect.

## Defining the output data set (SYSPRINT)

You can use SYSPRINT to produce a listing, as in:

```
//SYSPRINT DD SYSOUT=A
```

You can direct the output to a SYSOUT data set, a printer, a direct-access storage device, or a magnetic-tape device. The listing includes the results of the default or requested options of the PARM parameter (that is, diagnostic messages, the object code listing).

Specify the following for the data set that you define:

- The name of a sequential data set, the name of a PDS or PDSE member, or an HFS path.
- A data set LRECL of 133.
- A data set RECFM of F or B.
- A block size, using the BLKSIZE subparameter of the DCB parameter, or let the system set the system-determined default block size.

## Directing compiler messages to your terminal (SYSTEM)

If you are compiling under TSO, you can define the SYSTEM data set to send compiler messages to your terminal, as in:

```
ALLOC F(SYSTEM) DA(*)
```

You can define SYSTEM in various ways, including as a SYSOUT data set, a data set on disk, a file in the HFS, or to another print class.

## Creating object code (SYSLIN or SYSPUNCH)

When using the OBJECT compiler option, you can store the object code on disk as a traditional MVS data set or an HFS file or on tape. The compiler uses the file that you define in the SYSLIN or SYSPUNCH DD statement to store the object code.

```
//SYSLIN DD DSNAME=dsname,UNIT=SYSDA,
//      SPACE=(subparms),DISP=(MOD,PASS)
```

Use the DISP parameter of the SYSLIN DD statement to indicate whether the object code data set is to be:

- Passed to the linkage editor
- Cataloged
- Kept
- Added to an existing cataloged library

In the example above, the data is created and passed to another job step, the linkage editor job step.

Your installation might use the DECK option and the SYSPUNCH DD statement.

```
//SYSPUNCH DD SYSOUT=B
```

B is the standard output class for punch data sets.

You do not need the SYSLIN DD statement if the NOOBJECT option is in effect. You do not need the SYSPUNCH DD statement if the NODECK option is in effect.

RELATED REFERENCES

“OBJECT” on page 284

“DECK” on page 269

## Creating an associated data file (SYSADATA)

Define a SYSADATA data set if you use either the ADATA compiler option or the EVENTS compiler option.

```
//SYSADATA DD DSNAME=dsname,UNIT=SYSDA
```

The file defined in the SYSADATA DD statement will be a sequential file containing specific record types that have information about the program collected during compilation. The file can be a traditional MVS data set or an HFS file.

RELATED REFERENCES

“ADATA” on page 261

## Defining the output IDL data set (SYSIDL)

Add the SYSIDL DD statement if you are using the IDLGEN compiler option to request a SOM Interface Definition Language (IDL) file for object-oriented COBOL class definitions.

```
//SYSIDL DD DSNAME=pdsname(member),DISP=(MOD,PASS)
```

The file can be a traditional MVS data set as shown, or an HFS file.

## Defining the debug data set (SYSDEBUG)

When you compile from JCL or from TSO and specify the TEST(. . .,SYM,SEPARATE) compiler option, the symbolic debug information tables are written to the data set that you specify on the SYSDEBUG DD statement, as in:

```
//SYSDEBUG DD DSNAME=dsname,UNIT=SYSDA
```

Specify the following for the data set that you define:

- The name of a sequential data set, the name of a PDS or PDSE member, or an HFS path.
- A data set LRECL greater than or equal to 80, and less than or equal to 1024. The default LRECL for SYSDEBUG is 1024.
- A data set RECFM of F or FB.
- A block size, using the BLKSIZE subparameter of the DCB parameter, or let the system set the system-determined default block size.

RELATED REFERENCES

“TEST” on page 295

---

## Specifying compiler options under OS/390

The compiler is installed and set up with default compiler options. While installing the compiler, the system programmers for a site can fix compiler option settings to, for example, ensure better performance or maintain certain standards. You cannot override any compiler options that your site has set as fixed. For options that are not fixed, you can override the default settings by specifying compiler options in either of these ways:

- Code them on the PROCESS or CBL statement in your COBOL source.
- Include them when you start the compiler, either on the PARM parameter on the EXEC statement in your JCL or on the command line under TSO.

The compiler recognizes the options in the following order of precedence from highest to lowest:

1. Installation defaults that are fixed by your site
2. Values of the BUFSIZE, LIB, OUTDD, SIZE, and SQL compiler options in effect for the first program in a batch
3. Options specified on PROCESS (or CBL) statements, preceding the Identification Division
4. Options specified on the compiler invocation (JCL PARM parameter or the TSO CALL command)
5. Installation defaults that are not fixed

This order of precedence also determines which options are in effect when conflicting or mutually exclusive options are specified.

Most of the options come in pairs; you select one or the other. For example, the option pair for a cross-reference listing is XREF|NOXREF. If you want a cross-reference listing, specify XREF. If you do not want one, specify NOXREF.

Some options have subparameters. For example, if you want 44 lines per page on your listings, specify LINECOUNT(44).

“Example: specifying compiler options using JCL” on page 224

“Example: specifying compiler options under TSO” on page 224

### RELATED TASKS

“Specifying compiler options with the PROCESS (CBL) statement”

“Specifying compiler options in a batch compilation” on page 227

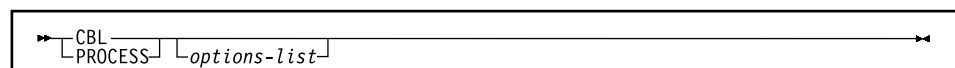
### RELATED REFERENCES

“Conflicting compiler options” on page 259

“Chapter 16. Compiler options” on page 257

## 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.

### Example: specifying compiler options using JCL

The following example shows how to specify compiler options under OS/390 using JCL:

```

//STEP1 EXEC PGM=IGYCRCTL,
//          PARM='LIST,NOCOMPILE(S),OBJECT,FLAG(E,E)'
```

### Example: specifying compiler options under TSO

The following example shows how to specify compiler options under TSO:

```

. . .
READY

CALL 'SYS1.LINKLIB(IGYCRCTL)' 'LIST,NOCOMPILE(S),OBJECT,FLAG(E,E)'
```

### Compiler options and compiler output under OS/390

When the compiler finishes processing your source program, it will have produced one or more of the following, depending on the compiler options you selected:

Compiler option	Compiler output	Type of output
ADATA or EVENTS	Information about the program being compiled	Associated data file
DLL	Object module that is enabled for DLL support	Object
DUMP	System dump, if compilation ended with abnormal termination (requires SYSUDUMP, SYSABEND, or SYSMDUMP DD statement); should be rarely used	Listing
EXPORTALL	Exported symbols for a DLL	Object
FLAG	List of errors that the compiler found in your program	Listing
IDLGEN	Interface Definition Language (IDL) file for your object-oriented COBOL class definition	Object

Compiler option	Compiler output	Type of output
LIST	Listing of object code in machine and assembler language	Listing
MAP	Map of the data items in your program	Listing
NUMBER	User-supplied line numbers shown in listing	Listing
OBJECT or DECK with COMPILE	Your object code	Object
OFFSET	Map of the relative addresses in your object code	Listing
OPTIMIZE	Optimized object code if OBJECT in effect	Object
RENT	Reentrant object code if OBJECT in effect	Object
SOURCE	Listing of your source program	Listing
SQL	SQL statements and host variable information for DB2 bind process	Database request module
SSRANGE	Extra code for checking references within tables	In object
TERMINAL	Progress and diagnostic messages sent to terminal	Terminal
TEST( <i>hook location suboption</i> )	Compiled-in hooks for Debug Tool	Extra code in object
TEST(SYM,NOSEP)	Information tables for Debug Tool and for formatted dumps	Object
TEST(SYM,SEP)	Information tables for Debug Tool and for formatted dumps	Debug information side file
VBREF	Cross-reference listing of verbs in your source program	Listing
XREF	Sorted cross-reference listing of names of procedures, programs, and data	Listing

Listing output from compilation will be in the data set defined by SYSPRINT; object output will be in SYSLIN or SYSPUNCH. Progress and diagnostic messages can be directed to the SYSTEM data set as well as included in the SYSPRINT data set. Associated data file records will be in the data set defined by SYSADATA. The database request module (DBRM) will be the data set defined in DBRMLIB. The debug information side file will be the data set defined in SYSDEBUG.

Save the listings you produced during compilation. You can use them during the testing of your work if you need to debug or tune.

After compilation, your next step will be to fix any errors that the compiler found in your program.

If no errors are detected, you can go to the next step in the process: link-editing, or binding, your program. (If you used compiler options to suppress object code generation, you must recompile to obtain it.)

#### RELATED REFERENCES

“Messages and listings for compiler-detected errors” on page 231  
“Chapter 16. Compiler options” on page 257

---

## Compiling multiple programs (batch compilation)

You can compile a sequence of separate COBOL programs with a single invocation of the compiler. You can link the object program produced from this compilation into one load module or separate load modules, controlled by the NAME compiler option.

You can also use a single invocation of the compiler to compile a sequence of class definitions or a sequence that mixes class definitions and program definitions.

When you compile several programs as part of a batch job, you need to:

- Determine whether you want to create one or more load modules.
- Terminate each program in the sequence.
- Specify compiler options, with an awareness of the effect of compiler options specified in programs within the batch job.

To create separate load modules, precede each set of modules with the NAME option. When the compiler encounters a NAME compiler option, the first program in the sequence and all subsequent programs are link-edited into a single load module, until the next NAME compiler option is encountered. Then, each successive program that is compiled with the NAME option is included in a separate load module.

Use the END PROGRAM header to terminate each program in the sequence (except the last program in the batch for which the END PROGRAM header is optional). Alternatively, you can precede each program in the sequence with a CBL or PROCESS statement. (If the CMPR2 compiler option is in effect, separate each program with the CBL form of the PROCESS or CBL statement.)

If you omit the END PROGRAM header from a program (other than the last program in a sequence of separate programs), the next program in the sequence will be nested in the preceding program. An error can occur in either of the following situations:

- A PROCESS statement is in a program that is now nested.
- A CBL statement is not coded entirely in the sequence number area (columns 1 through 6).

If a CBL statement is coded entirely in the sequence number area (columns 1 through 6), no error message is issued for the CBL statement because it is considered a label for the source statement line.

“Example: batch compilation”

### RELATED TASKS

“Specifying compiler options in a batch compilation” on page 227

### RELATED REFERENCES

“NAME” on page 282

“CMPR2” on page 264

## Example: batch compilation

The following example shows a batch compilation for three programs (PROG1, PROG2, and PROG3) creating two load modules, using one invocation of the IGYWCL cataloged procedure:

- PROG1 and PROG2 are link-edited together to form one load module with a name of PROG2. The entry point of this load module defaults to the first program in the load module, PROG1.



- PROG3 is link-edited by itself into a load module with the name PROG3. Because it is the only program in the load module, the entry point is also PROG3.

```
//jobname JOB acctno,name,MSGLEVEL=1
//stepname EXEC IGYWCL
//COBOL.SYSIN DD *
010100 IDENTIFICATION DIVISION.
010200 PROGRAM-ID PROG1.
. . .
019000 END PROGRAM PROG1.
020100 IDENTIFICATION DIVISION.
020200 PROGRAM-ID PROG2.
. . .
029000 END PROGRAM PROG2.
CBL NAME
030100 IDENTIFICATION DIVISION.
030200 PROGRAM-ID PROG3.
. . .
039000 END PROGRAM PROG3.
/*
//LKED.SYSLMOD DD DSN=&&GOSET (1)
/*
//P2 EXEC PGM=PROG2
//STEPLIB DD DSN=&&GOSET,DISP=(SHR,PASS) (2)
. . . (3)
/*
//P3 EXEC PGM=PROG3
//STEPLIB DD DSN=&&GOSET,DISP=(SHR,PASS) (2)
. . . (3)
/*
//
```

- (1) The data set name for the LKED step SYSLMOD is changed to the temporary name &&GOSET, without any member name.
- (2) The temporary data set &&GOSET is used as the STEPLIB for steps P2 and P3 to run the compiled programs. If the library does not reside in shared storage, the library data set CEE.V2R2M0.SCEERUN must be added as a DD statement for STEPLIB.
- (3) Other DD statements and input that are required to run PROG1 and PROG2 must be added.
- (4) Other DD statements and input that are required to run PROG3 must be added.

#### RELATED REFERENCES

Cataloged procedures for link-editing COBOL programs (*Language Environment Programming Guide*)

## Specifying compiler options in a batch compilation

You can specify compiler options for each program in the batch sequence in either of the usual ways:

- CBL or PROCESS statements preceding a program
- Invocation of the compiler

If a CBL or PROCESS statement is specified in the current program, the compiler resolves the CBL or PROCESS statements together with the options in effect before the first program. If the current program does not contain CBL or PROCESS statements, the compiler uses the settings of options in effect for the previous program.

You should be aware of the effect of certain compiler options on the precedence of compiler option settings for each program in the sequence:

1. Installation defaults that are fixed at your site
2. Values of the BUFSIZE, LIB, OUTDD, SIZE, and SQL compiler options in effect for the first program in the batch
3. Options on CBL or PROCESS statements, if any, for the current program
4. Options specified on the compiler invocation (JCL PARM or TSO CALL)
5. Installation defaults that are not fixed

If any program in the sequence requires the BUF, LIB, OUTDD, SIZE, or SQL option, that option must be in effect for the first program in the batch sequence. (When processing BASIS, COPY, or REPLACE statements, the compiler handles all programs in the batch as a single input file.)

If you specify the LIB option for the batch, you cannot change the NUMBER and SEQUENCE options during the batch compilation. The compiler treats all programs in the batch as a single input file during NUMBER and SEQUENCE processing under the LIB option; therefore, the sequence numbers of the entire input file must be in ascending order.

If the compiler diagnoses the LANGUAGE option on the CBL or PROCESS statement as an error, the language selection reverts to what was in effect before the compiler encountered the first CBL or PROCESS statement. The language in effect during a batch compilation conforms to the rules of processing CBL or PROCESS statements in that environment.

“Example: precedence of options in a batch compilation”

“Example: LANGUAGE option in a batch compilation” on page 229

## Example: precedence of options in a batch compilation

The following listing shows the compiler options hierarchy for a batch compile.

PP 5648-A25 IBM COBOL for OS/390 & VM 2.2.0      Date 03/30/2000 Time 10:34:14 Page 1

INVOCATION PARAMETERS:

NOTERM

PROCESS(CBL) statements:

CBL CMPR2,FLAG(I,I)

Options in effect:      All options are installation defaults unless otherwise noted:

NOADATA

ADV

QUOTE

NOAWO

BUFSIZE(4096)

CMPR2      Process option PROGRAM 1

.

.

FLAG(I,I)      Process option PROGRAM 1

.

.

NOTERM      INVOCATION option

.

.

End of compilation for program 1

.

.

PP 5648-A25 IBM COBOL for OS/390 & VM 2.2.0      Date 03/30/2000 Time 10:34:14 Page 4

PROCESS(CBL) statements:

```

          CBL APOST
Options in effect:
  NOADATA
  ADV
  APOST          Process option in effect for PROGRAM 2
  NOAWO
  BUFSIZE(4096)
  NOCMPR2       Returns to installation option for PROGRAM 2, and subsequent prog
  .
  .
  FLAG(I)       Returns to installation option
  .
  .
  NOTERM        INVOCATION option remains in effect
  .
  .
End of compilation for program 2

```

## Example: LANGUAGE option in a batch compilation

The following example shows the behavior of the LANGUAGE compiler option in a batch environment. The default installation option is ENGLISH (abbreviated to EN), and the invocation option is XX (a nonexistent language).

Source	Language in effect
CBL LANG(JP),FLAG(I,I),APOST,SIZE(MAX) IDENTIFICATION DIVISION. PROGRAM-ID. COMPILE1. . . . END PROGRAM COMPILE1.	EN   Installation default -- EN JP   Invocation -- XX :   :   :
CBL LANGUAGE(YY) CBL SIZE(2048K),LANGUAGE(JP),LANG(!) IDENTIFICATION DIVISION. PROGRAM-ID. COMPILE2. . . . END PROGRAM COMPILE2. IDENTIFICATION DIVISION. PROGRAM-ID. COMPILE3. . . . END PROGRAM COMPILE3.	EN   CBL resets language :   to EN. LANGUAGE(YY) JP   is ignored because it :   is superseded by (JP). :   (!! ) is not alpha- :   numeric and is :   discarded.
CBL LANGUAGE(JP),LANGUAGE(YY) . . . . . . . . .	EN   CBL resets language :   to EN. LANGUAGE(YY) :   supersedes (JP) but :   is nonexistent.

For the program COMPILE1, the default language English (EN) is in effect when the compiler scans the invocation options. A diagnostic message is issued in mixed-case English because XX is a nonexistent language identifier. The default EN remains in effect when the compiler scans the CBL statement. The unrecognized option APOST in the CBL statement is diagnosed in mixed-case English because the CBL statement has not completed processing and EN was the last valid language option. After the compiler processes the CBL options, the language in effect becomes Japanese (JP).

In the program COMPILE2, the compiler diagnoses CBL statement errors in mixed-case English because English is the language in effect before the first program is used. If more than one LANGUAGE option is specified, only the last valid language specified is used. In this example, the last valid language is Japanese (JP), and therefore Japanese becomes the language in effect when the compiler finishes

processing the CBL options. If you want diagnostics in Japanese for the options in the CBL or PROCESS statements, the language in effect before COMPILE1 must be Japanese.

The program COMPILE3 has no CBL statement, and so it inherits the language in effect, Japanese (JP), from the previous compilation.

After compiling COMPILE3, the compiler resets the language in effect to English (EN) because of the CBL statement. The language option in the CBL statement resolves the last-specified two-character alphanumeric language identifier, which is YY. Because YY is nonexistent, the language in effect remains English.

---

## 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”

### RELATED REFERENCES

“Messages and listings for compiler-detected errors” on page 231

## 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).

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 230

“Generating a list of compiler error messages” on page 230

### RELATED REFERENCES

“Format of compiler error messages”

“Severity codes for compiler error messages” on page 232

“FLAG” on page 274

## 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”

“FLAG” on page 274

## 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.”

<b>Line ID</b>	<b>Message code</b>	<b>Message text</b>
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.





---

## Chapter 14. Compiling under OS/390 UNIX

You can compile COBOL for OS/390 & VM programs under OS/390 UNIX using the `cob2` command. Under OS/390 UNIX, you can compile any COBOL source program that you compile under OS/390. The object code generated under OS/390 UNIX by the COBOL compiler can run under OS/390 or CMS, with certain run-time restrictions.

As part of the compilation step, you need to define the files needed for the compilation and specify any compiler options necessary for your program and for the output that you want.

The main job of the compiler is to translate your COBOL program into language that the computer can process (object code). The compiler also lists errors in your source statements and provides supplementary information to help you debug and tune your program. Use compiler-directing statements and compiler options to control your compilation.

### RELATED TASKS

"Setting environment variables under OS/390 UNIX"

"Specifying compiler options under OS/390 UNIX" on page 236

"Compiling and linking with the `cob2` command" on page 237

"Compiling using scripts" on page 241

### RELATED REFERENCES

"Run-time restrictions under CMS" on page 369

"Data sets used by the compiler under OS/390" on page 218

"Compiler options and compiler output under OS/390" on page 224

---

## Setting environment variables under OS/390 UNIX

An *environment variable* is a name associated with a string of characters. You use environment variables to set values that programs need, including the compiler. Set the environment variables for the compiler by using the `export` shell command. For example, to set the `SYSLIB` variable, issue the `export` command from the shell or from a script file:

```
export SYSLIB=/u/mystuff/copybooks
```

The value that you assign to an environment variable can include other environment variables or the variable itself. The values of these variables apply only when you compile from the shell where you issue the `export` command. If you do not set an environment variable, either a default value is applied or the variable is not defined. The environment variable names must be uppercase.

The environment variables that you can set for use by the compiler are as follows:

### COBOPT

Specify compiler options separated by blanks or commas. Separate suboptions with commas. Blanks at the beginning or the end of the variable value are ignored. Delimit the list of options with quotation marks if it contains blanks or characters significant to the OS/390 UNIX shell. For example:

```
export COBOPT="TRUNC(OPT) XREF"
```

## SYSLIB

Specify paths to directories to be used in searching for COBOL copy books when you do not specify an explicit library name on the COPY statement. Separate multiple paths with a colon. The paths are evaluated in order, from the first path to the last in the export command. If you set the variable with multiple files of the same name, the first located copy of the file is used.

For COPY statements where you have not coded an explicit library name, the compiler searches for your copy book in this order:

1. In the current directory
2. In the paths you specify with the `-I cob2` option
3. In the paths you specify in the SYSLIB environment variable

### *library-name*

Specify the directory path from which to copy when you specify an explicit *library-name* on the COPY statement. The environment variable name is identical to the *library-name* in your program. You must set an environment variable for each library; an error will occur otherwise. The environment variable name *library-name* must be uppercase.

### *text-name*

Specify the name of the file from which to copy text. The environment variable name is identical to the *text-name* in your program. The environment variable name *text-name* must be uppercase.

## RELATED TASKS

“Specifying compiler options under OS/390 UNIX”

“Compiling and linking with the cob2 command” on page 237

## RELATED REFERENCES

“Compiler-directing statements” on page 304

“Chapter 16. Compiler options” on page 257

COPY statement (*IBM COBOL Language Reference*)

---

## Specifying compiler options under OS/390 UNIX

The compiler is installed and set up with default compiler options. While installing the compiler, the system programmers for a site can fix compiler option settings to, for example, ensure better performance or maintain certain standards. You cannot override any compiler options that your site has set as fixed. For options that are not fixed, you can override the default settings by specifying compiler options in any of three ways:

1. Code them on the PROCESS or CBL statement in your COBOL source.
2. Specify the `-q` option of the cob2 command.
3. Set the COBOPT environment variable.

The compiler recognizes the options in the above order of precedence.

The order of precedence also determines which options are in effect when conflicting or mutually exclusive options are specified. When you compile using the cob2 command, compiler options are recognized in the following order of precedence from highest to lowest:

1. Installation defaults fixed as nonoverridable
2. The values of BUFSIZE, LIB, SQL, OUTDD, and SIZE options in effect for the first program in a batch compilation

3. The values that you specify on PROCESS or CBL statements within your COBOL source programs
4. The values that you specify in the cob2 command's -q option string
5. The values that you specify in the COBOPT environment variable
6. Installation defaults that are not fixed

#### RELATED TASKS

"Specifying compiler options with the PROCESS (CBL) statement" on page 223  
"Setting environment variables under OS/390 UNIX" on page 235  
"Compiling and linking with the cob2 command"

#### RELATED REFERENCES

"Conflicting compiler options" on page 259  
"Chapter 16. Compiler options" on page 257

---

## Compiling and linking with the cob2 command

Use the cob2 command to compile and link your COBOL programs from the OS/390 UNIX shell. You can specify the options and input file names in any order, using spaces to separate options and names. Any options that you specify apply to all files on the command line.

To compile multiple files (batch compilation), specify multiple source file names.

When you compile COBOL programs for OS/390 UNIX, the RENT option is required. The cob2 command automatically includes the COBOL compiler options RENT and TERM.

The cob2 command invokes the COBOL compiler that is found through the standard MVS search order. If the COBOL compiler is not installed in the linklist or if more than one level of IBM COBOL compiler is installed on your system, you can specify the compiler PDS that you want to use in the STEPLIB environment variable. For example:

```
export STEPLIB=IGY.V2R2M0.SIGYCOMP
```

The cob2 command implicitly uses the OS/390 UNIX shell command c89 for the link step. c89 is the shell interface to the linkage editor (the DFSMS program management binder).

## Defining input and output

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

Only files with the .cbl extension are passed to the compiler; cob2 passes all other files to the linker.

The linker causes execution to begin at the first main program.

Listing output that you request from the compilation of a COBOL source program *file.cbl* is written to *file.lst*. Listing output that you request from the linker is written to stdout.

## Creating a DLL

To create a DLL, you must specify the cob2 option `-bdll`. The COBOL compiler options `DLL`, `EXPORTALL`, and `RENT` are required when you create a DLL and are included for you when you specify the `-bdll cob2` option. For example:

```
cob2 -o mydll -bdll mysub.cbl
```

When you specify `cob2 -bdll`, the link step produces a DLL definition side file. This file contains `IMPORT` control statements for each of the names exported by the DLL. The name of the DLL definition side file is based on the output file name. If the output name has an extension, that extension is replaced with `"x"` to form the side file name. For example, if the output file name is `foo.dll`, the side file name is `foo.x`.

To use the DLL definition side files later when you create a module that calls the DLL, specify the side files with any other object files (*file.o*) that you specify for the linking. For example:

```
cob2 -o myapp1 -qdll myapp1.cbl mydll.x
```

“Example: using cob2 to compile under OS/390 UNIX”

### RELATED TASKS

“Compiling programs to create DLLs” on page 490  
*Language Environment Programming Guide*  
*OS/390 UNIX System Services User's Guide*

### RELATED REFERENCES

“cob2” on page 239  
“cob2 input and output files” on page 240  
“Data sets used by the compiler under OS/390” on page 218  
“Compiler options and compiler output under OS/390” on page 224  
*OS/390 UNIX System Services Command Reference*

## Example: using cob2 to compile under OS/390 UNIX

The following examples illustrate the use of `cob2`:

- To compile one file (called `alpha.cbl` here), enter:

```
cob2 -c alpha.cbl
```

The compiled file is named `alpha.o`.

- To compile two files (called `alpha.cbl` and `beta.cbl` here), enter:

```
cob2 -c alpha.cbl beta.cbl
```

The compiled files are named `alpha.o` and `beta.o`.

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

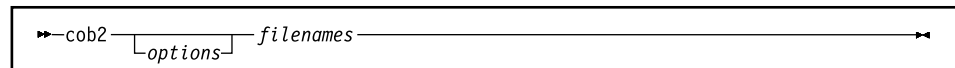
```
cob2 alpha.cbl beta.cbl -o gamma
```

This command creates `alpha.o` and `beta.o`, then links `alpha.o`, `beta.o`, and the COBOL libraries. If the link step is successful, it produces an executable program named `gamma`.

- To compile `alpha.cbl` with the `LIST` and `NODATA` options, enter:

```
cob2 -qlist,noadata alpha.cbl
```

## cob2



Do not capitalize cob2.

The options for cob2 are as follows:

**-bxxx** Passes the string *xxx* to the linker as parameters. *xxx* is a list of linker options in *name=value* format, separated by commas. You must spell out both the name and the value in full. They are case insensitive. Do not use any spaces between **-b** and *xxx*.

If you do not specify a value for an option, a default value of YES is used, except for the following options, which have the indicated default values:

- LIST=NOIMPORT
- ALIASES=ALL
- COMPAT=CURRENT
- DYNAM=DLL

One special value is when *xxx* is *dll*, which specifies that the executable module is to be a DLL. It is not passed to the linker.

**-c** Compiles programs but does not link them.

**-comprc\_ok=*n***

Controls cob2 behavior on the return code from the compiler. If the return code is less than or equal to *n*, cob2 continues to the link step or, in the compile-only case, exits with a zero return code. If the return code returned by the compiler is greater than *n*, cob2 exits with the same return code. When the *c89* command is implicitly invoked by cob2 for the link step, the exit value from the *c89* command is used as the return code from the cob2 command.

The default is **-comprc\_ok=4**.

**-e xxx** Specifies the name of a program to be used as the entry point of the module. If you do not specify **-e**, the default entry point is the first program (*file.cbl*) or object file (*file.o*) that you specify as a file name on the cob2 command invocation.

**-g** Prepares the program for debugging. Equivalent to specifying the TEST option with no suboptions.

**-Ixxx** Adds a path *xxx* to the directories to be searched for COPY files for which you do not specify a *library-name*.

To specify multiple paths, either use multiple **-I** options, or use a colon to separate multiple path names within a single **-I** option value.

For COPY statements where you have not coded an explicit library name, the compiler searches for your COPY book in this order:

1. In the current directory
2. In the paths you specify with the **-I** cob2 option
3. In the paths you specify in the SYSLIB environment variable

If you use the COPY statement, you must ensure that the LIB compiler option is in effect.

- L xxx** Specifies the directory paths to be used to search for archive libraries specified by the -l operand.
- l xxx** Specifies the name of an archive library for the linker. The cob2 command searches for the name libxxx.a in the directories specified on the -L option, then in the usual search order. (This is lowercase "el," not uppercase "eye.")
- o xxx** Names the object module xxx. If the -o option is not used, the name of the object module is a.out.
- qxxx** Passes xxx to the compiler, where xxx is a list of compiler options separated by blanks or commas.  
  
Enclose xxx in quotes if a parenthesis is part of the option or suboption, or if you use blanks to separate options. Do not insert spaces between -q and xxx.
- v** Displays the generated commands that are issued by cob2 for the compile and link steps, including the options being passed, and executes them. This is sample output:  

```
cob2 -v -o mini -qssrange mini.cbl  
compiler: ATTCRCTL PARM=RENT,TERM,SSRANGE /u/userid/cobol/mini.cbl  
PP 5648-A25 IBM COBOL for OS/390 & VM 2.2.0 in progress ...  
End of compilation 1, program mini, no statements flagged.  
linker: /bin/c89 -o mini -e // mini.o
```
- #** Displays compile and link steps, but does not execute them.

#### RELATED TASKS

"Compiling and linking with the cob2 command" on page 237

"Setting environment variables under OS/390 UNIX" on page 235

## cob2 input and output files

You can specify the following files as input file names when you use the cob2 command:

File name	Description	Comments
<i>file.cbl</i>	COBOL source file to be compiled and linked	Will not be linked if you specify the cob2 option -c
<i>file.a</i>	Archive file	Produced by the ar command, to be used during the link-edit phase
<i>file.o</i>	Object file to be link-edited	Can be produced by the COBOL compiler, the C/C++ compiler, or the assembler
<i>file.x</i>	DLL definition side file	Used during the link-edit phase of an application that references the dynamic link library (DLL)

When you use the cob2 command, the following files are created in the current directory:

File name	Description	Comments
<i>file</i>	Executable module or DLL	Created by the linker if you specify the cob2 option <code>-o file</code>
<i>a.out</i>	Executable module or DLL	Created by the linker if you do not specify the cob2 option <code>-o</code>
<i>file.adt</i>	Associated data (ADATA) file corresponding to input COBOL source program <i>file.cbl</i>	Created by the compiler if you specify the compiler option ADATA
<i>file.dbg</i>	Symbolic information tables for Debug Tool corresponding to input COBOL source program <i>file.cbl</i>	Created by the compiler if you specify the compiler option TEST ( <i>hook</i> , SYM, SEPARATE)
<i>file.idl</i>	Interface Definition Language (IDL) file for OO COBOL class definitions defined in input source program <i>file.cbl</i>	Created by the compiler if you specify the compiler option IDLGEN
<i>file.lst</i>	Listing file corresponding to input COBOL source program <i>file.cbl</i>	Created by the compiler
<i>file.o</i>	Object file corresponding to input COBOL source program <i>file.cbl</i>	Created by the compiler
<i>file.x</i>	DLL definition side file	Created during the cob2 linking phase when creating a DLL named <i>file.dll</i>

#### RELATED TASKS

“Compiling and linking with the cob2 command” on page 237

#### RELATED REFERENCES

“ADATA” on page 261

“TEST” on page 295

“IDLGEN” on page 277

*OS/390 UNIX System Services Command Reference*

---

## Compiling using scripts

To use a shell script to automate your cob2 tasks, use the following syntax to prevent the shell from passing syntax to cob2 that is not valid:

- Use an equal sign and colon rather than left and right parentheses, respectively, to specify compiler suboptions. For example, use `-qOPT=FULL: ,XREF` instead of `-qOPT(FULL) ,XREF`.
- Use an underscore rather than apostrophe where a compiler option requires apostrophes for delimiting a suboption.
- Do not use blanks in the option string.





---

## Chapter 15. Compiling under CMS

Under CMS, the COBOL compiler can compile any COBOL source program it can compile under OS/390 except programs that contain embedded SQL. The object code generated under CMS by the COBOL compiler can be run under OS/390 or CMS, with certain run-time restrictions.

To compile under CMS:

1. Make the compiler available to you on a minidisk through using the CP LINK and ACCESS commands. If IBM COBOL for OS/390 & VM is stored on your A-disk or another disk to which you already have access, you can omit this step.
2. Start the compiler by using the COBOL2 command. You can start the compiler by using ISPF/PDF menus if you have ISPF/PDF Version 3 installed at your site.

The compiler translates your COBOL program into language that the computer can process (object code). The compiler also lists errors in your source statements and provides supplementary information to help you debug and tune your program. Use compiler-directing statements and compiler options to control your compilation.

### RELATED TASKS

"Accessing the compiler (CP LINK and ACCESS)"

"Specifying a source program to compile" on page 244

"Using compiler-directing statements" on page 245

"Specifying compiler options under CMS" on page 246

"Naming generated files under CMS" on page 251

"Correcting errors" on page 252

"Chapter 22. Running COBOL programs under CMS" on page 369

### RELATED REFERENCES

"Run-time restrictions under CMS" on page 369

"COBOL2" on page 244

"Differences in compiler options under CMS" on page 248

"Files used by the compiler under CMS" on page 248

---

## Accessing the compiler (CP LINK and ACCESS)

To access the compiler under CMS:

1. Link to the mini disk containing the compiler by issuing the CP LINK command.
2. Issue the ACCESS command to identify the mini disk to CMS.

For example:

```
CP LINK PRODUCT 194 198 RR PASSWORD
ACCESS 198 B
```

This example links to disk 194 of the virtual machine that contains the IBM COBOL for OS/390 & VM product and for which the user ID is PRODUCT. It defines disk 194 as 198 to the VM session. It asks for read access to the disk (RR) and enters the read-share password for the 194 disk (PASSWORD). After linking to the 194 disk as 198, this code accesses the 198 disk as disk B.

If you can avoid having to access the compiler explicitly each time you need it, simply put the CP LINK and ACCESS commands into your PROFILE EXEC, which issues them for you each time you log on.

---

## Specifying a source program to compile

Use one of the following methods to indicate which COBOL source program to compile:

- Enter the file name of the COBOL source program with the COBOL2 command. For example:

```
COBOL2 PROG1 (SOURCE,VBREF,XREF,MAP)
```

This command compiles the COBOL source program named PROG1 using the compiler options SOURCE, VBREF, XREF, and MAP. The source program must have a file type of COBOL.

- Issue a FILEDEF for SYSIN before compiling. For example:

```
FILEDEF SYSIN DISK PROG2 COBOL A  
COBOL2 (SOURCE,VBREF,XREF,MAP)
```

This code compiles the COBOL program named PROG2 using the options entered with the COBOL2 command.

The compiler uses the source program that you enter with the FILEDEF, even if you provided an optional file name on the command line. The compiler uses the optional file name for the name of listing, text, SYSADATA, and work files. With FILEDEF, the file type need not be COBOL.

- Supply source statements from a user-supplied module by using the EXIT compiler option.

Neither a FILEDEF for SYSIN nor a file name in the COBOL2 command is required. For example:

```
COBOL2 (EXIT(INEXIT('ABCD',INMOD1)),VBREF,XREF)
```

This command compiles the source statements provided by the input user module named INMOD1 using the character string ABCD and the options VBREF and XREF.

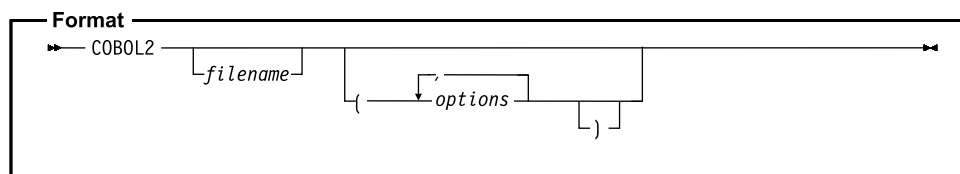
The compiler ignores FILEDEF for SYSIN when the INEXIT suboption of the EXIT option is specified. However, you can provide an optional file name with the COBOL2 command that will be used as the name for listing, text, and work files.

### RELATED REFERENCES

“COBOL2”

## COBOL2

You can compile programs under CMS by using the COBOL2 command.



*filename*

The name of your COBOL source program. Use one of three methods for specifying your source COBOL program.

*options*

Represents one or more compiler options, separated by a blank or comma, that you want in effect during compilation. These compiler options are equivalent to the options you would provide on the PARM parameter of an EXEC job-control statement, if you were invoking the compiler under OS/390.

The compiler options in effect at compile time are determined by the default options that were set when IBM COBOL for OS/390 & VM was installed and by the options you enter with the COBOL2 command. Three compiler options can be used only when running under CMS: DISK, PRINT, and NOPRINT.

Your installation might have created a synonym for COBOL2 when the compiler was installed. See your system programmer for the command name.

RELATED TASKS

- “Specifying a source program to compile” on page 244
- “Specifying compiler options under CMS” on page 246
- “Naming generated files under CMS” on page 251

RELATED REFERENCES

- “Additional compiler options under CMS” on page 247

---

## Using compiler-directing statements

You can put compiler-directing statements in your source program to help direct compilation.

If you have files that contain frequently used COBOL code, you can place them in a CMS file called a MACLIB. You can then use the compiler-directing COPY statement in your COBOL program and identify the MACLIB to be searched during compilation.

If you are using the compiler-directing BASIS statement, you also place the BASIS program in a CMS MACLIB file.

To make the MACLIB available, you must issue the CMS GLOBAL command:

```
GLOBAL MACLIB mylib
```

where *mylib* is the name of your MACLIB file (library).

If the GLOBAL command is issued, the FILEDEF command for SYSLIB will be issued by IBM COBOL for OS/390 & VM.

To use the COPY statement to copy from libraries other than SYSLIB, you must issue a FILEDEF command for each library. The ddname field of the FILEDEF command is identical to the library-name in your program. For these additional libraries, do not issue a GLOBAL command.

RELATED REFERENCES

- “Compiler-directing statements” on page 304

---

## Specifying compiler options under CMS

The compiler is installed and set up with default compiler options. The default options will be used unless you override them in either the PROCESS (synonym: CBL) statement or the COBOL2 command.

Compiler options under CMS have these differences:

- Three additional compiler options are available: DISK, PRINT, and NOPRINT.
- Certain compiler options behave differently: DUMP, TERMINAL, and LANGUAGE.
- Compiler options are recognized in a specific order of precedence. You can specify any compiler options on the PROCESS statement except DISK, PRINT, NOPRINT, EVENTS, ADATA, and EXIT. Specifying any of these options on a PROCESS statement is not allowed and causes a diagnostic message to be issued.

You can enter more than one compiler option on the COBOL2 command line, using a blank character or a comma as a separator. Do not mix commas and blanks within the command. If you choose blanks, use all blanks; if you choose commas, use all commas.

The COBOL2 command can diagnose some errors in compiler options from the command line by issuing a message to the terminal and then stopping the compilation. If the compilation is not ended, you should check the compiler listing for other compiler-detected errors. The listing shows subparameter values that are not valid. If a subparameter is optional, you can omit it and the separator character.

## Using parentheses in COBOL2

You must use parentheses with options that normally require them. For example, you can enter the same command in either of these different ways:

```
COBOL2 FILENAME (SZ(MAX),BUF(10K),LC(55),OPT)
```

```
COBOL2 FILENAME (SZ(MAX) BUF(10K) LC(55) OPT)
```

The format for specifying options in the COBOL2 command is the same as for specifying options on the PROCESS statement.

Option lists for existing applications that use only spaces and no parentheses will continue to work, and the compiler will issue a warning indicating that parentheses were added to the option list. If you change an existing options list by adding an option that requires parentheses, by adding commas, or by adding an option greater than eight characters, then parentheses must also be added to all the existing options that require them.

If you do not place commas between options or parenthesize subparameters, the compiler might place parentheses around any subparameters following an option. For example, NOC is an option that can be coded with or without the subparameter S, and SOURCE is an option that can be abbreviated as S. The option string NOC S might be interpreted as NOC(S) if the compiler assumes you are entering subparameters without parentheses. To ensure correct interpretation of the NOC option followed by the S option, you could code NOC,S.

## Abbreviating compiler options

When you enter the options in the command, you can use either the full option name or the allowed abbreviations. Options can be more than eight characters.

If you receive an error message that too many characters were specified for options in the COBOL2 command, you must reissue the command with fewer option characters. You can use more abbreviations for the options or enter some options on the PROCESS statement. However, moving options to the PROCESS statement could change their precedence.

## Using logical line-editing characters

When entering options with the COBOL2 command, be aware of the VM/ESA logical line-editing characters that are established for your terminal or virtual console. VM/ESA does not consider the logical line-editing characters as part of the input stream that is passed to the invoked program.

For example, the VM/ESA default setting for the logical escape character is the quotation mark ("). Suppose the compiler option CURRENCY("\$") was entered with the COBOL2 command:

```
COBOL2 programe (CURRENCY("$"))
```

VM/ESA removes the character " from the input stream and passes the following code:

```
CURRENCY($)
```

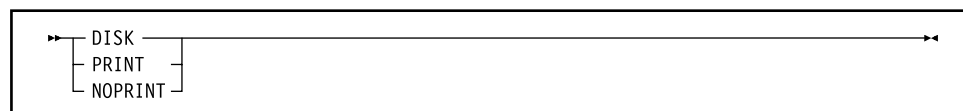
The COBOL for OS/390 & VM compiler then diagnoses the option CURRENCY(\$) as not valid because of the missing delimiters in the value.

### RELATED REFERENCES

- "Additional compiler options under CMS"
- "Files used by the compiler under CMS" on page 248
- "Error messages from COBOL2" on page 252

## Additional compiler options under CMS

Three compiler options are unique to COBOL under CMS: DISK, PRINT, and NOPRINT. They are valid only when used with the COBOL2 command.



Default is: DISK

Abbreviations are: DI, PRI, and NOPRI, respectively

Use DISK to request that a program listing be produced containing any of the following information:

- Page headings
- Line numbers of the statements in error
- Message identification numbers
- Severity levels
- Message texts
- Any other output requested by OFFSET, MAP, LIST, XREF, SOURCE, or VBREF

This listing will be written to the appropriate read/write disk with a file type of LISTING.

Use PRINT to request that the program listing be printed on your virtual printer. Use NOPRINT if you want no LISTING file to be produced.

If you have issued a FILEDEF for SYSPRINT to redirect the listing file produced by the compiler, the DISK, PRINT, and NOPRINT options are ignored.

DISK, PRINT, and NOPRINT are opposing options. If you enter more than one of these options with the COBOL2 command, the last one takes precedence.

## Differences in compiler options under CMS

The following differences apply to using the compiler options under CMS:

- The DUMP option specifies that there is to be an abend at the point of failure, but no output is produced.
- The default for the TERMINAL option is TERMINAL, whereas under OS/390 it is NOTERMINAL.
- The LANGUAGE option does not control the language of messages issued by the COBOL2 command processor. The language of these messages is determined by the installation default LANGUAGE selection and cannot be changed.

### RELATED REFERENCES

“Error messages from COBOL2” on page 252

## Precedence of compiler options under CMS

Compiler options under CMS are recognized in the following order of precedence from highest to lowest:

1. Installation defaults fixed by your site
2. Options specified on PROCESS (or CBL) statements
3. Options specified on the COBOL2 command
4. Installation defaults that are not fixed

This order of precedence also determines which options are in effect when conflicting or mutually exclusive options are specified.

### RELATED REFERENCES

“Conflicting compiler options” on page 259

“Chapter 16. Compiler options” on page 257

## Files used by the compiler under CMS

When you compile a COBOL program, the compiler generates work files and possibly output such as listings and error messages.

When the compiler is running, it uses and creates both work and default files, as shown below.

FILEDEF	File type	Function	Required?	Default device	Comments
SYSIN	COBOL	Reads the source file	Yes	Disk	
SYSLIB	MACLIB	Reads user source libraries	If program has COPY or BASIS statements (LIB option is required)	Disk	

FILEDEF	File type	Function	Required?	Default device	Comments
SYSUT1 to SYSUT4, SYSUT6, and SYSUT7 <sup>1</sup>	SYSUT1 to SYSUT4, SYSUT6, and SYSUT7	Work files used by the compiler during compilation	Yes	Disk	If you need to stop a compilation before it finishes, these files might still reside on one of your disks, and you should erase them.
SYSUT5 <sup>1</sup>	SYSUT5	Work file used by the compiler during compilation	If LIB is in effect	Disk	
SYSPRINT	LISTING	Writes listings and messages	If DISK is in effect	Disk	Written to the appropriate disk <sup>2</sup>
			If PRINT is in effect	Printer	Printed on virtual printer
SYSLIN	TEXT	Creates machine-language code; no error messages are produced with a severity level higher than the severity indicated in the COMPILE option.	Yes	Disk	Written to the appropriate disk <sup>2</sup>  You can relocate this TEXT file and run it after loading it into virtual storage.
SYSDEBUG	SYSDEBUG	Writes symbolic debug information tables to a data set separate from the object module	If TEST(...,SYM,SEPARATE) is in effect	Disk	
SYSTEM	None	Writes progress and diagnostic messages and compiler statistics	If TERM is in effect	Terminal	
SYSPUNCH	None	Writes object module on spooled punch	If DECK is in effect	Punch	
SYSADATA	SYSADATA	Writes record types that have machine-readable information about the compilation	If ADATA or EVENTS is in effect	Disk	Written to the appropriate disk <sup>2</sup>
<ol style="list-style-type: none"> <li>1. These files are written to the minidisk with the most available read-write space; they are erased at the end of compilation.</li> <li>2. LISTING, TEXT, and SYSADATA files are written to the minidisk where the source COBOL program resides, if that disk is accessed as read-write. If that disk is not accessed in read-write mode, the LISTING, TEXT, and SYSADATA files are written to the first read-write disk found using the search order.</li> </ol>					

### Compiler options and compiler output under CMS

If you use certain compiler options, the compiler will also create files with the file types of TEXT, LISTING, or SYSADATA. Their file names are the same as the name of the source file. If any TEXT, LISTING, or SYSADATA file with the same name already exists, it is erased, even when you use NOOBJECT or NOPRINT or both.

The following table lists the types of output you will get if you use various compiler options.

Compiler option	LISTING file	TEXT file	Terminal (with TERM option)
DECK		File is created; object code	
DISK	Writes the LISTING file to disk		
EXIT (PRTEXT)	Writes the LISTING file to a user-specified module		
FLAG(x[,y])	Warning messages and error messages to LISTING file		Warning messages and error messages are displayed
FLAGMIG	Warning messages and error messages to LISTING file		Warning messages and error messages are displayed
FLAGSTD(x[yy][,z])	Information messages to LISTING file		Information messages are displayed
LIST	Assembler language expansion, global tables, literal pools, WORKING-STORAGE location, length		
MAP	Glossary, global tables, literal pools		
NUMBER	User-supplied line numbers shown in listing		
OBJECT		Object code	
OFFSET	Condensed object listing in LISTING file		
OPTIMIZE		Optimized object code if OBJECT option in effect	
PGMNAME(LU), (LM)	Where appropriate, you will see long names in your listing		
PRINT	Prints LISTING file on virtual printer		
RENT		Reentrant object code if OBJECT option in effect	
SOURCE	Source module to LISTING file		
SPACE(1)	LISTING file is single-spaced		
SPACE(2)	LISTING file is double-spaced		
SPACE(3)	LISTING file is triple-spaced		
SSRANGE		Extra code generated for checking references within tables	
TERMINAL			Progress and diagnostic messages to terminal
TEST		Extra code generated	
VBREF	Verb cross-reference list to LISTING file		
XREF, XREF(FULL), XREF(SHORT)	Sorted procedure and data-names to LISTING file		



#### RELATED TASKS

“Naming generated files under CMS”

“Correcting errors” on page 252

---

## Naming generated files under CMS

The names for the files the compiler generates depend on how you code the COBOL2 command and whether you issue any FILEDEFs.

If you start the compiler using COBOL2 and the file name of the source program, the compiler will assign file names that are the same as the source program file name:  
COBOL2 *filename* (SOURCE,VBREF,XREF,MAP)

You can override the name of a compiler-created file (such as listing, text, or work files) by issuing a CMS FILEDEF command. When you issue a FILEDEF with a different file name or file type, the file is not erased when the compilation ends.

Indicate the file name to be assigned by the compiler for these files with any of the following methods when you start the compiler:

```
FILEDEF SYSIN DISK PROG2 COBOL A
COBOL2 PROG1 (SOURCE,VBREF,XREF,MAP)
```

or

```
COBOL2 PROG1 (EXIT(INEXIT('ABCD',INMOD1)),VBREF,XREF)
```

or

```
FILEDEF SYSIN DISK PROG2 COBOL A
COBOL2 PROG1 (EXIT(INEXIT('ABCD',INMOD1)),VBREF,XREF)
```

In all the above cases, the compiler assigns the file name PROG1.

## Overriding FILEDEF

By using the PRTEXT suboption of the EXIT option, you can override two specifications for file names:

- The listing output from the FILEDEF command for SYSPRINT
- Any optional file name that you specify in the COBOL2 command

For example:

```
FILEDEF SYSPRINT DISK PROG4 COBOL A
COBOL2 PROG2 (EXIT(PRTEXT('WXYZ',OUTMOD)),VBREF,XREF)
```

This code causes the SYSPRINT output to be transmitted to the user-supplied exit module named OUTMOD.

## Using system-generated names

CMS defines the listing, text, work, associated data (ADATA), and debugging information files for you if you do not provide a file name in the COBOL2 command and either of the following conditions exists:

- A FILEDEF for SYSIN is not supplied (for example, when an input user exit provides the source statements for the compiler).
- The FILEDEF for SYSIN does not include a file name (such as FILEDEF SYSIN READER).

The default file name provided by CMS is FILE. The table below shows the file types of the files.

File	CMS file type
Listing	SYSPRINT
Text	SYSLIN
Work files	SYSUT <i>n</i>
Associated data files	SYSADATA
Symbolic debugging information	SYSDEBUG

RELATED REFERENCES

“Files used by the compiler under CMS” on page 248

---

## Correcting errors

You can usually identify a problem from the diagnostic message you see displayed at the terminal, so that you do not have to examine a printed listing file. When you have an error, you can correct the source file immediately and recompile it.

Under CMS, the TERM option is in effect unless the default value for your site has been changed to TERM=NO or you have specified the NOTERM option. TERM specifies that all diagnostic messages issued by the compiler are displayed at your terminal, in addition to being written to a LISTING file.

RELATED REFERENCES

“Error messages from COBOL2”

## Error messages from COBOL2

The COBOL2 command can generate error messages. These have the identification DMSIGY, followed by a message number, severity indicator, and message text. Whether the message identification or text, or both, is displayed depends on the setting in your virtual machine of EMSG, which is a CP SET command function. Many of the DMSIGY messages issued by CMS are self-explanatory. For example:

```
COBOL2 myprog
DMSIGY002E Source file "MYPROG COBOL" was not found.
DMSIGY000E The COBOL2 command was terminated.
Ready(00040);
```

The error was issued because the specified input file was not found on any of the accessed disks. You should reissue the command using the correct file name.

The messages issued by the COBOL2 command will be in the language specified by your installation’s default setting for the LANGUAGE compiler option.

The error messages issued by the COBOL2 command are listed below.

Be sure to check your listing for any errors not detected by the COBOL2 command.

Message number	Message text
DMSIGY000E (page 253)	THE COBOL2 COMMAND WAS TERMINATED.
DMSIGY002E (page 253)	SOURCE FILE “filename filetype” WAS NOT FOUND.
DMSIGY003E (page 253)	AN ASTERISK “*” WAS USED AS A FILE NAME IN THE COBOL2 COMMAND.
DMSIGY004E (page 254)	AN ERROR OCCURRED WHILE ATTEMPTING TO LOAD MODULE “modname”.

DMSIGY005W (page 254)	THE OPTIONS PARAMETER LIST WAS PARENTHESIZED BY THE COMPILER. THE OPTIONS IN EFFECT MAY NOT BE AS INTENDED.
DMSIGY011E (page 254)	A CMS DISK WAS NOT ACCESSED IN READ/WRITE MODE.
DMSIGY012E (page 254)	DEVICE "device" WAS INVALID IN A USER FILEDEF FOR SYSIN.
DMSIGY021E (page 254)	SOURCE FILE "filename filetype" DID NOT HAVE "FIXED" RECORD FORMAT.
DMSIGY030E (page 255)	THE FOLLOWING OPTIONS SPECIFIED WERE INVALID:
DMSIGY031I (page 255)	"option-1 option-2...option-n".
DMSIGY032E (page 255)	OPTION "option" DID NOT HAVE A VALUE SPECIFIED.
DMSIGY033E (page 255)	TOO MANY CHARACTERS WERE SPECIFIED FOR OPTIONS IN THE COBOL2 COMMAND.
DMSIGY034E (page 256)	A LEFT PARENTHESIS "(" DID NOT PRECEDE THE OPTION LIST IN THE COBOL2 COMMAND.
DMSIGY036W (page 256)	THE OPTION LIST WAS TERMINATED BY A RIGHT PARENTHESIS ")". THE REMAINDER OF THE OPTION LIST WAS IGNORED.
DMSIGY037E (page 256)	THE OPTION "GRAPHIC" IS NOT SUPPORTED.

#### DMSIGY000E

THE COBOL2 COMMAND WAS TERMINATED.
------------------------------------

**Explanation:** An error was detected that will terminate the COBOL2 command. This message will be issued only once for invalid COBOL2 commands.

**Programmer response:** Reissue the command, correcting the errors indicated by previous messages.

**System action:** Execution of the COBOL2 command ends. The system status is the same as before the command executed, except that all user FILEDEFS issued without the PERM option will be cleared.

#### DMSIGY002E

SOURCE FILE "filename filetype" WAS NOT FOUND.
--

**Explanation:** The specified input file was not found on any of the accessed disks.

**Programmer response:** Reissue the command, specifying the correct input file.

**System action:** RC=40. Execution of the COBOL2 command ends. The system has the same status as before the command executed.

#### DMSIGY003E

AN ASTERISK "*" WAS USED AS A FILE NAME IN THE COBOL2 COMMAND.
--

**Explanation:** '\*' is not a valid file name for the COBOL2 command.

**Programmer response:** Reissue the command with a valid file name.

**System action:** RC=40. Execution of the COBOL2 command ends. The system has the same status as before the command executed.

#### DMSIGY004E

AN ERROR OCCURRED WHILE ATTEMPTING TO LOAD MODULE "modname".

**Explanation:** An error was detected by the command processor while attempting to load module *modname*.

**Programmer response:** Reissue the command.

**System action:** Compilation ends, and a user abend is initiated.

#### DMSIGY005W

THE OPTIONS PARAMETER LIST WAS PARENTHESESIZED BY THE COMPILER. THE OPTIONS IN EFFECT MAY NOT BE AS INTENDED.

**Explanation:** An options parameter list was used that contained no suboption parentheses.

**Programmer response:** Verify the options as shown in the listing.

**System action:** The compiler adds any required parentheses to the list.

#### DMSIGY011E

A CMS DISK WAS NOT ACCESSED IN READ/WRITE MODE.

**Explanation:** No read/write disk is available for compiler output and work files.

**Programmer response:** Access a CMS disk in read/write mode.

**System action:** RC=40. Execution of the COBOL2 command ends. The system has the same status as before the command executed.

#### DMSIGY012E

DEVICE "device" WAS INVALID IN A USER FILEDEF FOR SYSIN.

**Explanation:** A user FILEDEF specifying a device invalid for SYSIN has been issued.

**Programmer response:** Reissue the COBOL2 command specifying a valid device for SYSIN, or clear the SYSIN FILEDEF.

**System action:** RC=40. Execution of the COBOL2 command ends. The user-issued FILEDEF is cleared unless the PERM option was specified. Otherwise, the system has the same status as before the command executed.

#### DMSIGY021E

SOURCE FILE "filename filetype" DID NOT HAVE "FIXED" RECORD FORMAT.

**Explanation:** COBOL source file must have FIXED record format.

**Programmer response:** Change the record format of the source file to FIXED and reissue the command.

**System action:** RC=40. Execution of the COBOL2 command ends. The system has the same status as before the command executed.

#### DMSIGY030E

THE FOLLOWING OPTIONS SPECIFIED WERE INVALID:

**Explanation:** At least one invalid option is specified in the command line. The invalid options are listed under message DMSIGY031I that follows.

**Programmer response:** Reissue the COBOL2 command with valid options.

**System action:** RC=40. Execution of the COBOL2 command ends. The system has the same status as before the command executed.

#### DMSIGY031I

"option-1 option-2...option-n".

**Explanation:** This is the list of invalid options indicated by message DMSIGY030E.

**Programmer response:** Same as message DMSIGY030E

**System action:** Same as message DMSIGY030E

#### DMSIGY032E

OPTION "option" DID NOT HAVE A VALUE SPECIFIED.

**Explanation:** The option stated must have a value specified.

**Programmer response:** Reissue the COBOL2 command, specifying a value for the option.

**System action:** RC=40. Execution of the COBOL2 command ends. The system has the same status as before the command executed.

#### DMSIGY033E

TOO MANY CHARACTERS WERE SPECIFIED FOR OPTIONS IN THE COBOL2 COMMAND.

**Explanation:** User specified too many characters for options in the COBOL2 command line.

**Programmer response:** Reissue the COBOL2 command with fewer options or use abbreviations

**System action:** RC=40. Execution of the COBOL2 command ends. The system has the same status as before the command executed. for the options.

#### DMSIGY034E

A LEFT PARENTHESIS "(" DID NOT PRECEDE THE OPTION LIST IN THE COBOL2 COMMAND.

**Explanation:** Options following the file name in the COBOL2 command must be preceded by a left parenthesis.

**Programmer response:** Reissue the COBOL2 command with the correct syntax.

**System action:** RC=40. Execution of the COBOL2 command ends. The system has the same status as before the command executed.

#### DMSIGY036W

THE OPTION LIST WAS TERMINATED BY A RIGHT PARENTHESIS ")". THE REMAINDER OF THE OPTION LIST WAS IGNORED.

**Explanation:** A right parenthesis has prematurely ended the option list in the COBOL2 command.

**System action:** RC=0. Any options following a right parenthesis are ignored. The COBOL2 command will be executed unless other terminating errors are detected.

**Programmer response:** Reissue the command deleting the right parenthesis.

#### DMSIGY037E

THE OPTION "GRAPHIC" IS NOT SUPPORTED.

**Explanation:** The user specified GRAPHIC as an option of the COBOL2 command. This option is not supported in Release 2.0 or later.

**Programmer response:** Reissue the COBOL2 command without the GRAPHIC option.

**System action:** RC=40. Execution of the COBOL2 command ends. The system has the same status as before the command executed.

#### RELATED TASKS

"Correcting errors" on page 252

## Chapter 16. 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	"ARITH" on page 262	ARITH(COMPAT)	AR(C), AR(E)
	"QUOTE/APOST" on page 289	QUOTE	Q APOST
	"CMPR2" on page 264	NOCMPR2	None
	"CURRENCY" on page 265	NOCURRENCY	CURR NOCURR
	"DBCS" on page 269	NODBCS	None
	"LIB" on page 280	LIB	None
	"NUMBER" on page 283	NONUMBER	NUM NONUM
	"SEQUENCE" on page 291	SEQUENCE	SEQ NOSEQ
	"SQL" on page 293	NOSQL	None
	"WORD" on page 301	NOWORD	WD NOWD
Date processing	"DATEPROC" on page 267	NODATEPROC, or DATEPROC(FLAG,NOTRIG) if only DATEPROC is specified	DP NODP
	"INTDATE" on page 278	INTDATE(ANSI)	None
	"YEARWINDOW" on page 303	YEARWINDOW(1900)	YW
Maps and listings	"LANGUAGE" on page 279	LANGUAGE(ENGLISH)	LANG(EN UE JA JP)
	"LINECOUNT" on page 280	LINECOUNT(60)	LC
	"LIST" on page 281	NOLIST	None
	"MAP" on page 281	NOMAP	None
	"OFFSET" on page 285	NOFFSET	OFF NOFF
	"SOURCE" on page 292	SOURCE	S NOS
	"SPACE" on page 293	SPACE(1)	None
	"TERMINAL" on page 295	NOTERMINAL	TERM NOTERM
		"VBREF" on page 301	NOVBREF
	"XREF" on page 302	NOXREF	X NOX

Aspect of your program	Compiler option	Default	Abbreviations
Object deck generation	"COMPILE" on page 265	NOCOMPILE(S)	C NOC
	"DECK" on page 269	NODECK	D NOD
	"NAME" on page 282	NONAME, or NAME(NOALIAS) if only NAME is specified	None
	"OBJECT" on page 284	OBJECT	OBJ NOOBJ
	"PGMNAME" on page 287	PGMNAME(COMPAT)	PGMN(CO LU LM)
Object code control	"ADV" on page 262	ADV	None
	"AWO" on page 263	NOAWO	None
	"DLL" on page 270	NODLL	None
	"EXPORTALL" on page 272	NOEXPORTALL	EXP NOEXP
	"FASTSRT" on page 273	NOFASTSRT	FSRT NOFSRT
	"NUMPROC" on page 283	NUMPROC(NOPFD)	None
	"OPTIMIZE" on page 285	NOOPTIMIZE	OPT NOOPT
	"OUTDD" on page 286	OUTDD(SYSOUT)	OUT
	"TRUNC" on page 297	TRUNC(STD)	None
	"ZWB" on page 304	ZWB	None
Virtual storage usage	"BUFSIZE" on page 263	4096	BUF
	"SIZE" on page 291	SIZE(MAX)	SZ
	"DATA" on page 266	DATA(31)	None
	"DYNAM" on page 272	NODYNAM	DYN NODYN
	"RENT" on page 289	NORENT	None
	"RMODE" on page 290	AUTO	None
Debugging and diagnostics	"DIAGTRUNC" on page 269	NODIAGTRUNC	DTR, NODTR
	"DUMP" on page 271	NODUMP	DU NODU
	"FLAG" on page 274	FLAG(I)	F NOF
	"FLAGMIG" on page 275	NOFLAGMIG	None
	"FLAGSTD" on page 275	NOFLAGSTD	None
	"TEST" on page 295	NOTEST	None
	"SSRANGE" on page 294	NOSSRANGE	SSR NOSSR
	"TYPECHK" on page 300	NOTYPECHK	TC NOTC
Other	"ADATA" on page 261	NOADATA	None
	"ANALYZE" on page 261	NOANALYZE	None
	"EXIT" on page 272	NOEXIT	EX(INX,LIBX,PRTX,ADX)
	"IDLGEN" on page 277	NOIDLGEN	IDL

**Installation defaults:** 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. (In some installations, certain compiler options are set up as fixed so that you cannot override them. If you have problems, see your system administrator.) 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.



**Nonoverridable options:** In some installations, certain compiler options are set up so that you cannot override them. If you have problems, see your system administrator.

**CMS only:** The compiler options DISK, PRINT, and NOPRINT, which are not listed in the following table, are unique to COBOL under CMS. For a description, see the related information below.

**Performance considerations:** The DYNAM, FASTSRT, NUMPROC, OPTIMIZE, RENT, SSRANGE, TEST, and TRUNC compiler options can all affect run-time performance.

**RELATED REFERENCES**

- “Conflicting compiler options”
- “Compiler-directing statements” on page 304
- “Option settings for COBOL 85 Standard conformance”

**RELATED TASKS**

- “Chapter 13. Compiling under OS/390” on page 203
- “Compiling under TSO” on page 215
- “Chapter 15. Compiling under CMS” on page 243
- “Chapter 14. Compiling under OS/390 UNIX” on page 235
- “Chapter 33. Tuning your program” on page 535

---

## Option settings for COBOL 85 Standard conformance

The following compiler options are required to conform to the COBOL 85 Standard specification:

ADV	NAME (ALIAS) or NAME (NOALIAS)
NOCMPR2	NONUMBER
NODATEPROC	NUMPROC (NOPFD) or NUMPROC (MIG)
NODBCS	QUOTE
DYNAM	NOSEQUENCE
NOFASTSRT	TRUNC (STD)
NOFLAGMIG	NOWORD
LIB	ZWB

The following run-time options are required to conform to the COBOL 85 Standard:

- AIXBLD
- CBLQDA(ON)
- TRAP(ON)

**RELATED REFERENCES**

*Language Environment Programming Reference*

---

## Conflicting compiler options

The COBOL for OS/390 & VM compiler can encounter conflicting compiler options in two ways:

- Both the positive and negative form of a compiler option are specified on the same level in the hierarchy of precedence (such as specifying both DECK and NODECK in a PROCESS (or CBL) statement).

When conflicting options are specified at the same level in the hierarchy, the option specified last takes effect.

- Mutually exclusive options are specified at the same level in the hierarchy, regardless of order.

When you specify a mutually exclusive option, any conflicting options that you specify are ignored. For example, if you specify both OFFSET and LIST in your PROCESS statement, in any order, OFFSET takes effect and LIST is ignored.

However, options coded at a higher level of precedence override any options specified at a lower level of precedence. For example, if you code OFFSET in your JCL statement but LIST in your PROCESS statement, LIST will take effect because the options coded in the PROCESS statement and any options forced on by an option coded in the PROCESS statement have higher precedence.

The following table lists mutually exclusive compiler options.

Specified	Ignored <sup>1</sup>	Forced on <sup>1</sup>
ANALYZE	SQL NOADATA NOLIB	NOSQL ADATA LIB
CMPR2	ADATA ANALYZE ARITH(EXTEND) CURRENCY DATEPROC DBCS DIAGTRUNC DLL EXPORTALL FLAGSTD IDLGEN INTDATE(LILIAN) OPT(FULL) PGMNAME(LU) and PGMNAME(LM) RMODE(24) and RMODE(ANY) SQL TYPECHK YEARWINDOW	NOADATA NOANALYZE ARITH(COMPAT) NOCURRENCY NODATEPROC NODBCS NODIAGTRUNC NODLL NOEXPORTALL NOFLAGSTD NOIDLGEN INTDATE(ANSI) OPT(STD) PGMNAME(COMPAT) RMODE(AUTO) NOSQL NOTYPECHK YEARWINDOW(1900)
DBCS	FLAGMIG	NOFLAGMIG
DLL	DYNAM NORENT	NODYNAM RENT
EXIT	DUMP	NODUMP
EXPORTALL	NODLL DYNAM NORENT	DLL NODYNAM RENT
FLAGSTD	FLAGMIG DBCS	NOFLAGMIG NODBCS
NOCMPR2	FLAGMIG	NOFLAGMIG
OFFSET	LIST	NOLIST
SQL	NOLIB	LIB
TEST TEST(ALL) TEST(STMT) TEST(PATH) TEST(BLOCK)	NOOBJECT OPT(STD) or OPT(FULL)	OBJECT NOOPTIMIZE

Specified	Ignored <sup>1</sup>	Forced on <sup>1</sup>
WORD	FLAGSTD	NOFLAGSTD
1. Unless in conflict with a fixed installation default option.		

#### RELATED TASKS

“Specifying compiler options under OS/390” on page 223

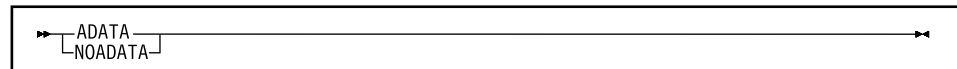
“Specifying compiler options in a batch compilation” on page 227

“Specifying compiler options under OS/390 UNIX” on page 236

“Specifying compiler options under CMS” on page 246

---

## ADATA



Default is: NOADATA

Abbreviations are: None

Use ADATA when you want the compiler to create a SYSADATA file, which contains records of additional compilation information. It is required for remote compile using IBM VisualAge COBOL. On OS/390, this file is written to DDNAME SYSADATA. On CMS, a file with file type SYSADATA is created. 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
- On the PARM field of JCL
- As a command option
- As an installation default

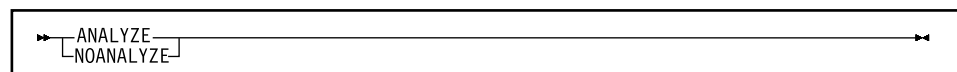
#### RELATED REFERENCES

“Appendix C. EXIT compiler option” on page 575

“Conflicting compiler options” on page 259

---

## 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:

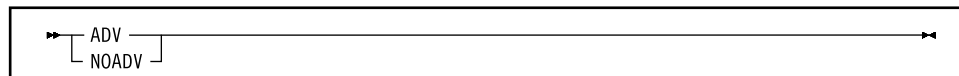
- `CICS`
- `EXEC`
- `END-EXEC`
- `SQL`

#### RELATED REFERENCES

“Conflicting compiler options” on page 259

---

## ADV



Default is: `ADV`

Abbreviations are: None

`ADV` has meaning only if you use `WRITE . . . ADVANCING` in your source code.

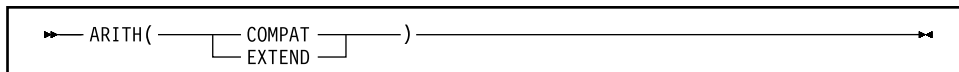
With `ADV` in effect, the compiler adds 1 byte to the record length to account for the printer control character.

`ADV` conforms to the COBOL 85 Standard.

Use `NOADV` if you have already adjusted your record length to include 1 byte for the printer control character.

---

## ARITH



Default is: `ARITH(COMPAT)`

Abbreviations are: `AR(C)`, `AR(E)`

When you specify `ARITH(EXTEND)`:

- The maximum number of digit positions that you can specify in the `PICTURE` clause for packed-decimal, zoned-decimal, and numeric-edited data items is raised from 18 to 31.
- The maximum number of digits that you can specify in a fixed-point numeric literal is raised from 18 to 31. You can use numeric literals with large precision anywhere that numeric literals are currently allowed, including:

- Operands of PROCEDURE DIVISION statements
- VALUE clauses (on numeric data items with large precision pictures)
- Condition-name values (on numeric data items with large precision pictures)
- The maximum number of digits that you can specify in the arguments to NUMVAL and NUMVAL-C is raised from 18 to 31.
- The maximum value of the integer argument to the FACTORIAL function is 29.
- Intermediate results in arithmetic statements use *extended mode*.

When you specify ARITH(COMPAT):

- The maximum number of digit positions in the PICTURE clause for packed-decimal, zoned-decimal, and numeric-edited data items is 18.
- The maximum number of digits in a fixed-point numeric literal is 18.
- The maximum number of digits in the arguments to NUMVAL and NUMVAL-C is 18.
- The maximum value of the integer argument to the FACTORIAL function is 28.
- Intermediate results in arithmetic statements use *compatibility mode*.

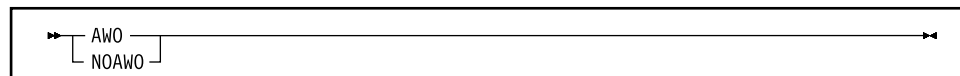
**RELATED CONCEPTS**

“Appendix A. Intermediate results and arithmetic precision” on page 559

**RELATED REFERENCES**

“Conflicting compiler options” on page 259

## AWO



Default is: NOAWO

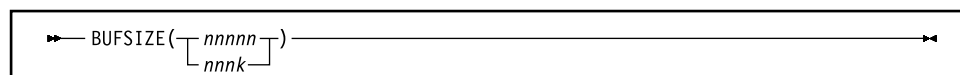
Abbreviations are: None

With AWO specified, an implicit APPLY WRITE-ONLY clause is activated for all files in the program that are eligible for this clause. To be eligible, a file must have physical sequential organization and blocked variable-length records.

**RELATED TASKS**

“Optimizing buffer and device space” on page 12

## BUFSIZE



Default is: 4096

Abbreviations are: BUF

*nnnnn* specifies a decimal number that must be at least 256.

*nnnK* specifies a decimal number in 1K increments.

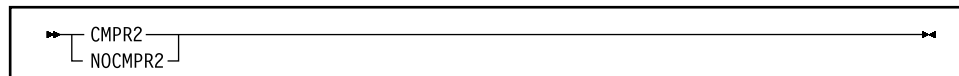
Use BUFSIZE to allocate an amount of main storage to the buffer for each compiler work data set (where 1K = 1024 bytes decimal). Usually, a large buffer size improves the performance of the compiler.

If you use both BUFSIZE and SIZE, the amount allocated to buffers is included in the amount of main storage available for compilation via the SIZE option.

BUFSIZE cannot exceed the track capacity for the device used, nor can it exceed the maximum allowed by data management services.

---

## CMPR2



Default is: NOCMPR2

Abbreviations are: None

Use CMPR2 when you want the compiler to generate code that is compatible with code generated by VS COBOL II Release 2. Valid COBOL source programs that compiled successfully under VS COBOL II Release 2 also compile successfully under COBOL for OS/390 & VM with CMPR2 in effect and will provide compatible results.

The only valid VS COBOL II Release 2 programs that do not compile successfully under CMPR2 are programs comparing an internal decimal item, a binary item, or an arithmetic expression to a group item.

Implementation of the COBOL 85 Standard created some instances where incompatibilities with VS COBOL II Release 2 can occur. Use the CMPR2 and FLAGMIG options to aid in the migration of programs written for VS COBOL II Release 2 to COBOL for OS/390 & VM.

NOCMPR2 conforms to the COBOL 85 Standard.

**Limited support:** In general, new language elements added to IBM COBOL since VS COBOL II Release 2 are not supported under CMPR2. CMPR2 is intended only as a migration aid to enable continued use of programs that are written at the VS COBOL II Release 2 language level.

### RELATED REFERENCES

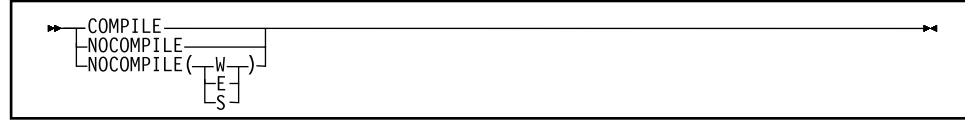
"FLAGMIG" on page 275

"Conflicting compiler options" on page 259

Differences between CMPR2 and NOCMPR2 (*COBOL for OS/390 & VM Compiler and Run-Time Migration Guide*)

---

## 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:

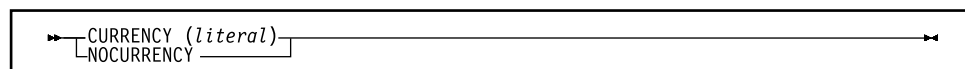
- LIST
- SSRANGE
- OPTIMIZE
- OBJECT
- TEST

### RELATED REFERENCES

“Messages and listings for compiler-detected errors” on page 231

---

## 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

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.

If you plan to specify the CURRENCY compiler option in the COBOL2 command under CMS, see the special usage notes under Specifying compiler options using the COBOL2 command in the related information below.

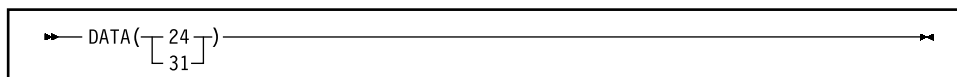
#### RELATED TASKS

“Specifying compiler options under CMS” on page 246

“Conflicting compiler options” on page 259

---

## DATA



Default is: DATA(31)

Abbreviations are: None

Language Environment provides services that control the storage used at run time. IBM COBOL for OS/390 & VM uses these services for all storage requests

For reentrant programs, the DATA(24|31) compiler option and the HEAP run-time option control whether storage for dynamic data areas (such as WORKING-STORAGE section and FD record areas) is obtained from below the 16-MB line or from unrestricted storage. (The DATA option does not affect the location of LOCAL-STORAGE section variables; the STACK run-time option controls that location instead.)

When you specify the run-time option HEAP(BELOW), the DATA(24|31) compiler option has no effect; the storage for all dynamic data areas is allocated from below the 16-MB line. However, with HEAP(ANYWHERE) as the run-time option, storage for



dynamic data areas is allocated from below the line if you compiled the program with the DATA(24) compiler option or from unrestricted storage if you compiled with the DATA(31) compiler option.

Specify the DATA(24) compiler option for programs running in 31-bit addressing mode that are passing data parameters to programs in 24-bit addressing mode. This ensures that the data will be addressable by the called program.

## External data

In addition to affecting how storage is obtained for dynamic data areas, the DATA(24|31) compiler option can also influence where storage for external data is obtained. Storage required for external data will be obtained from unrestricted storage if the following conditions are met:

- The program is compiled with the DATA(31) compiler option.
- The HEAP(ANYWHERE) run-time option is in effect.
- The ALL31(ON) run-time option is in effect.

In all other cases, the storage for external data will be obtained from below the 16-MB line. To specify the ALL31(ON) run-time option, all the programs in the run unit must be capable of running in 31-bit addressing mode.

## QSAM input/output buffers

On an OS/390 system, the DATA(24|31) compiler option can also influence where input/output buffers for QSAM files are obtained for reentrant programs. Storage required for a QSAM file's buffers will be acquired from unrestricted storage if the following conditions are met:

- The program is compiled with the DATA(31) and RENT compiler options or the program is compiled with the RMODE(ANY) and NORENT compiler options.
- The program is running in AMODE 31.
- The file is eligible for buffer allocation above the 16-MB line.
- If the file is an EXTERNAL file, the ALL31(ON) run-time option is in effect.

In all other cases, the storage for the QSAM file's buffers will be obtained from below the 16-MB line. To specify the ALL31(ON) run-time option, all the programs in the run unit must be capable of running in 31-bit addressing mode.

### RELATED TASKS

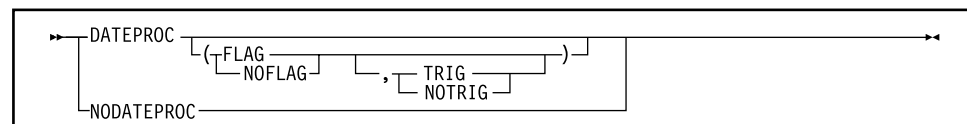
Using run-time options (*Language Environment Programming Guide*)

### RELATED REFERENCES

"Allocation of buffers for QSAM files" on page 127

---

## DATEPROC



Default is: NODATEPROC, or DATEPROC(FLAG,NOTRIG) 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.

#### **DATEPROC (TRIG)**

With DATEPROC (TRIG), the millennium language extensions are enabled, and the automatic windowing that the compiler applies to operations on windowed date fields is sensitive to specific trigger or limit values in the date fields and in other nondate fields that are stored into or compared with the windowed date fields. These special values represent invalid dates that can be tested for or used as upper or lower limits.

**Performance:** The DATEPROC (TRIG) option results in slower-performing code for windowed date comparisons.

#### **DATEPROC (NOTRIG)**

With DATEPROC (NOTRIG), the millennium language extensions are enabled, and the automatic windowing that the compiler applies to operations on windowed dates does not recognize any special trigger values in the operands. Only the value of the year part of dates is relevant to automatic windowing.

**Performance:** The DATEPROC (NOTRIG) option is a performance option that assumes valid date values in windowed date fields.

#### **NODATEPROC**

NODATEPROC indicates that the extensions are not enabled for this compilation unit. This affects date-related program constructs as follows:

- The DATE FORMAT clause is syntax-checked, but has no effect on the execution of the program.
- The DATEVAL and UNDATE intrinsic functions have no effect. That is, the value returned by the intrinsic function is exactly the same as the value of the argument.
- The YEARWINDOW intrinsic function returns a value of zero.

#### **Notes:**

1. NODATEPROC conforms to the COBOL 85 Standard.
2. You can specify the FLAG|NOFLAG and TRIG|NOTRIG suboptions in any order. If you omit either suboption, it defaults to the current setting.

#### **RELATED REFERENCES**

“FLAG” on page 274

“Conflicting compiler options” on page 259

---

## DBCS



Default is: NODBCS

Abbreviations are: None

Using DBCS causes the compiler to recognize X'0E' (SO) and X'0F' (SI) as shift codes for the double-byte portion of a nonnumeric literal.

With DBCS selected, the double-byte portion of the literal is syntax-checked and the literal remains category alphanumeric.

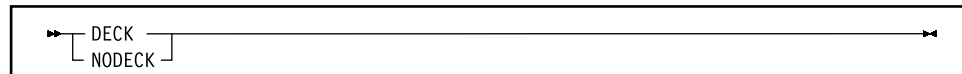
NODBCS conforms to the COBOL 85 Standard.

### RELATED REFERENCES

“Conflicting compiler options” on page 259

---

## DECK



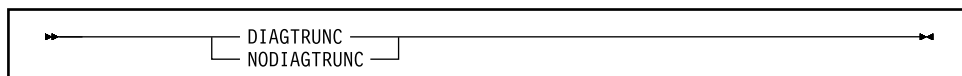
Default is: NODECK

Abbreviations are: D|NOD

Use DECK to produce object code in the form of 80-column card images. If you use the DECK option, be certain that SYSPUNCH is defined in your JCL for compilation.

---

## DIAGTRUNC



Default is: NODIAGTRUNC

Abbreviations are: DTR, NODTR

DIAGTRUNC causes the compiler to issue a severity-4 (Warning) diagnostic message for MOVE statements with numeric receivers when the receiving data item has fewer integer positions than the sending data item or literal. In statements with multiple receivers, the message is issued separately for each receiver that could be truncated.

The diagnostic message is also issued for implicit moves associated with statements such as these:

- INITIALIZE

- READ . . . INTO
- RELEASE . . . FROM
- RETURN . . . INTO
- REWRITE . . . FROM
- WRITE . . . FROM

The diagnostic is also issued for moves to numeric receivers from alphanumeric data-names or literal senders, except when the sending field is reference-modified.

There is no diagnostic for COMP-5 receivers, nor for binary receivers when you specify the TRUNC(BIN) option.

#### RELATED CONCEPTS

“Formats for numeric data” on page 36

“Reference modifiers” on page 87

#### RELATED REFERENCES

“TRUNC” on page 297

“Conflicting compiler options” on page 259

---

## DISK/PRINT

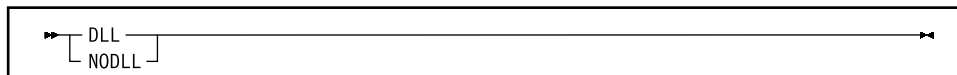
The DISK and PRINT|NOPRINT compiler options are unique to COBOL under CMS.

#### RELATED REFERENCES

“Additional compiler options under CMS” on page 247

---

## DLL



Default is: NODLL

Abbreviations are: None

Execution of programs compiled with the DLL option is allowed only under OS/390. You can specify the DLL option when compiling under VM/CMS, but the resulting application object modules must be moved to OS/390 for linking and for execution.

Use DLL to instruct the compiler to generate an object module that is enabled for dynamic link library (DLL) support. DLL enablement is required if the program will be part of a DLL, will reference DLLs, or if the program contains object-oriented COBOL syntax such as INVOKE statements, class definitions, or calls to SOMobjects API functions.

Specification of the DLL option requires that the RENT linkage editor option also be used.

NODLL instructs the compiler to generate an object module that is not enabled for DLL usage.

---

## DUMP



Default is: NODUMP

Abbreviations are: DU | NODU

**Not general:** This option is not intended for general use.

Use DUMP to produce a system dump at compile time for an internal compiler error. Under CMS, this option specifies that there is to be an abend at the point of failure, but no output is produced. The DUMP option should only be used at the request of an IBM representative.

The dump, which consists of a listing of the compiler's registers and a storage dump, is intended primarily for diagnostic personnel for determining errors in the compiler.

If you use the DUMP option, include a DD statement at compile time to define SYSABEND, SYSUDUMP, or SYSMDUMP.

With DUMP, the compiler will not issue a diagnostic message before abnormal termination processing. Instead, a user abend will be issued with an IGYppnnnn message. In general, a message IGYppnnnn corresponds to a compile-time user abend nnnn. However, both IGYpp5nnn and IGYpp1nnn messages produce a user abend of 1nnn. You can usually distinguish whether the message is really a 5nnn or a 1nnn by recompiling with the NODUMP option.

Use NODUMP if you want normal termination processing, including:

- Diagnostic messages produced so far in compilation
- A description of the error
- The name of the compiler phase currently executing
- The line number of the COBOL statement being processed when the error was found (if you have compiled with OPTIMIZE, the line number might not always be correct; for some errors, it will be the last line in the program)
- The contents of the general purpose registers

Using the DUMP and OPTIMIZE compiler options together could cause the compiler to produce a system dump instead of the following optimizer message:

```
"IGYOP3124-W This statement may cause a program exception at
execution time."
```

This situation is not a compiler error. Using the NODUMP option will allow the compiler to issue message IGYOP3124-W and continue processing.

### RELATED TASKS

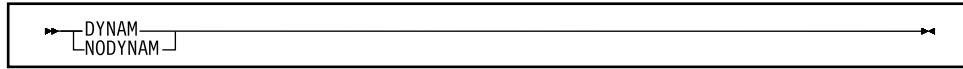
Understanding abend codes (*Language Environment Debugging Guide and Run-Time Messages*)

RELATED REFERENCES

“Conflicting compiler options” on page 259

---

## DYNAM



Default is: NODYNAM

Abbreviations are: DYN | NODYN

Use DYNAM to cause nonnested, separately compiled programs invoked through the CALL *literal* statement to be loaded (for CALL) and deleted (for CANCEL) dynamically at run time. Any CALL *identifier* statements that cannot be resolved in your program are also handled as dynamic calls. The DYNAM compiler option must not be used by programs that are translated by the CICS translator.

If your COBOL program calls programs that have been linked as dynamic link libraries (DLLs), then you must not use the DYNAM option. You must instead compile the program with the NODYNAM and DLL options. In particular, if your program calls SOMobjects API functions, then you must compile it with NODYNAM and DLL.

DYNAM conforms to the COBOL 85 Standard.

RELATED REFERENCES

“Making both static and dynamic calls” on page 465

“Conflicting compiler options” on page 259

---

## EXIT

For information on the EXIT compiler option, see the first reference below.

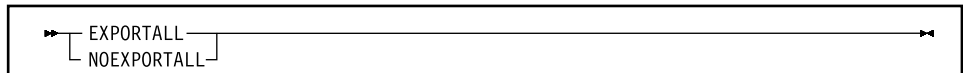
RELATED REFERENCES

“Appendix C. EXIT compiler option” on page 575

“Conflicting compiler options” on page 259

---

## EXPORTALL



Default is: NOEXPORTALL

Abbreviations are: EXP | NOEXP

Execution of programs compiled with the EXPORTALL option is only allowed under OS/390. You can specify the EXPORTALL option when compiling under VM/CMS, but you must move the resulting application object modules to OS/390 for linking and for execution.

Use EXPORTALL to instruct the compiler to automatically export the following symbols when the object deck is link-edited to form a DLL:

- From each program definition, export the PROGRAM-ID name and each alternate entry point name.
- From each class definition, export the external SOM symbols needed to bind to the class when it is a DLL, namely:
  - <classname>NewClass
  - <classname>ClassData
  - <classname>CClassData

With these symbols exported from the DLL:

- The exported program and entry point names can be called from programs in the root load module or in other DLL load modules in the application, as well as from programs that are linked into the same DLL.
- The methods in the classes can be invoked from programs in the *root* load module or in other DLL load modules in the application, as well as from programs that are linked into the same DLL.

Specification of the EXPORTALL option requires that the RENT linkage editor option is also used.

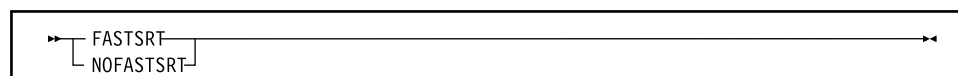
NOEXPORTALL instructs the compiler to not export any symbols. In this case the programs and class definitions are only accessible from other routines that are link-edited into the same load module together with this COBOL class or program definition.

#### RELATED REFERENCES

“Conflicting compiler options” on page 259

---

## FASTSRT



Default is: NOFASTSRT

Abbreviations are: FSRT | NOFSRT

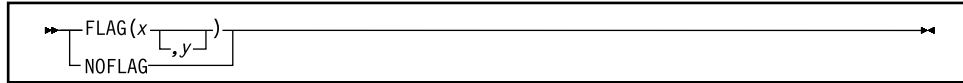
FASTSRT allows IBM DFSORT, or its equivalent, to perform the input and output instead of COBOL.

NOFASTSRT conforms to the COBOL 85 Standard

#### RELATED TASKS

“Improving sort performance with FASTSRT” on page 181

## 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==>  IGYPS2121-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 231



---

## FLAGMIG



Default is: NOFLAGMIG

Abbreviations are: None

Use FLAGMIG to identify language elements that might be implemented differently in VS COBOL II Release 2 than in COBOL for OS/390 & VM.

NOFLAGMIG conforms to the COBOL 85 Standard.

Implementation of the COBOL 85 Standard created some instances where incompatibilities with VS COBOL II Release 2 can occur. Using the CMPR2 and FLAGMIG options aids in the migration of programs written for VS COBOL II Release 2 to COBOL for OS/390 & VM.

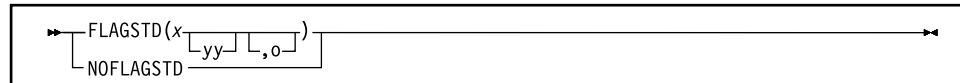
### RELATED REFERENCES

“Conflicting compiler options” on page 259

CMPR2, FLAGMIG, and NOCOMPILE compiler options (*COBOL for OS/390 & VM Compiler and Run-Time Migration Guide*)

---

## 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).

0 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")

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.

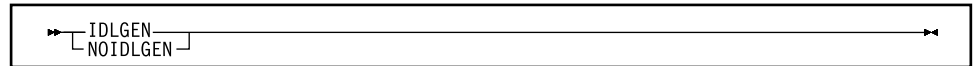
LINE	COL	CODE	FIPS MESSAGE TEXT
		IGYDS8211	Comment lines before "IDENTIFICATION DIVISION": nonconforming nonstandard, IBM extension to ANS/ISO 1985.
11.14		IGYDS8111	"GLOBAL clause": nonconforming standard, ANS/ISO 1985 high subset.
59.12		IGYPS8169	"USE FOR DEBUGGING statement": obsolete element in ANS/ISO 1985.
FIPS MESSAGES TOTAL			STANDARD      NONSTANDARD      OBSOLETE
			3                      1                      1                      1

#### RELATED REFERENCES

"Conflicting compiler options" on page 259

---

## IDLGEN



Default is: NOIDLGEN

Abbreviations are: IDL | NOIDL

**OS/390 only:** This option is not supported under VM/CMS.

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 is written to the data set specified on the SYSIDL DD statement, which must be included in the JCL for the compilation. The data set name on the SYSIDL DD statement must specify a member of a partitioned data set (PDS) that has fixed or fixed-blocked records (RECFM=F or RECFM=FB) and a record length (LRECL) of between 80 and 255 characters. The default record length is 255 characters. IDL does not support continuation characters, so if a line output by the compiler is longer than the LRECL of the IDL output data set, the line is truncated. This could occur if the class uses very long names for methods or the external class-name, and the LRECL of the IDL output data set is too short. A record length of 255 is recommended, since IDL created by COBOL compiler will always fit in 255-character lines.

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 with external class-names formed with eight or fewer characters, but if long external class-names are used, the IR entry is essential to enable the COBOL compiler to obtain the eight-character (or less) PDS member name to include.

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 OS/390, Windows, and AIX might differ; therefore you should regenerate the IDL for the target platform rather than port it between platforms. For example, the procedure-pointer data type in COBOL for OS/390 & VM is an 8-byte data item that does not map to any native IDL type, hence a COBOL-specific mapping is used. On Windows and AIX, procedure-pointers are 4-byte data items that map to IDL pointers. Another example is that on OS/390 or 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.

Do not use SIZE(MAX) when you are compiling large object-oriented applications with the IDLGEN option. You must leave storage space in the user region for Interface Repository data.

#### RELATED TASKS

“Chapter 24. Writing object-oriented programs” on page 375  
“Chapter 26. Using SOM IDL-based class libraries” on page 425  
*OS/390 SOMobjects Programmer’s Guide*

#### RELATED REFERENCES

“Conflicting compiler options” on page 259

---

## INTDATE

```
INTDATE( [ ANSI ] )
```

Default is: INTDATE(ANSI)

INTDATE(ANSI) instructs the compiler to use the ANSI COBOL Standard starting date for integer dates used with date intrinsic functions. Day 1 is Jan 1, 1601.

With INTDATE(ANSI), the date intrinsic functions return the same results as in COBOL/370 Release 1.

INTDATE(LILIAN) instructs the compiler to use the Language Environment Lilian starting date for integer dates used with date intrinsic functions. Day 1 is Oct 15, 1582.

With INTDATE(LILIAN), the date intrinsic functions return results compatible with the Language Environment date callable services. These results will be different than in COBOL/370 Release 1.

**Usage notes:**

1. When INTDATE(LILIAN) is in effect, CEECBLDY is not usable since you have no way to turn an ANSI integer into a meaningful date using either intrinsic functions or callable services. If you code a CALL *literal* statement with CEECBLDY as the target of the call with INTDATE(LILIAN) in effect, the compiler diagnoses this and converts the call target to CEEDAYS.
2. If your installation default is INTDATE(LILIAN), you should recompile all of your COBOL/370 Release 1 programs that use intrinsic functions to ensure that all of your code uses the Lilian integer date standard. This method is the safest, because you can store integer dates and pass them between programs, even between PL/I, COBOL, and C programs, and know that the date processing will be consistent.

RELATED REFERENCES

“Conflicting compiler options” on page 259

## LANGUAGE

→ LANGUAGE(*name*) ←

Default is: LANGUAGE(ENGLISH)

Abbreviations are: LANG(EN|UE|JA|JP)

Use the LANGUAGE option to select the language in which compiler output will be printed. The information that will be printed in the selected language includes diagnostic messages, source listing page and scale headers, FIPS message headers, message summary headers, compilation summary, and headers and notations that result from the selection of certain compiler options (MAP, XREF, VBREF, and FLAGSTD).

*name* specifies the language for compiler output messages. Possible values for the LANGUAGE option are shown in the table.

Name	Abbreviation <sup>1</sup>	Output language
ENGLISH	EN	Mixed-case English (the default)
JAPANESE	JA, JP	Japanese, using the Japanese character set
UENGLISH <sup>2</sup>	UE	Uppercase English

1. If your installation’s system programmer has provided a language other than those described, you must specify at least the first two characters of this other language’s name.
2. To specify a language other than UENGLISH, the appropriate language feature must be installed.

If the LANGUAGE option is changed at compile time (using CBL or PROCESS statements), some initial text will be printed using the language that was in effect at the time the compiler was started.

**NATLANG:** The NATLANG run-time option allows you to control the national language to be used for the run-time environment, including error messages, month names, and day-of-the-week names. The LANGUAGE compiler option and the NATLANG run-time option act independently of each other. You can use them together with neither taking precedence over the other.

---

## LIB

```
LIB
NOLIB
```

Default is: LIB

Abbreviations are: None

If your program uses COPY, BASIS, or REPLACE statements, you need to specify the LIB compiler option. In addition, for COPY and BASIS statements, you need to define the library or libraries from which the compiler can take the copied code. Define the libraries with DD statements, ALLOCATE commands, environment variables, or FILEDEF commands as appropriate for your environment. When using JCL, also include a DD statement to allocate SYSUT5.

LIB conforms to the COBOL 85 Standard.

### RELATED REFERENCES

“Compiler-directing statements” on page 304

“Conflicting compiler options” on page 259

---

## 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



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 and its location in the object code if the program is compiled with the NORENT option

The output is generated if:

- You specify the COMPILE option (or the NOCOMPILE(*x*) option is in effect and an error level *x* or higher does not occur).
- You do not specify the OFFSET option.

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.

### RELATED TASKS

"Getting listings" on page 319

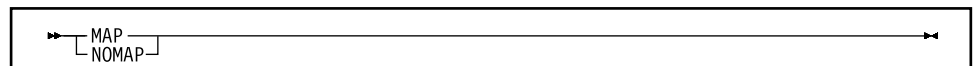
### RELATED REFERENCES

"Conflicting compiler options" on page 259

\*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 and its location in the object code if the program is compiled with the NORENT option

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 324

#### RELATED CONCEPTS

"Chapter 17. Debugging" on page 309

#### RELATED TASKS

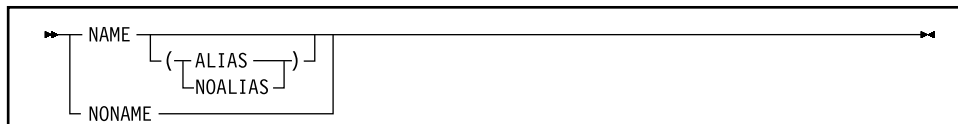
"Getting listings" on page 319

#### RELATED REFERENCES

\*CONTROL (\*CBL) statement (*IBM COBOL Language Reference*)

---

## NAME



Default is: NONAME, or NAME(NOALIAS) if only NAME is specified

Abbreviations are: None

Use NAME to generate a link-edit NAME card for each object module. NAME is also used to generate names for each load module when doing batch compilations. When NAME is specified, a NAME card is appended to each object module that is created. Load module names are formed using the rules for forming module names from PROGRAM-ID statements.

If you specify NAME(ALIAS) and if your program contains ENTRY statements, a link-edit ALIAS card is generated for each ENTRY statement.

The NAME or NAME(ALIAS) options cannot be used when compiling programs that will be prelinked with the Language Environment prelinker.

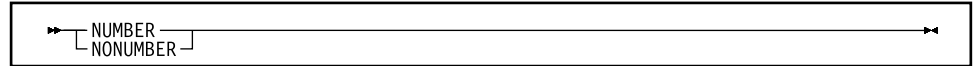
#### RELATED REFERENCES

PROGRAM-ID paragraph (*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.

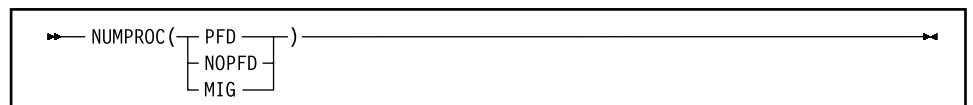
If you are doing a batch compilation and LIB and NUMBER are in effect, all programs in the batch compile will be treated as a single input file. The sequence numbers of the entire input file must be in ascending order.

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.

---

## NUMPROC



Default is: NUMPROC(NOPFD)

Abbreviations are: None

Use NUMPROC(NOPFD) if you want the compiler to perform invalid sign processing. This option is not as efficient as NUMPROC(PFD); object code size will be increased, and there could be an increase in run-time overhead to validate all signed data.

NUMPROC(NOPFD) and NUMPROC(MIG) conform to the COBOL 85 Standard.

NUMPROC(PFD) is a performance option that can be used to bypass invalid sign processing. Use this option **only** if your program data agrees exactly with the following IBM system standards:

**External decimal, unsigned:** High-order 4 bits of the sign byte contain X'F'.

**External decimal, signed overpunch:** High-order 4 bits of the sign byte contain X'C' if the number is positive or 0, X'D' if it is not.

**External decimal, separate sign:** Separate sign contains the character '+' if the number is positive or 0, '-' if it is not.

**Internal decimal, unsigned:** Low-order 4 bits of the low-order byte contain X'F'.

**Internal decimal, signed:** Low-order 4 bits of the low-order byte contain X'C' if the number is positive or 0, X'D' if it is not.

Data produced by COBOL arithmetic statements conforms to the above IBM system standards. However, using REDEFINES and group moves could change data so that it no longer conforms. If you use NUMPROC(PFD), use the INITIALIZE statement to initialize data fields, rather than using group moves.

Using NUMPROC(PFD) can affect class tests for numeric data. You should use NUMPROC(NOPFD) or NUMPROC(MIG) if a COBOL program calls programs written in PL/I or FORTRAN.

Sign representation is not only affected by the NUMPROC option, but also by the installation time option NUMCLS.

Use NUMPROC(MIG) to aid in migrating OS/VS COBOL programs to IBM COBOL for OS/390 & VM. When NUMPROC(MIG) is in effect, the following processing occurs:

- Preferred signs are created only on the output of MOVE statements and arithmetic operations.
- No explicit sign repair is done on input.
- Some implicit sign repair might occur during conversion.
- Numeric comparisons are performed by a decimal compare, not a logical compare.

#### RELATED TASKS

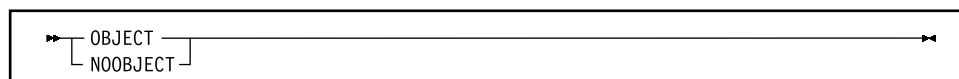
"Checking for incompatible data (numeric class test)" on page 42

#### RELATED REFERENCES

"Sign representation and processing" on page 41

---

## OBJECT



Default is: OBJECT

Abbreviations are: OBJ | NOOBJ

Use OBJECT to place the generated object code on disk or tape to be later used as input for the linkage editor.

If you specify OBJECT, include a SYSLIN DD statement in your JCL for compilation.

The only difference between DECK and OBJECT is in the routing of the data sets:

- DECK output goes to the data set associated with ddname SYSPUNCH.
- OBJECT output goes to the data set associated with ddname SYSLIN.

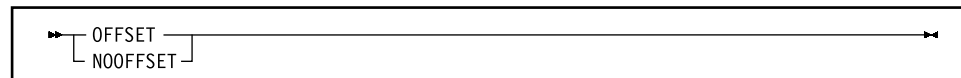
Use the option your installation guidelines recommend.

#### RELATED REFERENCES

“Conflicting compiler options” on page 259

---

## OFFSET



Default is: NOOFFSET

Abbreviations are: OFF | NOOFF

Use OFFSET to produce a condensed PROCEDURE DIVISION listing. With OFFSET, the condensed PROCEDURE DIVISION listing will contain line numbers, statement references, and the location of the first instruction generated for each statement. In addition, the following are produced:

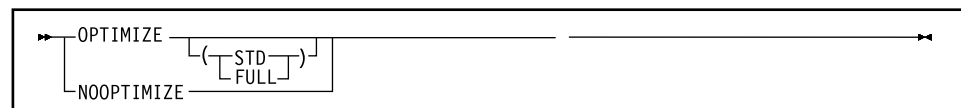
- Global tables.
- Literal pools.
- Size of the program’s WORKING-STORAGE, and its location in the object code if the program is compiled with the NORENT option.

#### RELATED REFERENCES

“Conflicting compiler options” on page 259

---

## 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. 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.

## Unused data items:

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.

### RELATED CONCEPTS

“Optimization” on page 542

### RELATED REFERENCES

“Conflicting compiler options” on page 259

---

## OUTDD

➔ OUTDD(*ddname*) ➔

Default is: OUTDD(SYSOUT)

Abbreviations are: OUT

Use OUTDD to specify that you want DISPLAY output that is directed to the system logical output device to go to a specific *ddname*. Note that you can specify a file in the hierarchical file system (HFS) with the *ddname* named in OUTDD. See the discussion of the DISPLAY statement for defaults and for behavior when this *ddname* is not allocated.

The MSGFILE run-time option allows you to specify the *ddname* of the file to which all run-time diagnostics and reports generated by the RPTOPTS and RPTSTG run-time options are written. The IBM-supplied default is MSGFILE(SYSOUT). If the OUTDD compiler option and the MSGFILE run-time option both specify the same *ddname*,

the error message information and DISPLAY output directed to the system logical output device are routed to the same destination.

**RELATED TASKS**

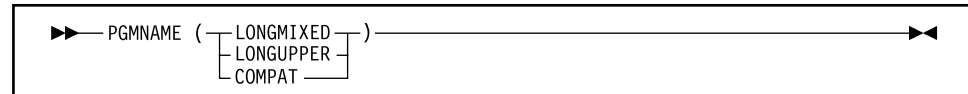
“Displaying values on a screen or in a file (DISPLAY)” on page 29

**RELATED REFERENCES**

MSGFILE (*Language Environment Programming Reference*)

---

## PGMNAME



Default is: PGMNAME (COMPAT)

Abbreviations are: PGMN(LM|LU|CO)

LONGUPPER can be abbreviated as UPPER, LU, or U. LONGMIXED can be abbreviated as MIXED, LM, or M.

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:
  - Calls to nested programs
  - Static calls to separately compiled programs
  - Static SET *procedure-pointer* TO ENTRY *literal* statement
  - CANCEL of a nested program

### PGMNAME(COMPAT)

With PGMNAME(COMPAT), program names are handled in a manner compatible with COBOL/370 Release 1, namely:

- 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, except that if the program name is entered in the literal format and is in the outermost program, then the literal can also contain the extension characters @, #, and \$.
- At least one character must be alphabetic.
- The hyphen cannot be used as the first or last character.

External program names are processed by the compiler as follows:

- They are folded to uppercase.
- They are truncated to eight characters.
- Hyphens are translated to zero (0).
- If the first character is not alphabetic, it is converted as follows:
  - 1-9 are translated to A-J.
  - Anything else is translated to J.

## PGMNAME(LONGUPPER)

With PGMNAME(LONGUPPER), 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 by the compiler as follows:

- They are folded to uppercase.
- Hyphens are translated to zero (0).
- If the first character is not alphabetic, it is converted as follows:
  - 1-9 are translated to A-J.
  - Anything else is translated to J.

Nested-program names are folded to uppercase by the compiler but otherwise are processed as is, without truncation or translation.

## PGMNAME(LONGMIXED)

With PGMNAME(LONGMIXED), program names are processed as is, without truncation, translation, or folding to uppercase.

With PGMNAME(LONGMIXED), all program name definitions must be specified using the literal format of the program name in the PROGRAM-ID paragraph or ENTRY statement.

The literal used for a program name (in any of the contexts listed above as affected by the PGMNAME option) can contain any character in the range X'41'-X'FE'.

## PGMNAME usage notes

- The following are not affected by the PGMNAME option:
  - Class names and method names.
  - System names (assignment-names in SELECT . . . ASSIGN, and text-names or library-names on COPY statements).
  - Dynamic calls. Dynamic calls are resolved with the target program name truncated to eight characters, folded to uppercase, and translation of embedded hyphens or a leading digit.
  - CANCELs of nonnested programs. Name resolution uses the same mechanism as for a dynamic call.
- The PGMNAME option does affect nested-program calls and static calls to programs that are linked together with the caller.

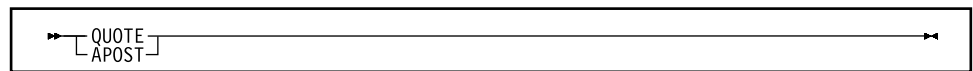
- COBOL source files containing calls to SOM API interfaces must be compiled with PGMNAME(LONGMIXED) because SOM API names are case sensitive and many are longer than eight characters.
- Dynamic calls are not permitted to COBOL programs compiled with the PGMNAME(LONGMIXED) or PGMNAME(LONGUPPER) options unless the program name is less than or equal to 8 bytes and all uppercase. In addition, the name of the program must be identical to the name of the module that contains it.
- PGMNAME(LONGUPPER) and PGMNAME(LONGMIXED) are supported only when NOCMR2 is specified. When CMR2 is specified, PGMNAME(COMPAT) is forced on.
- When using the extended character set supported by PGMNAME(LONGMIXED), be sure to use names that conform to the linker, prelinker, or system conventions that apply, depending on the mechanism used to resolve the names.  
Using characters such as commas or parentheses is not recommended, as these characters are used in the syntax of linker control statements.

RELATED REFERENCES

“Conflicting compiler options” on page 259

---

## QUOTE/APOST



Default is: QUOTE

Abbreviations are: Q | APOST

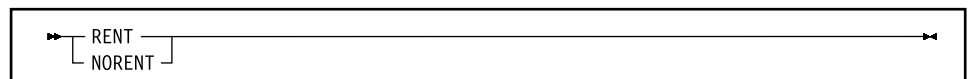
Use QUOTE if you want the figurative constant [ALL] QUOTE or [ALL] QUOTES to represent one or more quotation mark (") characters. QUOTE conforms to the COBOL 85 Standard.

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.

---

## RENT



Default is: NORENT

Abbreviations are: None

A program compiled as RENT is generated as a reentrant object module; a program compiled as NORENT is generated as a nonreentrant object module. Either can be invoked as a main program or subprogram.

**CICS:** You must use RENT for programs to be run under CICS.

**OS/390 UNIX:** You must use RENT for programs to be run in the OS/390 UNIX environment.

When a reentrant program is to be run with extended addressing, you can use the DATA(24|31) option to control whether dynamic data areas are allocated in unrestricted storage or in storage obtained from below 16 MB. Compile programs with RENT or RMODE(ANY) if they will be run with extended addressing in virtual storage addresses above 16 MB.

**DATA(24|31):** DATA(24|31) does not affect programs compiled with NORENT.

RENT|NORENT also affects the RMODE (residency mode) that your program will run under. All IBM COBOL for OS/390 & VM programs are AMODE(ANY).

**Link-edit considerations:** If all programs in a load module are compiled with RENT, then it is recommended that the load module be link-edited with the RENT link-edit attribute. (Use the REUS link-edit attribute instead if the load module will also contain any non-COBOL programs that are serially reusable.)

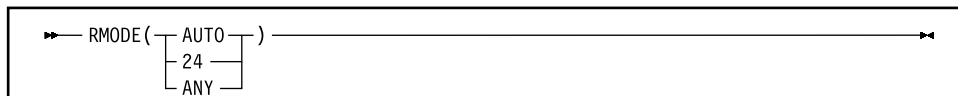
If any program in a load module is compiled with NORENT, the load module must not be link-edited with the RENT or REUS link-edit attributes. The NOREUS linkage-editor option is needed to ensure that the CANCEL statement will guarantee a fresh copy of the program on a subsequent CALL.

#### RELATED REFERENCES

“Conflicting compiler options” on page 259

---

## RMODE



Default is: AUTO

Abbreviations are: None

A program compiled with the RMODE(AUTO) option will have RMODE(24) if NORENT is specified, and RMODE(ANY) if RENT is specified. This is the same behavior as COBOL/370 Release 1 and VS COBOL II.

A program compiled with the RMODE(24) option will have RMODE(24) whether NORENT or RENT is specified.

A program compiled with the RMODE(ANY) option will have RMODE(ANY) whether NORENT or RENT is specified.

**Passing data:** IBM COBOL for OS/390 & VM NORENT programs compiled with RMODE(ANY) that are required to pass data to programs running in AMODE(24) must be link-edited with RMODE(24). The data areas for NORENT programs will be above the line or below the line depending on the RMODE of the program, even if DATA(24) has been specified.



IBM COBOL for OS/390 & VM RENT programs that are required to pass data to programs running in AMODE(24) must be compiled with DATA(24).

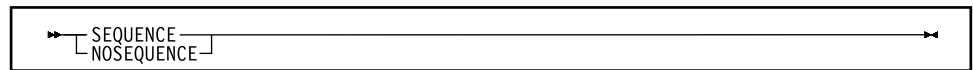
RELATED REFERENCES

“QSAM input/output buffers” on page 267

“Conflicting compiler options” on page 259

---

## 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.

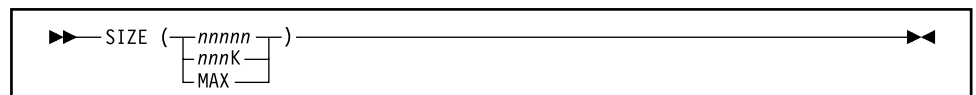
If you are doing a batch compilation and LIB and SEQUENCE are in effect, all programs in the batch compile will be treated as a single input file. The sequence numbers of the entire input file must be in ascending order.

Use NOSEQUENCE to suppress this checking and the diagnostic messages.

NOSEQUENCE conforms to the COBOL 85 Standard.

---

## SIZE



Default is: SIZE(MAX)

Abbreviations are: SZ

*nnnnn* specifies a decimal number that must be at least 851968.

*nnnK* specifies a decimal number in 1K increments. The minimum acceptable value is 832K.

MAX requests the largest available block of storage in the user region for use during compilation.

Use SIZE to indicate the amount of main storage available for compilation (where 1K = 1024 bytes decimal).

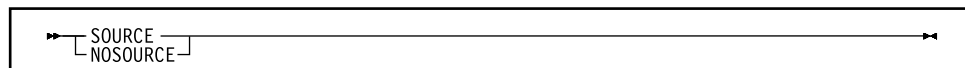
Do not use SIZE(MAX) if, when you invoke the compiler, you require it to leave a specific amount of unused storage available in the user region. If you specify SIZE(MAX) in an extended addressing environment, the compiler uses:

- Above the 16-MB line—all the storage in the user region
- Below the 16-MB line—storage for:
  - Work file buffers
  - Compiler modules that must be loaded below the line

Do not use SIZE(MAX) when you are compiling large object-oriented applications with either the IDLGEN or TYPECHK options. Storage space must be left in the user region for Interface Repository data.

---

## 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 324

### RELATED REFERENCES

\*CONTROL (\*CBL) statement (*IBM COBOL Language Reference*)

---

## SPACE



Default is: SPACE(1)

Abbreviations are: None

Use SPACE to select single-, double-, or triple-spacing in your source code listing.

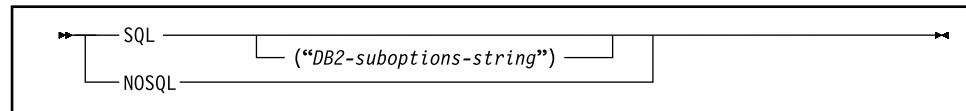
SPACE has meaning only when the SOURCE compiler option is in effect.

### RELATED REFERENCES

"SOURCE" on page 292

---

## SQL



Default is: NOSQL

Abbreviations are: None

**OS/390 only:** This option is not supported under VM/CMS.

Use the SQL compiler option to enable the DB2 coprocessor capability and to specify DB2 suboptions. You must specify the SQL option if your COBOL source program contains SQL statements and it has not been processed by the DB2 precompiler.

When you use the SQL option, the compiler writes the database request module (DBRM) to ddname DBRMLIB. Note that the compiler needs access to DB2 Version 7 or later.

If you specify the NOSQL option, any SQL statements found in the source program are diagnosed and discarded.

Use either quotes or apostrophes to delimit the string of DB2 suboptions.

You can partition a long suboption string into multiple suboption strings on multiple CBL statements. The DB2 suboptions are concatenated in the order of their appearance. For example:

```
//STEP1 EXEC IGYWC, . . .
// PARM.COBOL='SQL("string1")'
//COBOL.SYSIN DD *
    CBL SQL("string2")
    CBL SQL("string3")
    IDENTIFICATION DIVISION.
    PROGRAM-ID. DRIVER1.
    . . .
```

The compiler passes the following suboption string to the DB2 coprocessor:  
"string1 string2 string3"

The concatenated strings are delimited with single spaces as shown. If multiple instances of the same DB2 option are found, the last specification of each option prevails. The compiler limits the length of the concatenated DB2 suboptions string to 4K bytes.

**RELATED TASKS**

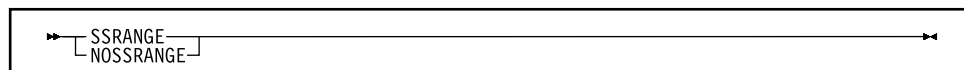
"Compiling with the SQL option" on page 356

**RELATED REFERENCES**

"Conflicting compiler options" on page 259

---

## SSRANGE



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 87

---

## TERMINAL



Default is: NOTERMINAL

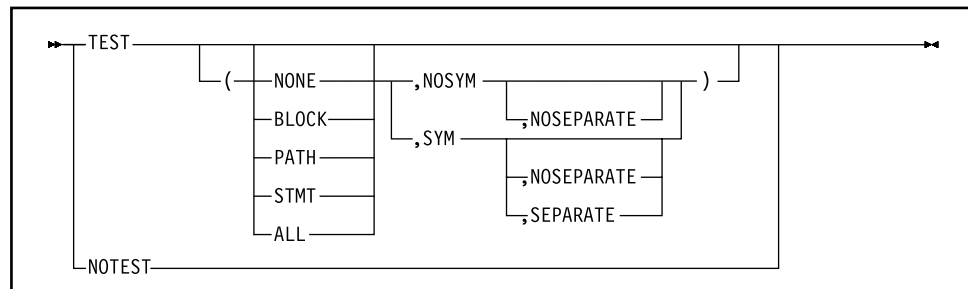
Abbreviations are: TERM | NOTERM

Use TERMINAL to send progress and diagnostic messages to the SYSTERM data set.

Use NOTERMINAL if this additional output is not desired.

---

## TEST



Default is: NOTEST

Abbreviations are: SEP for SEPARATE and NOSEP for NOSEPARATE.

Use TEST to produce object code that enables Debug Tool to perform batch and interactive debugging. The amount of debugging support available depends on which TEST suboptions you use. The TEST option also allows you to request that symbolic variables be included in the formatted dump produced by Language Environment.

Use NOTEST if you do not want to generate object code with debugging information and do not want the formatted dump to include symbolic variables.

TEST has three suboptions:

1. The first specifies whether compiled-in hooks will be generated by the compiler.
2. The second specifies whether symbolic information will be generated.
3. The third specifies whether that symbolic information will be part of the object program or in a separate file.

You can specify any combination of suboptions (one, two, or all) but you can specify SEPARATE only when SYM is in effect.

You can choose one of the following hook-location suboptions:

**NONE** No hooks will be generated.

**BLOCK** Hooks will be generated at all program entry and exit points.

**PATH** Hooks will be generated at all program entry and exit points and at all

path points. A path point is anywhere in a program where the logic flow is not necessarily sequential or can change. Some examples of path points are IF-THEN-ELSE constructs, PERFORM loops, ON SIZE ERROR phrases, and CALL statements.

**STMT** Hooks will be generated at every statement and label, and at all program entry and exit points. In addition, if the DATEPROC option is in effect, hooks will be generated at all date-processing statements.

**ALL** Hooks will be generated at all statements, all path points, and at all program entry and exit points (both outermost and contained programs.) In addition, if the DATEPROC option is in effect, hooks will be generated at all date-processing statements.

When you set the SYM suboption, you can specify the optional suboption SEPARATE, indicating that the symbolic information tables for the Debug Tool are generated in a data set separate from the object module. When you do not specify the SEPARATE suboption, the symbolic information is included in the object module.

When you invoke the COBOL compiler from JCL or from TSO and you specify the TEST(. . .,SYM,SEPARATE) option, the symbolic debug information tables are written to the data set specified in the SYSDEBUG DD statement. The SYSDEBUG DD statement must specify the name of a sequential data set, the name of a PDS or PDSE member, or an HFS path. The data set LRECL must be greater than or equal to 80, and less than or equal 1024. The default LRECL for SYSDEBUG is 1024. The data set RECFM can be F or FB. You can set the block size by either using the BLKSIZE subparameter of the DCB parameter, or leave it to the system to set it to the system-determined default block size.

When you invoke the COBOL compiler from the OS/390 UNIX shell and you specify the TEST(. . .,SYM,SEPARATE) option, the symbolic debug information tables are written to file *file.dbg* in the current directory, where *file* is the name of the COBOL source file.

**Production debugging:** With the latest levels of Debug Tool you can step through your program, even if there are no compiled-in hooks, using new Debug Tool function called *overlay hooks*. To take advantage of this function, compile with NOOPTIMIZE and TEST(NONE). This is sometimes referred to as *production debugging* or *production-level debugging*, because you avoid the performance overhead of compiled-in hooks yet most Debug Tool function is still available. Production debugging also refers to using the SEPARATE suboption to keep the load module size smaller and yet maintain Debug Tool and Language Environment formatted dump functionality.

When you use production debugging with overlay hooks, there are some restrictions on debugging function, such as:

- The GO TO Debug Tool command is not supported.

There are two ways that the TEST option can improve your formatted dumps from Language Environment:

- Use the NOSYM suboption of TEST to have line numbers in the dump that indicate the failing statement, rather than just an offset.
- Use the SYM suboption of TEST to have the values of the program variables listed in the dump.

With NOTEST, the dump will not have program variables and will not have a line number for the failing statement.

IBM COBOL for OS/390 & VM uses the Language Environment-provided dump services to produce dumps that are consistent in content and format to those produced by other Language Environment-conforming member languages. Whether Language Environment produces a dump for unhandled conditions depends on the setting of the run-time option TERMTHDACT. If you specify TERMTHDACT(DUMP), a dump is generated if a condition of severity 2 or greater goes unhandled.

#### RELATED CONCEPTS

Considerations for setting TERMTHDACT options (*Language Environment Debugging Guide and Run-Time Messages*)

#### RELATED TASKS

“Defining the debug data set (SYSDEBUG)” on page 222

Generating a dump (*Language Environment Debugging Guide and Run-Time Messages*)

Invoking Debug Tool using the run-time TEST option (*Debug Tool User’s Guide and Reference*)

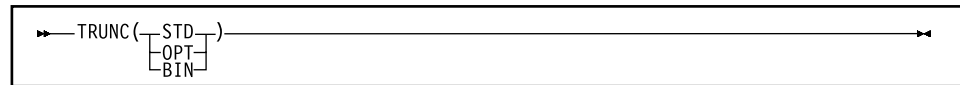
#### RELATED REFERENCES

“Conflicting compiler options” on page 259

TEST | NOTEST (*Language Environment Programming Reference*)

---

## 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).

#### Tips:

1. 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.
2. There are some cases when programs compiled with the TRUNC(OPT) option under COBOL for OS/390 & VM could give different results than the same programs compiled under OS/VSE COBOL with NOTRUNC. You must actually lose nonzero high-order digits for this difference to appear. For statements where loss of high-order digits might cause a difference in results between COBOL for OS/390 & VM and OS/VSE COBOL, COBOL for OS/390 & VM will issue a diagnostic message. If you receive this message, make sure that either the sending items will not contain large numbers or that the receivers are defined with enough digits in the PICTURE clause to handle the largest sending data items.

### 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 IMS, DB2, C/C++, FORTRAN, 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	Display
Sender	123451	00 01 E2 3B	123451
Receiver TRUNC(STD)	51	00 33	51



Data item	Decimal	Hex	Display
Receiver TRUNC(OPT)	-7621	E213B	2J
Receiver TRUNC(BIN)	-7621	E213B	762J

A halfword of storage is allocated for BIN-VAR. The result of this MOVE statement if the program is compiled with the TRUNC(STD) option is 51; the field is truncated to conform to the PICTURE clause.

If you compile the program with 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.

## 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	Display
Sender	1234567891	491961021D3	1234567891
Receiver TRUNC(STD)	567891	001081AA153	567891
Receiver TRUNC(OPT)	567891	531AA108100	567891
Receiver TRUNC(BIN)	1234567891	491961021D3	1234567891

When you specify TRUNC(STD), the sending data is truncated to six integer digits to conform to the PICTURE clause of the BINARY receiver.

When you specify TRUNC(OPT), the compiler assumes the sending data is not larger than the PICTURE clause precision of the BINARY receiver. The most efficient code sequence in this case is truncation as if TRUNC(STD) were in effect.

When you specify TRUNC(BIN), no truncation occurs because all of the sending data fits into the binary fullword allocated for BIN-VAR.

### RELATED CONCEPTS

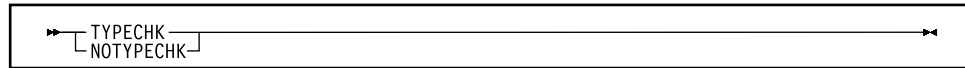
“Formats for numeric data” on page 36

### RELATED TASKS

“Preparing COBOL programs to run under CICS” on page 350

---

# TYPECHK



Default is: NOTYPECHK

Abbreviations are: TC|NOTC

**OS/390 only:** This option is not supported under VM/CMS.

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.

Do not use SIZE(MAX) when you are compiling large object-oriented applications with the TYPECHK option. Storage space must be left in the user region for Interface Repository data.

#### RELATED CONCEPTS

“Chapter 25. System Object Model” on page 415

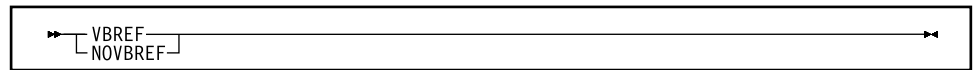
#### RELATED REFERENCES

“IDLGEN” on page 277

“Conflicting compiler options” on page 259

---

## VBREF



Default is: NOVBREF

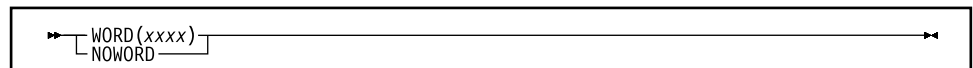
Abbreviations are: None

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.

---

## WORD



Default is: NOWORD

Abbreviations are: WD | NOWD

*xxxx* specifies the ending characters of the name of the reserved-word table (IGYC*xxxx*) to be used in your compilation. IGYC are the first four standard characters of the name, and *xxxx* can be one to four characters in length.

Use WORD(*xxxx*) to specify that an alternate reserved-word table is to be used during compilation.

Alternate reserved-word tables provide changes to the IBM-supplied default reserved-word table. Your systems programmer might have created one or more alternate reserved-word tables for your site. See your systems programmer for the names of alternate reserved-word tables.

IBM COBOL provides an alternate reserved-word table (IGYCCICS) specifically for CICS applications. It is set up to flag COBOL words not supported under CICS with an error message. If you want to use this CICS reserved-word table during your compilation, specify the compiler option WORD(CICS).

IBM COBOL provides an alternate reserved-word table (IGYCNOOO) specifically for your existing applications that use object-oriented reserved words (OBJECT, METHOD, FACTORY, and others) as user-defined words. IGYCNOOO does not reserve the words used for object-orientation, so your applications will still work if you refer to these words:

CLASS-ID	METAClass	RECURSIVE
END-INVOKE	METHOD	RETURNING
FACTORY	METHOD-ID	REPOSITORY
INHERITS	OBJECT	SELF
INVOKE	OVERRIDE	SUPER
LOCAL-STORAGE		

If you want to use this N000 reserved-word table during your compilation, specify the compiler option `WORD(N000)`.

`NOWORD` conforms to the COBOL 85 Standard.

#### RELATED TASKS

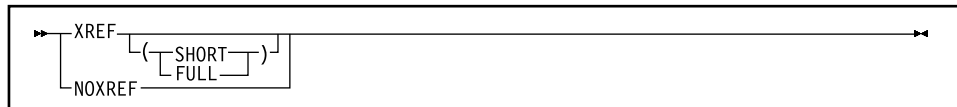
“Preparing COBOL programs to run under CICS” on page 350

#### RELATED REFERENCES

“Conflicting compiler options” on page 259

---

## XREF



Default is: `NOXREF`

Abbreviations are: `X|NOX`

You can choose `XREF`, `XREF(FULL)`, or `XREF(SHORT)`.

Use `XREF` to get a sorted cross-reference listing. EBCDIC data names and procedure names are listed in alphanumeric order. DBCS data names and procedure names are listed based on their physical order in the program, and appear before the EBCDIC data names and procedure names, unless the `DBCSEXREF` installation option is selected with a DBCS ordering program. In this case, DBCS data names and procedure names are ordered as specified by the DBCS ordering program.

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 DBCS data names and procedure-names as well as EBCDIC 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 17. Debugging” on page 309

RELATED TASKS

“Getting listings” on page 319

RELATED REFERENCES

COBOL compiler options (*Language Environment Debugging Guide and Run-Time Messages*)

---

## 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.

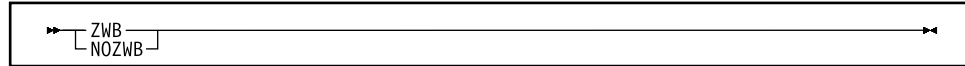
For example, if the current year is 2001, the DATEPROC option is in effect, and you use the YEARWINDOW(1900) option, the program will terminate with an error message.

RELATED REFERENCES

“Conflicting compiler options” on page 259

---

## ZWB



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.

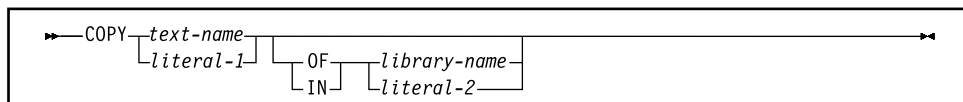
### **BASIS** statement

This extended source program library statement provides a complete COBOL program as the source for a compilation. For rules of formation and processing, 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.

### **COPY** statement



This library statement places prewritten text into a COBOL program. A user-defined word can be the same as a *text-name* or a *library-name*. The uniqueness of *text-name* and *library-name* is determined after the formation and conversion rules for a system-dependent name have been applied. If *library-name* is omitted, SYSLIB is assumed.

### **When compiling with JCL:**

*Text-name*, *library-name*, and *literal* are processed as follows:

- The name (which can be one to 30 characters long) is truncated to eight characters. Only the first eight characters of *text-name* and *library-name* are used as the identifying name. These eight characters must be unique within one COBOL library.
- The name is folded to uppercase.

- Hyphens that are not the first or last character are translated to zero (0), and a warning message is given.
- If the first character is numeric, then the characters (1-9) are translated to A-I, and a warning message is given. An error message is given if the first or last character is a hyphen (or if it is a hyphen, @, #, or \$ for a literal).

For example:

```
COPY INVOICES1Q
COPY "Company-#Employees" IN Personellib
```

In the IN/OF phrase, *library-name* is the ddname that identifies the partitioned data set to be copied from. Use a DD statement such as in the following example to define *library-name*:

```
//COPYLIB DD DSNAME=ABC.COB,VOLUME=SER=111111,
// DISP=SHR,UNIT=3380
```

To specify more than one copy library, use either JCL or a combination of JCL and the IN/OF phrase. Using just JCL, concatenate data sets on your DD statement for SYSLIB. Alternatively, define multiple DD statements and include the IN/OF phrase on your COPY statements.

The maximum block size for the copy library depends on the device on which your data set resides.

#### When compiling in the OS/390 UNIX System Services shell:

When you compile with the cob2 command, COPY text is included from the HFS. *Text-name*, *library-name*, and *literal* are processed as follows:

- User-defined words are folded to uppercase. Literals are not. Because OS/390 UNIX is case sensitive, if your file name is lowercase or mixed case, you must specify it as a literal.
- When *text-name* is a literal and *library-name* is omitted, *text-name* is used directly: as a file name, a relative path name, or an absolute path name (if the first character is /). For example:

```
COPY "MyInc"
COPY "x/MyInc"
COPY "/u/user1/MyInc"
```

- When *text-name* is a user-defined word and an environment variable of that name is defined, the value of the environment variable is used as the name of the file containing the copy text.

If an environment variable of that name is not defined, the copy text is searched for as the following names, in the order given:

1. *text-name.cpy*
2. *text-name.CPY*
3. *text-name.cbl*
4. *text-name.CBL*
5. *text-name.cob*
6. *text-name.COB*
7. *text-name*

- When *library-name* is a literal, it is treated as the actual path, relative or absolute, from which to copy file *text-name*.

- When *library-name* is a user-defined word, it is treated as an environment variable. The value of the environment variable is used as the path. If the environment variable is not set, an error occurs.
- If both *library-name* and *text-name* are specified, the compiler forms the path name for the copy text by concatenating *library-name* and *text-name* with a path separator (/) inserted between the two values. For example, suppose you have the following setting for COPY MYCOPY OF MYLIB:

```
export MYCOPY=mystuff/today.cpy
export MYLIB=/u/user1
```

These settings result in:

```
/u/user1/mystuff/today.cpy
```

When *library-name* is an environment variable that identifies the path from which copy text is to be copied, use an export command such as the following example to define *library-name*:

```
export COPYLIB=/u/mystuff/copybooks
```

The name of the environment variable must be uppercase. To specify more than one copy library, set the environment variable to multiple path names delimited by : (colon).

When *library-name* is omitted and *text-name* is not an absolute path name, the copy text is searched for in this order:

1. In the current directory
2. In the paths specified on the -I cob2 option
3. In the paths specified in the SYSLIB environment variable

#### **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.

#### **SERVICE LABEL statement**

This statement is generated by the CICS translator to indicate control flow. It is not intended for general use.

#### **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
- User label-handling procedures—LABEL
- Debugging lines and sections—DEBUGGING

**RELATED TASKS**

“Changing the header of a source listing” on page 6

“Eliminating repetitive coding” on page 551

“Using compiler-directing statements” on page 245

“Specifying compiler options under OS/390” on page 223

“Specifying compiler options under OS/390 UNIX” on page 236

“Setting environment variables under OS/390 UNIX” on page 235

“Specifying compiler options under CMS” on page 246

**RELATED REFERENCES**

“cob2” on page 239

*IBM COBOL Language Reference*



---

## Chapter 17. 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.

If the problem with your program is not easily detected and you do not have a debugger available, you might need to analyze a storage dump of your program.

Besides using the features inherent in COBOL, you can also use the Debug Tool, which is available in the Full Function offering of this compiler.

The Debug Tool offers these productivity enhancements:

- 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” on page 310
- “Debugging using compiler options” on page 313
- “Getting listings” on page 319
- “Preparing to use the debugger” on page 343

### RELATED REFERENCE

- Debug Tool User's Guide and Reference*
- Formatting and analyzing system dumps on OS/390 (*Language Environment Debugging Guide and Run-Time Messages*)
- Debugging example COBOL programs (*Language Environment Debugging Guide and Run-Time Messages*)

---

## 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 311

“Validating data” on page 311

“Finding uninitialized data” on page 311

“Generating information about procedures” on page 312

### 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 20

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 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, DBCS, KANJI, 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 (with any suboption value other than `NONE`) 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 data set. 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 316
- “Finding coding errors”
- “Finding line sequence problems” on page 315
- “Checking for valid ranges” on page 315
- “Finding program entity definitions and references” on page 317
- “Listing data items” on page 318
- “Getting listings” on page 319
- “Preparing to use the debugger” on page 343

#### RELATED REFERENCES

- “COMPILE” on page 265
- “SEQUENCE” on page 291
- “SSRANGE” on page 294
- “FLAG” on page 274
- “XREF” on page 302
- “MAP” on page 281
- “VBREF” on page 301
- “LIST” on page 281
- “TEST” on page 295

## 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.

If you are compiling in the TSO foreground, you can send the messages to your screen by defining your data set as the SYSTEM data set and using the TERM option when you compile your program.

### 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: DECK, OFFSET, 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 265

## 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 291

## 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 294

“Performance-related compiler options” on page 545

## 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 232

RELATED TASKS

“Generating a list of compiler error messages” on page 230

RELATED REFERENCES

“FLAG” on page 274

“Messages and listings for compiler-detected errors” on page 231

### 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.

```

090671** /
090672** *****
090673** *** INITIALIZE PARAGRAPH **
090674** *** Open files. Accept date, time and format header lines. **
090675** *** Load location-table. **
090676** *****
090677** 100-initialize-paragraph.
090678** move spaces to ws-transaction-record IMP 331
090679** move spaces to ws-commuter-record IMP 307
090680** move zeroes to commuter-zipcode IMP 318
090681** move zeroes to commuter-home-phone IMP 319
090682** move zeroes to commuter-work-phone IMP 320
090683** move zeroes to commuter-update-date IMP 324
090684** open input update-transaction-file 204
==090684==> IGYPS2052-S An error was found in the definition of file "LOCATION-FILE". The
reference to this file was discarded.
090685** location-file 193
090686** i-o commuter-file 181
090687** output print-file 217
090688** if commuter-file-status not = "00" and not = "97" 241
090689** 1 display "100-OPEN"
090690** 1 move 100 to comp-code 231
090691** 1 perform 500-vsam-error 91069
090692** 1 perform 900-abnormal-termination 91114
090693** end-if
090694** accept ws-date from date UND
==090694==> IGYPS2121-S "WS-DATE" was not defined as a data-name. The statement was discarded.
090695** move corr ws-date to header-date UND 455
==090695==> IGYPS2121-S "WS-DATE" was not defined as a data-name. The statement was discarded.
090696** accept ws-time from time UND
==090696==> IGYPS2121-S "WS-TIME" was not defined as a data-name. The statement was discarded.
090697** move corr ws-time to header-time UND 449
==090697==> IGYPS2121-S "WS-TIME" was not defined as a data-name. The statement was discarded.
090698** read location-file 193
==090698==> IGYPS2053-S An error was found in the definition of file "LOCATION-FILE". This
input/output statement was discarded.
090699** at end
090700** 1 set location-eof to true 256
090701** end-read

```

```

IGYSC0090-W 1700 sequence errors were found in this program.
IGYSC3002-I A severe error was found in the program. The "OPTIMIZE" compiler option was cancelled.
160 IGYDS1089-S "ASSIGN" was invalid. Scanning was resumed at the next area "A" item, level-number, or
the start of the next clause.
193 IGYGR1207-S The "ASSIGN" clause was missing or invalid in the "SELECT" entry for file "LOCATION-FILE".
The file definition was discarded.
269 IGYDS1066-S "REDEFINES" object "WS-DATE" was not the immediately preceding level-1 data item.
The "REDEFINES" clause was discarded.
90602 IGYPS2052-S An error was found in the definition of file "LOCATION-FILE". The reference to this file
was discarded. Same message on line: 90684
90694 IGYPS2121-S "WS-DATE" was not defined as a data-name. The statement was discarded.
Same message on line: 90695
90696 IGYPS2121-S "WS-TIME" was not defined as a data-name. The statement was discarded.
Same message on line: 90697
90698 IGYPS2053-S An error was found in the definition of file "LOCATION-FILE". This input/output statement
was discarded. Same message on line: 90709

```

Messages Total Informational Warning Error Severe Terminating  
 Printed: 13 1 1 1 11

\* Statistics for COBOL program IGYTCARA:  
 \* Source records = 1735  
 \* Data Division statements = 287  
 \* Procedure Division statements = 471  
 End of compilation 1, program IGYTCARA, highest severity 12.  
 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.

If your program contains DBCS user-defined words, these user-defined words are listed before the alphabetic list of EBCDIC user-defined words.

In the OS/390 environment, if a DBCS ordering program is specified in the DBCSXREF installation option, the DBCS user-defined words are listed in order according to the specified DBCS collating sequence. Otherwise, the DBCS user-defined words are listed in physical order according to their appearance in the source program.

In the CMS environment, the DBCS user-defined words are listed in physical order according to their appearance in the source program.

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 340

“Example: XREF output - program-name cross-references” on page 341

“Example: embedded cross-reference” on page 341

#### RELATED TASKS

“Getting listings” on page 319

#### RELATED REFERENCES

“XREF” on page 302

## 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. Use the MAP output to locate the contents of a data item in a system dump.

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
*CBL MAP                 *CBL MAP
```

“Example: MAP output” on page 324

#### RELATED TASKS

“Getting listings” on page 319

#### RELATED REFERENCES

“MAP” on page 281

## 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> ; 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 292
To find certain data items in a storage dump; 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 281 <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)	
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 302 <sup>2 3</sup>
		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>	Generated code	"LIST" on page 281 <sup>2 4</sup>
To verify you still have a valid logic path after you move or add PROCEDURE DIVISION sections	Condensed PROCEDURE DIVISION listing	Condensed verb listing, global tables, WORKING-STORAGE information, and literals	"OFFSET" on page 285
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 301

Use	Listing	Contents	Compiler option
<p>1. To eliminate messages, turn off the options (such as FLAG) that govern the level of compile diagnostic information. 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.</p> <p>2. The context of the procedure reference is indicated by the characters preceding the line number.</p> <p>3. 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.</p> <p>The output is generated if:</p> <ul style="list-style-type: none"> <li>• You specify the COMPILE option (or the NOCOMPILE(x) option is in effect and an error level x or higher does not occur).</li> <li>• You do not specify the OFFSET option. OFFSET and LIST are mutually exclusive options with OFFSET taking precedence.</li> </ul>			

“Example: short listing”

“Example: SOURCE and NUMBER output” on page 323

“Example: MAP output” on page 324

“Example: embedded map summary” on page 325

“Example: nested program map” on page 328

“Example: XREF output - data-name cross-references” on page 340

“Example: XREF output - program-name cross-references” on page 341

“Example: embedded cross-reference” on page 341

“Example: OFFSET compiler output” on page 342

“Example: VBREF compiler output” on page 343

#### RELATED TASKS

“Generating a list of compiler error messages” on page 230

“Reading LIST output” on page 328

Determining the source of error (*Language Environment Debugging Guide and Run-Time Messages*)

#### RELATED REFERENCES

“Messages and listings for compiler-detected errors” on page 231

## 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)

OPT

PROCESS(CBL) statements:

CBL RENT,NOSOURCE,TEST(ALL) (3)

(4)

IGYOS4022-W The "OPTIMIZE" option was discarded due to option conflict resolution.  
The "TEST" option from "PROCESS/CBL" statement took precedence.

Options in effect: (5)

NOADATA  
ADV  
QUOTE  
ARITH(COMPAT)  
NOAWO  
BUFSIZE(4096)  
NOCMPR2  
NOCMPLE(S)  
NOCURRENCY  
DATA(31)  
NODATEPROC  
NODBCS  
NODECK  
NODIAGTRUNC  
NODLL  
NODUMP  
NODYNAM  
NOEXIT  
NOEXPORTALL  
NOFASTSRT  
FLAG(1)  
NOFLAGMIG  
NOFLAGSTD  
NOIDLGEN  
INTDATE(ANSI)  
LANGUAGE(EN)  
NOLIB  
LINECOUNT(60)  
NOLIST  
NOMAP  
NONAME  
NONUMBER  
NUMPROC(NOPFD)  
OBJECT  
NOOFFSET  
NOOPTIMIZE  
OUTDD(SYSOUT)  
PGMNAME(COMPAT)  
RENT  
RMODE(AUTO)  
SEQUENCE  
SIZE(MAX)  
NOSOURCE  
SPACE(1)  
NOSSL  
NOSSRANGE  
NOTERM  
TEST(ALL,SYM, NOSEPARATE)  
TRUNC(STD)  
NOTYPECHK  
NOVBREF  
NOWORD  
NOXREF  
YEARWINDOW(1900)  
ZWB

DATA VALIDATION AND UPDATE PROGRAM (6) IGYTCARA Date 03/30/2000 Time 12:26:53 Page 2

LineID Message code Message text (7)

IGYDS0139-W Diagnostic messages were issued during processing of compiler options.  
These messages are located at the beginning of the listing.  
IGYSC0090-W 3 sequence errors were found in this program.  
160 IGYDS1089-S "ASSIGNN" was invalid. Scanning was resumed at the next area "A" item,  
level-number,or the start of the next clause.  
193 IGYGR1207-S The "ASSIGN" clause was missing or invalid in the "SELECT" entry for file  
"LOCATION-FILE". The file definition was discarded.  
269 IGYDS1066-S "REDEFINES" object "WS-DATE" was not the immediately preceding level-1 data item.  
The "REDEFINES" clause was discarded.  
901 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: 1008

Messages Total Informational Warning Error Severe Terminating (8)

Printed: 14 3 11

\* Statistics for COBOL program IGYTCARA: (9)

\* Source records = 1735

\* Data Division statements = 287

\* Procedure Division statements = 471

End of compilation 1, program IGYTCARA, highest severity 12. (10)

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.
  - RENT** The program was compiled to be reentrant to copy code from a library.
  - NOSOURCE**  
Turning SOURCE off removes the COBOL source code from the COBOL listing.
  - TEST(ALL)**  
The program was compiled for use with Debug Tool.
- (4) Deliberate option conflicts were forced by specifying the OPTIMIZE option on the compiler input parameter list. OPTIMIZE and the TEST(ALL) option specified on the CBL statement are mutually exclusive. As a result, the OPTIMIZE option is ignored.
- (5) Status of options at the start of this compilation.
- (6) Customized page header resulting from the COBOL program TITLE statement.
- (7) Program diagnostics. The first message refers you to any library phase diagnostics. Diagnostics for the library phase are presented at the beginning of the listing.
- (8) Count of diagnostic messages in this program, grouped by severity level.
- (9) Program statistics for the program IGYTCARA.
- (10) Program statistics for the compilation unit. When you perform a batch 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 03/30/2000 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 IGYTCARA - Beginning"
087900 perform 050-create-vsam-master-file.                      90633
088150 display "perform 050-create-vsam-master finished".
088151** 088125 perform 100-initialize-paragraph                90677
088200 display "perform 100-initialize-paragraph finished"
088300 read update-transaction-file into ws-transaction-record  204 331
088400 at end
1 088500 set transaction-eof to true                               254
088600 end-read
088700 display "READ completed"
088800 perform until transaction-eof                               254
1 088900 display "inside perform until loop"
1 089000 perform 200-edit-update-transaction                    90733
1 089100 display "After perform 200-edit "
1 089200 if no-errors                                           365
2 089300 perform 300-update-commuter-record                    90842
2 089400 display "After perform 300-update "
1 089650 else
089651** 2 089600 perform 400-print-transaction-errors          90995
2 089700 display "After perform 400-errors "
1 089800 end-if
1 089900 perform 410-re-initialize-fields                       91056
1 090000 display "After perform 410-reinitialize"
1 090100 read update-transaction-file into ws-transaction-record  204 331
1 090200 at end
2 090300 set transaction-eof to true                               254
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 03/30/2000 Time 12:26:53 Page 62

Data Division Map

(1)  
Data Definition Attribute codes (rightmost column) have the following meanings:  
 D = Object of OCCURS DEPENDING G = GLOBAL S = Spanned file  
 E = EXTERNAL O = Has OCCURS clause U = Undefined format file  
 F = Fixed-length file OG= Group has own length definition V = Variable-length file  
 FB= Fixed-length blocked file R = REDEFINES VB= Variable-length blocked file

(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
Source LineID	Hierarchy and Data Name		Base Locator	Hex-Displacement Blk	Structure	Asmblr Data Definition	Data Type	Data Def Attributes
4	PROGRAM-ID IGYTCARA							
181	FD COMMUTER-FILE						VSAM	F
183	1 COMMUTER-RECORD		BLF=0000	000		DS 0CL80	Group	
184	2 COMMUTER-KEY		BLF=0000	000	0 000 000	DS 16C	Display	
185	2 FILLER		BLF=0000	010	0 000 010	DS 64C	Display	
187	FD COMMUTER-FILE-MST						VSAM	F
189	1 COMMUTER-RECORD-MST		BLF=0001	000		DS 0CL80	Group	
190	2 COMMUTER-KEY-MST		BLF=0001	000	0 000 000	DS 16C	Display	
191	2 FILLER		BLF=0001	010	0 000 010	DS 64C	Display	
193	FD LOCATION-FILE						QSAM	FB
198	1 LOCATION-RECORD		BLF=0002	000		DS 0CL80	Group	
199	2 LOC-CODE		BLF=0002	000	0 000 000	DS 2C	Display	
200	2 LOC-DESCRIPTION		BLF=0002	002	0 000 002	DS 20C	Display	
201	2 FILLER		BLF=0002	016	0 000 016	DS 58C	Display	
204	FD UPDATE-TRANSACTION-FILE						QSAM	FB
209	1 UPDATE-TRANSACTION-RECORD		BLF=0003	000		DS 80C	Display	
217	FD PRINT-FILE						QSAM	FB
222	1 PRINT-RECORD		BLF=0004	000		DS 121C	Display	
229	1 WORKING-STORAGE-FOR-IGYTCARA		BLW=0000	000		DS 1C	Display	
231	77 COMP-CODE		BLW=0000	008		DS 2C	Binary	
232	77 WS-TYPE		BLW=0000	010		DS 3C	Display	
235	1 I-F-STATUS-AREA		BLW=0000	018		DS 0CL2	Group	
236	2 I-F-FILE-STATUS		BLW=0000	018	0 000 000	DS 2C	Display	
237	88 I-O-SUCCESSFUL							
240	1 STATUS-AREA		BLW=0000	020		DS 0CL8	Group	
241	2 COMMUTER-FILE-STATUS		BLW=0000	020	0 000 000	DS 2C	Display	
242	88 I-O-OKEY							
243	2 COMMUTER-VSAM-STATUS		BLW=0000	022	0 000 002	DS 0CL6	Group	
244	3 VSAM-R15-RETURN-CODE		BLW=0000	022	0 000 002	DS 2C	Binary	
245	77 UNUSED-DATA-ITEM		BLW=XXXX	022		DS 10C	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) Base locator used for this data item.
- (6) Hexadecimal displacement from the beginning of the base locator value.
- (7) Hexadecimal displacement from the beginning of the containing structure.
- (8) Pseudoassembler code showing how the data is defined. When a structure contains variable-length fields, the maximum length of the structure is shown.
- (9) Data type and usage.

- (10) Data definition attribute codes. The definitions are explained at the top of the DATA DIVISION map.
- (11) UNUSED-DATA-ITEM was not referenced in the PROCEDURE DIVISION. Because OPTIMIZE(FULL) was specified, UNUSED-DATA-ITEM was deleted, resulting in the base locator being set to XXXX.

RELATED REFERENCES

“Terms used in MAP output” on page 326  
 “Symbols used in LIST and MAP output” on page 326

### 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. IGYTCARA.
. . .
000177 Data division.
000178 File section.
000179
000180
000181 FD COMMUTER-FILE
000182 record 80 characters. (1) (2) (3) (4)
. . .
000222 01 print-record pic x(121). BLF=0004+000 121C
. . .
000228 Working-storage section.
000229 01 Working-storage-for-IGYTCARA pic x. BLW=0000+000 1C
000230
000231 77 comp-code pic S9999 comp. BLW=0000+008 2C
000232 77 ws-type pic x(3) value spaces. BLW=0000+010 3C
000233
000234
000235 01 i-f-status-area. BLW=0000+018 0CL2
000236 05 i-f-file-status pic x(2). BLW=0000+018,0000000 2C
000237 88 i-o-successful value zeroes.
000238
000239
000240 01 status-area. BLW=0000+020 0CL8
000241 05 commuter-file-status pic x(2). BLW=0000+020,0000000 2C
000242 88 i-o-okay value zeroes.
000243 05 commuter-vsam-status. BLW=0000+022,0000002 0CL6
000244 10 vsam-r15-return-code pic 9(2) comp. BLW=0000+022,0000002 2C
000245 10 vsam-function-code pic 9(1) comp. BLW=0000+024,0000004 2C
000246 10 vsam-feedback-code pic 9(3) comp. BLW=0000+026,0000006 2C
000247
000248 77 update-file-status pic xx. BLW=0000+028 2C
000249 77 loccode-file-status pic xx. BLW=0000+030 2C
000250 77 updprint-file-status pic xx. BLW=0000+038 2C
000251
000252 01 flags. BLW=0000+040 0CL3
000253 05 transaction-eof-flag pic x value space. BLW=0000+040,0000000 1C
000254 88 transaction-eof value "Y".
000255 05 location-eof-flag pic x value space. BLW=0000+041,0000001 1C
000256 88 location-eof value "Y".
000257 05 transaction-match-flag pic x. BLW=0000+042,0000002 1C
. . .
000876 procedure division.
000877 000-do-main-logic.
000878 display "PROGRAM IGYTCARA - Beginning"
000879 perform 050-create-vsam-master-file.

```

- (1) Base locator used for this data item
- (2) Hexadecimal displacement from the beginning of the base locator value
- (3) Hexadecimal displacement from the beginning of the containing structure

- (4) Pseudoassembler code showing how the data is defined

## Terms used in MAP output

This table describes the terms used in the listings produced by the MAP option.

Term	Definition	Description
GROUP	DS 0CLn <sup>1</sup>	Fixed-length group data item
ALPHABETIC	DS nC	Alphabetic data item (PICTURE A)
ALPHA-EDIT	DS nC	Alphabetic-edited data item
DISPLAY	DS nC	Alphanumeric data item (PICTURE X)
AN-EDIT	DS nC	Alphanumeric-edited data item
GRP-VARLEN	DS 0LCn <sup>1</sup>	Variable-length group data item
NUM-EDIT	DS nC	Numeric-edited data item
DISP-NUM	DS nC	External decimal data item (USAGE DISPLAY)
BINARY	DS 1H <sup>2</sup> , 1F <sup>2</sup> , 2F <sup>2</sup> , 2C, 4C, or 8C	Binary data item (USAGE BINARY or USAGE COMPUTATIONAL or USAGE COMPUTATIONAL-5)
COMP-1	DS 4C	Single-precision floating-point data item (USAGE COMPUTATIONAL-1)
COMP-2	DS 8C	Double-precision floating-point data item (USAGE COMPUTATIONAL-2)
PACKED-DEC	DS nP	Internal decimal data item (USAGE PACKED-DECIMAL or USAGE COMPUTATIONAL-3)
DBCS	DS nC	DBCS data item (USAGE DISPLAY-1)
DBCS-EDIT	DS nC	DBCS edited data item (USAGE DISPLAY-1)
INDX-NAME		Index name
INDEX		Index data item (USAGE INDEX)
OBJECT-REF		Object-reference data item (USAGE OBJECT-REFERENCE)
POINTER		Pointer data item (USAGE POINTER)
FD		File definition
VSAM, QSAM, LINESEQ		File processing method
1-49, 77		Level numbers for data descriptions
66		Level number for RENAMES
88		Level number for condition-names
SD		Sort file definition
<p>1. <i>n</i> is the size in bytes for fixed-length groups and the maximum size in bytes for variable-length groups.</p> <p>2. If the SYNCHRONIZED clause appears, these fields are used.</p>		

## Symbols used in LIST and MAP output

This table describes the symbols used in the listings produced by the LIST or MAP option.

Symbol	Definition
APBdisp=n <sup>1</sup>	ALL subscript parameter block displacement
AVN=n <sup>1</sup>	Variable name cell for ALTER statement

Symbol	Definition
BL=n <sup>1</sup>	Base locator for special registers
BLA=n <sup>1</sup>	Base locator for alphanumeric temporaries <sup>4</sup>
BLF=n <sup>1</sup>	Base locator for files
BLK=n <sup>1</sup>	Base locator for LOCAL-STORAGE
BLL=n <sup>1</sup>	Base locator for linkage section
BLO=n <sup>1</sup>	Base locator for object instance data
BLS=n <sup>1</sup>	Base locator for sort items
BLV=n <sup>1</sup>	Base locator for variably located data
BLW=n <sup>1</sup>	Base locator for WORKING-STORAGE
BLX=n <sup>1</sup>	Base locator for external data
CBL=n <sup>1</sup>	Base locator for constant global table (CGT)
CLLE=@	Load list entry address in TGT
CLO=n <sup>1</sup>	Class object cell
DOV=n <sup>1</sup>	DSA overflow cell
EVALUATE=n <sup>1</sup>	Evaluate boolean cell
FCB=n <sup>1</sup>	File control block (FCB) address
GN=n(hhhh) <sup>2</sup>	Generated procedure name and its offset in hexadecimal
IDX=n <sup>1</sup>	Base locator for index names
IDX=n <sup>1</sup>	Index cell number
ILS=n <sup>1</sup>	Index cell for LOCAL-STORAGE table or instance variable
ODOSAVE=n <sup>1</sup>	000 save cell number
OPT=nnnn <sup>3</sup>	Optimizer temporary storage cell
PBL=n <sup>1</sup>	Base locator for procedure code
PFM=n <sup>1</sup>	PERFORM n times cells
PGMLIT AT + nnnn <sup>3</sup>	Displacement for program literal from beginning of literal pool
PSV=n <sup>1</sup>	Perform save cell number
PVN=n <sup>1</sup>	Variable name cell for PERFORM statement
RBKST=n <sup>1</sup>	Register backstore cell
SFCB=n <sup>1</sup>	Secondary file control block for external file
SYSLIT AT + nnnn <sup>3</sup>	Displacement for system literal from beginning of system literal pool
TGT FDMP TEST INFO. AREA + nnnn <sup>3</sup>	FDUMP/TEST information area
TGTFIXD + nnnn <sup>3</sup>	Offset from beginning of fixed portion of task global table (TGT)
TOV=n <sup>1</sup>	TGT overflow cell number
TS1=aaaa	Temporary storage cell number in subpool 1
TS2=aaaa	Temporary storage cell number in subpool 2
TS3=aaaa	Temporary storage cell number in subpool 3
TS4=aaaa	Temporary storage cell number in subpool 4
V(routine name)	Assembler VCON for external routine
VLC=n <sup>1</sup>	Variable length name cell number (ODO)
VNI=n <sup>1</sup>	Variable name initialization

Symbol	Definition
WHEN=n <sup>1</sup>	Evaluate WHEN cell number
<ol style="list-style-type: none"> <li>n is the number of the entry.</li> <li>(hhhhh) is the program offset in hexadecimal.</li> <li>nnnn is the offset in decimal from the beginning of the entry.</li> <li>Alphanumeric temporaries are temporary data values used in processing alphanumeric intrinsic function and alphanumeric EVALUATE statement subjects.</li> </ol>	

## Example: nested program map

This example shows a map of nested procedures produced by specifying the MAP compiler option.

```

PP 5648-A25 IBM COBOL for OS/390 and VM 2.2.0          Date 03/30/2000 Time 12:26:53 Page 51
Nested Program Map
Program Attribute codes (rightmost column) have the following meanings:
  C = COMMON
  I = INITIAL (1)
  U = PROCEDURE DIVISION USING...
Source Nesting
LineID Level Program Name from PROGRAM-ID paragraph Program Attributes
  2      0 NESTMAIN. . . . . U
120     1 (4) SUBPRO1 . . . . . I,C,U
(2)199  2   NESTED1 . . . . . I,C,U
253     1   SUBPRO2 . . . . . U
335     2   NESTED2 . . . . . C,U
          (3)

```

- (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

## Reading LIST output

Parts of the LIST compiler output might be useful to you for debugging your program. You do not need to be able to program in assembler language to understand the output produced by LIST. The comments that accompany most of the assembler code provide you with a conceptual understanding of the functions performed by the code.

The LIST compiler option produces seven pieces of output:

- An assembler listing of the initialization code for the program (program signature information bytes) from which you can verify program characteristics such as these:
  - Compiler options in effect
  - Types of data items present
  - Verbs used in the PROCEDURE DIVISION
- An assembler listing of the source code for the program

From the address in storage of the instruction that was executing when an abend occurred, you can find the COBOL verb corresponding to that instruction. After you have found the address of the failing instruction, go to the assembler listing and find the verb for which that instruction was generated.
- Location of compiler-generated tables in the object module

- Map of the task global table (TGT), including information on the program global table (PGT) and constant global table (CGT)

Use the TGT to find information about the environment in which your program is running.

- Information on the location and size of WORKING-STORAGE and control blocks  
You can use the WORKING-STORAGE piece of LIST output to find the location of data items defined in WORKING-STORAGE. (The beginning location of WORKING-STORAGE is not shown for programs compiled with the RENT option.)

- Map of the dynamic save area (DSA)

The map of the DSA (also known as the stack frame) contains information about the contents of the storage acquired every time a separately compiled procedure is entered.

- Information on the location of literals and code for dynamic storage usage

“Example: program initialization code”

“Example: assembler code generated from source code” on page 336

“Example: TGT memory map” on page 338

“Example: location and size of WORKING-STORAGE” on page 339

“Example: DSA memory map” on page 339

#### RELATED TASKS

Interpret a particular instruction (*System/390 Reference Summary*)

#### RELATED REFERENCE

The stack frame section (*Language Environment Debugging Guide and Run-Time Messages*)

### **Example: program initialization code**

A listing of the program initialization code gives you information about the characteristics of the COBOL source program. Interpret the program signature information bytes to verify characteristics of your program.

(1)	(2)	(3)	(4)
000000		IGYTCARA	PROGRAM: IGYTCARA
000000	47F0 F028	DS OH	
000004	00	USING *,15	
000005	C3C5C5	B 40(,15)	BYPASS CONSTANTS. BRANCH TO @STM
000008	000004A8	DC AL1(0)	ZERO NAME LENGTH FOR DUMPS
00000C	00000014	DC CL3'CEE'	CEE EYE CATCHER (5)
000010	47F0 F001	DC X'000004A8'	STACK FRAME SIZE
000014		DC A(@PPA1-IGYTCARA)	OFFSET TO PPA1 FROM PRIMARY ENTRY
000014	98	B 1(,15)	RESERVED
000015	CE	@PPA1 DS OH	PPA1 STARTS HERE
000016	AC	DC X'98'	OFFSET TO LENGTH OF NAME FROM PPA1
000017	00	DC X'CE'	CEL SIGNATURE
000018	000000B6	DC X'AC'	CEL FLAGS: '10101100'B
00001C	00000000	DC X'00'	MEMBER FLAGS FOR COBOL
000020	00000000	DC A(@PPA2)	ADDRESS OF PPA2
000024	00000000	DC F'0'	OFFSET TO THE BDI (NONE)
000028		DC F'0'	ADDRESS OF ENTRY POINT DESCRIPTORS
000028	90EC D00C	DC F'0'	OFFSET FOR STACK OVERFLOW RETURN
00002C	4110 F038	@STM DS OH	STM STARTS HERE
000030	98EF F04C	STM 14,12,12(13)	@STM: SAVE CALLER'S REGISTERS
000034	07FF	LA 1,56(,15)	GET ADDRESS OF PARM LIST INTO R1
000036	0000	LM 14,15,76(15)	LOAD ADDRESSES FROM @BRVAL
000038		BR 15	DO ANY NECESSARY INITIALIZATION
000038	00000000	DC AL2'0'	AVAILABLE HALF-WORD
00003C	00000000	@MAINENT DS OH	PRIMARY ENTRY POINT ADDRESS
000040	000123D0	DC A(IGYTCARA)	@PARMS: 1) PRIMARY ENTRY POINT ADDRESS
000044	000000AE	DC AL4'0'	2) Available
000048	00000000	DC A(DAB)	3) DAB ADDRESS (6)
00004C	00002128	DC A(@EPNAM)	4) ENTRY POINT NAME ADDRESS
000050	00000000	DC A(IGYTCARA)	5) CURRENT ENTRY POINT ADDRESS
000054	000000CA	DC A(START)	@BRVAL: 6) PROCEDURE CODE ADDRESS
000058	00104001	DC V(IGZCBS0)	7) INITIALIZATION ROUTINE
00005C	00000000	DC A(@CEEPARM)	8) ADDRESS OF PARM LIST FOR CEEINT
000060	00000000	DC X'00104001'	DSA WORD 0 CONSTANT
000064	00000000	DC AL4'0'	AVAILABLE WORD
000068	F1F9F9F8	DC AL4'0'	AVAILABLE WORD
00006C	F0F2F2F7	DC CL4'1998'	@TIMEVRS: YEAR OF COMPILATION (8)
000070	F1F2F2F6	DC CL4'0227'	MONTH/DAY OF COMPILATION (9)
000074	F5F3	DC CL4'1226'	HOURS/MINUTES OF COMPILATION (10)
000076	F0F2F0F1F0F1	DC CL2'53'	SECONDS FOR COMPILATION DATE
00007C	00000000	DC CL6'020101'	VERSION/RELEASE/MOD LEVEL OF PROD (11)
000080	0000	DC AL4'0'	AVAILABLE WORD
000082	076C	DC X'0000'	INFO. BYTES 28-29
000084	287A7EDC00A1	DC X'076C'	SIGNED BINARY YEARWINDOW OPTION VALUE
00008A	0080D2A8010B	DC X'287A7EDC00A1'	INFO. BYTES 1-6
000090	8D3060000400	DC X'0080D2A8010B'	INFO. BYTES 7-12 (12)
000096	0000008004	DC X'8D3060000400'	INFO. BYTES 13-18
00009B	00	DC X'0000008004'	INFO. BYTES 19-23
00009C	00000127	DC X'00'	COBOL SIGNATURE LEVEL
0000A0	000001DE	DC X'00000127'	# DATA DIVISION STATEMENTS (13)
0000A4	200480	DC X'000001DE'	# PROCEDURE DIVISION STATEMENTS (14)
0000A7	00	DC X'200480'	INFO. BYTES 24-26
0000AB	40404040	DC X'00'	INFO. BYTE 27
0000AC	0008	DC C'	USER LEVEL INFO (LVLINFO) (15)
0000AE		DC X'0008'	LENGTH OF PROGRAM NAME
0000B6	C9C7E8E3C3C1D9C1	@EPNAM DS OH	ENTRY POINT NAME
0000B6	05	DC C'IGYTCARA'	PROGRAM NAME (7)
0000B7	00	@PPA2 DS OH	PPA2 STARTS HERE
0000B8	00	DC X'05'	CEL MEMBER IDENTIFIER
0000B9	01	DC X'00'	CEL MEMBER SUB-IDENTIFIER
0000BA	00000000	DC X'00'	CEL MEMBER DEFINED BYTE
0000BE	00000000	DC X'01'	CONTROL LEVEL OF PROLOG
0000C2	FFFFFFB2	DC V(CCEESTART)	VCON FOR LOAD MODULE
0000C6	00000000	DC F'0'	OFFSET TO THE CDI (NONE)
0000CA		DC A(@TIMEVRS-@PPA2)	OFFSET TO TIMESTAMP/VERSION INFO
0000CA	00000038	DC A(IGYTCARA)	ADDRESS OF CU PRIMARY ENTRY POINT
0000CE	00000008	@CEEPARM DS OH	PARM LIST FOR CEEINT
0000D2		DC A(@MAINENT)	POINTER TO PRIMARY ENTRY PT ADDR
0000D2	00000006	DC A(@PARMCCE-@CEEPARM)	OFFSET TO PARAMETERS FOR CEEINT
0000D6	00000038	@PARMCCE DS OH	PARAMETERS FOR CEEINT
0000DA	00000000	DC F'6'	1) NUMBER OF ENTRIES IN PARM LIST
0000DE	00000000	DC A(@MAINENT)	2) POINTER TO PRIMARY ENTRY PT ADDR
0000E2	00000000	DC V(CCEESTART)	3) ADDRESS OF CCEESTART
0000E6	00000000	DC V(CCEEBETBL)	4) ADDRESS OF CCEEBETBL
0000EA	00000000	DC F'5'	5) CEL MEMBER IDENTIFIER
0000EE	00000000	DC F'0'	6) FOR CEL MEMBER USE
0000F2	00000000	DC AL4'0'	AVAILABLE WORD
0000F6	00000000	DC AL4'0'	AVAILABLE WORD
0000FA	0000	DC AL4'0'	AVAILABLE WORD
		DC AL2'0'	AVAILABLE HALF-WORD

- (1) Offset from the start of the COBOL program.
- (2) Hexadecimal representation of assembler instructions.



- (3) Pseudoassembler code generated for the COBOL program.
- (4) Comments explaining the assembler code.
- (5) "Eye catcher" indicating the COBOL compiler used to compile this program.
- (6) Address of the task global table (TGT), or the address of the dynamic access block (DAB), if the program is reentrant.
- (7) Program name as used in the IDENTIFICATION DIVISION of the program.
- (8) Four-digit year when the program was compiled.
- (9) Month and the day when the program was compiled.
- (10) Time when the program was compiled.
- (11) Version, release, and modification level of the COBOL compiler used to compile this program (each represented in two digits).
- (12) Program signature information bytes. These provide information about the following for this program:
  - Compiler options
  - DATA DIVISION
  - ENVIRONMENT DIVISION
  - PROCEDURE DIVISION
- (13) Number of statements in the DATA DIVISION.
- (14) Number of statements in the PROCEDURE DIVISION.
- (15) 4-byte user-controlled level information field. The value of this field is controlled by the LVLINFO.

**RELATED REFERENCES**

"Signature information bytes: compiler options"

"Signature information bytes: DATA DIVISION" on page 333

"Signature information bytes: ENVIRONMENT DIVISION" on page 333

"Signature information bytes: PROCEDURE DIVISION verbs" on page 333

"Signature information bytes: more PROCEDURE DIVISION items" on page 335

**Signature information bytes: compiler options**

This table shows program signature information that is part of the listing of program initialization code provided when you use the LIST compiler option.

Byte	Bit	On	Off
1	0	ADV	NOADV
	1	APOST	QUOTE
	2	DATA(31)	DATA(24)
	3	DECK	NODECK
	4	DUMP	NODUMP
	5	DYNAM	NODYNAM
	6	FASTSRT	NOFASTSRT
	7	reserved	

Byte	Bit	On	Off
2	0	LIB	NOLIB
	1	LIST	NOLIST
	2	MAP	NOMAP
	3	NUM	NONUM
	4	OBJ	NOOBJ
	5	OFFSET	NOOFFSET
	6	OPTIMIZE	NOOPTIMIZE
	7	DDNAME supplied in OUTDD option will be used	Default DDNAME for OUTDD will be used
3	0	NUMPROC(PFD)	NUMPROC(NOPFD)
	1	RENT	NORENT
	2	reserved	
	3	SEQUENCE	NOSEQUENCE
	4	SIZE(MAX)	SIZE(value)
	5	SOURCE	NOSOURCE
	6	SSRANGE	NOSSRANGE
	7	TERM	NOTERM
4	0	TEST	NOTEST
	1	TRUNC(STD)	TRUNC(OPT)
	2	User-Supplied Reserved Word List	Installation Default Reserved Word List
	3	VBREF	NOVBREF
	4	XREF	NOXREF
	5	ZWB	NOZWB
	6	NAME	NONAME
	7	CMPR2	NOCMPR2
5	0	NUMPROC(MIG)	
	1	NUMCLS(ALT)	NUMCLS(PRIM)
	2	DBCS	NODBCS
	3	AWO	NOAWO
	4	TRUNC(BIN)	not TRUNC(BIN)
	6	CURRENCY	NOCURRENCY
	7	Compilation unit is a CLASS	Compilation unit is a program
26	0	RMODE(ANY)	RMODE(24)
	1	TEST(STMT)	not TEST(STMT)
	2	TEST(PATH)	not TEST(PATH)
	3	TEST(BLOCK)	not TEST(BLOCK)
	4	OPT(FULL)	OPT(STD) or NOOPT
	5	INTDATE(LILIAN)	INTDATE(ANSI)
	6	TEST(SEPARATE)	not TEST(SEPARATE)

Byte	Bit	On	Off
27	0	PGMNAME(LONGUPPER)	not PGMNAME(LONGUPPER)
	1	PGMNAME(LONGMIXED)	not PGMNAME(LONGMIXED)
	2	DLL	NODLL
	3	EXPORTALL	NOEXPORTALL
	4	DATEPROC	NODATEPROC
	5	ARITH(EXTEND)	ARITH(COMPAT)

### Signature information bytes: DATA DIVISION

This table shows program signature information that is part of the listing of program initialization code provided when you use the LIST compiler option.

Byte	Bit	Item
6	0	QSAM file descriptor
	1	VSAM sequential file descriptor
	2	VSAM indexed file descriptor
	3	VSAM relative file descriptor
	4	CODE-SET clause (ASCII files) in file descriptor
	5	Spanned records
	6	PIC G or PIC N (DBCS data item)
	7	OCCURS DEPENDING ON clause in data description entry
7	0	SYNCHRONIZED clause in data description entry
	1	JUSTIFIED clause in data description entry
	2	USAGE IS POINTER item
	3	Complex OCCURS DEPENDING ON clause
	4	External floating-point items in the DATA DIVISION
	5	Internal floating-point items in the DATA DIVISION
	6	Line-sequential file
	7	USAGE IS PROCEDURE-POINTER item

#### RELATED REFERENCES

“LIST” on page 281

### Signature information bytes: ENVIRONMENT DIVISION

This table shows program signature information that is part of the listing of program initialization code provided when you use the LIST compiler option.

Byte	Bit	Item
8	0	FILE STATUS clause in FILE-CONTROL paragraph
	1	RERUN clause in I-O-CONTROL paragraph of INPUT-OUTPUT SECTION
	2	UPS I switch defined in SPECIAL-NAMES paragraph

### Signature information bytes: PROCEDURE DIVISION verbs

This table shows program signature information that is part of the listing of program initialization code provided when you use the LIST compiler option.

Byte	Bit	Item
9	0	ACCEPT
	1	ADD
	2	ALTER
	3	CALL
	4	CANCEL
	6	CLOSE
10	0	COMPUTE
	2	DELETE
	4	DISPLAY
	5	DIVIDE
11	1	END-PERFORM
	2	ENTER
	3	ENTRY
	4	EXIT
	6	GO TO
	7	IF
12	0	INITIALIZE
	1	INVOKE
	2	INSPECT
	3	MERGE
	4	MOVE
	5	MULTIPLY
	6	OPEN
	7	PERFORM
13	0	READ
	2	RELEASE
	3	RETURN
	4	REWRITE
	5	SEARCH
	7	SET
14	0	SORT
	1	START
	2	STOP
	3	STRING
	4	SUBTRACT
	7	UNSTRING

Byte	Bit	Item
15	0	USE
	1	WRITE
	2	CONTINUE
	3	END-ADD
	4	END-CALL
	5	END-COMPUTE
	6	END-DELETE
	7	END-DIVIDE
16	0	END-EVALUATE
	1	END-IF
	2	END-MULTIPLY
	3	END-READ
	4	END-RETURN
	5	END-REWRITE
	6	END-SEARCH
	7	END-START
17	0	END-STRING
	1	END-SUBTRACT
	2	END-UNSTRING
	3	END-WRITE
	4	GOBACK
	5	EVALUATE
	7	SERVICE statement
18	0	END-INVOKE

**Check return code:** A return code greater than 4 from the compiler could mean that some of the verbs shown as being in the program in information bytes might have been discarded because of an error.

### Signature information bytes: more PROCEDURE DIVISION items

This table shows program signature information that is part of the listing of program initialization code provided when you use the LIST compiler option.

Byte	Bit	Item
21	0	Hexadecimal literal
	1	Altered GO TO
	2	I-O ERROR declarative
	3	LABEL declarative
	4	DEBUGGING declarative
	5	Program segmentation
	6	OPEN. . .EXTEND
	7	EXIT PROGRAM

Byte	Bit	Item
22	0	CALL literal
	1	CALL identifier
	2	CALL. . .ON OVERFLOW
	3	CALL. . .LENGTH OF
	4	CALL. . .ADDRESS OF
	5	CLOSE. . .REEL/UNIT
	6	Exponentiation used
	7	Floating-point items used
23	0	COPY
	1	BASIS
	2	DBCS name in program
	3	SHIFT-OUT and SHIFT-IN in program
	4-7	Highest error severity at entry to ASM2 module IGYBINIT
24	0	DBCS literal
	1	REPLACE
	2	Reference modification was used
	3	Nested program
	4	INITIAL
	5	COMMON
	6	SELECT . . . OPTIONAL
	7	EXTERNAL
25	0	GLOBAL
	1	RECORD IS VARYING
	2	ACCEPT FROM SYSIPT used in LABEL declarative
	3	DISPLAY UPON SYSLST used in LABEL declarative
	4	DISPLAY UPON SYSPCH used in LABEL declarative
	5	Intrinsic function was used

RELATED REFERENCES  
 "LIST" on page 281

**Example: assembler code generated from source code**

The following example shows a listing of the assembler code that is generated from source code when you use the LIST compiler option. You can use this listing to find the COBOL verb corresponding to the instruction that failed.

000433 MOVE  
 000435 READ  
 000436 SET (1)

(2)	(3)	(5)	(6)
000F26	92E8 A00A	MVI 10(10),X'E8'	LOCATION-EOF-FLAG
000F2A		EQU *	
000F2A	47F0 B426	GN=13 BC 15,1062(0,11)	GN=75(000EFA)
000F2E		GN=74 EQU *	
000439	IF		
000F2E	95E8 A00A	CLI 10(10),X'E8'	LOCATION-EOF-FLAG
000F32	4780 B490	BC 8,1168(0,11)	GN=14(000F64)
000440	DISPLAY		
000F36	5820 D05C	L 2,92(0,13)	TGTFIXD+92
000F3A	58F0 202C	L 15,44(0,2)	V(IGZCDSP )
000F3E	4110 97FF	LA 1,2047(0,9)	PGMLIT AT +1999
000F42	05EF	BALR 14,15	
000443	CALL		
000F44	4130 A012	LA 3,18(0,10)	COMP-CODE
000F48	5030 D21C	ST 3,540(0,13)	TS2=4
000F4C	9680 D21C	OI 540(13),X'80'	TS2=4
000F50	4110 D21C	LA 1,540(0,13)	TS2=4
000F54	58F0 9000	L 15,0(0,9)	V(ILBOABN0)
000F58	05EF	BALR 14,15	
000F5A	50F0 D078	ST 15,120(0,13)	TGTFIXD+120
000F5E	BF38 D089	ICM 3,8,137(13)	TGTFIXD+137
000F62	0430	SPM 3,0	
000F64	(4)	GN=14 EQU *	
000F64	5820 D154	L 2,340(0,13)	VN=3
000F68	07F2	BCR 15,2	

- (1) Source line number and COBOL verb, paragraph name, or section name  
 In line 000436, SET is the COBOL verb. An asterisk (\*) before a name indicates that the name is a paragraph name or a section name.
- (2) Relative location of the object code instruction in the module, in hexadecimal notation
- (3) Object code instruction, in hexadecimal notation  
 The first two or four hexadecimal digits are the instruction, and the remaining digits are the instruction operands. Some instructions have two operands.
- (4) Compiler-generated names (GN) for code sequences
- (5) Object code instruction in a form that closely resembles assembler language
- (6) Comments about the object code instruction:
  - One or two operands participating in the machine instructions are displayed on the right. An asterisk immediately follows the data names that are defined in more than one structure (in that way made unique by qualification in the source program).
  - The relative location of any generated label appearing as an operand is displayed in parentheses.

RELATED REFERENCES

“Symbols used in LIST and MAP output” on page 326

## Example: TGT memory map

The following example shows LIST output for the task global table (TGT) with information about the environment in which your program runs.

DATA VALIDATION AND UPDATE PROGRAM IGYTCARA Date 03/30/2000 Time 12:26:53 Page 132

```
*** TGT MEMORY MAP ***
(1)      (2)      (3)
PGMLOC  TGTLOC

005160  000000  RESERVED - 72 BYTES
0051A8  000048  TGT IDENTIFIER
0051AC  00004C  RESERVED - 4 BYTES
0051B0  000050  TGT LEVEL INDICATOR
0051B1  000051  RESERVED - 3 SINGLE BYTE FIELDS
0051B4  000054  32 BIT SWITCH
0051B8  000058  POINTER TO RUNCOM
0051BC  00005C  POINTER TO COBVEC
0051C0  000060  POINTER TO PROGRAM DYNAMIC BLOCK TABLE
0051C4  000064  NUMBER OF FCB'S
0051C8  000068  WORKING-STORAGE LENGTH
0051CC  00006C  RESERVED - 4 BYTES
0051D0  000070  ADDRESS OF IGZESMG WORK AREA
0051D4  000074  ADDRESS OF 1ST GETMAIN BLOCK (SPACE MGR)
0051D8  000078  RESERVED - 2 BYTES
0051DA  00007A  RESERVED - 2 BYTES
0051DC  00007C  RESERVED - 2 BYTES
0051DE  00007E  MERGE FILE NUMBER
0051E0  000080  ADDRESS OF CEL COMMON ANCHOR AREA
0051E4  000084  LENGTH OF TGT
0051E8  000088  RESERVED - 1 SINGLE BYTE FIELD
0051E9  000089  PROGRAM MASK USED BY THIS PROGRAM
0051EA  00008A  RESERVED - 2 SINGLE BYTE FIELDS
0051EC  00008C  NUMBER OF SECONDARY FCB CELLS
0051F0  000090  LENGTH OF THE ALTER VN(VNI) VECTOR
0051F4  000094  COUNT OF NESTED PROGRAMS IN COMPILE UNIT
0051F8  000098  DDNAME FOR DISPLAY OUTPUT
005200  0000A0  RESERVED - 8 BYTES
005208  0000A8  POINTER TO COM-REG SPECIAL REGISTER
00520C  0000AC  CALC ROUTINE REGISTER SAVE AREA
005240  0000E0  ALTERNATE COLLATING SEQUENCE TABLE PTR.
005244  0000E4  ADDRESS OF SORT G.N. ADDRESS BLOCK
005248  0000E8  ADDRESS OF PGT
00524C  0000EC  CURRENT INTERNAL PROGRAM NUMBER
005250  0000F0  POINTER TO 1ST IPCB
005254  0000F4  ADDRESS OF THE CLLE FOR THIS PROGRAM
005258  0000F8  POINTER TO ABEND INFORMATION TABLE
00525C  0000FC  POINTER TO TEST INFO FIELDS IN THE TGT
005260  000100  ADDRESS OF START OF COBOL PROGRAM
005264  000104  POINTER TO ALTER VNI'S IN CGT
005268  000108  POINTER TO ALTER VN'S IN TGT
00526C  00010C  POINTER TO FIRST PBL IN THE PGT
005270  000110  POINTER TO FIRST FCB CELL
005274  000114  WORKING-STORAGE ADDRESS
005278  000118  POINTER TO FIRST SECONDARY FCB CELL
00527C  00011C  POINTER TO STATIC CLASS INFO BLOCK

*** VARIABLE PORTION OF TGT ***

005280  000120  BASE LOCATORS FOR SPECIAL REGISTERS
005288  000128  BASE LOCATORS FOR WORKING-STORAGE (4)
005290  000130  BASE LOCATORS FOR LINKAGE-SECTION
005294  000134  BASE LOCATORS FOR FILES
0052A8  000148  BASE LOCATORS FOR ALPHANUMERIC TEMPS
0052AC  00014C  CLLE ADDR. CELLS FOR CALL LIT. SUB-PGMS.
0052C8  000168  INDEX CELLS
```



```

0052EC 00018C FCB CELLS
005300 0001A0 ALL PARAMETER BLOCK
005364 000204 INTERNAL PROGRAM CONTROL BLOCKS

```

- (1) Hexadecimal offset of the TGT field from the start of the COBOL program (not shown for programs compiled with the RENT option)
- (2) Hexadecimal offset of the TGT field from the start of the TGT
- (3) Explanation of the contents of the TGT field
- (4) TGT fields for the base locators of COBOL data areas

### Example: location and size of WORKING-STORAGE

The following example shows LIST output about the WORKING-STORAGE for a NORENT program.

```

(1)          (2)          (3)
WRK-STOR LOCATED AT 0066D8 FOR 00001598 BYTES
(1) WORKING-STORAGE identification
(2) Hexadecimal offset of WORKING-STORAGE from the start of the COBOL
    program
(3) Length of WORKING-STORAGE in hexadecimal

```

### Example: DSA memory map

The following example shows LIST output for the dynamic save area (DSA), which contains information about the contents of the storage acquired when a separately compiled procedure is entered.

```

DATA VALIDATION AND UPDATE PROGRAM   IGYTCARA   Date 03/30/2000 Time 12:26:53 Page 139
*** DSA MEMORY MAP ***
(1)  (2)
DSALOC

000000 REGISTER SAVE AREA
00004C STACK NAB (NEXT AVAILABLE BYTE)
00005C ADDRESS OF TGT
000058 ADDRESS OF INLINE-CODE PRIMARY DSA
000080 PROCEDURE DIVISION RETURNING VALUE

*** VARIABLE PORTION OF DSA ***
000084 BACKSTORE CELLS FOR SYMBOLIC REGISTERS
00009C VARIABLE-LENGTH CELLS
0000A8 ODO SAVE CELLS
0000B4 VARIABLE NAME (VN) CELLS FOR PERFORM
000124 PERFORM SAVE CELLS
000250 TEMPORARY STORAGE-1
000260 TEMPORARY STORAGE-2
000430 OPTIMIZER TEMPORARY STORAGE

```

- (1) Hexadecimal offset of the dynamic save area (DSA) field from the start of the DSA
- (2) Explanation of the contents of the DSA field

## 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.

PP 5648-A25 IBM COBOL for OS/390 and VM 2.2.0 Date 03/30/2000 Time 12:26:53 Page 4

(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 204 340
000884      at end
000885      1 set transaction-eof to true                        254
000886      end-read.
. . .
000984      100-initialize-paragraph.
000985      move spaces to ws-transaction-record                IMP 340 (2)
000986      move spaces to ws-commuter-record                IMP 316
000987      move zeroes to commuter-zipcode                 IMP 327
000988      move zeroes to commuter-home-phone              IMP 328
000989      move zeroes to commuter-work-phone              IMP 329
000990      move zeroes to commuter-update-date            IMP 333
000991      open input update-transaction-file             204
000992      location-file                                  193
000993      i-o commuter-file                              181
000994      output print-file                                217
. . .
001442      1100-print-i-f-headings.
001443
001444      open output print-file.                            217
001445
001446      move function when-compiled to when-comp.          IFN 698 (2)
001447      move when-comp (5:2) to compile-month.          698 640
001448      move when-comp (7:2) to compile-day.            698 642
001449      move when-comp (3:2) to compile-year.          698 644
001450
001451      move function current-date (5:2) to current-month. IFN 649
001452      move function current-date (7:2) to current-day. IFN 651
001453      move function current-date (3:2) to current-year. IFN 653
001454
001455      write print-record from i-f-header-line-1          222 635
001456      after new-page.                                   138
. . .

```

- (1) Line number of the definition of the data name or procedure name in the program
- (2) Special definition symbols:  
**UND** The user name is undefined.

- DUP** The user name is defined more than once.
- IMP** Implicitly defined name, such as special registers and figurative constants
- IFN** Intrinsic function reference
- EXT** External reference
- \*** The program name is unresolved because the NOCOMPILE option is in effect.

## Example: OFFSET compiler output

The following example shows a listing that has a condensed verb listing, global tables, WORKING-STORAGE information, and literals. It is output from the OFFSET compiler option.

DATA VALIDATION AND UPDATE PROGRAM IGYTCARA Date 03/30/2000 Time 12:26:53 Page 54

(1)	(2)	(3)						
LINE #	HEXLOC	VERB	LINE #	HEXLOC	VERB	LINE #	HEXLOC	VERB
000880	0026F0	DISPLAY	000881	002702	PERFORM	000933	002702	OPEN
000934	002722	IF	000935	00272C	DISPLAY	000936	002736	PERFORM
001389	002736	DISPLAY	001390	002740	DISPLAY	001391	00274A	DISPLAY
001392	002754	DISPLAY	001393	00275E	DISPLAY	001394	002768	DISPLAY
001395	002772	DISPLAY	000937	00277C	PERFORM	001434	00277C	DISPLAY
001435	002786	STOP	000939	0027A2	MOVE	000940	0027AC	WRITE
000941	0027D6	IF	000942	0027E0	DISPLAY	000943	0027EA	PERFORM
001389	0027EA	DISPLAY	001390	0027F4	DISPLAY	001391	0027FE	DISPLAY
001392	002808	DISPLAY	001393	002812	DISPLAY	001394	00281C	DISPLAY
001395	002826	DISPLAY	000944	002830	DISPLAY	000945	00283A	PERFORM
001403	00283A	DISPLAY	001404	002844	DISPLAY	001405	00284E	DISPLAY
001406	002858	DISPLAY	001407	002862	CALL	000947	002888	CLOSE

- (1) Line number. Your line numbers or compiler-generated line numbers are listed.
- (2) Offset, from the start of the program, of the code generated for this verb (in hexadecimal notation).  
  
The verbs are listed in the order in which they occur and once for each time they are used.
- (3) Verb used.

RELATED REFERENCES  
"OFFSET" on page 285

## 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
		140 143 148 148 149 149 149 152 152 153 162
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
		107 107 108 108 108 108 108 108 109 110 111 112 113 113 114
		114 114 115 115 116 116 117 117 117 118 118 118 119 119 120 121
		121 121 121 121 121 121 121 121 121 122 122 122 122 123 123
		123 123 123 123 123 124 124 124 125 125 125 125 125 125 126
		126 126 126 126 127 127 127 127 128 128 129 129 130 130 130
		131 131 131 131 131 132 132 132 132 132 132 133 133 133 133
		134 134 134 134 134 135 135 135 135 135 135 136 136 137 137
		137 137 138 138 138 138 141 141 142 142 144 144 144 145 145
		145 145 146 149 150 150 150 151 151 155 156 156 157 158 158
		159 159 160 160 161 161 162 162 162 168 168 168 169 170 171
		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 97 100 100 101 102 104 109 109 111 116 117 117
		117 118 118 118 118 119 119 119 120 120 124 125 127 128 133 134
		135 136 136 137 150 151 151 153 153
8	READ . . . . .	88 89 96 101 102 108 149 151
1	REWRITE . . . . .	118
4	SEARCH . . . . .	106 106 141 142
46	SET . . . . .	88 89 101 103 104 105 106 108 108 136 141 142 149 150 151 152 154
		155 156 156 156 156 157 157 157 157 158 158 158 158 159 159 159
		159 160 160 160 160 161 161 161 162 162 164 164
2	STOP . . . . .	92 143
4	STRING . . . . .	123 126 132 134
33	WRITE . . . . .	94 116 129 129 129 129 129 130 130 130 130 145 146 146 146 147
		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

---

## Preparing to use the debugger

Use the TEST option to prepare your executable COBOL program for use with the debugger.

The Debug Tool debugger is provided with the Full Function feature of the COBOL for OS/390 & VM compiler. To use Debug Tool to step through a run of your program, specify the TEST compiler option. For remote debugging, the Distributed Debugger provides the client graphical user interface to the debug information provided by the Debug Tool engine running on OS/390 or OS/390 UNIX.

You can specify the TEST suboption SEPARATE to have the symbolic information tables for the Debug Tool generated in a data set separate from your object module. And you can enable your COBOL program for debugging using overlay hooks (*production debugging*), rather than compiled-in hooks, which have some performance degradation even when the run-time TEST option is off.

### RELATED TASKS

*Debug Tool User's Guide and Reference*

RELATED REFERENCES  
"TEST" on page 295

---

## Part 3. Targeting COBOL programs for certain environments

<b>Chapter 18. Developing COBOL programs for CICS</b> . . . . .	347
Coding COBOL programs to run under CICS . . . . .	347
Coding file input and output . . . . .	348
Retrieving the system date and time. . . . .	348
Displaying the contents of data items . . . . .	348
Calling to or from COBOL programs . . . . .	348
Coding a COBOL program to run above the 16-MB line . . . . .	349
Determining the success of ECI calls. . . . .	349
Preparing COBOL programs to run under CICS . . . . .	350
Using the CICS translator . . . . .	350
Compiling your CICS program . . . . .	350
CICS reserved-word table . . . . .	351
Handling errors by using CICS HANDLE . . . . .	352
Example: handling errors by using CICS HANDLE . . . . .	352
<b>Chapter 19. Programming for a DB2 environment</b> . . . . .	355
Coding SQL statements . . . . .	355
Using SQL INCLUDE . . . . .	355
Using binary items . . . . .	355
Determining the success of SQL statements . . . . .	356
Compiling with the SQL option . . . . .	356
Compiling in batch . . . . .	357
Separating DB2 suboptions. . . . .	357
DB2 coprocessor . . . . .	358
<b>Chapter 20. Running COBOL programs under IMS</b> . . . . .	359
Compiling and linking COBOL programs for running under IMS . . . . .	359
<b>Chapter 21. Running COBOL programs under OS/390 UNIX</b> . . . . .	361
Running in OS/390 UNIX environments . . . . .	361
Setting and accessing environment variables . . . . .	362
Setting environment variables that affect execution. . . . .	362
Environment variables of interest for COBOL programs. . . . .	363
Resetting environment variables . . . . .	363
Accessing environment variables . . . . .	363
Example: accessing environment variables. . . . .	363
Calling UNIX/POSIX APIs . . . . .	364
fork, exec, and spawn . . . . .	364
Samples . . . . .	365
Accessing main program parameters . . . . .	365
Example: accessing main program parameters . . . . .	366
<b>Chapter 22. Running COBOL programs under CMS</b> . . . . .	369
Run-time restrictions under CMS. . . . .	369
Handling QSAM files under CMS . . . . .	369
Run-time message IGZ0002S . . . . .	370
<b>Chapter 23. Accessing COBOL programs interactively with ISPF.</b> . . . . .	371





---

## Chapter 18. Developing COBOL programs for CICS

COBOL for OS/390 & VM programs written for CICS can run under CICS/ESA and CICS Transaction Server. You must use the CICS command level interface to write CICS COBOL application programs. COBOL programs cannot use the CICS macro level interface.

The CICS translator converts the EXEC CICS commands in your source program to COBOL code. The translator replaces each EXEC CICS command with one or more COBOL statements, one of which is a CALL statement.

After compiling and link-editing your program, you need to do some other steps such as updating CICS tables before you can run the COBOL program under CICS. However, these CICS topics are beyond the focus of this COBOL information.

### RELATED TASKS

“Coding COBOL programs to run under CICS”

“Preparing COBOL programs to run under CICS” on page 350

“Handling errors by using CICS HANDLE” on page 352

---

## Coding COBOL programs to run under CICS

To code your program, you need to know how to code CICS commands in the PROCEDURE DIVISION. They have the following basic format:

```
EXEC CICS command name and command options  
END-EXEC
```

Observe these guidelines when coding your COBOL programs to run under CICS:

- Do not use EXEC, CICS, or END-EXEC for variable names.
- Do not use the FILE-CONTROL entry in the ENVIRONMENT DIVISION, unless the FILE-CONTROL entry is being used for a SORT statement.
- Do not use the FILE SECTION of the DATA DIVISION, unless the FILE SECTION is being used for a SORT statement.
- Do not use user-specified parameters to the main program.
- Do not use USE declaratives (except USE FOR DEBUGGING).
- Do not use these COBOL language statements:

```
ACCEPT format 1: data transfer (you can use format-2 ACCEPT to retrieve the  
system date and time)  
CLOSE  
DELETE  
DISPLAY UPON SYSPUNCH  
DISPLAY UPON CONSOLE  
MERGE  
OPEN  
READ  
RERUN  
REWRITE  
START  
STOP literal  
WRITE
```

- REPLACE statements that contain EXEC commands must occur after the PROCEDURE DIVISION statement of the program for the EXEC commands to be translated.
- When coding nested (contained) programs, pass DFHEIBLK and DFHCOMMAREA as parameters to any nested programs that contain EXEC commands or references to the EIB (EXEC interface block). The same parameters must also be passed to any program that forms part of the control hierarchy between such a program and its top level program.
- The space character is not interchangeable with a comma or semicolon within EXEC commands. In such commands, use the space as a word separator.
- You cannot use EXEC CICS statements within object-oriented COBOL class definitions. However, COBOL methods can make calls to COBOL subprograms that contain EXEC CICS statements. Also, COBOL programs containing EXEC CICS statements can use COBOL INVOKE statements to invoke methods.

## Coding file input and output

You must use CICS commands for most input and output processing. Therefore, do not describe files or code any OPEN, CLOSE, READ, START, REWRITE, WRITE, or DELETE statements. Instead, use CICS commands to retrieve, update, insert, and delete data.

## Retrieving the system date and time

You cannot not use a format-1 ACCEPT statement in a CICS program.

You can use these format-2 ACCEPT statements in the CICS environment to get the system date:

- ACCEPT *identifier-2* FROM DATE
- ACCEPT *identifier-2* FROM DATE YYYYMMDD
- ACCEPT *identifier-2* FROM DAY
- ACCEPT *identifier-2* FROM DAY YYYYDDD
- ACCEPT *identifier-2* FROM DAY-OF-WEEK

You can use this format-2 ACCEPT statement in the CICS environment to get the system time:

- ACCEPT *identifier-2* FROM TIME

The recommended way of retrieving system date and time information is using the ACCEPT statement, because it works in all environments (CICS and non-CICS).

## Displaying the contents of data items

DISPLAY to the system logical output device (SYSOUT, SYSLIST, SYSLST) is supported under CICS. The DISPLAY output is written to the Language Environment message file (transient data queue CESE).

DISPLAY . . . UPON CONSOLE and DISPLAY . . . UPON SYSPUNCH, however, are disallowed.

## Calling to or from COBOL programs

You can make calls to or from VS COBOL II, COBOL for MVS & VM, and COBOL for OS/390 & VM programs using the CALL statement. However, COBOL for OS/390 & VM, COBOL for MVS & VM, and VS COBOL II programs cannot call or be called by OS/VS COBOL programs with the CALL statement. You must use EXEC

CICS LINK instead. If you are calling a COBOL program that has been translated, you must pass DFHEIBLK and DFHCOMMAREA as the first two parameters in the CALL statement.

You can use CALL identifier with the NODYNAM compiler option to dynamically call a program. Called programs can contain any function supported by CICS for the language. You must define dynamically called programs in the CICS program processing table (PPT), if you are not using CICS autoinstall.

The ON OVERFLOW phrase and ON EXCEPTION phrase of the CALL statement are supported under CICS unless the program has been compiled with the CMPR2 compiler option. In that case, no conditions under CICS will cause the statement specified by the ON OVERFLOW to be performed.

Support for interlanguage communication (ILC) with other HLL languages is available. Where ILC is not supported, you can use CICS LINK, XCTL, and RETURN instead.

Assembler-language programs that conform to the interface described in the *Language Environment Programming Guide* are called *Language Environment-conforming* assembler programs. Those that do not conform to the interface are *non-Language Environment-conforming* assembler programs. The following table shows the calling relationship between COBOL and assembler-language programs.

Calls between COBOL and assembler programs	Language Environment-conforming assembler program	Non-Language Environment-conforming assembler program
From a COBOL for OS/390 & VM program to the assembler program?	Yes	Yes
From the assembler program to a COBOL for OS/390 & VM program?	Yes	No

## Coding a COBOL program to run above the 16-MB line

Under OS/390, these restrictions apply when you code a COBOL program to run above the 16-MB line:

- If you are using IMS/ESA Version 3 (or later) without DBCTL, DL/I CALL statements are supported only if all the data passed on the call resides below the 16-MB line. Therefore, you must specify the DATA(24) compiler option. However, if you are using IMS/ESA Version 3 (or later) with DBCTL, you can use the DATA(31) compiler option instead and pass data that resides above the 16-MB line.  
If you use EXEC DLI instead of DL/I CALL statements, you can specify DATA(31), regardless of the IMS product level.
- If the receiving program is link-edited with AMODE=31, addresses passed must be 31 bits long, or 24 bits long with the leftmost byte set to zeros.
- If the receiving program is link-edited with AMODE=24, addresses passed must be 24 bits long.

## Determining the success of ECI calls

The external CICS interface (ECI) does not clear register 15 at termination, regardless of whether your COBOL program executes normally or not. Therefore,

even if your COBOL program terminates normally after successfully using the external CICS interface, the job step could end with an undefined return code.

To ensure that a meaningful return code is given at termination, set the job step return code before terminating your program. Keep the COBOL convention of storing its RETURN-CODE special register into register 15 by saving the response code from the last call to the external CICS interface and moving it to special register RETURN-CODE before terminating.

#### RELATED TASKS

Writing ILC applications (*Language Environment Writing ILC Applications*)

#### RELATED REFERENCES

*CICS/ESA V4R1 External CICS Interface, SC33-1390*

---

## Preparing COBOL programs to run under CICS

To prepare your COBOL program to run under CICS you must use the CICS translator to convert the CICS commands to COBOL statements and compile and link the program to create the executable module.

### Using the CICS translator

To run a COBOL program that includes CICS commands, you must use the CICS translator to convert the EXEC CICS commands into COBOL code. When you translate COBOL for OS/390 & VM programs, use the COBOL3 translator option.

When you use the COBOL3 translator option, it causes the following line to be inserted:

```
CBL RENT,NODYNAM,LTB
```

You can suppress the insertion of a CBL statement by using the CICS translator option NOCBLCARD.

CICS features the translator option ANSI85, which supports these language features (introduced by the COBOL 85 Standard):

- Blank lines intervening in literals
- Sequence numbers containing any character
- Lowercase characters supported in all COBOL words
- REPLACE statement
- Batch compilation
- Nested programs
- Reference modification
- GLOBAL variables
- Interchangeability of comma, semicolon, and space
- Symbolic character definition

### Compiling your CICS program

To run a COBOL program under CICS, you must use the following compiler options when you compile the program:

Required compiler option	Condition
RENT	

Required compiler option	Condition
NODYNAM	The program is translated by the CICS translator.
LIB	The program contains a COPY or BASIS statement.

In addition, you might want to use the following recommended options under certain circumstances:

Recommended compiler option	Condition
TRUNC(OPT)	All binary data items conform to the PICTURE and USAGE clause for those data items.
TRUNC(BIN)	Not all binary data items conform to the PICTURE and USAGE clause for those data items.
WORD(CICS)	You want the COBOL language elements not supported under CICS to be flagged at compile time.

For example, if you have a data item defined as PIC S9(8) BINARY that might receive a value greater than eight digits, use TRUNC(BIN) or change the PICTURE clause.

You might also want to avoid using an option that has no effect:

ADV  
FASTSORT  
OUTDD

The input data set for the compiler is the data set that you received as result of translation, which is SYSPUNCH by default.

## CICS reserved-word table

COBOL provides an alternate reserved-word table (IGYCCICS) for CICS application programs. If you use the compiler option WORD(CICS), COBOL words not supported under CICS are flagged by the compiler with an error message.

In addition to the COBOL words restricted by the IBM-supplied default reserved-word table, the IBM-supplied CICS reserved-word table restricts the following COBOL words:

CLOSE	I-O-CONTROL	REWRITE
DELETE	MERGE	<i>SD</i>
FD	OPEN	<i>SORT</i>
<i>FILE</i>	READ	START
<i>FILE-CONTROL</i>	RERUN	WRITE
<i>INPUT-OUTPUT</i>		

If you intend to use the SORT statement under CICS (COBOL supports an interface for the SORT statement under CICS), you must change the CICS reserved-word table before using it. You must remove the words *italicized* above from the list of words marked as restricted, because they are required for the SORT function.

### RELATED TASKS

“Preparing COBOL programs to run under CICS” on page 350

---

## Handling errors by using CICS HANDLE

The setting of the CBLPSHPOP run-time option affects the state of the HANDLE specifications when a program calls a COBOL subprogram.

When CBLPSHPOP is ON and a COBOL subprogram (not a nested program) is called, the following happens:

- As part of program initialization, the run time suspends the HANDLE specifications of the calling program (with an EXEC CICS PUSH HANDLE).
- The default actions for HANDLE apply until the called program issues its own HANDLE commands.
- As part of program termination, the run time reinstates the HANDLE specifications of the calling program (with an EXEC CICS POP HANDLE).

When CBLPSHPOP is OFF, the run time does not perform the CICS PUSH or POP on a call to a COBOL subprogram.

Run with CBLPSHPOP(ON) if any of your called COBOL subprograms uses one or more of the following CICS commands:

```
CICS HANDLE CONDITION
CICS HANDLE AID
CICS HANDLE ABEND
CICS IGNORE CONDITION
CICS PUSH HANDLE
CICS POP HANDLE
```

If you use the CICS HANDLE CONDITION or HANDLE AID commands, the LABEL specified for the CICS HANDLE command must be in the same PROCEDURE DIVISION as the CICS command that causes branching to the CICS HANDLE label. You cannot use the CICS HANDLE commands with the LABEL option to handle conditions, aids, and abends that were caused by another program invoked using the COBOL CALL statement. Attempts to perform cross-program branching by using the CICS HANDLE command with the LABEL option result in a transaction abend.

If a condition, aid, or abend occurs in a nested program, the LABEL for the condition, aid, or abend must be in the same nested program; otherwise unpredictable results will occur.

"Example: handling errors by using CICS HANDLE"

### Example: handling errors by using CICS HANDLE

The following sample code illustrates the use of CICS HANDLE in COBOL programs. Program A has a CICS HANDLE CONDITION command and program B has no CICS HANDLE commands. Program A calls program B; program A also calls nested program A1.

The following shows how a condition is handled in three scenarios.

- (1) CBLPSHPOP(ON): If the CICS READ command in program B causes a condition, the condition will not be handled by program A (the HANDLE specifications have been suspended because the run time performed a CICS PUSH HANDLE). The condition will turn into a transaction abend.
- (2) CBLPSHPOP(OFF): If the CICS READ command in program B causes a

condition, the condition will not be handled by program A (the run time will diagnose the attempt to perform cross-program branching by using a CICS HANDLE command with the LABEL option). The condition will turn into a transaction abend.

- (3) If the CICS READ command in nested program A1 causes a condition, the flow of control goes to label ERR-1, and unpredictable results will occur.

```

*****
* Program A *
*****
ID DIVISION.
PROGRAM-ID. A.
. . .
PROCEDURE DIVISION.
    EXEC CICS HANDLE CONDITION
        ERROR(ERR-1)
    END-EXEC.
    CALL 'B' USING DFHEIBLK DFHCOMMAREA.
    CALL 'A1' USING DFHEIBLK DFHCOMMAREA.
. . .
THE-END.
    EXEC CICS RETURN END-EXEC.
ERR-1.
. . .
* Nested program A1.
ID DIVISION.
PROGRAM-ID. A1.
PROCEDURE DIVISION.
    EXEC CICS READ (3)
        FILE('LEDGER')
        INTO(RECORD)
        RIDFLD(ACCTNO)
    END-EXEC.
END PROGRAM A1.
END PROGRAM A.
*
*****
* Program B *
*****
ID DIVISION.
PROGRAM-ID. B.
. . .
PROCEDURE DIVISION.
    EXEC CICS READ (1) (2)
        FILE('MASTER')
        INTO(RECORD)
        RIDFLD(ACCTNO)
    END-EXEC.
. . .
END PROGRAM B.

```





---

## Chapter 19. 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.
- Use the DB2 precompiler or compile with the SQL compiler option if using DB2 for OS/390 Version 7 or later.

### RELATED CONCEPTS

“DB2 coprocessor” on page 358

### RELATED TASKS

“Coding SQL statements”

“Compiling with the SQL option” on page 356

Coding SQL statements in a COBOL application (*IBM DB2 Application Programming and SQL Guide*)

### RELATED REFERENCES

*IBM DB2 SQL Reference*

---

## 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.

### Using SQL INCLUDE

An SQL INCLUDE statement is treated identically to a native COBOL COPY statement. 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.

The library search order for SQL INCLUDE statements is the same SYSLIB concatenation as the compiler uses to resolve COBOL COPY statements that do not specify a library name.

### 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 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 SQLCA (with one exception on OS/390) to indicate whether the operation succeeded or failed. Your program should test the return code and take any necessary action.

The exception on OS/390 occurs when a program runs under DSN from one of the alternate entry points of the TSO batch mode module IKJEFT01 (IKJEFT1A or IKJEFT1B). In this case, the return code is passed in register 15.

Because DB2 usually does not pass the return code in register 15, the COBOL RETURN-CODE special register might contain a value that is not valid. Your COBOL program should set the RETURN-CODE special register to a meaningful value before returning to its caller, to keep the COBOL convention of storing its RETURN-CODE special register into register 15.

### RELATED CONCEPTS

“Formats for numeric data” on page 36

---

## Compiling with the SQL option

You use the SQL compiler option on OS/390 to enable the DB2 coprocessor capability and specify the DB2 suboptions. The SQL compiler option works with embedded SQL statements only if the compiler has access to DB2 for OS/390 Version 7 or later.

You can specify the SQL option in any of the compiler option sources: compiler invocation, PROCESS or CBL statements, or installation default. You cannot specify DB2 suboptions when the SQL option is the COBOL installation default, but you can specify default DB2 suboptions by customizing the DB2 product installation defaults.

The DB2 suboption string that you provide on the SQL compiler option is made available to the DB2 coprocessor. Only the DB2 coprocessor views the contents of the string.

When you use the DB2 coprocessor, you must compile with these options:

Compiler option	Comment
SQL	If you use also NOLIB, LIB is forced on. If you use also ANALYZE, SQL is forced off.
LIB	Must be specified with SQL
SIZE( <i>xxx</i> )	<i>xxx</i> is a size value (not MAX) that leaves enough storage in the user region for the DB2 coprocessor services.

You can use standard JCL procedural statements to compile your program with the DB2 coprocessor. In addition to specifying the above compiler options, specify the following items in your JCL:

- DBRMLIB DD statement with the location for the generated database request module (DBRM).
- STEPLIB override for the COBOL step, adding the data set that contains the DB2 coprocessor services, unless these services are in the LNKLST. Typically, this data set is DSN710.SDSNLOAD, but your installation might have changed the name.

For example, you might have the following lines in your JCL:

```
//DBRMLIB DD DSN=PAYROLL.MONTHLY.DBRMLIB.DATA(MASTER),DISP=SHR
//STEPLIB DD DSN=DSN710.SDSNLOAD,DISP=SHR
```

## Compiling in batch

When you use the SQL option to compile a source file that contains a sequence of COBOL programs (a batch compile sequence), the option must be in effect for the first program of the batch sequence. If the SQL option is specified on CBL or PROCESS cards, the CBL or PROCESS cards must precede the first program in the batch compile sequence.

## Separating DB2 suboptions

Because of the concatenation of multiple SQL option specifications, you can separate DB2 suboptions (which might not fit into single CBL statement) into multiple CBL statements.

The DB2 suboptions that you include in the suboption string are cumulative. The compiler concatenates these suboptions from multiple sources in the order that they are specified. For example, suppose that your source file has the following code:

```
//STEP1 EXEC IGYWC, . . .
// PARM.COBOL='SQL("DATABASE xxxx")'
//COBOL.SYSIN DD *
    CBL SQL("PACKAGE USER xxxx")
    CBL SQL("USING xxxx")
    IDENTIFICATION DIVISION.
    PROGRAM-ID. DRIVER1.
```

During compilation, the compiler passes the following suboption string to the DB2 coprocessor:

```
"DATABASE xxxx PACKAGE USER xxxx USING xxxx"
```

The concatenated strings are delimited with single spaces. When the compiler finds multiple instances of the same DB2 suboption, the last specification of the suboption in the concatenated string will be in effect. The compiler limits the length of the concatenated DB2 suboption string to 4K bytes.

### RELATED CONCEPTS

"DB2 coprocessor" on page 358

### RELATED REFERENCES

"SQL" on page 293

*IBM DB2 Command Reference*

## DB2 coprocessor

When you use the DB2 coprocessor (called *SQL statement coprocessor* by DB2), the compiler handles your source program containing embedded SQL statements without your having to use a separate precompile step. 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.

Although the use of a separate precompile step continues to be supported, use of the coprocessor is recommended. Interactive debugging with Debug Tool is enhanced when you use the coprocessor because you only see the SQL statements in the listing (and not the generated COBOL source). However, you must have DB2 for OS/390 Version 7 or later. The DB2 coprocessor is not supported on VM.

Compiling with the DB2 coprocessor generates a DB2 database request module (DBRM) along with the usual COBOL compiler outputs such as object module and listing. The DBRM writes to the data set that you specified on the DBRMLIBB DD statement in the JCL for the COBOL compile step. As input to the DB2 bind process, the DBRM data set contains information about the SQL statements and host variables in the program.

The COBOL compiler listing includes the error diagnostics (such as syntax errors in the SQL statements) that the DB2 coprocessor generates.

Certain restrictions on the use of COBOL language that apply when you use the precompile step do not apply when you use the DB2 coprocessor:

- You can use SQL statements in any nested program. (With the precompiler, SQL statements are restricted to the outermost program.)
- You can use SQL statements in copy books.
- REPLACE statements work on SQL statements.

### RELATED TASKS

“Compiling with the SQL option” on page 356

---

## Chapter 20. Running COBOL programs under IMS

Although much of the coding of a COBOL program will be the same when running under IMS, be aware of the following recommendations and restrictions.

In COBOL, IMS message processing programs (MPPs) do not use non-IMS input or output statements such as READ, WRITE, REWRITE, OPEN, and CLOSE.

With COBOL for OS/390 & VM you can invoke IMS facilities using the following interfaces:

- CBLTDLI call
- Language Environment callable service CEETDLI

You code calls to CEETDLI the same way as calls to CBLTDLI. CEETDLI behaves essentially the same way as CBLTDLI.

---

### Compiling and linking COBOL programs for running under IMS

For best performance in the IMS environment, use the RENT compiler option. It causes COBOL to generate reentrant code. You can then run your application programs in either *preloaded* mode (the programs remain resident in storage) or *nonpreload* mode, without having to recompile with different options.

IMS allows COBOL programs to be preloaded. This preloading can boost performance because subsequent requests for the program can be handled faster when the program is already inside the processor (rather than being fetched from a library each time it is needed).

You must use the RENT compiler option to compile a program that is to be used only preloaded or in both a preloaded and a nonpreloaded way. When you preload a load module that contains COBOL programs, all of the COBOL programs in that load module must be compiled with the RENT option.

In an application with any mixture of COBOL for OS/390 & VM, COBOL for MVS & VM, VS COBOL II, and OS/VS COBOL programs, the following compiler options are recommended:

COBOL for OS/390 & VM	COBOL for MVS & VM	VS COBOL II	OS/VS COBOL
RENT	RENT	RENT and RES	RES and NOENDJOB for preloaded programs
			RES and ENDJOB for nonpreloaded programs

You can place programs compiled with the RENT option in the OS/390 link pack area. There they can be shared among the IMS dependent regions.

To run above the 16-MB line, your application program must be compiled with either of the following compiler options, depending on your IMS environment:

RENT  
NORENT and RMODE(ANY)

With IMS, the data for IMS application programs can reside above the 16-MB line, and you can use DATA(31) RENT, or RMODE(ANY) NORENT, for programs that use IMS services.

The recommended link-edit attributes for proper execution of COBOL programs under IMS are as follows:

- Link as RENT load modules that contain only COBOL for OS/390 & VM programs compiled with the RENT compiler option.
- To link load modules that contain a mix of COBOL for OS/390 & VM RENT programs and other programs, you can use the link-edit attributes recommended for the other programs.

**RELATED TASKS**

CEETDLL, and condition handling under IMS (*Language Environment Programming Guide*)

**RELATED REFERENCES**

“DATA” on page 266

“RENT” on page 289

Appendix M. IMS considerations (*COBOL for OS/390 & VM Compiler and Run-Time Migration Guide*)

---

## Chapter 21. Running COBOL programs under OS/390 UNIX

To run your COBOL programs in the OS/390 UNIX environment, you must compile them with the IBM COBOL for OS/390 & VM or the COBOL for MVS & VM compiler. They must be reentrant: use the compiler and linker option RENT. If you are going to run them from the HFS, use the linker option AMODE(31). Any AMODE(24) program that you call from within an OS/390 UNIX application must reside in an MVS PDS or PDSE.

The following restrictions apply to running under OS/390 UNIX:

- SORT and MERGE statements are not supported.
- You cannot use the interfaces for preinitialization (run-time option RTEREUS and functions IGZERRE and ILBOSTP0) to establish a reusable environment in the OS/390 UNIX environment.
- You cannot run your COBOL program in more than one thread. If you start a COBOL application in a second thread, you get a software condition from the COBOL run time. You can run your COBOL programs in the initial process thread (IPT) or in one non-IPT that you create from a C or PL/I routine.
- You can use Debug Tool to debug your OS/390 UNIX programs using a remote debugging session from IBM VisualAge COBOL only.

### RELATED TASKS

“Chapter 14. Compiling under OS/390 UNIX” on page 235

“Running in OS/390 UNIX environments”

“Setting and accessing environment variables” on page 362

“Calling UNIX/POSIX APIs” on page 364

“Accessing main program parameters” on page 365

*Language Environment Programming Guide*

### RELATED REFERENCE

“RENT” on page 289

---

## Running in OS/390 UNIX environments

You can run your OS/390 UNIX COBOL programs in any of the OS/390 UNIX execution environments:

- From a UNIX shell, either:
  - The OS/390 shell (OMVS) or
  - The OS/390 ISPF shell (ISHELL)

Enter the program name at the shell prompt.

The program must be in the current directory or in your search path. When running your UNIX programs, you can specify run-time options only by setting the environment variable `_CEE_RUNOPTS` before starting your program.

You can run your MVS executable program that resides in a cataloged MVS data set from a shell by using the `tso` utility. For example:

```
tso "call 'my.loadlib(myprog)'"
```

The ISPF shell can direct stdout and stderr only to an HFS file, not to your terminal.

- From outside a shell, either:
  - TSO/E or
  - OS/390 batch

To call your OS/390 UNIX COBOL program that resides in an HFS file from the TSO/E ready prompt, use the BPXBATCH utility or a spawn() syscall in a REXX exec.

To call your OS/390 UNIX COBOL program that resides in an HFS file with the JCL EXEC statement, use the BPXBATCH utility.

#### RELATED TASKS

“Setting and accessing environment variables”

“Calling UNIX/POSIX APIs” on page 364

“Accessing main program parameters” on page 365

Running POSIX-enabled programs using an OS/390 UNIX shell (*Language Environment Programming Guide*)

Running POSIX-enabled programs outside the OS/390 UNIX shells (*Language Environment Programming Guide*)

“Defining and allocating QSAM files” on page 121

“Defining and allocating line-sequential files” on page 165

“Allocating VSAM files” on page 156

“Displaying values on a screen or in a file (DISPLAY)” on page 29

#### RELATED REFERENCES

“TEST” on page 295

BPXBATCH utility (*OS/390 UNIX System Services User’s Guide*)  
*Language Environment Programming Reference*

---

## Setting and accessing environment variables

You can set environment variables for your OS/390 UNIX COBOL program in either of these ways:

- From the shell with commands export and set
- From your program

Although setting and resetting environment variables from the shell before you begin to run your program is probably a typical procedure, you can set, reset, and access environment variables from your program while it is running.

If you are running your program with BPXBATCH, you can set environment variables by using an STDENV DD statement.

### Setting environment variables that affect execution

To set environment variables from a shell, use the export or set command. For example, to set the environment variable MYFILE:

```
export MYFILE=/usr/mystuff/notes.txt
```

Call POSIX functions setenv() and putenv() to set environment variables from your program.



## Environment variables of interest for COBOL programs

### `_CEE_RUNOPTS`

Run-time options.

### `LIBPATH`

Directory paths of your dynamic link libraries.

### `_IGZ_SYSOUT`

stdout or stderr to direct your DISPLAY output. These are the only allowable values.

### `STEPLIB`

Location of programs not in the LNKLST.

## Resetting environment variables

You can reset an environment variable from the shell or from your program. To reset an environment variable as if it had not been set, use the OS/390 UNIX shell command `unset`. To reset an environment variable from your COBOL program, call the `setenv()` function.

## Accessing environment variables

To see the values of all environment variables, you can use the `export` command with no parameters.

To access the value of an environment variable from your COBOL program, call the `getenv()` function.

“Example: accessing environment variables”

### RELATED TASKS

“Running in OS/390 UNIX environments” on page 361

“Calling UNIX/POSIX APIs” on page 364

“Accessing main program parameters” on page 365

“Displaying values on a screen or in a file (DISPLAY)” on page 29

### RELATED REFERENCES

*Language Environment Programming Reference*

*OS/390 MVS JCL Reference*

## Example: accessing environment variables

The following example shows how you can access and set environment variables from your COBOL programs by calling the standard POSIX functions `getenv()` and `putenv()`.

Because these are C functions, you must pass arguments BY VALUE. Pass character strings as BY VALUE pointers that point to null-terminated strings.

```
CBL pgmname(longmixed)
  Identification division.
  Program-id. "envdemo".
  Data division.
  Working-storage section.
  01 P pointer.
  01 PATH pic x(5) value Z"PATH".
  01 var-ptr pointer.
  01 var-len pic 9(4) binary.
  01 putenv-arg pic x(14) value Z"MYVAR=ABCDEFGH".
  01 rc pic 9(9) binary.
```

```

Linkage section.
01 var pic x(5000).
Procedure division.
* Retrieve and display the PATH environment variable
  Set P to address of PATH
  Call "getenv" using by value P returning var-ptr
  If var-ptr = null then
    Display "PATH not set"
  Else
    Set address of var to var-ptr
    Move 0 to var-len
    Inspect var tallying var-len
      for characters before initial X"00"
    Display "PATH = " var(1:var-len)
  End-if
* Set environment variable MYVAR to ABCDEFG
  Set P to address of putenv-arg
  Call "putenv" using by value P returning rc
  If rc not = 0 then
    Display "putenv failed"
  Stop run
End-if
Goback.

```

---

## Calling UNIX/POSIX APIs

You can call standard UNIX/POSIX functions from your UNIX programs and from traditional MVS COBOL programs. These functions are part of Language Environment.

Because these are C functions, you must pass arguments BY VALUE. Pass character strings as BY VALUE pointers that point to null-terminated strings. You must use the compiler options NODYNAM and PGMNAME(LONGMIXED) when you compile programs that call these functions.

### fork, exec, and spawn

You can call the `fork`, `exec`, and `spawn` functions from your COBOL program, or from a non-COBOL program in the same process as COBOL programs. However, you need to be aware of these restrictions:

- From a forked process you cannot access any COBOL sequential, indexed, or relative files that were open when you issued the `fork`. You get a file status code 92 when you attempt such an access (`CLOSE`, `READ`, `WRITE`, `REWRITE`, `DELETE`, or `START`). You can access line-sequential files that were open at the time of a `fork`.
- You cannot use the `fork` function in a process in which any of the following is true:
  - A COBOL `SORT` or `MERGE` is running.
  - A declarative is running.
  - The process has more than one Language Environment enclave (COBOL run unit).
  - The process has used any of the COBOL reusable environment interfaces.
  - The process has ever run an OS/VS COBOL or VS COBOL II program.
- With one exception, MVS DD allocations are not inherited from a parent process to a child process. The exception is the local `spawn`, which creates a child process in the same address space as the parent process. You request a local `spawn` by setting the environment variable `_BPX_ SHAREAS=YES` before you invoke the `spawn` function.

The exec and spawn functions start a new Language Environment enclave in the new UNIX process. Therefore, the target program of the exec or spawn is a main program, and all COBOL programs in the process start in initial state with all files closed.

## Samples

Sample code is provided with the product for calling some of the POSIX routines. The sample source code is in the SIGYSAMP data set.

Purpose	Sample	Functions used
Shows how to use some of the file and directory routines.	IGYTFL1	getcwd mkdir rmdir access
Shows how to use the iconv routines to convert data.	IGYTCNV	iconv_open iconv iconv_close
Shows the use of the exec routine to run a new program along with other process related routines.	IGYTEXC IGYTEXC1	fork getpid getppid execl perror wait
Shows how to get the errno value.	IGYTERNO IGYTGETE	perror fopen
Shows the use of the interprocess communication message routines.	IGYTMSQ IGYTMSQ2	ftok msgget msgsnd perror fopen fclose msgrcv msgctl perror

### RELATED TASKS

“Running in OS/390 UNIX environments” on page 361

“Setting and accessing environment variables” on page 362

“Accessing main program parameters”

*Language Environment Programming Guide*

### RELATED REFERENCES

*OS/390 C/C++ Run-Time Library Reference*

*OS/390 UNIX System Services Programming: Assembler Callable Services Reference*

---

## Accessing main program parameters

When you run a COBOL program from the OS/390 UNIX shell command line, or with an exec or spawn function, the parameter list consists of three parameters passed by reference:

### argument count

A binary fullword integer containing the number of elements in each of the arrays that are passed as the second and third parameters.

### argument length list

An array of pointers. The *n*th entry in the array is the address of a fullword binary integer containing the length of the *n*th entry in the argument list.

### argument list

An array of pointers. The *n*th entry in the array is the address of the *n*th character string passed as an argument on the spawn or exec functions, or the command invocation. Each character string is null-terminated.

This array is never empty. The first argument is the character string that represents the name of the file associated with the process being started.

You can access these parameters with standard COBOL coding.

“Example: accessing main program parameters”

#### RELATED TASKS

“Running in OS/390 UNIX environments” on page 361

“Setting and accessing environment variables” on page 362

“Calling UNIX/POSIX APIs” on page 364

## Example: accessing main program parameters

The following example shows the three parameters that are passed by reference.

```
Identification division.
Program-id. "EXECED".
*****
* This sample program displays arguments received via exec() *
* function of OS/390 UNIX System Services *
*****
Data division.
Working-storage section.
01 curr-arg-count pic 9(9) binary value zero.
Linkage section.
01 arg-count pic 9(9) binary. (1)
01 arg-length-list. (2)
    05 arg-length-addr pointer occurs 1 to 99999
        depending on curr-arg-count.
01 arg-list. (3)
    05 arg-addr pointer occurs 1 to 99999
        depending on curr-arg-count.
01 arg-length pic 9(9) binary.
01 arg pic X(65536).
Procedure division using arg-count arg-length-list arg-list.
*****
* Display number of arguments received *
*****
    Display "Number of arguments received: " arg-count
*****
* Display each argument passed to this program *
*****
    Perform arg-count times
        Add 1 to curr-arg-count
* *****
* * Set address of arg-length to address of current *
* * argument length and display *
* *****
        Set Address of arg-length
            to arg-length-addr(curr-arg-count)
        Display
            "Length of Arg " curr-arg-count " = " arg-length
* *****
```

```

* * Set address of arg to address of current argument *
* * and display *
* *****
  Set Address of arg to arg-addr(curr-arg-count)
  Display "Arg " curr-arg-count " = " arg (1:arg-length)
End-Perform
Display "Display of arguments complete."
Goback.

```

- (1) This count contains the number of elements in each of the arrays that are passed as the second and third parameters.
- (2) This array includes a pointer to the length of the *n*th entry in the argument list.
- (3) This array includes a pointer to the *n*th character string passed as an argument on the spawn or exec function or the command invocation.



---

## Chapter 22. Running COBOL programs under CMS

You can code and compile COBOL programs under both CMS and OS/390, but you should be aware of certain differences in how programs run in the CMS environment, including file handling and other run-time restrictions.

---

### Run-time restrictions under CMS

The following restrictions apply when you run COBOL programs under CMS:

- SORT and MERGE verbs are supported on XA-mode machines if DFSORT/CMS Version 2 Release 1 or an equivalent is installed.
- The RERUN phrase is tolerated, but no checkpoints are taken because CMS has no checkpoint/restart feature.
- Only if you use the CMS LKED command to generate independent load modules for each program in the object deck does CMS support the following code: CANCEL, dynamic CALL *literal*, and CALL *identifier* targeting a subprogram compiled into the same object program as the calling program using the batch compile facility.
- You cannot use the Debug Tool with programs compiled with the TEST(SEPARATE) compiler option.
- You cannot run COBOL programs in the VM/ESA OpenEdition environment.
- You cannot use files defined as ORGANIZATION LINE SEQUENTIAL.

CMS does not support using the following COBOL code:

- User label-handling functions. Therefore, the label user-handling format of USE and the data-name option of the LABEL RECORDS clause are invalid. USER DECLARATIVE is never invoked for label processing.
- IGZBRDGE macro for call conversions.
- Object-oriented COBOL language extensions (references to objects, methods, or classes, for example).
- Programs compiled with the DLL compiler option.

---

### Handling QSAM files under CMS

Processing of QSAM files under CMS is slightly different than under OS/390.

QSAM unblocked files or files with a small block size can slow performance under CMS. To improve performance, use files with a block size 50 to 100 times larger than the logical record length.

Also, CMS does not diagnose as errors any conflicts in fixed file attributes when opening an existing QSAM file for output (not update). These attributes refer to:

- Record format (RECFM)
- Logical record length (LRECL)
- Data set organization (DSORG)
- Attribute indicating that ASCII or EBCDIC translation is requested (OPTCD=Q)

Conflicts in these attributes can occur on an OPEN of an existing QSAM file when the COBOL program (DCB) and the data set label for the file or the CMS FILEDEF command specify different values for any of them. However, because CMS does

not diagnose these conflicts as errors, Language Environment does not return a file status code 39 to the COBOL program, although such a code is what COBOL expects for such errors.

Certain VM/ESA rules govern the precedence of sources for CMS file attributes (sources include the DCB for the program, CMS FILEDEF command, and existing data set label). These rules affect the behavior of the file status code 39.

---

## Run-time message IGZ0002S

Language Environment issues COBOL run-time message IGZ0002S when a SYNAD error has occurred on a QSAM file, and no invalid key clause, file status key, or user error declarative was specified in the COBOL program for the output operation tried. The text of IGZ0002S is provided by the system SYNADAF routine. Under CMS the format of message IGZ0002S is:

```
IGZ0002S '120S INPUT/OUTPUT ERROR nnn ON ddname'
```

where

**120S** Is the CMS message number for SYNAD errors

**INPUT** Indicates an input operation was tried

**OUTPUT** Indicates an output operation was tried

*nnn* Is the associated error code

*ddname*

Is the ddname of the related file

A CMS SYNAD error also occurs when there is not enough minidisk space available for a CMS output file. When this SYNAD error occurs on QSAM files used by COBOL programs, and no invalid key clause, file status key, or user error declarative was specified in the COBOL program for the output operation tried, Language Environment issues COBOL run-time message IGZ0002S as follows:

```
IGZ0002S '120S OUTPUT ERROR 013 ON ddname'
```

After issuing message IGZ0002S, the COBOL program ends.

### RELATED REFERENCES

Precedence rules for CMS file attributes (*VM/ESA CMS Application Development Guide*)



---

## Chapter 23. Accessing COBOL programs interactively with ISPF

If you code your application program using ISPF panels, you can interactively access your COBOL program.

Under CMS, you must specify the CMS option when you use the ISPF command ISPEXEC SELECT PGM to invoke a COBOL program in order to pass user arguments or run-time options specified and have them take effect. Under OS/390, you cannot pass any user arguments or run-time options when you use the ISPEXEC SELECT PGM command to invoke a COBOL program.

You can run COBOL programs concurrently in both screens of the ISPF split-screen mode.

One of the sample programs that is shipped with COBOL for OS/390 & VM, IGYTCARB, runs under ISPF.

### RELATED CONCEPTS

“IGYTCARB: interactive program” on page 592



---

## Part 4. Developing object-oriented programs

<b>Chapter 24. Writing object-oriented programs</b>	375
Example: mail-order catalog	375
Subclasses	377
Defining a class	378
CLASS-ID paragraph for defining a class	378
REPOSITORY paragraph for defining a class	379
WORKING-STORAGE SECTION for defining a class	379
Example: defining a class	380
Defining a class method	381
METHOD-ID paragraph for defining a class method	381
Overriding a method	381
INPUT-OUTPUT SECTION for defining a method	382
DATA DIVISION for defining a method	382
PROCEDURE DIVISION for defining a method	383
Coding special methods	383
Attribute methods	383
Method somInit for customizing object creation	384
Method somUninit for customizing object destruction	384
Example: defining a method	384
Defining a client program	389
REPOSITORY paragraph for defining a client	389
WORKING-STORAGE SECTION for defining a client	390
Creating and freeing instances of classes	390
Manipulating object references	391
Invoking methods	391
Example: defining a client	392
Defining a subclass	393
CLASS-ID paragraph for defining a subclass	394
REPOSITORY paragraph for defining a subclass	394
WORKING-STORAGE SECTION for defining a subclass	395
Defining a subclass method	395
METHOD-ID paragraph for defining a subclass method	395
Example: defining a subclass (with methods)	396
Defining a metaclass	405
CLASS-ID paragraph for defining a metaclass	406
REPOSITORY paragraph for defining a metaclass	406
WORKING-STORAGE SECTION for defining a metaclass	406
Defining a metaclass method	406
Invoking a metaclass constructor method	407
Changing the definition of a class or subclass	407
Changing a client program	407
Example: defining a metaclass (with methods)	408
<b>Chapter 25. System Object Model</b>	415
SOM Interface Repository	415
Accessing the SOM Interface Repository	415
Populating the SOM Interface Repository	416
SOM environment variables	417
Example: sample JCL for an object-oriented application	418
Notes to sample JCL	419
SOM services	420
SOM methods and functions	420
Compiling and linking programs that call SOM functions	421
Class initialization	421
Changing SOM class interfaces	422
<b>Chapter 26. Using SOM IDL-based class libraries</b>	425
SOM objects	425
SOM IDL	426
Mapping IDL to COBOL	427
Using IDL operations	427
Expressing IDL attributes	428
Common IDL types	429
Examples: common IDL types	429
enum	430
interface	430
long (signed and unsigned)	430
short (signed and unsigned)	431
string	431
Complex IDL types	432
any	432
array	433
sequence	433
struct	434
union	434
Passing COBOL arguments and return values	435
Passing literal arguments	435
Passing arguments of complex types	435
Conventions for passing arguments and return values	436
Passing enumerated arguments	437
Passing string arguments	437
Example: passing string arguments	439
Example: using a SOM IDL-based class library	440
Handling errors and exceptions	442
Passing environment variables	443
Checking the exception type field	443
Handling exceptions	443
Example: checking SOM exceptions	443
Creating and initializing object instances	445
Looking at the IDL file	446
Avoiding memory leaks	447
Example: COBOL variable-length string class	448
Source code for helper routines	450
<b>Chapter 27. Wrapping or converting procedure-oriented programs</b>	451
OO view of COBOL programs	451
Wrapping procedure-oriented programs	452

Coordinating procedural code with interface actions . . . . .	452
Integrating procedural code into OO systems	452
Changing procedural code . . . . .	453
Converting from procedure-oriented to OO programs. . . . .	453
Identifying objects. . . . .	454
Analyzing data flow and usage . . . . .	454
Reallocating code to objects . . . . .	454
Writing the object-oriented code . . . . .	455

---

## Chapter 24. 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 378

“Defining a class method” on page 381

---

### 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**

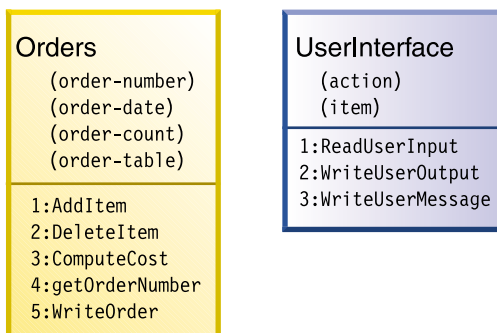
Write the final order

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.

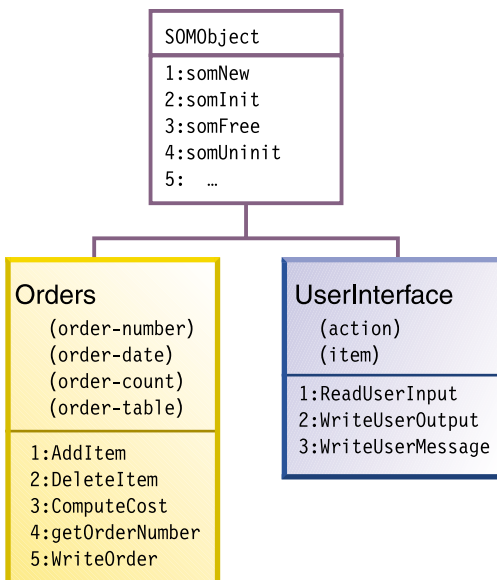


The words in parentheses are instance data and the words after the number and colon are methods.

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.

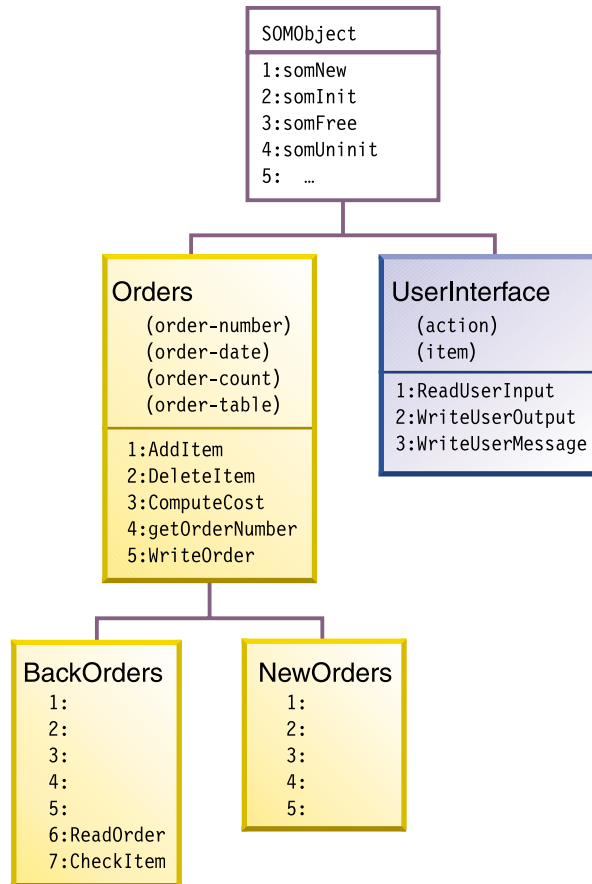
All classes in COBOL inherit directly or indirectly from SOMObject. When multiple inheritance is used, the class structure might not be a tree; it could be a graph. However, the SOMObject class will always be at the root of the tree or graph.

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 378
- “Defining a class method” on page 381
- “Defining a subclass” on page 393
- “Defining a metaclass” on page 405

---

## 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 379 (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 379 (optional)
PROCEDURE (optional)	Define methods.	“Defining a class method” on page 381

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.

Any application containing class definitions needs to be prelinked.

“Example: mail-order catalog” on page 375

“Example: defining a class” on page 380

### RELATED TASKS

“Defining a subclass” on page 393

“Defining a metaclass” on page 405

“Changing the definition of a class or subclass” on page 407

“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.           Required
Configuration Section.         Required
Repository.                    Required
    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 382 (optional)
DATA (optional)	Define external files. Allocate a copy of the data.	“DATA DIVISION for defining a method” on page 382 (optional)
PROCEDURE (optional)	Code the executable statements to complete the service provided by the method.	“PROCEDURE DIVISION for defining a method” on page 383 (optional)

“Example: mail-order catalog” on page 375

“Example: defining a method” on page 384

### RELATED TASKS

“Defining a subclass method” on page 395

“Defining a metaclass method” on page 406

“Overriding a method”

“Invoking methods” on page 391

“Coding special methods” on page 383

## 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.           Required  
Method-Id. "somInit" Override.    Required
```

## 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 OrdrFile.
```

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 12

“Sharing data by using the EXTERNAL clause” on page 483

## 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
  - SEGMENTATION
  - 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 17

#### RELATED TASKS

“Processing the data” on page 17

## 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" on page 390 (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 375

"Example: defining a client" on page 392

### RELATED TASKS

"Creating and freeing instances of classes" on page 390

"Manipulating object references" on page 391

"Invoking methods" on page 391

"Changing a client program" on page 407

## 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
```

```
Source-Computer.  IBM-370.  
Object-Computer.  IBM-370.
```

```
Repository.      Required  
  Client UserInterface is 'UserInterface'  
  Client Orders is 'Orders'.
```

The SOURCE-COMPUTER and OBJECT-COMPUTER (and SPECIAL-NAMES) paragraphs are optional as usual.

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" (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 395 (optional)
PROCEDURE (optional)	Define methods.	Method definitions: "Defining a subclass method" on page 395

You must properly terminate a subclass definition with an END CLASS statement.

"Example: mail-order catalog" on page 375

"Example: defining a subclass (with methods)" on page 396

## 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 375

"Example: defining a subclass (with methods)" on page 396

### RELATED REFERENCES

Multiple inheritance (*OS/390 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” on page 406 (required)
ENVIRONMENT (required)	Relate the metaclass and class names to external SOM names.	CONFIGURATION section (required) “REPOSITORY paragraph for defining a metaclass” on page 406 (required)
DATA (optional)	Describe instance data the metaclass needs.	“WORKING-STORAGE SECTION for defining a metaclass” on page 406 (optional)
PROCEDURE (optional)	Define methods.	“Defining a metaclass method” on page 406

You must properly terminate a metaclass definition with an END CLASS statement.

“Example: mail-order catalog” on page 375

“Example: defining a metaclass (with methods)” on page 408

## RELATED CONCEPTS

SOMClass Class Object (*OS/390 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.                Required
Class-Id. MetaBackOrders INHERITS SOMClass. 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 395 (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 metaclass method definition with an END METHOD statement.

"Example: mail-order catalog" on page 375

"Example: defining a metaclass (with methods)" on page 408

#### RELATED TASKS

"Invoking a metaclass constructor method"

"Defining a class method" on page 381

"Defining a subclass method" on page 395

### 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 25. 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.

The object-oriented COBOL language support is based on OS/390 SOMobjects. This support is not available on VM/CMS.

### RELATED CONCEPTS

“SOM Interface Repository”

“SOM services” on page 420

### RELATED TASKS

“Defining a class” on page 378

“Chapter 26. Using SOM IDL-based class libraries” on page 425

*OS/390 SOMobjects Programmer’s Guide*

### RELATED REFERENCES

“SOM methods and functions” on page 420

---

## 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 421

“Populating the SOM Interface Repository” on page 416

## 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:

```
SOMIR=/'prefix.class.SOMIR';
```

*prefix.class.SOMIR* is the data set name (DSN) of the IR file. Substitute your OS/390 user ID for *prefix*, and a meaningful name for *class*.

You set this and other SOM environment variables in a SOM configuration file. See the cross-references below for further information on setting up configuration files and on SOM environment variables.

SOM provides a predefined interface repository data set called *somhlq.SGOSIR*, which contains the interfaces of all the SOM system classes. (*somhlq* represents the high-level qualifiers of the data sets where SOM is installed on your system.) Do not add your class interfaces to this IR. Define your own IR files and set SOMIR to an ordered list of IR files. SOM reads IR files in the list from left to right, but updates only the rightmost IR file. So specify your development IR file as the rightmost in the list. You can of course have application IR files (which are not subject to change) earlier in the list. For example:

```
SOMIR=/'somhlq.SGOSIR';/'prefix.app.SOMIR';/'prefix.user.IR';
```

In this case, *somhlq.SGOSIR* and *prefix.app.SOMIR* are the SOM and application IR files, respectively, and are not changed. *prefix.user.IR* is the working IR file, which is updated.

When you compile an application with the IDLGEN or TYPECHK compiler options, you must:

1. Define an IR file.
2. Create a SOM configuration file whose SOMIR setting specifies the IR as the rightmost in the list of IR files.
3. Provide a SOMENV DD statement (or equivalent TSO ALLOCATE command) that refers to the configuration file.

#### RELATED CONCEPTS

“SOM Interface Repository” on page 415

“SOM services” on page 420

#### RELATED TASKS

Setting up configuration files (*OS/390 SOMobjects Programmer's Guide*)

#### RELATED REFERENCES

“SOM environment variables” on page 417

“SOM methods and functions” on page 420

## 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.

“Example: sample JCL for an object-oriented application” on page 418

#### RELATED CONCEPTS

“SOM Interface Repository” on page 415

#### RELATED REFERENCES

“Class initialization” on page 421

---

## SOM environment variables

Environment variables provide information needed by the COBOL compiler and run-time environment and the SOM compiler and run-time environment. You specify the environment variables in a SOM *configuration file* (also known as a SOM *profile*).

The configuration file is organized into stanzas that contain specific environment variables. Each stanza is headed by the name of the stanza in brackets. For example, `[somc]` introduces the stanza for the SOM compiler.

Set the following environment variables, within the proper stanza, as shown:

### SOM kernel stanza:

#### SMLANG

Specifies the language for SOM messages.

For example:

```
[somk]  
SMLANG=EnUs
```

### SOM compiler stanza:

#### SMEMIT

Specifies which emitters the SOM compiler runs.

For a COBOL class, the most frequently used emitters are the `ir` emitter, which populates the interface repository, and the `.h` emitter, which produces a header file for use by a C client of the COBOL class.

#### SMINCLUDE

Specifies where the SOM compiler looks for `#include` files referred to by the IDL file being compiled.

For example:

```
[somc]  
SMEMIT=ir:h  
SMINCLUDE=//'somhlq.SGOSIDL';//'prefix.MYCLASS.IDL';
```

### SOM interface repository stanza:

## SOMIR

Specifies the location of the interface repository files.

For example:

```
[somi r]  
SOMIR=// 'somhlq.SGOSIR' ;// 'prefix.LOCAL.SOMIR' ;
```

See a complete sample SOM configuration file in note (6) to the JCL in the example below (“Example: sample JCL for an object-oriented application”).

### RELATED TASKS

Setting up configuration files (*OS/390 SOMObjects Programmer’s Guide*)

## Example: sample JCL for an object-oriented application

The following series of JCL statements compiles a COBOL class definition with type checking and IDL generation, and then prelinks and link-edits the class to create a DLL. The generated IDL for the class is compiled with the SOM compiler to populate the interface repository. The COBOL client program is then compiled with type checking, validating that its use of the class interface is correct, and the client code is prelinked and linked to form the main load module. Finally the application is run, and the client load module accesses both the COBOL class and the SOM kernel from DLLs.

The example uses standard cataloged procedures, augmented by additional DD statements required for processing object-oriented programs.

```
//OOSAMPLE JOB ,  
// TIME=(1),MSGLEVEL=(1,1),MSGCLASS=H,CLASS=A,NOTIFY=&SYSUID  
// SET SOMPFX='SOMMVS'  
// SET LEPFX='CEE.V2R10M0'  
//*-----  
//* Compile class definition with type checking and IDL generation,  
//* and exporting external symbols needed for a SOM DLL, then  
//* prelink and link-edit the class to form a DLL.  
//*-----  
//STEP1 EXEC IGYWCPL,REGION=16M,GOPGM=classnm,  
// PARM.COBOL='RENT,PGMN(LM),TYPECHK,IDLGEN,DLL,EXPORTALL', (1)  
// PARM.PLKED='DLLNAME(classnm)', (2)  
// PARM.LKED='RENT,LIST,XREF,LET,MAP'  
//COBOL.SYSIN DD DSN=user.COBOL.SOURCE(classnm),DISP=SHR  
//COBOL.SYSIDL DD DSN=user.COBOL.IDL(classnm),DISP=SHR (3)  
//COBOL.STEPLIB DD  
// DD DSN=&LEPFX..SCEERUN,DISP=SHR (4)  
// DD DSN=&SOMPFX..SGOSLOAD,DISP=SHR (5)  
//COBOL.SOMENV DD DSN=user.SOM.PROFILE(myprof),DISP=SHR (6)  
//PLKED.SYSIN DD  
// DD DSN=&SOMPFX..SGOSIMP(GOSSOMK),DISP=SHR (7)  
//PLKED.SYSDEFSD DD DSN=&&SIDEDECK,UNIT=SYSDA,DISP=(NEW,PASS),  
// SPACE=(TRK,(1,1)) (8)  
//LKED.SYSIN DD DUMMY  
//*-----  
//* SOM compile the IDL for the class, updating interface repository  
//*-----  
//STEP2 EXEC PGM=SC,REGION=80M,  
// PARM=(' -V -usir 'user.COBOL.IDL(classnm)''')  
//STEPLIB DD DSN=&LEPFX..SCEERUN,DISP=SHR  
// DD DSN=&SOMPFX..SGOSLOAD,DISP=SHR  
//SOMENV DD DSN=user.SOM.PROFILE(myprof),DISP=SHR (6)  
//SYSOUT DD SYSOUT=*  
//*-----
```



```

/** Compile the client program with type checking, prelink and link
/** the client program importing the class DLL and the SOM kernel DLL
/**-----
//STEP3 EXEC IGYWCPL,REGION=16M,GOPGM=clientnm,
//      PARM.COBOL='RENT,PGMNAME(LM),TYPECHK,DLL', (9)
//      PARM.LKED='RENT,LIST,XREF,LET,MAP'
//COBOL.SYSIN DD DSN=user.COBOL.SOURCE(clientnm),DISP=SHR
//COBOL.STEPLIB DD
//          DD DSN=&LEPFX..SCEERUN,DISP=SHR (4)
//          DD DSN=&SOMPFX..SGOSLOAD,DISP=SHR (5)
//COBOL.SOMENV DD DSN=user.SOM.PROFILE(myprof),DISP=SHR (6)
//PLKED.SYSIN DD
//          DD DSN=&&SIDEDECK,DISP=(OLD,DELETE) (10)
//          DD DSN=&SOMPFX..SGOSIMP(GOSSOMK),DISP=SHR
//LKED.SYSIN DD DUMMY
/**-----
/** Execute the client program, accessing the class and SOM from DLLs.
/**-----
//STEP4 EXEC PGM=clientnm,REGION=16M
//STEPLIB DD DSN=&&GOSET,DISP=(OLD,DELETE) (11)
//          DD DSN=&LEPFX..SCEERUN,DISP=SHR
//          DD DSN=&SOMPFX..SGOSLOAD,DISP=SHR
//SOMENV DD DSN=user.SOM.PROFILE(myprof),DISP=SHR (6)
//SYSOUT DD SYSOUT=*
//CEEDUMP DD SYSOUT=*

```

## Notes to sample JCL

- (1) Type check the class definition, generate IDL for use in setting up type checking of the client, enable DLL support, and export the SOM external symbols needed to make the class a DLL.
- (2) Indicate to the prelinker the name of the DLL that will be created, and that will appear on generated IMPORT control statements.
- (3) Data set to hold the IDL generated from the compilation of the COBOL class.
- (4) The Language Environment SCEERUN data set is needed during compilation of object-oriented COBOL programs (as well as during the SOM compilation of the IDL file and during the execution of the program).
- (5) The SOMobjects kernel data set SGOSLOAD is needed during compilation of object-oriented COBOL programs and for SOM compile of IDL.
- (6) The SOM profile data set is needed in each of the job steps. At the minimum, the profile contains environment variables specifying the language for SOM messages, the location of #include files referenced in IDL, and the location of the interface repository files. A sample profile for this example might be:

```

[somk]
SMLANG=ENUS
[somc]
SMINCLUDE=//'somhlq.SGOSIDL';//'somhlq.SGOSEFW';//'user.COBOL.IDL';
[somir]
SOMIR=//'somhlq.SGOSIR';//'user.SOM.IR';

```
- (7) DLL definition side deck containing IMPORT statements for the SOM kernel DLL.
- (8) Data set for the prelinker to write the generated DLL definition side deck containing IMPORT statements for the COBOL class definition DLL.

- (9) Type check the client usage of the COBOL class definition using the information in the interface repository created in the previous step, and enable DLL support for the client.
- (10) DLL definition side deck containing `IMPORT` statements for the COBOL class DLL (created via item (8) above), indicating to the prelinker that the class is accessed from the DLL rather than being linked together with the client.
- (11) Include the data sets containing referenced DLL load modules in the `STEPLIB`. In this case, the COBOL class DLL is in `&&G0SET` together with the client module, and the SOM kernel DLL is in the `&SOMPFX..SGOSLOAD` data set.

**RELATED TASKS**

“Chapter 13. Compiling under OS/390” on page 203

“Populating the SOM Interface Repository” on page 416

---

## 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 TASKS**

*OS/390 SOMObjects Programmer's Guide*

**RELATED REFERENCES**

“SOM methods and functions”

“Class initialization” on page 421

## 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.
- `DLL`, because the SOM functions are packaged in dynamic load libraries. You will get an unresolved reference when you link your program unless you specify this option.

When you link your program, specify the SOM kernel side deck as input to the binder. The data set name of the side deck is `somhlq.SGOSIMP(GOSSOMK)`, where `somhlq` represents the high-level qualifiers of the data sets where SOM is installed on your system.

Your invocations of SOM methods do not require any special considerations. The correct linkage conventions are used automatically for method invocations.

#### **RELATED TASKS**

“Chapter 30. Creating a DLL or a DLL application” on page 489

## **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 420

#### RELATED REFERENCES

"SOM methods and functions" on page 420

## 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.

#### RELATED REFERENCES

Method resolution (*OS/390 SOMobjects Programmer's Guide*)



---

## Chapter 26. 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 440

### RELATED CONCEPTS

"SOM objects"

"SOM IDL" on page 426

### RELATED TASKS

Understanding Somobjects programming (*OS/390 SOMobjects Programmer's Guide*)

"Mapping IDL to COBOL" on page 427

"Handling errors and exceptions" on page 442

"Creating and initializing object instances" on page 445

"Avoiding memory leaks" on page 447

### RELATED REFERENCES

"Common IDL types" on page 429

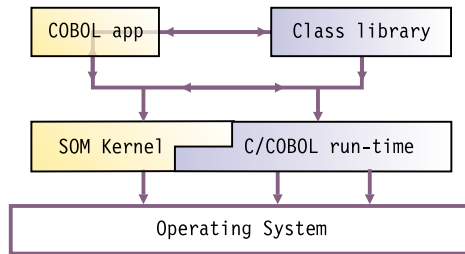
---

## 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 427

---

## 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 427

RELATED REFERENCES

“Common IDL types” on page 429

“Complex IDL types” on page 432



---

## 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 428

"Passing COBOL arguments and return values" on page 435

### RELATED REFERENCES

"Common IDL types" on page 429

"Complex IDL types" on page 432

## 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 435

“Handling errors and exceptions” on page 442

#### RELATED REFERENCES

“Common IDL types” on page 429

## 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 435

"Handling errors and exceptions" on page 442

## 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 430	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 430	OBJECT REFERENCE	
"long (signed and unsigned)" on page 430	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 431	COMPUTATIONAL-5 PICTURE S9(4)	Unsigned forms of binary data
"string" on page 431	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 437

### 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-1-max binary pic 9(9) value 101.  
1 that-1 binary pic 9(9).  
1 that.  
2 that-v pic x occurs 1 to 101 depending that-1.
```

The -1 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 437

**RELATED REFERENCES**

"Source code for helper routines" on page 450

## 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 433	Table
"sequence" on page 433	Group + variable-length table
"struct" on page 434	Group
"union" on page 434	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 `OMMAllocate`, 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 437

“Passing string arguments” on page 437

**RELATED REFERENCES**

“Conventions for passing arguments and return values”

“Source code for helper routines” on page 450

*OS/390 SOMobjects Programmer's 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 440

#### RELATED REFERENCES

“Source code for helper routines” on page 450

### 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),dll
*****
* 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) and the DLL compiler options to be able to call SOM APIs such as somGetGlobalEnvironment. The options do not affect INVOKE statements, but they do 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, `evp`, 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 `someSetException` 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 `someExceptionFree` 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

CORBA standard exceptions (*OS/390 SOMObjects Programmer's Guide*)

## 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 448

### RELATED TASKS

“Passing COBOL arguments and return values” on page 435

### RELATED REFERENCES

“Source code for helper routines” on page 450

## 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 strsze 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 strsze
      Call 'OMMAllocate' using strsze returning vstptr
      Set address of vstval to vstptr
```

```

        Move setval(1:strsze) to vstval(1:strsze)
        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 strsze 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&0x7fffffff)
#include <som.h>

/* Object Memory Management: allocate memory.                    */
somToken OMMAlloc(size_t *size){
    size_t *s;
    Clean(s,size);
    return SOMMAlloc(*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 27. 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 452

“Converting from procedure-oriented to OO programs” on page 453

---

## 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:

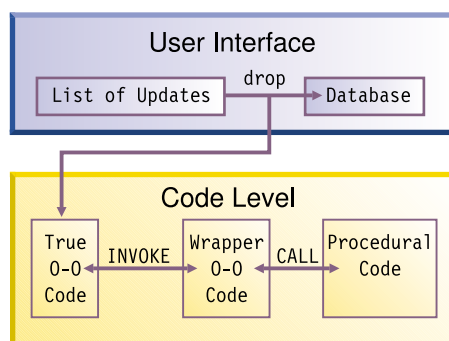
- To enclose as if with a protective covering
- To conceal as if by enveloping

*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:

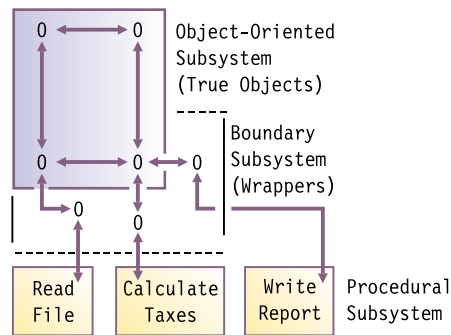


## Integrating procedural code into OO systems

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 454: create a list of objects with instance data for each object.
2. “Analyzing data flow and usage” on page 454: create an object relationship table that lists all the inheritance and collaboration relationships between objects.
3. “Reallocating code to objects” on page 454: complete the method definitions.
4. “Writing the object-oriented code” on page 455: write a class definition and client program.

### RELATED CONCEPTS

“OO view of COBOL programs” on page 451

### RELATED TASKS

- “Defining a class” on page 378
- “Defining a class method” on page 381
- “Defining a client program” on page 389

## 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	WwwwwWork
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.

## 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 454

“Analyzing data flow and usage” on page 454

“Defining a class method” on page 381

## 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 454

“Reallocating code to objects” on page 454

“Defining a class” on page 378

“Defining a client program” on page 389



## Part 5. Working with more complex applications

<b>Chapter 28. Using subprograms</b> . . . . .	459	Prelinking certain DLLs . . . . .	492
Main programs, subprograms, and calls . . . . .	459	Example: sample JCL for a procedural DLL application . . . . .	492
Ending and reentering main programs or subprograms . . . . .	460	Using CALL identifier with DLLs . . . . .	493
Transferring control to another program . . . . .	461	Search order for DLLs in HFS . . . . .	494
Making static calls. . . . .	462	Using DLL linkage and dynamic calls together . . . . .	494
Making dynamic calls . . . . .	462	Using procedure-pointers with DLLs . . . . .	495
Canceling a subprogram. . . . .	463	Calling DLLs from non-DLLs . . . . .	496
When to use a dynamic call . . . . .	463	Example: calling DLLs from non-DLLs . . . . .	496
Performance considerations of static and dynamic calls . . . . .	464	Using COBOL DLLs with C/C++ programs . . . . .	498
Making both static and dynamic calls . . . . .	465	Using DLLs in OO COBOL applications . . . . .	498
Example: static and dynamic CALL statements	465	<b>Chapter 31. Interrupts and checkpoint/restart</b>	501
NODYNAM restriction (CMS only) . . . . .	467	Setting checkpoints . . . . .	501
Example: mixing static and dynamic calls to the same subprogram under CMS . . . . .	467	Designing checkpoints . . . . .	502
Example: mixing static and dynamic calls that can cause a protection exception under CMS . . . . .	468	Testing for a successful checkpoint . . . . .	502
Calling nested COBOL programs . . . . .	468	DD statements for defining checkpoint data sets	503
Nested programs . . . . .	469	Examples: defining checkpoint data sets . . . . .	503
Example: structure of nested programs . . . . .	470	Messages generated during checkpoint . . . . .	504
Scope of names. . . . .	471	Restarting programs . . . . .	504
Making recursive calls . . . . .	472	Requesting automatic restart . . . . .	505
Calling to and from object-oriented programs . . . . .	472	Requesting deferred restart . . . . .	505
Using procedure pointers . . . . .	472	Formats for requesting deferred restart . . . . .	506
Calling a C function-pointer . . . . .	473	Example: requesting a deferred restart . . . . .	506
Calling to alternate entry points . . . . .	473	Resubmitting jobs for restart . . . . .	507
Making programs reentrant . . . . .	474	Example: restarting a job at a specific checkpoint step . . . . .	507
<b>Chapter 29. Sharing data</b> . . . . .	475	Example: requesting a step restart . . . . .	507
Passing data. . . . .	475	Example: resubmitting a job for a step restart	507
Describing arguments in the calling program	477	Example: resubmitting a job for a checkpoint restart . . . . .	508
Describing parameters in the called program	477	<b>Chapter 32. Processing two-digit-year dates</b>	509
Coding the LINKAGE SECTION . . . . .	477	Millennium language extensions (MLE) . . . . .	510
Coding the PROCEDURE DIVISION for passing arguments . . . . .	478	Principles and objectives of these extensions . . . . .	510
Grouping data to be passed . . . . .	478	Resolving date-related logic problems . . . . .	511
Handling null-terminated strings . . . . .	478	Using a century window . . . . .	512
Using pointers to process a chained list . . . . .	479	Example: century window . . . . .	513
Example: using pointers to process a chained list . . . . .	480	Using internal bridging . . . . .	513
Passing return code information . . . . .	482	Example: internal bridging . . . . .	514
Understanding the RETURN-CODE special register . . . . .	482	Moving to full field expansion. . . . .	514
Using PROCEDURE DIVISION RETURNING . . . . .	482	Example: converting files to expanded date form . . . . .	515
Specifying CALL . . . . . RETURNING . . . . .	483	Using year-first, year-only, and year-last date fields	516
Sharing data by using the EXTERNAL clause. . . . .	483	Compatible dates . . . . .	517
Sharing files between programs (external files) . . . . .	483	Example: comparing year-first date fields . . . . .	518
Example: using external files . . . . .	484	Using other date formats . . . . .	518
<b>Chapter 30. Creating a DLL or a DLL application</b>	489	Example: isolating the year. . . . .	518
Dynamic link libraries (DLLs) . . . . .	489	Manipulating literals as dates . . . . .	519
Compiling programs to create DLLs . . . . .	490	Assumed century window . . . . .	520
Linking DLLs . . . . .	491	Treatment of nondates . . . . .	521
		Setting triggers and limits . . . . .	521
		Example: using limits . . . . .	522
		Using sign conditions . . . . .	523
		Sorting and merging by date . . . . .	523
		Example: sorting by date and time . . . . .	524

Performing arithmetic on date fields. . . . .	525
Allowing for overflow from windowed date fields . . . . .	525
Specifying the order of evaluation . . . . .	526
Controlling date processing explicitly . . . . .	527
Using DATEVAL . . . . .	527
Using UNDATE . . . . .	527
Example: DATEVAL . . . . .	528
Example: UNDATE . . . . .	528
Analyzing and avoiding date-related diagnostic messages . . . . .	528
Avoiding problems in processing dates . . . . .	530
Avoiding problems with packed-decimal fields	530
Moving from expanded to windowed date fields	530



---

## Chapter 28. Using subprograms

Many applications consist of several separately compiled programs linked together. A *run unit* (the COBOL term synonymous with *enclave* in Language Environment) includes one or more object programs and can include object programs written in other Language Environment member languages.

Language Environment provides interlanguage support that allows your COBOL for OS/390 & VM programs to call and be called by programs that meet the requirements of Language Environment. Full details on interlanguage communication (ILC) and information on the register conventions required for assembler calls are documented in the Language Environment information.

**Attention:** Do not use program names that start with prefixes used by IBM products. If you try to use programs whose names start with any of the following, your CALL statements might resolve to IBM library or compiler routines rather than to your intended program:

AFB	AFH	CBC	CEE	EDC
IBM	IFY	IGY	IGZ	ILB

### RELATED CONCEPTS

“Main programs, subprograms, and calls”

### RELATED TASKS

“Ending and reentering main programs or subprograms” on page 460

“Transferring control to another program” on page 461

“Making recursive calls” on page 472

“Calling to and from object-oriented programs” on page 472

“Using procedure pointers” on page 472

“Making programs reentrant” on page 474

*Language Environment Writing ILC Applications*

### RELATED REFERENCES

Register conventions (*Language Environment Programming Guide*)

---

## 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”

“Making recursive calls” on page 472

“Transferring control to another program” on page 461

**RELATED REFERENCES**

*Language Environment Programming Guide*

---

## Ending and reentering main programs or subprograms

You can use any of three termination statements in a main program or subprogram, but they have different effects, as shown in the table below:

Termination statement	Main program	Subprogram
EXIT PROGRAM	No action taken.	Return to calling program without ending the run unit. An implicit EXIT PROGRAM statement is generated if the called program has no next executable statement.
STOP RUN	Return to calling program. <sup>1</sup> (Might be the operating system, and job will end.)  STOP RUN terminates the run unit, and deletes all dynamically called programs in the run unit and all programs link-edited with them. (It does not delete the main program.)	Return directly to the program that called the main program. <sup>1</sup> (Might be the operating system, and job will end.)
GOBACK	Return to calling program. <sup>1</sup> (Might be the operating system, and job will end.)  Same effect as STOP RUN.	Return to calling program.
1. If the main program is called by a program written in another language that does not follow Language Environment linkage conventions, return will be to this calling 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 15

RELATED TASKS

“Calling nested COBOL programs” on page 468

“Making recursive calls” on page 472

---

## Transferring control to another program

You can use several different methods to transfer control to another program:

- Static calls
- Dynamic calls
- Calls to nested programs
- Calls to dynamic link libraries (DLLs)

In addition to making calls between COBOL for OS/390 & VM programs, you can also make static and dynamic calls between COBOL for OS/390 & VM, COBOL for MVS & VM, and VS COBOL II programs in all environments (including CICS).

**Exception:** You cannot call VS COBOL II in the OS/390 UNIX environment.

When you want to use OS/VS COBOL with COBOL for OS/390 & VM, there are differences in support between non-CICS and CICS:

### In a non-CICS environment

You can make static and dynamic calls between COBOL for OS/390 & VM, COBOL for MVS & VM, and OS/VS COBOL programs.

**Exception:** You cannot call OS/VS COBOL in the OS/390 UNIX environment.

### In a CICS environment

You must use EXEC CICS LINK to transfer control between OS/VS COBOL programs and COBOL for OS/390 & VM, COBOL for MVS & VM, and OS/VS COBOL programs.

Calls to nested programs allow you to create applications using structured programming techniques. You can use nested programs in place of PERFORM procedures to prevent unintentional modification of data items. Call nested programs using either the CALL *literal* or CALL *identifier* statement.

Calls to dynamic link libraries (DLLs) are an alternative to COBOL dynamic CALL, and are well suited to object-oriented COBOL applications, OS/390 UNIX programs, and applications that interoperate with C/C++

Under OS/390, linking two load modules together results logically in a single program with a primary entry point and an alternate entry point, each with its own name. Each name by which a subprogram is to be dynamically called must be known to the system. You must specify each such name in linkage editor control statements as either a NAME or an ALIAS of the load module that contains the subprogram.

#### RELATED CONCEPTS

“Nested programs” on page 469

#### RELATED TASKS

“Making static calls”

“Making dynamic calls”

“Making both static and dynamic calls” on page 465

“Calling nested COBOL programs” on page 468

## Making static calls

When you use the *CALL literal* statement in a program that is compiled using the NODYNAM and NODLL compiler options, a static call occurs. With these options, all calls of the *CALL literal* format are handled as static calls.

In the static *CALL* statement, the COBOL program and all called programs are part of the same load module. When control is transferred, the called program already resides in storage, and a branch to it takes place. Subsequent executions of the *CALL* statement make the called program available in its last-used state, unless the called program has the INITIAL attribute. In that case, the called program and each program directly or indirectly contained within it are placed into its initial state every time the called program is called within a run unit.

If you specify alternate entry points, a static *CALL* statement can use any alternate entry point to enter the called subprogram.

“Example: static and dynamic *CALL* statements” on page 465

#### RELATED CONCEPTS

“Performance considerations of static and dynamic calls” on page 464

#### RELATED TASKS

“Calling to and from object-oriented programs” on page 472

“Making dynamic calls”

“Making both static and dynamic calls” on page 465

#### RELATED REFERENCES

“NODYNAM restriction (CMS only)” on page 467

“DYNAM” on page 272

“DLL” on page 270

*CALL* statement (*IBM COBOL Language Reference*)

## Making dynamic calls

When you use the *CALL literal* statement in a program that is compiled using the DYNAM and the NODLL compiler options, or when you use the *CALL identifier* statement in a program that is compiled using the NODLL compiler option, a dynamic call occurs. The program name in the PROGRAM-ID paragraph or ENTRY statement must be identical to the corresponding load module name or load module alias of the load module that contains it.

In this form of the *CALL* statement, the called COBOL subprogram is not link-edited with the main program, but is instead link-edited into a separate load module, and, at run time, is loaded only when it is required (that is, when called).

Each subprogram that is called with a dynamic *CALL* statement can be part of a different load module that is a member of either the system link library or a

user-supplied private library. In either case it must be in an OS/390 load library; it cannot reside in the hierarchical file system. When a dynamic CALL statement calls a subprogram that is not resident in storage, the subprogram is loaded from secondary storage into the region or partition containing the main program and a branch to the subprogram is performed.

The first dynamic call to a subprogram within a run unit obtains a fresh copy of the subprogram. Subsequent calls to the same subprogram (by either the original caller or any other subprogram within the same run unit) result in a branch to the same copy of the subprogram in its last-used state, provided the subprogram does not possess the INITIAL attribute. Therefore, the reinitialization of either of the following items is your responsibility:

- GO TO statements that have been altered
- Data items

If the same COBOL program is called under different run units, a separate copy of working storage is allocated for each run unit.

**Restriction:** Dynamic calls are not permitted to:

- COBOL DLL programs
- COBOL programs compiled with the PGMNAME(LONGMIXED) option, unless the program name is less than or equal to eight characters in length and is all uppercase
- COBOL programs compiled with the PGMNAME(LONGUPPER) option, unless the program name is less than or equal to eight characters in length
- More than one entry point in the same COBOL program (unless an intervening CANCEL statement has been executed)

### Canceling a subprogram

When a CANCEL statement is issued for a subprogram, the storage occupied by the subprogram is freed, and a subsequent call to the subprogram functions as though it were the first call. You can cancel a subprogram from a program other than the original caller.

If the called subprogram has more than one entry point, ensure an intervening CANCEL statement is issued before you specify different entry points in the dynamic CALL statement.

After a CANCEL statement is processed for a dynamically called contained program, the program will be in the first-used state. However, the program is not loaded with the initial call, and storage is not freed after the program is canceled.

### When to use a dynamic call

Use a dynamic CALL statement in any of the following circumstances:

- The load module that you want to dynamically call is in an OS/390 load library rather than in the hierarchical file system.
- You are concerned about ease of maintenance. Applications do not have to be link-edited again when dynamically called subprograms are changed.
- The subprograms called are used infrequently or are very large.

If the subprograms are called on only a few conditions, dynamic calls can bring in the subprogram only when needed.

If the subprograms are very large or there are many of them, using static calls might require too much main storage. Less total storage might be required to call and cancel one, then call and cancel another, than to statically call both.

- You want to call subprograms in their unused state, and you cannot use the INITIAL attribute.

When you cannot use the INITIAL attribute to ensure that a subprogram is placed in its unused state each time it is called, you can set the unused state by using a combination of dynamic CALL and CANCEL statements. When you cancel the subprogram that was first called by a COBOL program, the next call will cause the subprogram to be reinitialized to its unused state.

Using the CANCEL statement to explicitly cancel a subprogram that was dynamically loaded and branched to by a non-COBOL program does not result in any action being taken to release the subprogram's storage or to delete the subprogram.

- You have an OS/VS COBOL or other AMODE(24) program in the same run unit with COBOL for OS/390 & VM programs that you want to run in 31-bit addressing mode. OS/VS COBOL, VS COBOL II, COBOL for MVS & VM, and COBOL for OS/390 & VM dynamic call processing include AMODE switching for AMODE(24) programs calling AMODE(31) programs, and vice versa. To have this implicit AMODE switching done, you must use the Language Environment run-time option ALL31(OFF). AMODE switching is not performed when ALL31(ON) is set.

When AMODE switching is performed, control is passed from the caller to a Language Environment library routine. After the switching is performed, control passes to the called program; the save area for the library routine will be positioned between the save area for the caller program and the save area for the called program.

- You do not know the program name to be called until run time. Here, use the format CALL *identifier*, where the *identifier* is a data item that will contain the name of the called program at run time. In terms of practical application, you might use CALL *identifier* when the program to be called is variable, depending on conditional processing in your program. CALL *identifier* is always dynamic, even if you use the NODYNAM compiler option.

“Example: static and dynamic CALL statements” on page 465

#### RELATED CONCEPTS

“Performance considerations of static and dynamic calls”

#### RELATED TASKS

“Making both static and dynamic calls” on page 465

#### RELATED REFERENCES

“NODYNAM restriction (CMS only)” on page 467

“DYNAM” on page 272

CALL statement (*IBM COBOL Language Reference*)

ENTRY statement (*IBM COBOL Language Reference*)

*Language Environment Programming Reference*

## Performance considerations of static and dynamic calls

Because a statically called program is link-edited into the same load module as the calling program, a static call is faster than a dynamic call. A static call is the preferred method if your application does not require the services of the dynamic call.

Statically called programs cannot be deleted (using CANCEL), so static calls might take more main storage. If storage is a concern, think about using dynamic calls. Storage usage of calls depends on whether:

- The subprogram is called only a few times. Regardless of whether it is called, a statically called program is loaded into storage; a dynamically called program is loaded only when it is called.
- You subsequently delete the dynamically called subprogram with a CANCEL statement.

You cannot delete a statically called program, but you can delete a dynamically called program. Using a dynamic call and then a CANCEL statement to delete the dynamically called program after it is no longer needed in the application (and not after each call to it) might require less storage than using a static call.

#### RELATED TASKS

“Making static calls” on page 462

“Making dynamic calls” on page 462

## Making both static and dynamic calls

You can specify both static and dynamic CALL statements in the same program if you compile the program with the NODYNAM compiler option. In this case, with the CALL *literal* statement the called subprogram will be link-edited with the main program into one load module. The CALL *identifier* statement results in the dynamic invocation of a separate load module.

When a dynamic CALL statement and a static CALL statement to the same subprogram are issued within one program, a second copy of the subprogram is loaded into storage. Because this arrangement does not guarantee that the subprogram will be left in its last-used state, results can be unpredictable.

#### RELATED REFERENCES

“NODYNAM restriction (CMS only)” on page 467

“DYNAM” on page 272

## Example: static and dynamic CALL statements

This example has three parts:

- Code that uses a static call to call a subprogram
- Code that uses a dynamic call to call the same subprogram
- The subprogram that is called by the two types of calls

The following example shows how you would code a static call:

```
PROCESS NODYNAM NODLL
IDENTIFICATION DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 RECORD-2                PIC X.                (6)
01 RECORD-1.              (2)
   05 PAY                  PICTURE S9(5)V99.
   05 HOURLY-RATE          PICTURE S9V99.
   05 HOURS                PICTURE S99V9.
. . .
PROCEDURE DIVISION.
CALL "SUBPROG" USING RECORD-1.          (1)
CALL "PAYMASTR" USING RECORD-1 RECORD-2. (5)
STOP RUN.
```

The following example shows how you would code a dynamic call:

```

DATA DIVISION.
WORKING-STORAGE SECTION.
77 PGM-NAME          PICTURE X(8).
01 RECORD-2          PIC x.                (6)
01 RECORD-1.        (2)
   05 PAY            PICTURE S9(5)V99.
   05 HOURLY-RATE    PICTURE S9V99.
   05 HOURS          PICTURE S99V9.
. . .
PROCEDURE DIVISION.
. . .
   MOVE "SUBPROG" TO PGM-NAME.
   CALL PGM-NAME USING RECORD-1.          (1)
   CANCEL PGM-NAME.
   MOVE "PAYMASTR" TO PGM-NAME.          (4)
   CALL PGM-NAME USING RECORD-1 RECORD-2. (5)
   STOP RUN.

```

The following example shows a called subprogram that is called by each of the two preceding calling programs:

```

IDENTIFICATION DIVISION.
PROGRAM-ID. SUBPROG.
DATA DIVISION.
LINKAGE SECTION.
01 PAYREC.                (2)
   10 PAY          PICTURE S9(5)V99.
   10 HOURLY-RATE PICTURE S9V99.
   10 HOURS        PICTURE S99V9.
77 PAY-CODE         PICTURE 9.        (6)
PROCEDURE DIVISION USING PAYREC.    (1)
. . .
   EXIT PROGRAM.                (3)
   ENTRY "PAYMASTR" USING PAYREC PAY-CODE. (5)
. . .
   GOBACK.                        (7)

```

- (1) Processing begins in the calling program. When the first CALL statement is executed, control is transferred to the first statement of the PROCEDURE DIVISION in SUBPROG, which is the called program.

In each of the CALL statements, the operand of the first USING option is identified as RECORD-1.

- (2) When SUBPROG receives control, the values within RECORD-1 are made available to SUBPROG; however, in SUBPROG they are referred to as PAYREC.

The PICTURE character-strings within PAYREC and PAY-CODE contain the same number of characters as RECORD-1 and RECORD-2, although the descriptions are not identical.

- (3) When processing within SUBPROG reaches the EXIT PROGRAM statement, control is returned to the calling program. Processing continues in that program until the second CALL statement is issued.

- (4) In the example of a dynamically called program, because the second CALL statement refers to another entry point within SUBPROG, a CANCEL statement is issued before the second CALL statement.

- (5) With the second CALL statement in the calling program, control is again transferred to SUBPROG, but this time processing begins at the statement following the ENTRY statement in SUBPROG.

- (6) The values within RECORD-1 are again made available to PAYREC. In addition, the value in RECORD-2 is now made available to SUBPROG through the corresponding USING operand PAY-CODE.



When control is transferred the second time from the statically linked program, SUBPROG is made available in its last-used state (that is, if any values in SUBPROG storage were changed during the first execution, those changed values are still in effect). When control is transferred from the dynamically linked program, however, SUBPROG is made available in its initial state, because of the CANCEL statement that has been executed.

- (7) When processing reaches the GOBACK statement, control is returned to the calling program at the statement immediately following the second CALL statement.

In any given execution of the called program and either of the two calling programs, if the values within RECORD-1 are changed between the time of the first CALL and the second, the values passed at the time of the second CALL statement will be the changed, not the original, values. If you want to use the original values, they must be saved.

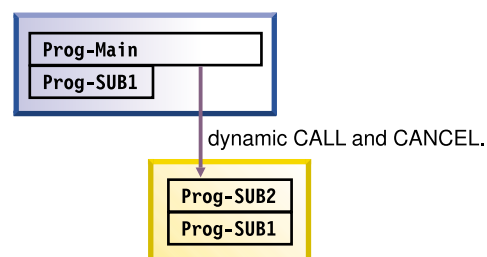
## NODYNAM restriction (CMS only)

The following restriction applies only under CMS if you are using TEXT, TXTLIB, or nonrelocatable modules.

When you use the NODYNAM option, do not mix a dynamic CALL *identifier* and a static CALL *literal* for the same subprogram. Under CMS, certain combinations can cause unpredictable results; some of these combinations are illustrated below.

### Example: mixing static and dynamic calls to the same subprogram under CMS

This example shows a mix of a static CALL with a dynamic CALL and CANCEL of subprograms under CMS. Prog-MAIN and Prog-SUB1 are available as TEXT files and Prog-SUB2 is a load module in a CMS LOADLIB (created with the LKED command) that is link-edited with a copy of Prog-SUB1.



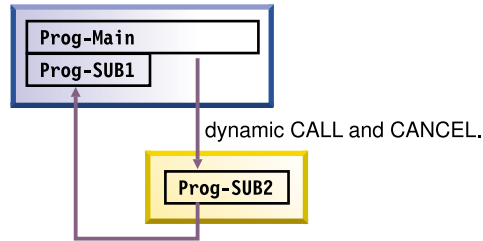
In this example, the following process occurs:

1. Prog-MAIN statically calls Prog-SUB1, which causes it to be loaded with Prog-MAIN.
2. Prog-MAIN dynamically calls Prog-SUB2, which causes a separate load module to be loaded.
3. Prog-SUB2 statically calls Prog-SUB1, which was link-edited with Prog-SUB2.

Now there are two copies of Prog-SUB1. Its last-used state will depend on which program called which version of Prog-SUB1.

## Example: mixing static and dynamic calls that can cause a protection exception under CMS

This example shows a mix of a static CALL with a dynamic CALL and CANCEL of subprograms under CMS that can cause a protection exception. Here all of the programs are available as TEXT files; no load modules are used.



In this example, the following process occurs:

1. Prog-MAIN statically calls Prog-SUB1, which causes it to be loaded with Prog-MAIN.
2. Prog-MAIN dynamically calls Prog-SUB2 but does not also load another copy of Prog-SUB1 because the user did not create a load module with Prog-SUB1 link-edited to Prog-SUB2.
3. Prog-SUB2 statically calls Prog-SUB1, maybe causing the CMS system to use the copy previously loaded for Prog-MAIN, instead of a separate copy.
4. When CMS uses the copy of Prog-SUB1 that was loaded for Prog-MAIN and Prog-MAIN issues a CANCEL for Prog-SUB2, Prog-SUB1 is also deleted.
5. Prog-MAIN again statically calls Prog-SUB1.

The run unit fails because the run unit control blocks still have pointers to Prog-SUB1 (because Prog-MAIN has not yet completed), but Prog-SUB1 is no longer in dynamic storage.

## 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 contained program only from its directly containing program, unless you identify the contained program as **COMMON** in its **PROGRAM-ID** clause. In that case, you can call the *common program* from any program that is contained (directly or indirectly) in the same program as the common program. Only contained 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 contained 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. Contained programs cannot have a **CONFIGURATION SECTION**.

- Include each contained program in the containing program immediately before its End Program header.
- Use an End Program header to terminate contained and containing programs.

RELATED CONCEPTS

“Nested programs”

RELATED REFERENCES

“Scope of names” on page 471

CALL statement (*IBM COBOL Language Reference*)

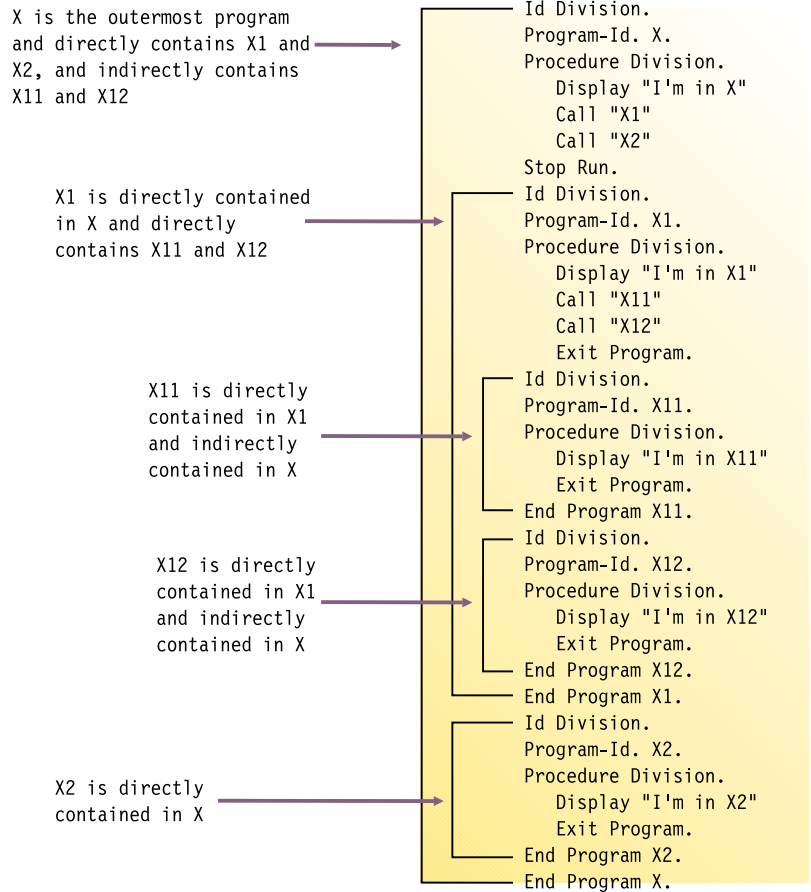
## Nested programs

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

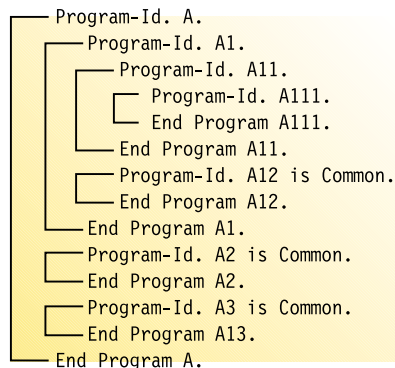
“Calling nested COBOL programs” on page 468

RELATED REFERENCES

“Scope of names” on page 471

**Example: structure of nested programs**

The following example shows a nested structure with some contained 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.

---

## Making recursive calls

A called program can directly or indirectly execute its caller. For example, program X calls program Y, program Y calls program Z, and program Z then calls program X. This type of call is *recursive*.

To make a recursive call, you must code the RECURSIVE clause on the PROGRAM-ID paragraph of the recursively called program. If you try to recursively call a COBOL program that does not have the RECURSIVE clause coded on its PROGRAM-ID paragraph, a condition is signaled. If the condition remains unhandled, the run unit will end.

### RELATED TASKS

“Identifying a program as recursive” on page 6

### RELATED REFERENCES

RECURSIVE attribute (*IBM COBOL Language Reference*)

---

## Calling to and from object-oriented programs

When you create applications that contain object-oriented programs (programs that contain class definitions or INVOKE statements), the object-oriented COBOL programs are COBOL DLL programs and can be in one or more dynamic link libraries (DLLs). Calls to or from COBOL DLL programs must either use DLL linkage or be static calls. COBOL dynamic calls to or from COBOL DLL programs are not supported.

If you must call a COBOL DLL program from a COBOL non-DLL program, other methods, which ensure the DLL linkage mechanism is followed, are available.

---

## Using procedure pointers

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. For example, to have a user-written error-handling routine take control when an exception condition occurs, you must first pass the entry address of the routine to CEEHDLR, a condition management Language Environment callable service, to have it registered.
- 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 will return 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 format-6 of the SET statement. The SET statement sets the procedure pointer to refer either to an entry point in the same load module as your program, to a separate load module, or to

an entry point exported from a DLL, depending on the DYNAM|NODYNAM and DLL|NODLL compiler options. Therefore, think about these factors when using procedure-pointer data items:

- If you compile your program with the NODYNAM and NODLL options and set your procedure-pointer item to a literal value (to an actual name of an entry point), the value must refer to an entry point in the same load module as your program. Otherwise the reference cannot be resolved.
- If you compile your program with the NODLL option and either set your procedure-pointer item to an identifier that will contain the name of the entry point at run time or set your procedure-pointer item to a literal and compile with the DYNAM option, then your procedure-pointer item, whether a literal or variable, must point to an entry point in a separate load module. The entry point can be either the primary entry point or an alternate entry point named in an ALIAS linkage editor statement.
- If you compile with the NODYNAM and DLL options and set your procedure-pointer item to a literal value (the actual name of an entry point), the value must refer to an entry point in the same load module as your program or to an entry point name that is exported from a DLL module. In the latter case you must include the DLL side deck for the target DLL module in the link edit of your program load module.
- If you compile with the NODYNAM and DLL options and set your procedure-pointer item to an identifier (a data item that contains the entry point name at run time), the identifier value must refer to the entry point name that is exported from a DLL module. In this case the DLL module name must match the name of the exported entry point.

If you set your procedure-pointer item to an entry address in a dynamically called load module and your program subsequently cancels that dynamically called module, then your procedure-pointer item becomes undefined. Reference to it thereafter is not reliable.

## Calling a C function-pointer

Many callable services written in C return function-pointers. Your COBOL program can call such a C function-pointer by the following technique:

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. DEMO.  
ENVIRONMENT DIVISION.  
DATA DIVISION.  
*****  
WORKING-STORAGE SECTION.  
01 FP USAGE POINTER  
01 PP USAGE PROCEDURE-POINTER  
*****  
PROCEDURE DIVISION.  
    CALL "c-function" RETURNING FP.  
    SET PP TO FP.  
    CALL PP.  
    . . .
```

## Calling to alternate entry points

You can specify another entry point where the program will begin running by using the ENTRY label in the called program. However, this is not recommended in a structured program.

Static calls to alternate entry points work without restriction.

Dynamic calls to alternate entry points require:

- NAME or ALIAS linkage editor control statements.
- The NAME compiler option.
- An intervening CANCEL for dynamic calls to the same module at differing entry points. CANCEL causes the program to be invoked in initial state when it is called at a new entry point.

“Example: static and dynamic CALL statements” on page 465

#### RELATED TASKS

“Using procedure-pointers with DLLs” on page 495

#### RELATED REFERENCES

“DYNAM” on page 272

“NAME” on page 282

CANCEL statement (*IBM COBOL Language Reference*)

Procedure pointer (*IBM COBOL Language Reference*)

ENTRY statement (*IBM COBOL Language Reference*)

---

## Making programs reentrant

If more than one user will run an application program at the same time (for example, users in different address spaces accessing the same program residing in the link pack area), you must make the program *reentrant* by using the RENT option when you compile it. You do not need to worry about multiple copies of variables. The compiler creates the necessary reentrancy controls in your object module.

The following COBOL for OS/390 & VM programs must be reentrant:

- Programs to be used with CICS
- Programs to be preloaded with IMS
- Programs to be used as DB2 stored procedures
- Programs to be run in the OS/390 UNIX environment

For reentrant programs, use the DATA(24|31) compiler option and the HEAP and ALL31 run-time options to control whether dynamic data areas, such as WORKING-STORAGE, are obtained from storage below 16 MB or from unrestricted storage.

#### RELATED REFERENCES

“RENT” on page 289

“DATA” on page 266

ALL31 run-time option (*Language Environment Programming Reference*)

HEAP run-time option (*Language Environment Programming Reference*)



---

## Chapter 29. 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 477

“Coding the PROCEDURE DIVISION for passing arguments” on page 478

“Passing return code information” on page 482

“Specifying CALL . . . RETURNING” on page 483

“Sharing data by using the EXTERNAL clause” on page 483

“Sharing files between programs (external files)” on page 483

---

## 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 REFERENCE.	To pass data control blocks to assembler programs	Use EXTERNAL or GLOBAL files to share files between COBOL programs.  If the file-name is for a QSAM sequential file, the COBOL compiler passes the address of the data control block (DCB) as this entry of the parameter list. The file-name cannot be a VSAM file-name or a line-sequential file-name. <sup>1</sup>
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	
1. File-names as CALL operands are allowed as an IBM extension to COBOL. Any use of the extension generally depends on the specific internal implementation of the compiler. Control block field settings might change in future releases. Any changes made to the control block are the user's responsibility and not supported by IBM.		

## 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.

Do not pass parameters allocated in storage above 16 MB to AMODE(24) subprograms. Use the DATA(24) option if the RENT option is in effect, or the RMODE(24) option if the NORENT option is in effect.

## 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 483

“Sharing data by using the EXTERNAL clause” on page 483

“Sharing files between programs (external files)” on page 483

### RELATED REFERENCES

CALL statement (*IBM COBOL Language Reference*)

INVOKE statement (*IBM COBOL Language Reference*)

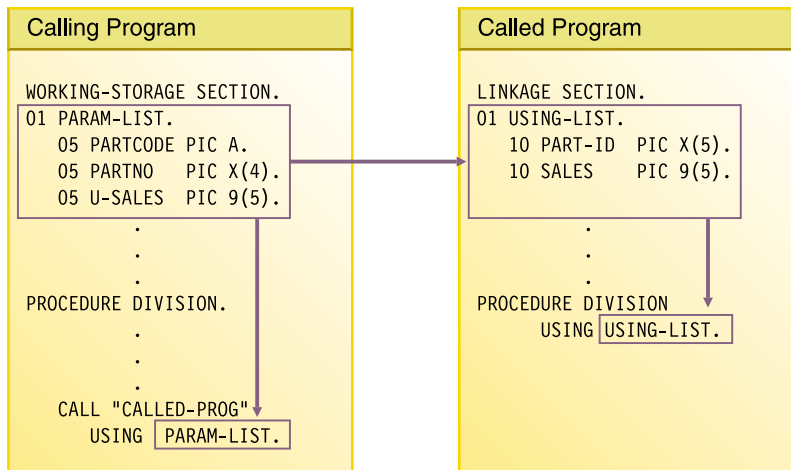
---

## 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 477

## 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 85

#### 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 477

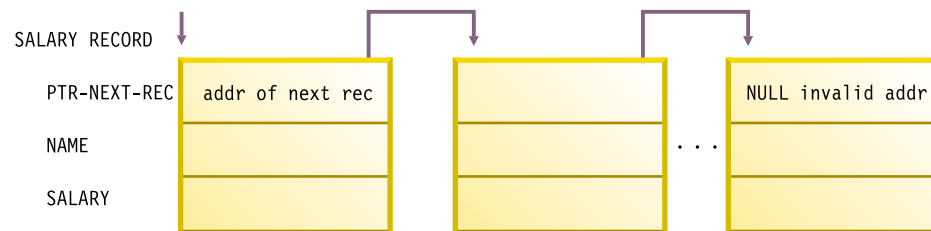
“Coding the PROCEDURE DIVISION for passing arguments” on page 478

**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.           (1)
77 DEPT-TOTAL         PIC 9(4) VALUE IS 0.
*****
LINKAGE SECTION.
01 SALARY-REC.
   02 PTR-NEXT-REC    POINTER.                          (2)
   02 NAME            PIC X(20).
   02 DEPT            PIC 9(4).
   02 SALARY          PIC 9(6).
01 DEPT-X            PIC 9(4).
```

```

*****
PROCEDURE DIVISION USING DEPT-X.
*****
* FOR EVERYONE IN THE DEPARTMENT RECEIVED AS DEPT-X,
* GO THROUGH ALL THE RECORDS IN THE CHAINED LIST BASED ON THE
* ADDRESS OBTAINED FROM THE PROGRAM CHAIN-ANCH
* AND CUMULATE THE SALARIES.
* IN EACH RECORD, PTR-NEXT-REC IS A POINTER TO THE NEXT RECORD
* IN THE LIST; IN THE LAST RECORD, PTR-NEXT-REC IS NULL.
* DISPLAY THE TOTAL.
*****
CALL "CHAIN-ANCH" USING PTR-FIRST                (3)
SET ADDRESS OF SALARY-REC TO PTR-FIRST           (4)
*****
PERFORM WITH TEST BEFORE UNTIL ADDRESS OF SALARY-REC = NULL (5)

    IF DEPT = DEPT-X
        THEN ADD SALARY TO DEPT-TOTAL
        ELSE CONTINUE
    END-IF
    SET ADDRESS OF SALARY-REC TO PTR-NEXT-REC     (6)

END-PERFORM
*****
DISPLAY DEPT-TOTAL
GOBACK.

```

- (1) PTR-FIRST is defined as a pointer data item with an initial value of NULL. On a successful return from the call to CHAIN-ANCH, PTR-FIRST contains the address of the first record in the chained list. If something goes 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 479

## 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 stored into register 15. When control is returned to a COBOL program or method from a call, the contents of register 15 are stored into the RETURN-CODE special register of the calling program or method. When control is returned from a COBOL program to the operating system, the special register contents are returned as a user return code.

You might need to think about this handling of the RETURN-CODE when control is returned to a COBOL program from a non-COBOL program. If the non-COBOL program does not use register 15 to pass back the return code, then the RETURN-CODE special register of the COBOL program might be updated with a value that is not valid. Unless you set this special register to a meaningful value before your COBOL for OS/390 & VM program returns to the operating system, a return code that is not valid will be passed back to the system.

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
ef1openo	Opens the external file for output and checks the file status code
ef1write	Writes a record to the external file and checks the file status code
ef1openi	Opens the external file for input and checks the file status code
ef1read	Reads a record from the external file and checks the file status code
ef1close	Closes the external file and checks the file status code

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 ef1
Assign To ef1
File Status Is efs1
Organization Is Sequential.
```
- The second is named effile and is placed in the FILE SECTION.
 

```
Fd ef1 Is External
      Record Contains 80 Characters
      Recording Mode F.
01 ef-record-1.
02 ef-item-1 Pic X(80).
```
- 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.
    ef1.
*
* 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 ef1.
Identification Division.
Program-Id.
  eflopeno.
*
* This program opens the external file for output.
*
Environment Division.
Input-Output Section.
File-Control.
  Copy efselect.
Data Division.
File Section.
  Copy effile.
Working-Storage Section.
  Copy efwrkstg.
Procedure Division.
  Open Output ef1
  If efs1 Not = 0
    Display "file status " efs1 " on open output"
    Stop Run
  End-If
  Goback.
End Program eflopeno.
Identification Division.
Program-Id.
  eflwrite.
*
* This program writes a record to the external file.
*
Environment Division.
Input-Output Section.
File-Control.
  Copy efselect.
Data Division.
File Section.
  Copy effile.
Working-Storage Section.
  Copy efwrkstg.
Procedure Division.
  Move "First record" to ef-record-1
  Write ef-record-1
  If efs1 Not = 0
    Display "file status " efs1 " on write"
    Stop Run
  End-If
  Goback.
End Program eflwrite.

```

```

Identification Division.
Program-Id.
    eflopeni.
*
* This program opens the external file for input.
*
Environment Division.
Input-Output Section.
File-Control.
    Copy efselect.
Data Division.
File Section.
    Copy effile.
Working-Storage Section.
    Copy efwrkstg.
Procedure Division.
    Open Input efl
    If efs1 Not = 0
        Display "file status " efs1 " on open input"
    Stop Run
    End-If
    Goback.
End Program eflopeni.
Identification Division.
Program-Id.
    eflread.
*
* This program reads a record from the external file.
*
Environment Division.
Input-Output Section.
File-Control.
    Copy efselect.
Data Division.
File Section.
    Copy effile.
Working-Storage Section.
    Copy efwrkstg.
Procedure Division.
    Read efl
    If efs1 Not = 0
        Display "file status " efs1 " on read"
    Stop Run
    End-If
    Goback.
End Program eflread.
Identification Division.
Program-Id.
    eflclose.
*
* This program closes the external file.
*
Environment Division.
Input-Output Section.
File-Control.
    Copy efselect.
Data Division.
File Section.
    Copy effile.
Working-Storage Section.
    Copy efwrkstg.
Procedure Division.
    Close efl
    If efs1 Not = 0
        Display "file status " efs1 " on close"

```

```
    Stop Run  
End-If  
Goback.  
End Program ef1close.
```



---

## Chapter 30. Creating a DLL or a DLL application

Creating a dynamic link library (DLL) or a DLL application is similar to creating a regular COBOL application. It involves writing, compiling, and linking your source code.

Special considerations when writing a DLL or a DLL application include:

- Determining how the parts of the load module or the application relate to each other or to other DLLs
- Deciding what linking or calling mechanisms to use

Depending on whether you want a DLL load module or a load module that references a separate DLL, you need to use slightly different compiling and linking options.

### RELATED CONCEPTS

Dynamic link libraries (DLLs)

### RELATED TASKS

“Using CALL identifier with DLLs” on page 493

“Using DLL linkage and dynamic calls together” on page 494

“Using COBOL DLLs with C/C++ programs” on page 498

“Using DLLs in OO COBOL applications” on page 498

“Compiling programs to create DLLs” on page 490

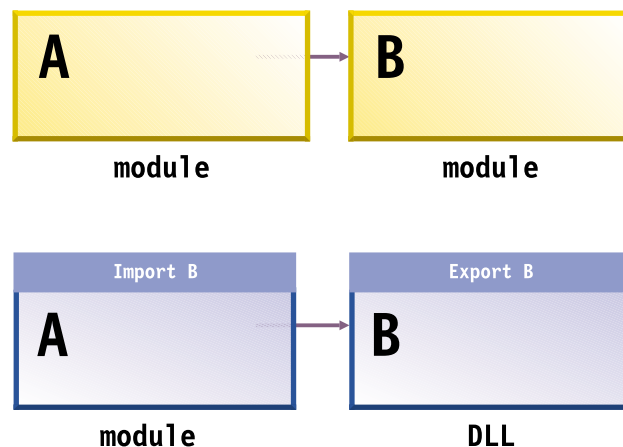
“Linking DLLs” on page 491

“Using procedure pointers” on page 472

---

## Dynamic link libraries (DLLs)

A DLL is a load module or a program object that can be accessed from other separate load modules. A DLL differs from a traditional load module in that it *exports* definitions of programs, functions, or variables to DLLs, DLL applications, or non-DLLs. Therefore, you do not need to link the target routines into the same load module as the referencing routine. When an application references a separate DLL for the first time, the system automatically loads the DLL into memory. In other words, calling a program in a DLL is similar to calling a load module with a dynamic CALL.



A DLL application is an application that references imported definitions of programs, functions, or variables.

Although some functions of OS/390 DLLs overlap the functions provided by COBOL dynamic CALL statements, DLLs have several advantages over regular OS/390 load modules and dynamic calls:

- DLLs are common across COBOL, C, and C++, thus providing better interoperation for applications that use multiple programming languages. Reentrant COBOL and C/C++ DLLs can also interoperate smoothly.
- You can make calls to programs in separate DLL modules that have long program names. (Dynamic call resolution truncates program names to eight characters.) Using the COBOL option PGMNAME(LONGUPPER) or PGMNAME(LONGMIXED) and the COBOL DLL support, you can make calls between load modules with names of up to 160 characters.
- DLLs are the primary means for packaging SOM class libraries, both those provided by IBM and those written by users. The OS/390 SOMobjects product is shipped as a set of DLLs. The COBOL DLL support enables object-oriented COBOL applications to access SOM from the DLLs. You do not have to statically link a copy of the SOM kernel into OO COBOL application load modules.

DLLs are supported by IBM Language Environment, based on function provided by the OS/390 DFSMS program management binder. DLL support is available for applications running under OS/390 in batch or in TSO, CICS, UNIX, or IMS environments. DLL support is not available under VM.

#### RELATED TASKS

Building class libraries (*OS/390 SOMobjects Programmer's Guide*)

#### RELATED REFERENCES

"PGMNAME" on page 287

Binder support for DLLs (*OS/390 DFSMS: Program Management*)

---

## Compiling programs to create DLLs

When you compile a COBOL program with the DLL option, it becomes enabled for DLL support. Applications that use DLL support must be reentrant. Therefore, you must compile them with the RENT compiler option and link them with the RENT binder option.

In an application with DLL support, use the following compiler options depending on where the programs or classes are:

Programs or classes in:	Compile with:
Root load module	DLL, RENT, NOEXPORTALL
DLL load modules used by other load modules	DLL, RENT, EXPORTALL

If a DLL load module includes some programs or classes that are used only from within the DLL module, you can hide these routines inside the DLL by compiling them with NOEXPORTALL.

"Example: sample JCL for a procedural DLL application" on page 492

#### RELATED TASKS

"Linking DLLs" on page 491



“Prelinking certain DLLs” on page 492  
“Chapter 30. Creating a DLL or a DLL application” on page 489

**RELATED REFERENCES**

“DLL” on page 270  
“EXPORTALL” on page 272  
“RENT” on page 289

---

## Linking DLLs

You can link your DLL-enabled object modules into separate DLL load modules, or you can link them together statically. You can decide whether to package the application as one module or as several DLL modules at link time.

Use of the DLL support in the DFSMS binder is recommended for linking DLL applications. The DFSMS binder can now directly receive the output of the COBOL compilers, thus eliminating the prelink step. However, you must use the prelinker if your DLL must reside in a PDS load library or you will deploy the DLL on a CICS system at a level lower than CICS Transaction Server 1.3.

A binder-based DLL must reside in a PDSE or in an HFS file, rather than in a PDS.

When linking a DLL application by using the DFSMS binder, use the following binder options:

Type of code	Link using binder parameters:
DLL applications	DYNAM(DLL), RENT, COMPAT(PM3) or COMPAT(CURRENT)
Applications that use mixed-case exported program names	CASE(MIXED)
Class definitions or INVOKE statements	

You must specify a SYSDEFSD DD statement to indicate the data set where the binder should create a DLL definition side file. This side file contains IMPORT control statements for each symbol exported by a DLL. The binder SYSLIN input (the binding code that references the DLL code) must include the DLL definition side files for the DLLs that are to be referenced from the module being linked.

If there are programs in the module that you do not want to make available with DLL linkage, you can edit the definition side file to remove these programs.

If a DLL must reside in a PDS load library or if the output will be deployed on a CICS system at a level lower than CICS Transaction Server 1.3, you must prelink the application by using the Language Environment prelinker before standard linkage editing.

“Example: sample JCL for a procedural DLL application” on page 492

**RELATED TASKS**

“Chapter 30. Creating a DLL or a DLL application” on page 489  
“Compiling programs to create DLLs” on page 490  
“Prelinking certain DLLs” on page 492  
*OS/390 DFSMS: Program Management*

## Prelinking certain DLLs

You need to use the Language Environment prelinker before standard linkage editing in the following cases:

- The DLL must reside in a PDS load library, rather than in a PDSE or an HFS file.
- The output will be deployed on a CICS system at a level lower than CICS Transaction Server 1.3.

After compiling your source DLL, prelink the object modules to form a single object module:

1. Specify a SYSDEFSD DD statement for the prelink step to indicate the data set where the prelinker should create a DLL definition side file for the DLL. This side file contains `IMPORT` prelinker control statements for each symbol exported by the DLL. The prelinker uses this side file to prelink other modules that reference the new DLL.
2. Specify the `DLLNAME(xxx)` prelinker option to indicate the DLL load module name for the prelinker to use in constructing the `IMPORT` control statements in the side file. Alternatively, the prelinker can obtain the DLL load module name from the `NAME` prelinker control statement or from the PDS member name in the `SYSMOD` DD statement for the prelink step.
3. If this DLL references any other DLLs, include the definition side files for these DLLs together with the object decks that are input to this prelink step. These side files instruct the prelinker to resolve the symbolic references in the current module to the symbols exported from the other DLLs.

Use the linkage editor or binder as usual to create the DLL load module from the object module produced by the prelinker. Specify the `RENT` option of the linkage editor or binder.

“Example: sample JCL for a procedural DLL application”

### RELATED TASKS

“Compiling programs to create DLLs” on page 490

“Linking DLLs” on page 491

## Example: sample JCL for a procedural DLL application

The following sample shows how to create an application that consists of a main program calling a DLL subprogram. The first step creates the DLL load module containing the subprogram `DemoDLLSubprogram`. The second step creates the main load module containing the program `MainProgram`. The third step runs the application.

```
//DILLSAMP JOB ,
// TIME=(1),MSGLEVEL=(1,1),MSGCLASS=H,CLASS=A,
// NOTIFY=&SYSUID,USER=&SYSUID
// SET LEPFX='SYS1'
//*-----
//* Compile COBOL subprogram, bind to form a DLL.
//*-----
//STEP1 EXEC IGYWCL,REGION=80M,GOPGM=DEMOLL,
//      PARM.COBOL='RENT,PGMN(LM),DLL,EXPORTALL',
//      PARM.LKED='RENT,LIST,XREF,LET,MAP,DYNAM(DLL),CASE(MIXED)'
//COBOL.SYSIN DD *
      Identification division.
      Program-id. "DemoDLLSubprogram".
```

```

        Procedure division.
        Display "Hello from DemoDLLSubprogram!".
        End program "DemoDLLSubprogram".
/*
//LKED.SYSDEFSD DD DSN=&&SIDEDECK,UNIT=SYSDA,DISP=(NEW,PASS),
//              SPACE=(TRK,(1,1))
//LKED.SYSLMOD  DD DSN=&&GOSET(&GOPGM),DSNTYPE=LIBRARY,DISP=(MOD,PASS)
//LKED.SYSIN    DD DUMMY
//*-----
//* Compile and bind COBOL main program
//*-----
//STEP2 EXEC IGYWCL,REGION=80M,GOPGM=MAINPGM,
//          PARM.COBOL='RENT,PGMNAME(LM),DLL',
//          PARM.LKED='RENT,LIST,XREF,LET,MAP,DYNAM(DLL),CASE(MIXED)'
//COBOL.SYSIN  DD *
        Identification division.
        Program-id. "MainProgram".
        Procedure division.
        Call "DemoDLLSubprogram"
        Stop Run.
        End program "MainProgram".
/*
//LKED.SYSIN    DD DSN=&&SIDEDECK,DISP=(OLD,DELETE)
//*-----
//* Execute the main program, calling the subprogram DLL.
//*-----
//STEP3 EXEC PGM=MAINPGM,REGION=80M
//STEPLIB DD DSN=&&GOSET,DISP=(OLD,DELETE)
//          DD DSN=&LEPFX..SCEERUN,DISP=SHR
//SYSOUT DD SYSOUT=*
//CEEDUMP DD SYSOUT=*

```

---

## Using CALL identifier with DLLs

In a COBOL program that has been compiled with the DLL option, you can use CALL *identifier* statements as well as CALL *literal* statements to make calls to DLLs. However, there are a few additional considerations for the CALL *identifier* case.

For the contents of the *identifier* or for the literal in the CALL statement, use the name of either of the following programs:

- A nested program in the same compilation unit as is eligible to be called from the program containing the CALL *identifier* statement.
- A program in a separately bound DLL module. The target program name must be exported from the DLL, and the DLL module name must match the exported name of the target program.

In the nonnested case, the run-time environment interprets the program name in the *identifier* according to the setting of the PGMNAME compiler option of the program containing the CALL statement, and interprets the program name that is exported from the target DLL according to the setting of the PGMNAME option used when the target program was compiled.

The search for the target DLL in the hierarchical file system (HFS) is case sensitive. If the target DLL is a PDS or PDSE member, then the DLL member name must be eight characters or less. For the purpose of the search for the DLL as a PDS or PDSE member, the run time automatically converts the name to uppercase.

If the run-time environment cannot resolve the CALL statement in either of these cases, control is transferred to the ON EXCEPTION or ON OVERFLOW phrase of the CALL statement. If the CALL statement does not specify one of these phrases in this situation, Language Environment raises a severity-3 condition.

RELATED TASKS

- “Using DLL linkage and dynamic calls together”
- “Compiling programs to create DLLs” on page 490
- “Linking DLLs” on page 491

RELATED REFERENCES

- “DLL” on page 270
- “PGMNAME” on page 287
- CALL statement (*IBM COBOL Language Reference*)
- “Search order for DLLs in HFS”

## Search order for DLLs in HFS

When you use the hierarchical file system (HFS), the search order for resolving a DLL reference in a CALL statement depends on the setting of the Language Environment POSIX run-time option.

If the POSIX run-time option is set to ON, the search order is as follows:

1. The run-time environment looks for the DLL in the HFS. If the LIBPATH environment variable is set, the run time searches each directory listed. Otherwise, it searches just the current directory. The search for the DLL in the HFS is case sensitive.
2. If the run-time environment does not find the DLL in the HFS, it tries to load the DLL from the MVS load library search order of the caller. In this case, the DLL name must be eight characters or less. The run time automatically converts the DLL name to uppercase for this search.

If the POSIX run-time option is set to OFF, the search order is reversed:

1. The run-time environment tries to load the DLL from the search order for the load library of the caller.
2. If the run-time environment cannot load the DLL from this load library, it tries to load the DLL from the HFS.

RELATED TASKS

- “Using CALL identifier with DLLs” on page 493

RELATED REFERENCES

- POSIX (*Language Environment Programming Reference*)

---

## Using DLL linkage and dynamic calls together

For applications (that is, Language Environment enclaves) that are structured as multiple, separately bound modules, you should use exclusively one form of linkage between modules: either dynamic call linkage or DLL linkage. *DLL linkage* refers to a call in a program that is compiled with the DLL and NODYNAM options where the call resolves to an exported name in a separate module. DLL linkage can also refer to an invocation of a method that is defined in a separate module.

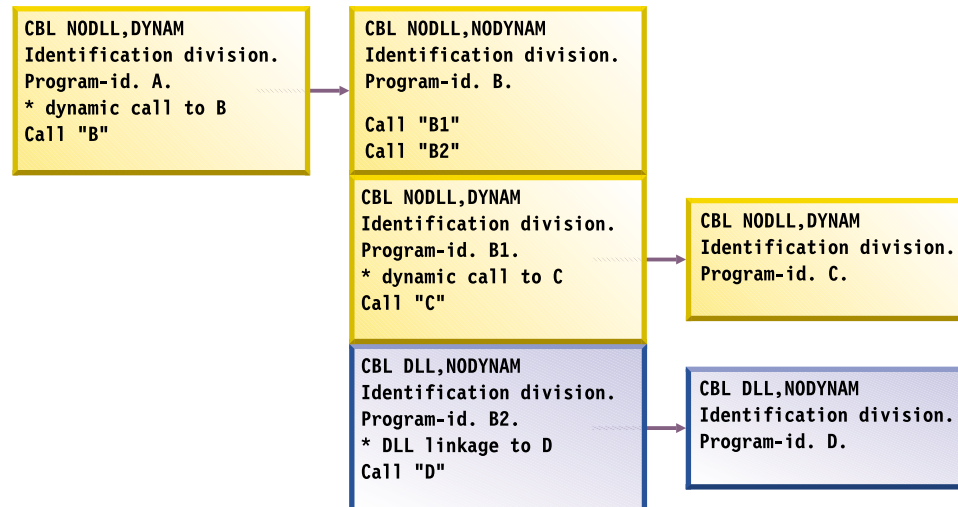
However, some applications require more flexibility. In these cases, you can use both DLL linkage and COBOL dynamic call linkage within a Language Environment enclave if the programs are compiled as follows:

ProgramA	ProgramB	Compile both with:
Contains dynamic call	Target of dynamic call	NODLL
Uses DLL linkage	Contains target program or method	DLL

**Important:** This support is available only in Language Environment V2R9 and higher.

If a program contains a CALL statement for a separately compiled program and you compile one program with the DLL compiler option and the other program with NODLL, then the CALL is supported only if you bind the two programs together in the same module.

The following diagram shows several separately bound modules that mix dynamic calls and DLL linkage:



You cannot cancel programs that are called using DLL linkage.

All components of a DLL application must have the same AMODE. The automatic AMODE switching normally provided by COBOL dynamic calls is not available for DLL linkages.

## Using procedure-pointers with DLLs

In run units that contain a mix of DLLs and non-DLLs, use procedure-pointers with care. When you use the SET *procedure-pointer-1* TO ENTRY *entryname* statement in a program that is compiled with the NODLL option, you must not pass the procedure-pointer to a program that is compiled with the DLL option. However, when you use this statement in a program that is compiled with the DLL option, you can pass the procedure-pointer to a program that is in a separately bound DLL module.

If you compile with the NODYNAM and DLL options and *entryname* is an identifier, the identifier value must refer to the entry point name that is exported from a DLL module. The DLL module name must match the name of the exported entry point. The following are further considerations in this case:

- The program name that is contained in the identifier is interpreted according to the setting of the PGMNAME (COMPAT|LONGUPPER|LONGMIXED) compiler option of the program containing the CALL statement.
- The program name that is exported from the target DLL is interpreted according to the setting of the PGMNAME option used when compiling the target program.
- The search for the target DLL in the HFS is case sensitive.

- If the target DLL is a PDS or PDSE member, then the DLL member name must be eight characters or less. For the purpose of the search for the DLL as a PDS or PDSE member, the name is automatically converted to uppercase.

## Calling DLLs from non-DLLs

It is possible to call a DLL from a COBOL program that is compiled with the NODLL option, but there are restrictions. You can use the following methods to ensure that the DLL linkage is followed:

- Put the COBOL DLL programs that you want to call from the COBOL non-DLL programs in the load module that contains the main program. Use static calls from the COBOL non-DLL programs to call the COBOL DLL programs.  
The COBOL DLL programs in the load module that contains the main program can call COBOL DLL programs in other DLLs.
- Put the COBOL DLL programs in DLLs and call them from COBOL non-DLL programs with `CALL procedure-pointer`, where the `procedure-pointer` is set to a function descriptor of the program to be called. You can obtain the address of the function descriptor for the program in the DLL by calling a C routine that uses `dllload` and `dllqueryfn`.

“Example: calling DLLs from non-DLLs”

### RELATED CONCEPTS

“Dynamic link libraries (DLLs)” on page 489

### RELATED TASKS

“Using CALL identifier with DLLs” on page 493

“Using procedure pointers” on page 472

“Compiling programs to create DLLs” on page 490

“Linking DLLs” on page 491

### RELATED REFERENCES

“DLL” on page 270

“EXPORTALL” on page 272

## Example: calling DLLs from non-DLLs

The following sample code shows how a COBOL program that is not in a DLL (COB0L1) can call a COBOL program that is in a DLL (program ooc05R in DLL OOC05R):

```
CBL NODYNAM
IDENTIFICATION DIVISION.
PROGRAM-ID. 'COB0L1'.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
DATA DIVISION.
FILE SECTION.
WORKING-STORAGE SECTION.
01 DLL-INFO.
   03 DLL-LOADMOD-NAME PIC X(12).
   03 DLL-PROGRAM-NAME PIC X(160).
   03 DLL-PROGRAM-HANDLE POINTER.
77 DLL-RC                PIC S9(9) BINARY.
77 DLL-PROGRAM-PTR      PROCEDURE-POINTER.

77 DLL-STATUS           PIC X(1) VALUE 'N'.
88 DLL-LOADED           VALUE 'Y'.
88 DLL-NOT-LOADED      VALUE 'N'.
```

PROCEDURE DIVISION.

```
        IF DLL-NOT-LOADED
        THEN
*       Move the names in. They must be null terminated.
        MOVE Z'00C05R' TO DLL-LOADMOD-NAME
        MOVE Z'ooc05r' TO DLL-PROGRAM-NAME

*       Call the C routine to load the DLL and to get the
*       function descriptor address.
        CALL 'A1CCDLGT' USING BY REFERENCE DLL-INFO
                        BY REFERENCE DLL-RC

        IF DLL-RC = 0
        THEN
            SET DLL-LOADED TO TRUE
        ELSE
            DISPLAY 'A1CCDLGT failed with rc = '
                DLL-RC
            MOVE 16 TO RETURN-CODE
            STOP RUN
        END-IF
    END-IF

*       Move the function pointer to a procedure pointer
*       so that we can use the call statement to call the
*       program in the DLL.
    SET DLL-PROGRAM-PTR TO DLL-PROGRAM-HANDLE

*       Call the program in the DLL.
    CALL DLL-PROGRAM-PTR

    GOBACK.
```

```
#include <stdio.h>
#include <dll.h>
#pragma linkage (A1CCDLGT,COBOL)

typedef struct dll_lm {
    char    dll_loadmod_name[(12)];
    char    dll_func_name[(160)];
    void    (*fptr) (void); /* function pointer */
} dll_lm;

void A1CCDLGT (dll_lm *dll, int *rc)
{
    dllhandle *handle;
    void (*fptr1)(void);
    *rc = 0;
    /* Load the DLL */
    handle = dllload(dll->dll_loadmod_name);
    if (handle == NULL) {
        perror("A1CCDLGT failed on call to load DLL./n");
        *rc = 1;
        return;
    }

    /* Get the address of the function */
    fptr1 = (void (*)(void))
        dllqueryfn(handle,dll->dll_func_name);
    if (fptr1 == NULL) {
        perror("A1CCDLGT failed on retrieving function./n");
        *rc = 2;
        return;
    }
}
```

```

/* Return the function pointer          */
dll->fptr = fptr1;
return;
}

```

---

## Using COBOL DLLs with C/C++ programs

COBOL support for DLLs interoperates with the DLL support in the OS/390 C/C++ products, except for COBOL EXTERNAL data. COBOL data items that are declared with the EXTERNAL attribute are independent of DLL support; these data items are accessible by name from any COBOL program in the run unit that declares them, regardless of whether the programs are in DLLs or not.

In particular, COBOL applications can call:

- Functions that are exported from C/C++ DLLs
- Methods in SOM class library DLLs that are implemented in C or C++

Similarly, C/C++ applications can call:

- COBOL programs that are exported from COBOL DLLs
- Methods in SOM class library DLLs that are implemented in COBOL

The COBOL options DLL, RENT, and EXPORTALL work much the same way as the DLL, RENT, and EXPORTALL options in C/C++. However, the C++ compiler produces DLL-enabled code by default. The DLL option applies only to C.

You can pass a C/C++ DLL function-pointer to COBOL and use it within COBOL. However, the COBOL application must receive the C/C++ function pointer as a pointer data item rather than as a COBOL procedure pointer, because C/C++ function pointers are 4 bytes long and COBOL procedure pointers are 8 bytes long.

The following example shows a COBOL call to a C function that returns a function-pointer to a service, followed by a COBOL call to the service.

```

Identification Division.
Program-id. Demo.
Data Division.
Working-storage section.
01 fp usage pointer.
01 pp usage procedure-pointer.
Procedure Division.
    Call "c-function" returning fp.
    Set pp to fp.
    Call pp.

```

### RELATED TASKS

“Compiling programs to create DLLs” on page 490

“Linking DLLs” on page 491

### RELATED REFERENCES

“DLL” on page 270

“EXPORTALL” on page 272

“RENT” on page 289

EXTERNAL (*IBM COBOL Language Reference*)

---

## Using DLLs in OO COBOL applications

To fully reuse COBOL classes, typically you should compile each class definition with the DLL, RENT, and EXPORTALL options, and link-edit them into a separate, individual DLL module.



Object-oriented (OO) COBOL applications can access SOM services from the DLLs that are part of the OS/390 SOMobjects product. For each class in a SOM class library DLL, you need to export only the following three SOM external symbols:

- <classname>NewClass
- <classname>ClassData
- <classname>CClassData

You can easily export these symbols by compiling the class definitions with the EXPORTALL option provided by the COBOL, C, or C++ compiler. You do not need to export the individual methods explicitly, regardless of the implementation language (COBOL, C, C++).

Every OO COBOL application accesses the SOMobjects kernel implicitly. Object-oriented COBOL applications can explicitly access SOM services through a user-coded INVOKE or CALL to the SOM facilities.

With OS/390 Release 3 SOMobjects (or later releases), access to the SOMobjects facilities must use the SOMobjects DLLs. You cannot link the SOM object modules into the application load module.

“Example: sample JCL for an object-oriented application” on page 418

#### RELATED TASKS

“Compiling programs to create DLLs” on page 490

“Linking DLLs” on page 491

#### RELATED REFERENCES

“DLL” on page 270

“EXPORTALL” on page 272

“RENT” on page 289



---

## Chapter 31. Interrupts and checkpoint/restart

When programs run for an extended period of time, interruptions might halt processing before the end of a job. The checkpoint/restart functions of OS/390 allow an interrupted program to be restarted at the beginning of a job step or at a checkpoint that you have set.

Because the checkpoint/restart functions cause a lot of extra processing, use them only when you anticipate interruptions caused by machine malfunctions, input or output errors, or intentional operator intervention.

**OS/390 only:** The checkpoint/restart functions are not available under VM/CMS.

The checkpoint routine starts from the COBOL load module containing your program. While your program is running, the checkpoint routine creates records at points you have designated using the COBOL RERUN clause. A checkpoint record contains a snapshot of the information in the registers and main storage when the program reached the checkpoint.

The restart routine restarts an interrupted program. You can perform a restart at any time after the program was interrupted: either immediately (automatic restart), or later (deferred restart).

### RELATED TASKS

“Setting checkpoints”

“Restarting programs” on page 504

“Resubmitting jobs for restart” on page 507

*OS/390 DFSMS: Checkpoint/Restart*

### RELATED REFERENCES

“DD statements for defining checkpoint data sets” on page 503

“Messages generated during checkpoint” on page 504

“Formats for requesting deferred restart” on page 506

---

## Setting checkpoints

To set checkpoints, use job control statements, and use the RERUN clause in the ENVIRONMENT DIVISION. Associate each RERUN clause with a particular COBOL file.

The RERUN clause indicates that a checkpoint record is to be written onto a checkpoint data set whenever a specified number of records in the COBOL file has been processed, or when END OF VOLUME is reached. You cannot use the RERUN clause with files that have been defined with the EXTERNAL attribute.

You can write checkpoint records from several COBOL files onto one checkpoint data set, but you must use a separate data set exclusively for checkpoint records. You cannot embed checkpoint records in one of your program data sets.

A checkpoint data set must have sequential organization. You cannot write checkpoints on VSAM data sets or on data sets allocated to extended-format QSAM data sets. Also, a checkpoint cannot be taken if any program in the run unit has an extended-format QSAM data set open.

Checkpoint records are written on the checkpoint data set defined by a DD statement. In the DD statement, you also choose the checkpoint method:

**Single (store single checkpoints)**

Only one checkpoint record exists at any given time. After the first checkpoint record is written, any succeeding checkpoint record overlays the previous one.

This method is acceptable for most programs. You save space on the checkpoint data set, and you can restart your program at the latest checkpoint.

**Multiple (store multiple contiguous checkpoints)**

Checkpoints are recorded and numbered sequentially. Each checkpoint is saved.

Use this method when you want to restart a program at a checkpoint other than the latest one taken.

You must use the multiple checkpoint method for complete compliance to the COBOL 85 Standard.

Checkpoints during sort operations have the following requirements:

- If checkpoints are to be taken during a sort operation, add a DD statement for SORTCKPT in the job control procedure for execution.
- You can take checkpoint records on ASCII-collated sorts, but the *system-name* indicating the checkpoint data set must not specify an ASCII file.

## Designing checkpoints

Design your checkpoints at critical points in your program so that data can be easily reconstructed. Do not change the contents of files between the time of a checkpoint and the time of the restart.

In a program using disk files, design the program so that you can identify previously processed records. For example, consider a disk file containing loan records that are periodically updated for interest due. If a checkpoint is taken, records are updated, and then the program is interrupted, you would want to test that the records updated after the last checkpoint are not updated again when the program is restarted. To do this, set up a date field in each record, and update the field each time the record is processed. Then, after the restart, test the field to determine whether the record was already processed.

For efficient repositioning of a print file, take checkpoints on the file only after printing the last line of a page.

## Testing for a successful checkpoint

After each input or output statement that issues a checkpoint, the RETURN-CODE special register is updated with the return code from the checkpoint routine. Therefore, you can test whether the checkpoint was successful and decide whether conditions are right to allow a restart. If the return code is greater than 4, an error has occurred in the checkpoint. Check the return code to prevent a restart that could cause incorrect output.

**RELATED TASKS**

“Using checkpoint/restart with DFSORT under OS/390” on page 186

#### RELATED REFERENCES

“DD statements for defining checkpoint data sets”  
Return codes (*OS/390 DFSMS: Checkpoint/Restart*)

## DD statements for defining checkpoint data sets

To define checkpoint data sets, use the following DD statements:

### For tape:

```
//ddname DD DSN= data-set-name ,  
//          [VOLUME=SER=volser ,]UNIT=device-type ,  
//          DISP=( {NEW|MOD} ,PASS)
```

### For direct-access devices:

```
//ddname DD DSN= data-set-name ,  
//          [VOLUME=(PRIVATE,RETAIN,SER=volser) ,]  
//          UNIT=device-type ,SPACE=(subparms) ,  
//          DISP=( {NEW|MOD} ,PASS,KEEP)
```

#### *ddname*

Provides a link to the DD statement. The same as the *ddname* portion of the *assignment-name* used in the COBOL RERUN clause.

#### *data-set-name*

Identifies the checkpoint data set to the restart procedure. The name given to the data set used to record checkpoint records.

*volser* Identifies the volume by serial number.

#### *device-type*

Identifies the device.

#### *subparms*

Specifies the amount of track space needed for the data set.

**MOD** Specifies the multiple contiguous checkpoint method.

**NEW** Specifies the single checkpoint method.

**PASS** Prevents deletion of the data set at successful completion of the job step, unless the job step is the last in the job. If it is the last step, the data set is deleted.

**KEEP** Keeps the data set if the job step abnormally ends.

“Examples: defining checkpoint data sets”

## Examples: defining checkpoint data sets

The following examples show the job control language and COBOL coding you can use to define checkpoint data sets.

### Writing single checkpoint records, using tape:

```
//CHECKPT DD DSN=CHECK1,VOLUME=SER=ND0003,  
//          UNIT=TAPE,DISP=(NEW,KEEP),LABEL=(,NL)
```

```
.....  
ENVIRONMENT DIVISION.
```

```
.....  
RERUN ON CHECKPT EVERY  
5000 RECORDS OF ACCT-FILE.
```

### Writing single checkpoint records, using disk:

```
//CHEK DD DSNAME=CHECK2,
//      VOLUME=(PRIVATE,RETAIN,SER=DB0030),
//      UNIT=3380,DISP=(NEW,KEEP),SPACE=(CYL,5)
      . . .
ENVIRONMENT DIVISION.
      . . .
RERUN ON CHEK EVERY
20000 RECORDS OF PAYCODE.
RERUN ON CHEK EVERY
30000 RECORDS OF IN-FILE.
```

#### Writing multiple contiguous checkpoint records, using tape:

```
//CHEKPT DD DSNAME=CHECK3,VOLUME=SER=111111,
//        UNIT=TAPE,DISP=(MOD,PASS),LABEL=(,NL)
      . . .
ENVIRONMENT DIVISION.
      . . .
RERUN ON CHEKPT EVERY
10000 RECORDS OF PAY-FILE.
```

## Messages generated during checkpoint

The system checkpoint routine advises the operator of the status of the checkpoints taken by displaying informative messages on the console.

Each time a checkpoint has been successfully completed, a message is displayed associating the jobname (*ddname*, *unit*, *volser*) with a *checkid*.

*checkid* is the identification name of the checkpoint taken. The control program assigns *checkid* as an eight-character string. The first character is the letter *C*, followed by a decimal number indicating the checkpoint. For example, the following message indicates the fourth checkpoint taken in the job step:

```
checkid C0000004
```

---

## Restarting programs

The system restart routine does the following:

- Retrieves the information recorded in a checkpoint record
- Restores the contents of main storage and all registers
- Restarts the program

You can begin the restart routine in one of two ways:

- Automatically, at the time an interruption stopped the program
- At a later time, as a deferred restart

The *RD* parameter of the job control language determines the type of restart. You can use the *RD* parameter on either the *JOB* or the *EXEC* statement. If coded on the *JOB* statement, the parameter overrides any *RD* parameters on the *EXEC* statement.

To suppress both restart and writing checkpoints, code *RD=NC*.

**Restriction:** If you try to restart at a checkpoint taken by a COBOL program during a *SORT* or *MERGE* operation, an error message is issued and the restart is canceled. Only checkpoints taken by *DFSORT* are valid.

Data sets that have the *SYSOUT* parameter coded on their *DD* statements are handled in various ways depending on the type of restart.

If the checkpoint data set is multivolume, include in the VOLUME parameter the sequence number of the volume on which the checkpoint entry was written. If the checkpoint data set is on a 7-track tape with nonstandard labels or no labels, the SYSCHK DD statement must contain DCB=(TRTCH=C,. . .).

## Requesting automatic restart

Automatic restart occurs only at the latest checkpoint taken. If no checkpoint was taken before interruption, automatic restart occurs at the beginning of the job step.

Whenever automatic restart is to occur, the system repositions all devices except unit-record devices.

If you want automatic restart, code RD=R or RD=RNC as follows:

- RD=R indicates that restart is to occur at the latest checkpoint. Code the RERUN clause for at least one data set in the program in order to record checkpoints. If no checkpoint is taken before interruption, restart occurs at the beginning of the job step.
- RD=RNC indicates that no checkpoint is to be written, and any restart is to occur at the beginning of the job step. In this case, RERUN clauses are unnecessary; if any are present, they are ignored.

If you omit the RD parameter, the CHKPT macro instruction remains active, and checkpoints can be taken during processing. If an interrupt occurs after the first checkpoint, automatic restart will occur.

To restart automatically, a program must satisfy the following conditions:

- In the program you must request restart by using the RD parameter or by taking a checkpoint.
- An abend that terminated the job must return a code that allows restart.
- The operator must authorize the restart.

“Example: requesting a step restart” on page 507

## Requesting deferred restart

Deferred restart can occur at any checkpoint, not necessarily the latest one taken. You can restart your program at a checkpoint other than at the beginning of the job step.

When a deferred restart has been successfully completed, the system displays a message on the console stating that the job has been restarted. Control is then given to your program.

If you want deferred restart, code the RD parameter as RD=NR. This form of the parameter suppresses automatic restart, but allows a checkpoint record to be written provided that a RERUN clause has been coded.

Request a deferred restart by using the RESTART parameter on the JOB card and a SYSCHK DD statement to identify the checkpoint data set. If a SYSCHK DD statement is present in a job and the JOB statement does not contain the RESTART parameter, the SYSCHK DD statement is ignored. If a RESTART parameter without the CHECKID subparameter is included in a job, a SYSCHK DD statement must not appear before the first EXEC statement for the job.

“Example: restarting a job at a specific checkpoint step” on page 507

RELATED TASKS

“Using checkpoint/restart with DFSORT under OS/390” on page 186

RELATED REFERENCES

“Formats for requesting deferred restart”

## Formats for requesting deferred restart

The formats for the RESTART parameter on the JOB statement and the SYSCHK DD statements are as follows:

```
//jobname JOB MSGLEVEL=1,RESTART=(request[,checkid])
//SYSCHK DD DSN=dataset-name,
//          DISP=OLD[,UNIT=device-type,
//          VOLUME=SER=volser]
```

**MSGLEVEL=1 (or MSGLEVEL=(1,y))**

MSGLEVEL is required.

**RESTART=(request,[checkid])**

Identifies the particular checkpoint at which restart is to occur.

*request*

Takes one of the following forms:

\* Indicates restart at the beginning of the job.

**stepname**

Indicates restart at the beginning of a job step.

**stepname.procstep**

Indicates restart at a procedure step within the job step.

*checkid*

Identifies the checkpoint where restart is to occur.

**SYSCHK** The ddname used to identify a checkpoint data set to the control program. The SYSCHK DD statement must immediately precede the first EXEC statement of the resubmitted job, and must follow any JOBLIB statement.

*data-set-name*

Identifies the checkpoint data set. It must be the same name that was used when the checkpoint was taken.

*device-type and volser*

Identify the device type and the serial number of the volume containing the checkpoint data set.

“Example: requesting a deferred restart”

### Example: requesting a deferred restart

A restart of the GO step of an IGYWCLG procedure, at checkpoint identifier (CHECKID) C0000003, might appear as follows:

```
//jobname JOB MSGLEVEL=1,RESTART=(stepname.GO,C0000003)
//SYSCHK DD DSN=CHKPT,
//          DISP=OLD[,UNIT=3380,VOLUME=SER=111111]
. . .
```



## Resubmitting jobs for restart

When you resubmit a job for restart, be careful with any DD statements that might affect the execution of the restarted job step. The restart routine uses information from DD statements in the resubmitted job to reset files for use after restart. Pay attention to the following:

- If you want a data set to be deleted at the end of a job step, give it a conditional disposition of PASS or KEEP (rather than DELETE) when you run it. This disposition allows the data set to be available if an interruption forces a restart. If you want to restart a job at the beginning of a step, you must first discard any data set created (defined as NEW on a DD statement) in the previous run, or change the DD statement to mark the data set as OLD.
- At restart time, position input data sets on cards as they were at the time of the checkpoint. The system automatically repositions input data sets on tape or disk.

“Example: resubmitting a job for a step restart”

“Example: resubmitting a job for a checkpoint restart” on page 508

## Example: restarting a job at a specific checkpoint step

This example shows a sequence of job control statements for restarting a job at a specific step:

```
//PAYROLL JOB MSGLEVEL=1,REGION=80K,
//          RESTART=(STEP1,CHEKPT4)
//JOBLIB DD DSNAME=PRIV.LIB3,DISP=OLD
//SYSCHK DD DSNAME=CHKPTLIB,
//          [UNIT=TAPE,VOL=SER=456789,]
//          DISP=(OLD,KEEP)
//STEP1 EXEC PGM=PROG4,TIME=5
```

## Example: requesting a step restart

This example shows the use of the RD parameter. Here, the RD parameter requests step restart for any abnormally terminated job step. The DD statement DDCKPNT defines a checkpoint data set. For this step, after a RERUN clause is performed, only automatic checkpoint restart can occur unless a CHKPT cancel is issued.

```
//J1234 JOB 386,SMITH,MSGLEVEL=1,RD=R
//S1 EXEC PGM=MYPROG
//INDATA DD DSNAME=INVENT[,UNIT=TAPE],DISP=OLD,
//          [VOLUME=SER=91468,]
//          LABEL=RETPD=14
//REPORT DD SYSOUT=A
//WORK DD DSNAME=T91468,DISP=(,KEEP),
//          UNIT=SYSDA,SPACE=(3000,(5000,500)),
//          VOLUME=(PRIVATE,RETAIN,,6)
//DDCKPNT DD UNIT=TAPE,DISP=(MOD,PASS,CATLG),
//          DSNAME=C91468,LABEL=(,NL)
```

## Example: resubmitting a job for a step restart

This example shows the changes that you might make to control statements before you resubmit the job for step restart:

- The job name has been changed (from J1234 to J3412) to distinguish the original job from the restarted job.
- The RESTART parameter has been added to the JOB statement, and indicates that restart is to begin with the first job step.
- The WORK DD statement was originally assigned a conditional disposition of KEEP for this data set:

- If the step terminated normally in the previous run of the job, the data set was deleted and no changes need to be made to this statement.
- If the step abnormally terminated, the data set was kept. In that case, define a new data set (S91468 instead of T91468, as shown below), or change the status of the data set to OLD before resubmitting the job.
- A new data set (R91468 instead of C91468) has also been defined as the checkpoint data set.

```
//J3412 JOB 386,SMITH,MSGLEVEL=1,RD=R,RESTART=*
//S1 EXEC PGM=MYPROG
//INDATA DD DSN=INVENT,UNIT=TAPE,DISP=OLD,
// [VOLUME=SER=91468,]LABEL=RETPD=14
//REPORT DD SYSOUT=A
//WORK DD DSN=S91468,
// DISP=(,KEEP),UNIT=SYSDA,
// SPACE=(3000,(5000,500)),
// VOLUME=(PRIVATE,RETAIN,,6)
//DDCHKPNT DD UNIT=TAPE,DISP=(MOD,PASS,CATLG),
// DSN=R91468,LABEL=(,NL)
```

“Example: requesting a step restart” on page 507

## Example: resubmitting a job for a checkpoint restart

This example shows the changes that you might make to control statements before you resubmit a job for checkpoint restart:

- The job name has been changed (from J1234 to J3412) to distinguish the original job from the restarted job.
- The RESTART parameter has been added to the JOB statement, and indicates that restart is to begin with the first step at the checkpoint entry named C0000002.
- The DD statement DDCKPNT was originally assigned a conditional disposition of CATLG for the checkpoint data set:
  - If the step terminated normally in the previous run of the job, the data set was kept. In that case, the SYSCHK DD statement must contain all of the information necessary for retrieving the checkpoint data set.
  - If the job abnormally terminated, the data set was cataloged. In that case, the only parameters required on the SYSCHK DD statement (as shown below) are DSN and DISP.

If a checkpoint is taken in a job that is running when V=R is specified, the job cannot be restarted until adequate nonpageable dynamic storage becomes available.

```
//J3412 JOB 386,SMITH,MSGLEVEL=1,RD=R,
// RESTART=(*,C0000002)
//SYSCHK DD DSN=C91468,DISP=OLD
//S1 EXEC PGM=MYPROG
//INDATA DD DSN=INVENT,UNIT=TAPE,DISP=OLD,
// VOLUME=SER=91468,LABEL=RETPD=14
//REPORT DD SYSOUT=A
//WORK DD DSN=T91468,DISP=(,KEEP),
// UNIT=SYSDA,SPACE=(3000,(5000,500)),
// VOLUME=(PRIVATE,RETAIN,,6)
//DDCKPNT DD UNIT=TAPE,DISP=(MOD,KEEP,CATLG),
// DSN=C91468,LABEL=(,NL)
```

---

## Chapter 32. 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 510

### RELATED TASKS

"Resolving date-related logic problems" on page 511

"Using year-first, year-only, and year-last date fields" on page 516

"Manipulating literals as dates" on page 519

"Setting triggers and limits" on page 521

"Sorting and merging by date" on page 523

"Performing arithmetic on date fields" on page 525

"Controlling date processing explicitly" on page 527

"Analyzing and avoiding date-related diagnostic messages" on page 528

"Avoiding problems in processing dates" on page 530

### RELATED REFERENCES

DATE FORMAT clause (*IBM COBOL Language Reference*)

"DATEPROC" on page 267

"YEARWINDOW" on page 303

---

## Millennium language extensions (MLE)

The term *millennium language extensions* refers to the features of COBOL for OS/390 & VM 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 267

“YEARWINDOW” on page 303

## 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 513

“Example: internal bridging” on page 514

“Example: converting files to expanded date form” on page 515

#### RELATED TASKS

“Using a century window”

“Using internal bridging” on page 513

“Moving to full field expansion” on page 514

## 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 512

“Performing arithmetic on date fields” on page 525

“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 YYXXXX.      (1)
    03  REMINDER-DATE  PIC X(6) DATE FORMAT YYXXXX.
    03  DUE-AMOUNT     PIC S9(5)V99 COMP-3.

FD  OUTPUT-FILE
    RECORDING MODE IS F.
01  OUTPUT-RECORD.
    03  CUST-NAME.
        05  FIRST-NAME  PIC X(10).
        05  LAST-NAME   PIC X(15).
    03  ACCOUNT-NUM    PIC 9(8).
    03  DUE-DATE       PIC X(8) DATE FORMAT YYYYXXXX.    (2)
    03  REMINDER-DATE  PIC X(8) DATE FORMAT YYYYXXXX.
    03  DUE-AMOUNT     PIC S9(5)V99 COMP-3.

WORKING-STORAGE SECTION.

01  INPUT-FILE-STATUS  PIC 99.
01  OUTPUT-FILE-STATUS PIC 99.

PROCEDURE DIVISION.

    OPEN INPUT INPUT-FILE.
    OPEN OUTPUT OUTPUT-FILE.

```

```

READ-RECORD.
  READ INPUT-FILE
    AT END GO TO CLOSE-FILES.
  MOVE CORRESPONDING INPUT-RECORD TO OUTPUT-RECORD.    (3)
  WRITE OUTPUT-RECORD.

  GO TO READ-RECORD.

CLOSE-FILES.
  CLOSE INPUT-FILE.
  CLOSE OUTPUT-FILE.

  EXIT PROGRAM.

END PROGRAM CONVERT.

```

**Notes:**

- (1) The fields DUE-DATE and REMINDER-DATE in the input record are 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 518

RELATED CONCEPTS

“Compatible dates”

RELATED TASKS

“Sorting and merging by date” on page 523

“Using other date formats” on page 518

## 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 yyxxxx.  
     05 Review-Year Pic XX Date Format yy.  
     05 Review-M-D Pic XXXX.
```

- The PROCEDURE DIVISION usage deals with how date fields can be used together in operations such as comparisons, moves, and arithmetic expressions. For year-first and year-only date fields, to be considered compatible, date fields must have the same number of nonyear characters. For example, a field with DATE FORMAT YYXXXX is compatible with another field that has the same date format, and with a YYYYXXXX field, but not with a YYXXX field.

Year-last date fields must have identical DATE FORMAT clauses. In particular, operations between windowed date fields and expanded year-last date fields are not allowed. For example, you can move a date field 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 yyxxxx.  
01 Date-Julian-Win    Pic 9(5) Packed-Decimal Date Format yyxxx.  
01 Date-Gregorian-Exp Pic 9(8) Packed-Decimal Date Format yyyyxxxx.
```

The following statement is inconsistent because the number of nonyear digits is different between the two fields:

```
If Date-Gregorian-Win Less than Date-Julian-Win . . .
```

The following statement is accepted because the number of nonyear digits is the same for both fields:

```
If Date-Gregorian-Win Less than Date-Gregorian-Exp . . .
```

In this case the century window is applied to the windowed date field (Date-Gregorian-Win) to ensure that the comparison is meaningful.

When a nondate is used in conjunction with a date field, the nondate 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 "YYXXXX")  
If Date-Due = Function DATEVAL (20051220 "YYYYXXXX")
```

With a level-88 condition-name, you can specify the THRU option on the VALUE clause, but you must specify a fixed century window on the YEARWINDOW compiler option rather than a sliding window. For example:

```
05 Year-Field Pic 99 Date Format yy.  
88 In-Range Value 98 Thru 06.
```

With this form, the windowed value of the second item 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 521

RELATED TASKS

"Controlling date processing explicitly" on page 527

## 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 527

## 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 523

## 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 yyxxx.
. . .
   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 521

### RELATED TASKS

"Setting triggers and limits" on page 521

"Controlling date processing explicitly" on page 527

---

## 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

*DFSORT Application Programming Guide* (for information on DFSORT and Y2PAST)

#### RELATED REFERENCES

“DATEPROC” on page 267

“YEARWINDOW” on page 303

## 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 513

---

## Controlling date processing explicitly

There may 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 may 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 528

## 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 527

**RELATED REFERENCES**

*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 33. Tuning your program</b> . . . . .	535
Using an optimal programming style . . . . .	535
Using structured programming . . . . .	536
Factoring expressions. . . . .	536
Using symbolic constants . . . . .	536
Grouping constant computations . . . . .	536
Grouping duplicate computations . . . . .	537
Choosing efficient data types . . . . .	537
Computational data items . . . . .	537
Consistent data types. . . . .	538
Arithmetic expressions . . . . .	538
Exponentiations . . . . .	538
Handling tables efficiently . . . . .	539
Optimization of table references . . . . .	540
Optimization of constant and variable items	541
Optimization of duplicate items . . . . .	541
Optimization of variable-length items . . . . .	541
Comparison of direct and relative indexing	541
Optimizing your code . . . . .	542
Optimization . . . . .	542
Contained program procedure integration	543
PERFORM procedure integration. . . . .	543
Example: PERFORM procedure integration . . . . .	544
Choosing compiler features to enhance performance. . . . .	544
Performance-related compiler options . . . . .	545
Evaluating performance . . . . .	548
Running efficiently with CICS, IMS, or VSAM . . . . .	548
CICS . . . . .	548
IMS . . . . .	549
VSAM. . . . .	549
<b>Chapter 34. Simplifying coding.</b> . . . . .	551
Eliminating repetitive coding . . . . .	551
Example: using the COPY statement. . . . .	552
Using Language Environment callable services . . . . .	553
Sample list of Language Environment callable services . . . . .	554
Calling Language Environment services . . . . .	555
Example: Language Environment callable services . . . . .	556



---

## Chapter 33. Tuning your program

When a program is comprehensible, you can assess its performance. A program that has a tangled control flow is difficult to understand and maintain. The tangled control flow also inhibits the optimization of the code. Therefore, before you try to improve the performance directly, you need to assess certain aspects:

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, IMS, or using VSAM: various tips can help make these programs run efficiently.

### RELATED CONCEPTS

“Optimization” on page 542

### RELATED TASKS

“Using an optimal programming style”

“Choosing efficient data types” on page 537

“Handling tables efficiently” on page 539

“Optimizing your code” on page 542

“Choosing compiler features to enhance performance” on page 544

Specifying run-time options (*Language Environment Programming Guide*)

“Running efficiently with CICS, IMS, or VSAM” on page 548

### RELATED REFERENCES

“Performance-related compiler options” on page 545

Storage performance considerations (*Language Environment Programming Guide*)

---

## 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. The code generated for PACKED-DECIMAL data items can be as fast as that for BINARY data items in some cases, especially if the statement is complicated or specifies rounding.

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
- 15 or fewer digits in the PICTURE specification to avoid using library routines for multiplication and division

## 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 36

---

## 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 55

“Choosing efficient data types” on page 537

#### RELATED REFERENCES

“SSRANGE” on page 294

## 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, . . . are constant data items and that V1, V2, . . . are variable data items. Then, for the table element reference ELEMENT(V1 C1 C2) the compiler can eliminate only the individual terms comp\_c1 \* d2 and comp\_c2 \* d3 as constant from the expression:

$$\text{comp\_v1} * \text{d1} + \text{comp\_c1} * \text{d2} + \text{comp\_c2} * \text{d3} + \text{base\_address}$$

However, for the table element reference ELEMENT(C1 C2 V1) the compiler can eliminate the entire subexpression comp\_c1 \* d1 + comp\_c2 \* d2 as constant from the expression:

$$\text{comp\_c1} * \text{d1} + \text{comp\_c2} * \text{d2} + \text{comp\_v1} * \text{d3} + \text{base\_address}$$

In the table element reference ELEMENT(C1 C2 C3) all the subscripts are constant, and so no subscript computation is done at run time. The expression is:

$$\text{comp\_c1} * \text{d1} + \text{comp\_c2} * \text{d2} + \text{comp\_c3} * \text{d3} + \text{base\_address}$$

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 ELEMENT(V1 V3 V4) and ELEMENT(V2 V3 V4) only the individual terms comp\_v3 \* d2 and comp\_v4 \* d3 are common subexpressions in the expressions needed to reference the table elements:

$$\begin{aligned} &\text{comp\_v1} * \text{d1} + \text{comp\_v3} * \text{d2} + \text{comp\_v4} * \text{d3} + \text{base\_address} \\ &\text{comp\_v2} * \text{d1} + \text{comp\_v3} * \text{d2} + \text{comp\_v4} * \text{d3} + \text{base\_address} \end{aligned}$$

However, for the two table element references ELEMENT(V1 V2 V3) and ELEMENT(V1 V2 V4) the entire subexpression comp\_v1 \* d1 + comp\_v2 \* d2 is common between the two expressions needed to reference the table elements:

$$\begin{aligned} &\text{comp\_v1} * \text{d1} + \text{comp\_v2} * \text{d2} + \text{comp\_v3} * \text{d3} + \text{base\_address} \\ &\text{comp\_v1} * \text{d1} + \text{comp\_v2} * \text{d2} + \text{comp\_v4} * \text{d3} + \text{base\_address} \end{aligned}$$

In the two references ELEMENT(V1 V2 V3) and ELEMENT(V1 V2 V3), the expressions are the same:

$$\begin{aligned} &\text{comp\_v1} * \text{d1} + \text{comp\_v2} * \text{d2} + \text{comp\_v3} * \text{d3} + \text{base\_address} \\ &\text{comp\_v1} * \text{d1} + \text{comp\_v2} * \text{d2} + \text{comp\_v3} * \text{d3} + \text{base\_address} \end{aligned}$$

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 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 539

---

## 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 281  
"OPTIMIZE" on page 285

## 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 both a PERFORM statement and 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.
- Delete from the program, and identify with a warning message, code that can never be performed (unreachable code elimination).
- 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.

### **PERFORM procedure integration**

PERFORM procedure integration is the process whereby a PERFORM statement is replaced by its performed procedures. The advantage is that the resulting program runs faster without the overhead of PERFORM linkage and with more linear control flow.

*Program size:* If the performed procedures are invoked by several PERFORM statements and replace each such statement, the program could become large. The optimizer limits this increase to no more than 50 percent, after which it no longer integrates these procedures. If you are concerned about program size, you can prevent procedure integration in specific instances by using a priority number on section names.

If you do not want a PERFORM statement to be replaced by its performed procedures, put the PERFORM statement in one section and put the performed procedures in another section with a different priority number. The optimizer then chooses the next best optimization for the PERFORM statement; the linkage overhead can be as few as two instructions.

*Unreachable code:* Because of procedure integration, one PERFORM procedure might be repeated several times. As further optimization proceeds on each copy of the procedure, portions might be found to be unreachable, depending on the context into which the code was copied

“Example: PERFORM procedure integration” on page 544

RELATED CONCEPTS

“Optimization of table references” on page 540

RELATED REFERENCES

“OPTIMIZE” on page 285

## Example: PERFORM procedure integration

All the PERFORM statements in the following program will be transformed by procedure integration:

```
1 SECTION 5.
11. PERFORM 12
    STOP RUN.
12. PERFORM 21
    PERFORM 21.
2 SECTION 5.
21. IF A < 5 THEN
    ADD 1 TO A
    DISPLAY A
    END-IF.
```

The program will be compiled as if it had originally been written as follows:

```
1 SECTION 5.
11.
12. IF A < 5 THEN
    ADD 1 TO A
    DISPLAY A
    END-IF.
    IF A < 5 THEN
    ADD 1 TO A
    DISPLAY A
    END-IF.
    STOP RUN.
```

By contrast, in the following program only the first PERFORM statement, PERFORM 12, will be optimized by procedure integration:

```
1 SECTION.
11. PERFORM 12
    STOP RUN.
12. PERFORM 21
    PERFORM 21.
2 SECTION 5.
21. IF A < 5 THEN
    ADD 1 TO A
    DISPLAY A
    END-IF.
```

RELATED CONCEPTS

“Optimization of table references” on page 540

RELATED TASKS

“Optimizing your code” on page 542

“Chapter 33. Tuning your program” on page 535

---

## 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.

Although COBOL allows segmentation language, you will not improve storage allocation by using it, because COBOL does not perform overlay.

RELATED CONCEPTS

“Optimization” on page 542

RELATED TASKS

“Optimizing your code” on page 542

“Getting listings” on page 319

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
("AW0" on page 263)	To get optimum use of buffer and device space	Can result in performance savings, because this option results in fewer calls to data management services to handle input and output	In general, none	When you use AW0, the APPLY WRITE-ONLY clause is in effect for all files in the program that are physical sequential with V-mode records.
DATA(31) (see ("DATA" on page 266))	To have DFSMS allocate QSAM buffers in unrestricted storage (by using the RENT and DATA(31) compiler options)	Because extended-format QSAM data sets can require many buffers, allocating the buffers in unrestricted storage avoids virtual storage constraint problems.	In general, none	On an OS/390 system with DFSMS, if your application processes striped extended-format QSAM data sets, use the RENT and DATA(31) compiler options to have the input-output buffers for your QSAM files allocated from unrestricted storage above the 16-MB line.

Compiler option	Purpose	Performance advantages	Performance disadvantages	Usage notes
("DYNAM" on page 272)	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 Language Environment routine.	To free virtual storage that is no longer needed, issue the CANCEL statement.
("FASTSORT" on page 273)	To specify that the IBM DFSORT product (or equivalent) will handle all of the input and output	Eliminates the overhead of returning to COBOL for OS/390 & VM after each record is processed	None	FASTSORT is recommended when direct work files are used for the sort work files. Not all sorts are eligible for this option.
NUMPROC (PFD) (see ("NUMPROC" on page 283))	To have invalid sign processing bypassed for numeric operations	Generates significantly more efficient code for numeric comparisons	For most references to COMP-3 and DISPLAY numeric data items, NUMPROC (PFD) inhibits extra code from being generated to "fix up" signs. This extra code might also inhibit some other types of optimizations. The extra code is generated with NUMPROC (MIG) and NUMPROC (NOPFD).	When you use NUMPROC (PFD), the compiler assumes that the data has the correct sign and bypasses the sign "fix up" process. Because not all external data files contain the proper sign for COMP-3 or DISPLAY signed numeric data, NUMPROC (PFD) might not be applicable for all programs. For performance-sensitive applications, NUMPROC (PFD) is recommended.
("OPTIMIZE" on page 285)	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.
("RENT" on page 289)	To generate a reentrant program	Enables the program to be placed in shared storage (LPA/ELPA) for faster execution	Generates additional code to ensure that the program is reentrant.	
RMODE (ANY) (see ("RMODE" on page 290))	To let the program be loaded anywhere	RMODE (ANY) with NORENT lets the program and its WORKING-STORAGE be located above the 16-MB line, relieving storage below the line.	In general, none	



Compiler option	Purpose	Performance advantages	Performance disadvantages	Usage notes
NOSSRANGE (see ("SSRANGE" on page 294))	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.
TEST(NONE) or NOTEST (see ("TEST" on page 295))	To avoid the additional object code that would be produced to take full advantage of Debug Tool, use TEST(NONE) or NOTEST. Additionally, when using TEST(NONE), you can use the SEPARATE suboption of TEST option to further reduce the size of your object code..	Because the TEST compiler option with any hook-location suboption other than NONE (that is, ALL, STMT, PATH, BLOCK) generates additional code, it can cause significant performance degradation when used in a production environment. The more compiled-in hooks you specify, the more additional code is generated and the greater performance degradation might be.	None	TEST without a hook-location suboption or with any one other than NONE forces the NOOPTIMIZE compiler option into effect. For production runs, using NOTEST or TEST(NONE,SYM) with or without the SEPARATE suboption, is recommended. This results in overlay hooks rather than compiled-in hooks.  If during the production run, you want a symbolic dump of the variables in a formatted dump when the program abends, compile with TEST(NONE,SYM) with or without the SEPARATE suboption.
TRUNC(OPT) (see ("TRUNC" on page 297))	To avoid having code generated to truncate the receiving fields of arithmetic operations	Does not generate extra code and generally improves performance	Both TRUNC(BIN) and TRUNC(STD) generate extra code whenever a BINARY data item is changed. TRUNC(BIN) is usually the slowest of these options.	TRUNC(STD) conforms to the 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 542

#### RELATED TASKS

"Generating a list of compiler error messages" on page 230

"Evaluating performance" on page 548

"Optimizing buffer and device space" on page 12

"Choosing compiler features to enhance performance" on page 544

"Improving sort performance with FASTSORT" on page 181

"Using striped extended-format QSAM data sets" on page 126

"Handling tables efficiently" on page 539

RELATED REFERENCES

“Sign representation and processing” on page 41

## 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?
AWO	Do you use the AWO option when possible?	
DATA	When you use QSAM striped data sets, do you use the DATA(31) option?	
DYNAM	Do you use NODYNAM? Consider the performance tradeoffs.	
FASTSRT	When you use direct work files for the sort work files, have you selected the FASTSRT option?	
NUMPROC	Do you use NUMPROC(PFD) when possible?	
OPTIMIZE	Do you use OPTIMIZE for production runs? Can you use OPTIMIZE(FULL)?	
RENT	Do you use NORENT? Consider the performance tradeoffs.	
RMODE(ANY)	Do you use RMODE(ANY) with your NORENT programs? Consider the performance tradeoffs with storage usage.	
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 544

RELATED REFERENCES

“Performance-related compiler options” on page 545

---

## Running efficiently with CICS, IMS, or VSAM

You can improve performance for online programs running under CICS or IMS, or programs that use VSAM, by following these tips:

### CICS

If your application runs under CICS, convert EXEC CICS LINKs to COBOL CALLs to improve transaction response time.

## IMS

If your application is running under IMS, preloading the application program and the library routines can help to reduce the overhead of load and search. It can also reduce the input-output activity.

For better system performance, use the RENT compiler option and preload the applications and library routines when possible.

You can also use the library routine retention (LRR) function to improve performance in IMS/TM regions.

## VSAM

When you use VSAM files, increase the number of data buffers for sequential access or index buffers for random access. Also, select a control interval size (CISZ) appropriate for the application. Smaller CISZ results in faster retrieval for the random processing at the expense of inserts, and a larger CISZ is more efficient for sequential processing.

For better performance, access the records sequentially and avoid using multiple alternate indexes when possible. If you do use alternate indexes, access method services builds them more efficiently than the AIXBLD run-time option.

### RELATED TASKS

“Improving VSAM performance” on page 159  
*Language Environment for OS/390 Customization*

### RELATED REFERENCES

*Language Environment Programming Guide*



---

## Chapter 34. Simplifying coding

This material provides techniques for improving your productivity. By using the COPY statement, COBOL intrinsic functions, and Language Environment 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 Language Environment 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 44

“Math and date Language Environment services” on page 45

### RELATED TASKS

“Eliminating repetitive coding”

“Converting data items (intrinsic functions)” on page 94

“Evaluating data items (intrinsic functions)” on page 96

“Using Language Environment callable services” on page 553

---

## 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, use either multiple system definitions or a combination of multiple definitions and the IN/OF phrase (*IN/OF library-name*):

### OS/390 batch

Use JCL to concatenate data sets on your SYSLIB DD statement.

Alternatively, define multiple DD statements and use the IN/OF phrase of the COPY statement.

**TSO** Use the ALLOCATE command to concatenate data sets for SYSLIB.

Alternatively, issue multiple ALLOCATE statements and use the IN/OF phrase of the COPY statement.

## OS/390 UNIX

Use the SYSLIB environment variable to define multiple paths to your copy files. Alternatively, use multiple environment variables and use the IN/OF phrase of the COPY statement.

**CMS** Use the GLOBAL command to concatenate libraries for SYSLIB. Alternatively, issue multiple FILEDEF statements and use the IN/OF phrase of the COPY statement.

For example:

```
COPY MEMBER1 OF COPYLIB
```

If you omit this qualifying phrase, the default is SYSLIB.

**COPY and debugging line:** In order for the text copied to be treated as debug lines, for example, as if there were a D inserted in column 7, put the D on the first line of the COPY statement. A COPY statement 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 304

## 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 551

“Using compiler-directing statements” on page 245

#### RELATED REFERENCES

“Compiler-directing statements” on page 304

---

## Using Language Environment callable services

Language Environment callable services make many types of programming tasks easier. Called with standard COBOL CALL statements, Language Environment services help you with the following tasks:

- Handling conditions

The Language Environment condition-handling facilities allow COBOL applications to react to unexpected errors. You can use language constructs or run-time options to select the level at which you want to handle each condition. For example, you can decide to handle a particular error in your COBOL program, let Language Environment take care of it, or have the operating system handle it.

In support of Language Environment condition handling, COBOL provides procedure-pointer data items.

- Managing dynamic storage

These services enable you to get, free, and reallocate storage. In addition, you can create your own storage pools.

- Calculating dates and times

With the date and time services, you can get the current local time and date in several formats, and perform date and time conversions. Two callable services, CEEQCEN and CEESCEN, provide a predictable way to handle two-digit years, such as 91 for 1991 or 02 for 2002.

- Making math calculations

Calculations that are easy to perform with mathematical callable services include logarithmic, exponential, trigonometric, square root, and integer functions.

COBOL also supports a set of intrinsic functions that include some of the same mathematical and date functions as those provided by the callable services. The Language Environment callable services and intrinsic functions provide equivalent results, with few exceptions. You should be familiar with these differences before deciding which to use.

- Handling messages

Message-handling services include getting, dispatching, and formatting messages. Messages for non-CICS applications can be directed to files or printers; CICS messages are directed to a CICS transient data queue. Language Environment splits messages to accommodate the record length of the destination, and presents messages in the correct national language such as Japanese or English.

- Supporting national languages

These services make it easy for your applications to support the language desired by application users. You can set the language and country, and obtain default date, time, number, and currency formats. For example, you might want dates to appear as 23 June 00 or as 6,23,00.

- General services such as starting Debug Tool and obtaining a Language Environment formatted dump

Debug Tool provides advanced debugging functions for COBOL applications, including both batch and interactive debugging of COBOL-CICS programs. Debug Tool enables you to debug a COBOL application from the host or, in conjunction with VisualAge COBOL Version 2.0 (or higher), from the workstation.

Depending on the options that you select, the Language Environment formatted dump might contain the names and values of variables, and information about conditions, program tracebacks, control blocks, storage, and files. All Language Environment dumps have a common, well-labeled, and easy-to-read format.

“Example: Language Environment callable services” on page 556

#### RELATED CONCEPTS

“Sample list of Language Environment callable services”

“Numeric intrinsic functions” on page 44

“Math and date Language Environment services” on page 45

#### RELATED TASKS

“Calling Language Environment services” on page 555

“Using procedure pointers” on page 472

## Sample list of Language Environment callable services

The following table gives some examples of the callable services available with Language Environment. Many more services are available than those listed.

Function type	Service	Purpose
Condition-handling services	CEEHDLR	To register a user condition handler
	CEESGL	To raise or signal a condition
	CEEMRCR	To indicate where the program will resume running after the condition handler has completed
Dynamic storage services	CEEGTST	To get storage
	CEECZST	To change the size of a previously allocated storage block
	CEEFRST	To free storage
Date and time	CEECBLDY	To convert a string that represents a date into COBOL integer date format, which represents a date as the number of days since 31 December 1600
	CEEQCEN, CEESCEN	To query and set the Language Environment century window, which is valuable when a program uses two digits to express a year
	CEEGMTO	To calculate the difference between the local system time and Greenwich Mean Time
	CEELOCT	To get the current local time in your choice of three formats



Function type	Service	Purpose
Math services	CEESIABS	To calculate the absolute value of an integer
	CEESSNWN	To calculate the nearest whole number for a single-precision floating-point number
	CEESSCOS	To calculate the cosine of an angle
Message-handling services	CEEMOUT	To dispatch a message
	CEEMGET	To retrieve a message
National language support services	CEE3LNG	To change or query the current national language
	CEE3CTY	To change or query the current national country
	CEE3MCS	To obtain the default currency symbol for a given country
General services	CEE3DMP	To obtain a Language Environment formatted dump
	CEETEST	To start a debugging tool, such as Debug Tool

#### RELATED REFERENCES

*Language Environment Programming Reference*

## Calling Language Environment services

To invoke a Language Environment service, use a CALL statement with the correct parameters for that particular service:

```
CALL "CESSSQT" using argument, feedback-code, result
```

Define the variables for the CALL statement in the DATA DIVISION with the definitions required by the function you are calling:

```
77 argument          comp-1.
77 feedback-code    pic x(12) display.
77 result           comp-1.
```

In this example, Language Environment service CESSSQT calculates the value of the square root of the variable argument and returns this value in the variable result.

You can choose whether you want to specify the feedback code parameter. If you specify the feedback code, the value returned in feedback-code indicates whether the service completed successfully. If you specify OMITTED instead of the feedback code and the service is not successful, a Language Environment condition is automatically signaled to the Language Environment condition manager. You can handle such a condition by recovery logic implemented in a user-written condition handler, or allow the default Language Environment processing for unhandled conditions to occur. In any case, this avoids the requirement to write logic to check the feedback code explicitly after each call.

When you call a Language Environment callable service and specify OMITTED for the feedback code, the RETURN-CODE special register is set to 0 if the service is successful. It is not altered if the service is unsuccessful. If you do not specify OMITTED for the feedback code, the RETURN-CODE special register is always set to 0 regardless of whether the service completed successfully.

“Example: Language Environment callable services” on page 556

RELATED CONCEPTS

General callable services (*Language Environment Programming Guide*)

RELATED REFERENCES

Language Environment callable services (*Language Environment Programming Reference*)

CALL statement (*IBM COBOL Language Reference*)

## Example: Language Environment callable services

Many callable services offer you entirely new function that would require extensive coding using previous versions of COBOL. This example shows a sample COBOL program that uses Language Environment services CEEDAYS and CEEDATE to format and display a date from the results of a COBOL ACCEPT statement. Using CEEDAYS and CEEDATE reduces the code that would be required without Language Environment.

```
ID DIVISION.
PROGRAM-ID. HOHOHO.
*****
* FUNCTION:  DISPLAY TODAY'S DATE IN THE FOLLOWING FORMAT: *
*           WWWWWWWW, MMMMMMM DD, YYYY                  *
*           For example: MONDAY, MARCH 13, 2000         *
*           *                                           *
*****
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 LE DATE/TIME CALLABLE SERVICES TO PRINT OUT      *
*   TODAY'S DATE FROM COBOL ACCEPT STATEMENT.           *
*****
ACCEPT CHRDATE-STRING FROM DATE.
*
MOVE "YYMMDD" TO PICSTR-STRING.
MOVE 6 TO PICSTR-LENGTH.
CALL "CEEDAYS" USING CHRDATE , PICSTR , LILIAN , OMITTED.
*
MOVE " WWWWWWWWZ, MMMMMMMMZ 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.
```

---

## Part 7. Appendixes



---

## Appendix A. 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 561

The precision of intermediate results depends on whether you compile using the default option ARITH(COMPAT) (referred to as *compatibility mode*), or using ARITH(EXTEND) (referred to as *extended mode*), explained below.

In compatibility mode, evaluation of arithmetic operations is unchanged from that in earlier releases of IBM COBOL:

- A maximum of 30 digits is used for fixed-point intermediate results.
- Floating-point intrinsic functions return long-precision (64-bit) floating-point results.
- Expressions containing floating-point operands, fractional exponents, or floating-point intrinsic functions are evaluated as if all operands that aren't in floating point are converted to long-precision floating point and floating-point operations are used to evaluate the expression.
- Floating-point literals and external floating-point data items are converted to long-precision floating point for processing.

In extended mode, evaluation of arithmetic operations has the following characteristics:

- A maximum of 31 digits is used for fixed-point intermediate results.
- Floating-point intrinsic functions return extended-precision (128-bit) floating-point results.
- Expressions containing floating-point operands, fractional exponents, or floating-point intrinsic functions are evaluated as if all operands that aren't in floating point are converted to extended-precision floating point and floating-point operations are used to evaluate the expression.
- Floating-point literals and external floating-point data items are converted to extended-precision floating point for processing.

Further details are provided in the references below.

#### RELATED CONCEPTS

“Formats for numeric data” on page 36

“Fixed-point versus floating-point arithmetic” on page 49

#### RELATED REFERENCES

“Fixed-point data and intermediate results” on page 561

“Floating-point data and intermediate results” on page 566

“ON SIZE ERROR and intermediate results” on page 567

“Arithmetic expressions in nonarithmetic statements” on page 568

“ARITH” on page 262

---

## 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 *ir1*.
2. Multiply B by C yielding *ir2*.
3. Divide E into D yielding *ir3*.
4. Add A to *ir2* yielding *ir4*.
5. Subtract *ir3* from *ir4* yielding *ir5*.
6. Add *ir5* to *ir1* yielding Y.

### RELATED CONCEPTS

“Arithmetic expressions” on page 43

### RELATED REFERENCES

“Terminology used for intermediate results” on page 560

---

## 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	$d1$ or $d2$ , whichever is greater
*	$i1 + i2$	$d1 + d2$
/	$i2 + d1$	$(d2 - d1)$ or $dmax$ , whichever is greater

You must define the operands of any arithmetic statements with enough decimal places to obtain the accuracy you want in the final result.

The following table shows the number of places the compiler carries for fixed-point intermediate results of arithmetic operations involving addition, subtraction, multiplication, or division in *compatibility mode* (that is, when you compile using the default compiler option ARITH(COMPAT)):

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

Value of $i + d$	Value of $d$	Value of $i + d_{max}$	Number of places carried for $ir$
>30	< $d_{max}$ = $d_{max}$	Any value	30- $d$ integer and $d$ decimal places
	> $d_{max}$	<30 =30	$i$ integer and 30- $i$ decimal places
		>30	30- $d_{max}$ integer and $d_{max}$ decimal places

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 in *extended mode* (that is, when you compile using option ARITH(EXTEND)):

Value of $i + d$	Value of $d$	Value of $i + d_{max}$	Number of places carried for $ir$
<31 =31	Any value	Any value	$i$ integer and $d$ decimal places
>31	< $d_{max}$ = $d_{max}$	Any value	31- $d$ integer and $d$ decimal places
	> $d_{max}$	<31 =31	$i$ integer and 31- $i$ decimal places
		>31	31- $d_{max}$ integer and $d_{max}$ decimal places

## Exponentiation

Exponentiation is represented by the expression  $op1 ** op2$ . Based on the characteristics of  $op2$ , the compiler handles exponentiation of fixed-point numbers in one of three ways:

- When  $op2$  is expressed with decimals, floating-point instructions are used.
- When  $op2$  is an integral literal or constant, the value  $d$  is computed as  $d = d1 * |op2|$

and the value  $i$  is computed based on the characteristics of  $op1$ :

- When  $op1$  is a data name or variable,  $i = i1 * |op2|$
- When  $op1$  is a literal or constant,  $i$  is set equal to the number of integers in the value of  $op1 ** |op2|$ .

In compatibility mode (compilation using ARITH(COMPAT)), the compiler having calculated  $i$  and  $d$  takes the action indicated in the table below to handle the intermediate results  $ir$  of the exponentiation.

Value of $i + d$	Other conditions	Action taken
<30	Any.	$i$ integer and $d$ decimal places are carried for $ir$ .



Value of $i + d$	Other conditions	Action taken
=30	$op1$ has an odd number of digits.	$i$ integer and $d$ decimal places are carried for $ir$ .
	$op1$ has an even number of digits.	Same action as when $op2$ is an integral data name or variable (shown below). Exception: for a 30-digit integer raised to the power of literal 1, $i$ integer and $d$ decimal places are carried for $ir$ .
>30	Any.	Same action as when $op2$ is an integral data name or variable (shown below).

In extended mode (compilation using ARITH(EXTEND)), the compiler having calculated  $i$  and  $d$  takes the action indicated in the table below to handle the intermediate results  $ir$  of the exponentiation.

Value of $i + d$	Other conditions	Action taken
<31	Any.	$i$ integer and $d$ decimal places are carried for $ir$ .
=31 >31	Any.	Same action as when $op2$ is an integral data name or variable (shown below). Exception: for a 31-digit integer raised to the power of literal 1, $i$ integer and $d$ decimal places are carried for $ir$ .

In either mode, if  $op2$  is negative, the value of 1 is then divided by the result produced by the preliminary computation. The values of  $i$  and  $d$  that are used are calculated following the division rules for fixed-point data already shown above.

- When  $op2$  is an integral data name or variable,  $dmax$  decimal places and  $30-dmax$  (compatibility mode) or  $31-dmax$  (extended mode) integer places are used.  $op1$  is multiplied by itself ( $|op2| - 1$ ) times for nonzero  $op2$ .

If  $op2$  is equal to 0, the result is 1. Division-by-0 and exponentiation SIZE ERROR conditions apply.

Fixed-point exponents with more than nine significant digits are always truncated to nine digits. If the exponent is a literal or constant, an E-level compiler diagnostic message is issued; otherwise, an informational message is issued at run time.

“Example: exponentiation in fixed-point arithmetic”

#### RELATED REFERENCES

“Terminology used for intermediate results” on page 560

“Truncated intermediate results” on page 564

“Binary data and intermediate results” on page 564

“Floating-point data and intermediate results” on page 566

“Intrinsic functions evaluated in fixed-point arithmetic” on page 565

“ARITH” on page 262

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 560  
“Fixed-point data and intermediate results” on page 561

## Truncated intermediate results

Whenever the number of digits in an intermediate result exceeds 30 in compatibility mode or 31 in extended mode, the compiler truncates to 30 (compatibility mode) or 31 (extended mode) digits 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 36

#### RELATED REFERENCES

“Fixed-point data and intermediate results” on page 561  
“ARITH” on page 262

## 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 561  
“ARITH” on page 262

---

## 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-INTEGGER	0	8
DATE-TO-YYYYMMDD	0	8
DAY-OF-INTEGGER	0	7
DAY-TO-YYYYDDD	0	7
FACTORIAL	0	30 in compatibility mode, 31 in extended mode
INTEGER-OF-DATE	0	7
INTEGER-OF-DAY	0	7
LENGTH	n/a	9
MOD	0	$\min(i1\ i2)$
ORD	n/a	3
ORD-MAX		9
ORD-MIN		9
YEAR-TO-YYYY	0	4
INTEGER		For a fixed-point argument: one more digit than in the argument. For a floating-point argument: 30 in compatibility mode, 31 in extended mode.
INTEGER-PART		For a fixed-point argument: same number of digits as in the argument. For a floating-point argument: 30 in compatibility mode, 31 in extended mode.

### Mixed functions

A *mixed* intrinsic function is a function whose result type depends on the type of its arguments. A mixed function is fixed point if all of its arguments are numeric and none of its arguments is floating point. (If any argument of a mixed function is floating point, the function is evaluated with floating-point instructions and returns a floating-point result.) When a mixed function is evaluated with fixed-point arithmetic, the result is integer if all of the arguments are integer; otherwise, the result is fixed point.

For the mixed functions MAX, MIN, RANGE, REM, and SUM, the *outer-dmax* is always equal to the *inner-dmax* (and both are thus zero if all the arguments are integer). To determine the precision of the result returned for these functions, apply the rules for fixed-point arithmetic and intermediate results (as referred to below) to each step in the algorithm.

#### MAX

1. Assign the first argument to the function result.
2. For each remaining argument, do the following:
  - 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 560  
 “Fixed-point data and intermediate results” on page 561  
 “Floating-point data and intermediate results”  
 “ARITH” on page 262

---

## 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.

In compatibility mode, if an expression is computed in floating-point arithmetic, the precision used to evaluate the arithmetic operations is determined as follows:

- Single precision is used if all receivers and operands are COMP-1 data items and the expression contains no multiplication or exponentiation operations.
- In all other cases, long precision is used.

Whenever long-precision floating point is used for one operation in an arithmetic expression, all operations in the expression are computed as if long floating-point instructions were used.

In extended mode, if an expression is computed in floating-point arithmetic, the precision used to evaluate the arithmetic operations is determined as follows:

- Single precision is used if all receivers and operands are COMP-1 data items and the expression contains no multiplication or exponentiation operations.
- Long precision is used if all receivers and operands are COMP-1 or COMP-2 data items, at least one receiver or operand is a COMP-2 data item, and the expression contains no multiplication or exponentiation operations.
- In all other cases, extended precision is used.

Whenever extended-precision floating point is used for one operation in an arithmetic expression, all operations in the expression are computed as if extended-precision floating-point instructions were used.

**Alert:** If a floating-point operation has an intermediate result field in which exponent overflow occurs, the job is abnormally terminated.

## Exponentiations evaluated in floating-point arithmetic

In compatibility mode, floating-point exponentiations are always evaluated using long floating-point arithmetic. In extended mode, floating-point exponentiations are always evaluated using extended-precision floating-point arithmetic.

The value of a negative number raised to a fractional power is undefined in COBOL. For example,  $(-2) ** 3$  is equal to  $-8$ , but  $(-2) ** (3.000001)$  is undefined. When an exponentiation is evaluated in floating point and there is a possibility that the result is undefined, the exponent is evaluated at run time to determine if it has an integral value. If not, a diagnostic message is issued.

## Intrinsic functions evaluated in floating-point arithmetic

In compatibility mode, floating-point intrinsic functions always return a long (64-bit) floating-point value. In extended mode, floating-point intrinsic functions always return an extended-precision (128-bit) floating-point value.

Mixed functions with at least one floating-point argument are evaluated using floating-point arithmetic.

### RELATED REFERENCES

“Terminology used for intermediate results” on page 560

“ARITH” on page 262

---

## ON SIZE ERROR and intermediate results

When the CMPR2 compiler option is in effect, the ON SIZE ERROR phrase for MULTIPLY and DIVIDE statements applies to intermediate and to final results. For other arithmetic statements, ON SIZE ERROR applies only to the final results.

#### RELATED TASKS

“Handling errors in arithmetic operations” on page 191

#### RELATED REFERENCES

“CMPR2” on page 264

---

## 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 49

#### RELATED REFERENCES

“Terminology used for intermediate results” on page 560

“Fixed-point data and intermediate results” on page 561

“Floating-point data and intermediate results” on page 566

IF statement (*IBM COBOL Language Reference*)

EVALUATE statement (*IBM COBOL Language Reference*)

Conditional expressions (*IBM COBOL Language Reference*)

---

## Appendix B. 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 571

“Preventing overlay when adding elements to a variable table” on page 571

### RELATED REFERENCES

“Effects of change in ODO object value” on page 570

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 569) 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 570

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).

- Restore the variably located data items from the data area where you saved them.

In the following example, suppose you want to add an element to the table VARY-FIELD-1, whose number of elements depends on the ODO object CONTROL-1. VARY-FIELD-1 is followed by the nonsubordinate variably located data item GROUP-ITEM-1, whose elements could potentially be overlaid.

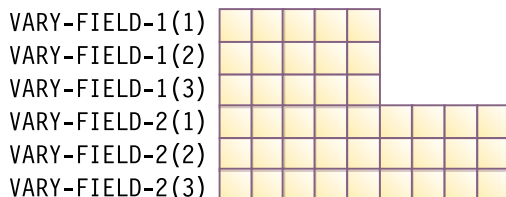
WORKING-STORAGE SECTION.

```

01 VARIABLE-REC.
   05 FIELD-1                               PIC X(10).
   05 CONTROL-1                             PIC S99.
   05 CONTROL-2                             PIC S99.
   05 VARY-FIELD-1 OCCURS 1 TO 10 TIMES
       DEPENDING ON CONTROL-1               PIC X(5).
   05 GROUP-ITEM-1.
       10 VARY-FIELD-2
           OCCURS 1 TO 10 TIMES
           DEPENDING ON CONTROL-2           PIC X(9).
01 STORE-VARY-FIELD-2.
   05 GROUP-ITEM-2.
       10 VARY-FLD-2
           OCCURS 1 TO 10 TIMES
           DEPENDING ON CONTROL-2           PIC X(9).

```

Each element of VARY-FIELD-1 has 5 bytes, and each element of VARY-FIELD-2 has 9 bytes. If CONTROL-1 and CONTROL-2 both contain the value 3, you can picture storage for VARY-FIELD-1 and VARY-FIELD-2 as follows:



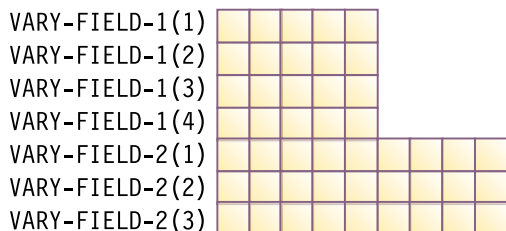
To add a fourth element to VARY-FIELD-1, code as follows to prevent overlaying the first 5 bytes of VARY-FIELD-2. (GROUP-ITEM-2 serves as temporary storage for the variably located GROUP-ITEM-1.)

```

MOVE GROUP-ITEM-1 TO GROUP-ITEM-2.
ADD 1 TO CONTROL-1.
MOVE five-byte-field TO
    VARY-FIELD-1 (CONTROL-1).
MOVE GROUP-ITEM-2 TO GROUP-ITEM-1.

```

You can picture the updated storage for VARY-FIELD-1 and VARY-FIELD-2 as follows:



Note that the fourth element of VARY-FIELD-1 did not overlay the first element of VARY-FIELD-2.

**RELATED REFERENCES**

“Effects of change in ODO object value” on page 570

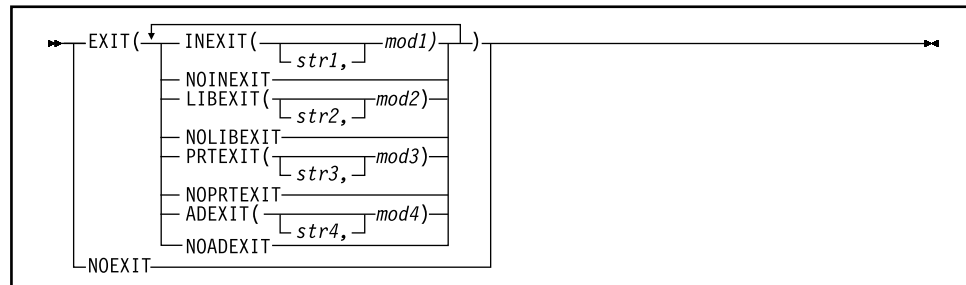


---

## Appendix C. EXIT compiler option

Use the EXIT option to allow the compiler to accept user-supplied modules in place of SYSIN, SYSLIB (or copy library), and SYSPRINT.

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. Under CMS, the user-supplied modules must be relocatable MODULE files.



Default is: NOEXIT

Abbreviations are: EX(INX|NOINX, LIBX|NOLIBX, PRTX|NOPRTX, ADX|NOADX)

If you specify the EXIT option without providing a least one suboption, NOEXIT will be in effect. You can specify the suboptions in any order and separate them by either commas or spaces. If you specify both the positive and negative form of a suboption (INEXIT|NOINEXT, LIBEXIT|NOLIBEXIT, PRTEXTIT|NOPRTEXTIT, or ADEXIT|NOADEXIT), the form specified last takes effect. If you specify the same suboption more than once, the last one that you specified takes effect.

You can specify the EXIT option only at invocation in the PARM field of JCL (under TSO, in a command argument) or at installation time. Do not specify the EXIT option in a PROCESS (CBL) statement.

### **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.

### **PRTEXTIT([*str3*'],*mod3*)**

The compiler passes printer-destined output to the user-supplied load module (where *mod3* is the module name), instead of SYSPRINT.

### **ADEXIT([*str4*'],*mod4*)**

The compiler passes the SYSADATA output to the user-supplied load module (where *mod4* is the module name).

The module names *mod1*, *mod2*, *mod3*, and *mod4* can refer to the same module.

The suboptions *str1*, *str2*, *str3*, and *str4* are character strings that are passed to the load module. These strings are optional. You can make them up to 64 characters in length and must enclose them in apostrophes. Any character is allowed, but included apostrophes must be doubled, and lowercase characters are folded to uppercase.

If one of *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.

“Example: SYSIN user exit” on page 583

#### RELATED TASKS

“Using the user-exit work area”

“Calling from exit modules”

#### RELATED REFERENCES

“Processing of INEXIT” on page 577

“Processing of LIBEXIT” on page 578

“Processing of PRTEXTIT” on page 580

“Processing of ADEXIT” on page 582

“Error handling for exit modules” on page 583

---

## Using the user-exit work area

When you use an exit, the compiler provides a user-exit work area where you can save the address of GETMAIN storage obtained by the exit module. This work area allows the module to be reentrant.

The user-exit work area is 4 fullwords residing on a fullword boundary. These fullwords are initialized to binary zeros 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.

You 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 PRTEXTIT module uses the third word.

#### RELATED REFERENCES

“Appendix C. EXIT compiler option” on page 575

---

## Calling from exit modules

Use COBOL standard linkage in your exit modules to call COBOL programs or library routines. You need to be aware of these conventions in order to trace the call chain correctly.

When a call is made to a program or to a routine, the registers are set up as follows:

**R1** Points to the parameter list passed to the called program or library routine.

- R13 Points to the register save area provided by the calling program or routine.
- R14 Holds the return address of the calling program or routine.
- R15 Holds the address of the called program or routine.

Exit modules must have the RMODE attribute of **24** and the AMODE attribute of **ANY**.

RELATED REFERENCES

“Appendix C. EXIT compiler option” on page 575

## Processing of INEXIT

The exit module is used in place of SYSIN, to read source code from a user-supplied load module.

The processing of INEXIT is as follows:

Action by compiler	Resulting action by exit module
Loads the exit module ( <i>mod1</i> ) during initialization.	
Calls the exit module with an OPEN operation code (op code).	Prepares its source for processing. Passes the status of the OPEN request back to the compiler.
Calls the exit module with a GET op code when a source statement is needed.	Returns either the address and length of the next statement or the end-of-data indication (if no more source statements exist).
Calls the exit module with a CLOSE op code when the end-of-data is presented.	Releases any resources that are related to its output.

## Parameter list for INEXIT

The compiler uses a parameter list to communicate with the exit module. The parameter list consists of 10 fullwords containing addresses, and register 1 contains the address of the parameter list. 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 following table describes the contents of the parameter list.

Offset	Contains address of	Description of item
00	User-exit type	Halfword identifying which user exit is to perform the operation. 1=INEXIT
02	Operation code	Halfword indicating the type of operation. 0=OPEN; 1=CLOSE; 2=GET
04	Return code	Fullword, placed by the exit module, indicating the success of the requested operation. 0=Operation was successful 4=End-of-data 12=Operation failed
08	User-exit work area	4-fullword work area provided by the compiler, for use by the user-exit module.
12	Data length	Fullword, placed by the exit module, specifying the length of the record being returned by the GET operation (must be 80).

Offset	Contains address of	Description of item
16	Data or <i>str1</i>	Fullword, placed by the exit module, containing the address of the record in a user-owned buffer, for the GET operation.  <i>str1</i> applies only to OPEN. The first halfword (on a halfword boundary) contains the length of the string, followed by the string.
20	Not used	
24	Not used	
28	Not used	
32	Not used	

“Example: SYSIN user exit” on page 583

RELATED REFERENCES

“Appendix C. EXIT compiler option” on page 575

---

## Processing of LIBEXIT

The exit module is used in place of the SYSLIB, or *library-name*, data set. Calls are made to the module by the compiler to obtain copy text whenever COPY or BASIS statements are encountered.

If LIBEXIT is specified, the LIB compiler option must be in effect.

The processing of LIBEXIT is as follows:

Action by compiler	Resulting action by exit module
Loads the exit module ( <i>mod2</i> ) during initialization.	
Calls the exit module with an OPEN operation code (op code).	Prepares the specified <i>library-name</i> for processing. Passes the status of the OPEN request to the compiler.
Calls the exit module with a FIND op code if the <i>library-name</i> has successfully opened.	Establishes positioning at the requested <i>text-name</i> (or <i>basis-name</i> ) in the specified <i>library-name</i> ; this place becomes the active copy source. Passes an appropriate return code to the compiler when positioning is complete.
Calls the exit module with a GET op code.	Passes the compiler either the length and address of the record to be copied from the active copy source or the end-of-data indicator.
Calls the exit module with a CLOSE op code when the end-of-data is presented.	Releases any resources that are related to its input.

## Processing of LIBEXIT with 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.)



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.

The following table shows how the processing of LIBEXIT changes when there is one or more valid COPY statements:

Action by compiler		Resulting action by exit module
Loads the exit module ( <i>mod2</i> ) during initialization.		
Calls the exit module with an OPEN operation code (op code).		Prepares the specified <i>library-name</i> for processing. Passes the status of the OPEN request to the compiler.
Calls the exit module with a FIND op code if the <i>library-name</i> has successfully opened.		Establishes positioning at the requested <i>text-name</i> (or <i>basis-name</i> ) in the specified <i>library-name</i> ; this place becomes the active copy source. Passes an appropriate return code to the compiler when positioning is complete.
When the compiler encounters a valid nested COPY statement	If the requested <i>library-name</i> from the nested COPY statement was not previously opened, calls the exit module with an OPEN op code.	Pushes its control information about the active copy source onto a stack. Completes the requested action (OPEN or FIND). The newly requested <i>text-name</i> (or <i>basis-name</i> ) becomes the active copy source.
	Calls the exit module with a FIND op code for the requested new <i>text-name</i> .	
	Calls the exit module with a GET op code.	Passes the compiler either the length and address of the record to be copied from the active copy source or the end-of-data indicator. At end-of-data, pops its control information from the stack.
Calls the exit module with a FIND op code if the <i>library-name</i> has successfully opened.		Reestablishes positioning at the previous active copy source. Passes an appropriate return code to the compiler when positioning is complete.
Calls the exit module with a GET op code.  Verifies that the same record was passed.		Passes the compiler the same record as was passed previously from this copy source. After verification, passes either the length and address of the record to be copied from the active copy source or the end-of-data indicator.
Calls the exit module with a CLOSE op code when the end-of-data is presented.		Releases any resources that are related to its input.

## Parameter list for LIBEXIT

The compiler uses a parameter list to communicate with the exit module. The parameter list consists of 10 fullwords containing addresses, and register 1 contains the address of the parameter list. 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 following table describes the contents of the parameter list used for LIBEXIT.

Offset	Contains address of	Description of item
00	User-exit type	Halfword identifying which user exit is to perform the operation. 2=LIBEXIT
04	Operation code	Halfword indicating the type of operation. 0=OPEN; 1=CLOSE; 2=GET; 4=FINN
08	Return code	Fullword, placed by the exit module, indicating the success of the requested operation. 0=Operation was successful 4=End-of-data 12=Operation failed
12	User-exit work area	4-fullword work area provided by the compiler for use by the user-exit module.
16	Data length	Fullword, placed by the exit module, specifying the length of the record being returned by the GET operation (must be 80).
20	Data or <i>str2</i>	Fullword, placed by the exit module, containing the address of the record in a user-owned buffer, for the GET operation.  <i>str2</i> applies only to OPEN. The first halfword (on a halfword boundary) contains the length of the string, followed by the string.
24	System <i>library-name</i>	Eight-character area containing the <i>library-name</i> from the COPY statement. Processing and conversion rules for a program-name are applied. Padded with blanks if required. Applies to OPEN, CLOSE, and FIND.
28	System <i>text-name</i>	Eight-character area containing the <i>text-name</i> from the COPY statement ( <i>basis-name</i> from BASIS statement). Processing and conversion rules for a <i>program name</i> are applied. Padded with blanks if required. Applies only to FIND.
32	Library-name	30-character area containing the full <i>library-name</i> from the COPY statement. Padded with blanks if required, and used as-is (not folded to uppercase). Applies to OPEN, CLOSE, and FIND.
36	Text-name	30-character area containing the full <i>text-name</i> from the COPY statement. Padded with blanks if required, and used as-is (not folded to uppercase). Applies only to FIND.

#### RELATED REFERENCES

“Appendix C. EXIT compiler option” on page 575

---

## Processing of PRTEXT

The exit module is used in place of the SYSPRINT data set.

The processing of PRTEXT is as follows:

Action by compiler	Resulting action by exit module
Loads the exit module ( <i>mod3</i> ) during initialization.	
Calls the exit module with an OPEN operation code (op code).	Prepares its output destination for processing. Passes the status of the OPEN request to the compiler.
Calls the exit modules with a PUT op code when a line is to be printed, supplying the address and length of the record that is to be printed.	Passes 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.
Calls the exit module with a CLOSE op code when the end-of-data is presented.	Releases any resources that are related to its output destination.

## Parameter list for PRTEXT

The compiler uses a parameter list to communicate with the exit module. The parameter list consists of 10 fullwords containing addresses, and register 1 contains the address of the parameter list. The return code, data length, and data buffer 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 following table describes the contents of the parameter list used for PRTEXT.

Offset	Contains address of	Description of item
00	User-exit type	Halfword identifying which user exit is to perform the operation. 3=PRTEXT
04	Operation code	Halfword indicating the type of operation. 0=OPEN; 1=CLOSE; 3=PUT
08	Return code	Fullword, placed by the exit module, indicating the success of the requested operation. 0=Operation was successful 12=Operation failed
12	User-exit work area	4-fullword work area provided by the compiler, for use by the user-exit module.
16	Data length	Fullword specifying the length of the record being supplied by the PUT operation (the compiler sets this value to 133).
20	Data buffer or <i>str3</i>	Fullword containing the address of the data buffer where the compiler has placed the record to be printed by the PUT operation.  <i>str3</i> applies only to OPEN. The first halfword (on a halfword boundary) contains the length of the string, followed by the string.
24	Not used	
28	Not used	
32	Not used	
36	Not used	

## Processing of ADEXIT

Use of the ADEXIT module requires:

- Compiler option `ADATA` to produce `SYSADATA` output
- `DD` statement `SYSADATA`

The processing of ADEXIT is as follows:

Action by compiler	Resulting action by exit module
Loads the exit module ( <i>mod4</i> ) during initialization.	
Calls the exit module with an <code>OPEN</code> operation code (op code).	Prepares its output destination for processing. Passes the status of the <code>OPEN</code> request to the compiler.
Calls the exit modules with a <code>PUT</code> op code when the compiler has written a <code>SYSADATA</code> record, supplying the address and length of the <code>SYSADATA</code> record.	Passes the status of the <code>PUT</code> request to the compiler by a return code.
Calls the exit module with a <code>CLOSE</code> op code when the end-of-data is presented.	Releases any resources.

## Parameter list for ADEXIT

The compiler uses a parameter list to communicate with the exit module. The parameter list consists of 10 fullwords containing addresses, and register 1 contains the address of the parameter list. The return code, data length, and data buffer 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 following table describes the contents of the parameter list used for ADEXIT.

Offset	Contains address of	Description of item
00	User-exit type	Halfword identifying which user exit is to perform the operation. 4=ADEXIT
04	Operation code	Halfword indicating the type of operation. 0=OPEN; 1=CLOSE; 3=PUT
08	Return code	Fullword, placed by the exit module, indicating the success of the requested operation. 0=Operation was successful 12=Operation failed
12	User-exit work area	4-fullword work area provided by the compiler, for use by the user-exit module.
16	Data length	Fullword specifying the length of the record being supplied by the <code>PUT</code> operation.

Offset	Contains address of	Description of item
20	Data buffer or <i>str4</i>	Fullword containing the address of the data buffer where the compiler has placed the record to be printed by the PUT operation.  <i>str4</i> applies only to OPEN. The first halfword (on a halfword boundary) contains the length of the string, followed by the string.
24	Not used	
28	Not used	
32	Not used	
36	Not used	

RELATED REFERENCES

“Appendix C. EXIT compiler option” on page 575

---

## Error handling for exit modules

The compiler reports an error message whenever an exit module cannot be loaded or an exit module returns an “operation failed” or inappropriate return code.

Message IGYSI5008 is written to the operator, and the compiler terminates with return code 16 when any of the following happens:

- An exit module cannot be loaded.
- A nonzero return code is received from INEXIT during an OPEN request.
- A nonzero return code is received from PRTEXT during an OPEN request.

The exit type and operation (OPEN or LOAD) are identified in the message.

Any other error from INEXIT or PRTEXT causes the compiler to terminate.

The compiler detects and reports the following conditions:

- 5203 PUT request to SYSPRINT user exit failed with return code *nm*.
- 5204 Record address not set by *exit-name* user exit.
- 5205 GET request from SYSIN user exit failed with return code *nm*.
- 5206 Record length not set by *exit-name* user exit.

RELATED REFERENCES

“Appendix C. EXIT compiler option” on page 575

---

## Example: SYSIN user exit

The following example shows a SYSIN user-exit module in COBOL.

```
*****
*                                                                 *
* Name:  SKELINX                                                *
*                                                                 *
* Function: Example of a SYSIN user-exit written                *
*           in the COBOL language.                               *
*                                                                 *
*****
```

Identification Division.

Program-ID. Skelinx.

Environment Division.

Data Division.

WORKING-STORAGE Section.

```

* *****
* *
* * Local variables.
* *
* *****

77 Operation          Pic 9(4)  Comp.
01 Record-Variable   Pic X(80).
* *****
* *
* * Definition of the User-Exit Parameter List, which *
* * is passed from the COBOL compiler to the user exit *
* * passed from the COBOL compiler to the user exit *
* * module.
* *
* *
* *****

```

Linkage Section.

```

01 Exit-Type          Pic 9(4)  Comp.
01 Exit-Operation     Pic 9(4)  Comp.
01 Exit-ReturnCode    Pic 9(5)  Comp.
01 Exit-WorkArea.
   05 Sysin-Slot      Pic 9(5)  Comp.
   05 Syslib-Slot     Pic 9(5)  Comp.
   05 Sysprint-Slot   Pic 9(5)  Comp.
   05 Reserved-Slot  Pic 9(5)  Comp.
01 Exit-DataLength    Pic 9(5)  Comp.
01 Exit-DataArea      Pic 9(9)  Comp.
01 Exit-Open-Parm     Redefines Exit-DataArea.
   05 String-Len      Pic 9(4)  Comp.
   05 Open-String     Pic X(64).
01 Exit-Print-Line    Redefines Exit-DataArea Pic X(133).
01 Exit-Syslibrary    Pic X(8).
01 Exit-Systext       Pic X(8).
01 Exit-CBLLibrary    Pic X(30).
01 Exit-CBLText       Pic X(30).

```

```

*****
*
* Begin PROCEDURE DIVISION
*
* Invoke the section to handle the exit.
*
*
*****

```

```

Procedure Division Using Exit-Type          Exit-Operation
                        Exit-ReturnCode Exit-WorkArea
                        Exit-DataLength Exit-DataArea
                        Exit-Syslibrary Exit-Systext
                        Exit-CBLLibrary Exit-CBLText.

```

```

Add 1 To Exit-Operation Giving Operation
Go To Handle-Sysin
   Handle-Syslib
   Handle-Sysprint
   Depending On Exit-Type.
Move 16 To Exit-ReturnCode
Goback.

```

```

*****
*   SYSIN EXIT PROCESSOR   *
*****
Handle-Sysin.

    Go To Sysin-Open
      Sysin-Close
      Sysin-Get
    Depending On Operation.
    Move 16 To Exit-ReturnCode
    Goback.

Sysin-Open.
* -----
*   Prepare for reading source
* -----
    Goback.

Sysin-Close.
* -----
*   Release resources
* -----
    Goback.

Sysin-Get.
* -----
*   Retrieve next source record
* -----

* -----
*   The following can be used to return the address of the
*   record to the compiler.
* -----
    Call "GETADDRESS" Using
      By Reference Record-Variable
      By Reference Exit-DataArea

* -----
*   Set length of record in User-Exit Parameter List
* -----
    Move 80 To Exit-DataLength

    Goback.

*****
*   SYSLIB EXIT PROCESSOR   *
*****
Handle-Syslib.
    Display "**** This module for SYSIN only"
    Move 16 To Exit-ReturnCode
    Goback.

*****
*   SYSPRINT EXIT PROCESSOR   *
*****
Handle-Sysprint.
    Display "**** This module for SYSIN only"
    Move 16 To Exit-ReturnCode
    Goback.

*****
**                                     **
**   Internal program to obtain the address of an   **
**   item in the caller's WORKING-STORAGE SECTION **
**                                     **
*****
*****

```

```
Identification Division.  
  Program-ID.  GetAddress.  
Environment Division.  
Data Division.  
  Linkage Section.  
    01 The-Item      Pic X(80).  
    01 Its-Address   Pointer.  
Procedure Division Using The-Item Its-Address.  
  Set Its-Address To Address Of The-Item.  
  Goback.  
End Program GetAddress.  
  
End Program Skelinx.
```



---

## Appendix D. Sample programs

This material contains information about the sample programs that are included on your product tape:

- Overview of the programs, including program charts for two of the samples
- Format and sample of the input data
- Sample of reports produced
- Information on how to run the programs
- List of the language elements and concepts that are illustrated

Pseudocode and other comments regarding these programs are included in the program prologue, which you can obtain in a program listing.

The sample programs in this material demonstrate many language elements and concepts of COBOL:

- IGYTCARA is an example of using QSAM files and VSAM indexed files and shows how to use many COBOL intrinsic functions.
- IGYTCARB is an example of using the IBM Interactive System Product Facility (ISPF).
- IGYTSALE is an example of using several of the Language Environment callable services features.

### RELATED CONCEPTS

“IGYTCARA: batch application”

“IGYTCARB: interactive program” on page 592

“IGYTSALE: nested program application” on page 595

---

### IGYTCARA: batch application

A company with several local offices wants to establish employee carpools. This batch application needs to perform two tasks:

- Produce reports of employees who can share rides from the same home location to the same work location.
- Update the carpool data:
  - Add data for new employees.
  - Change information for participating employees.
  - Delete employee records.
  - List update requests that are not valid.

Using QSAM files and VSAM indexed files, this program validates transaction file entries (sequential file processing) and updates a master file (indexed file processing).

The following diagram shows the parts of the application and how they are organized:



3. Home code
4. Work code
5. Commuter name
6. Home address
7. Home phone
8. Work phone
9. Home location code
10. Work location code
11. Driving status code

The sample below shows a section of the input file:

```

A10111ROBERTS AB1021 CRYSTAL COURTSAN FRANCISCOCA9990141555501904155551387H1W1D
A20212KAHN DE789 EMILY LANE SAN FRANCISCOCA9992141555518904155552589H2W2D
P48899 99ASDFG0005557890123ASDFGHJ T
R10111ROBERTS AB1221 CRYSTAL COURTSAN FRANCISCOCA9990141555501904155551387H1W1D
A20212KAHN DE789 EMILY LANE SAN FRANCISCOCA9992141555518904155552589H2W2D
D20212KAHN DE
D20212KAHN DE
A20212KAHN DE789 EMILY LANE SAN FRANCISCOCA9992141555518904155552589H2W2D
A10111BONNICK FD1025 FIFTH AVENUE SAN FRANCISCOCA9990541555595904155557895H8W3
A10111PETERSON SW435 THIRD AVENUE SAN FRANCISCOCA9990541555546904155553717H3W4
. . .

```

## Report produced by IGYTCARA

The following sample shows the first page of the output report produced by IGYTCARA. Your actual output might vary slightly in appearance, depending on your system.

1REPORT #:		IGYTCAR1		COMMUTER FILE UPDATE LIST		PAGE #:		1	
-PROGRAM #:		IGYTCAR1		RUN TIME: 01:40		RUN DATE: 11/30/1999			
TRANS CODE	RE-CORD TYPE	SHIFT HOME CODE WORK CODE	COMMUTER NAME	HOME ADDRESS	HOME PHONE WORK PHONE	HOME LOCATION WORK LOCATION	JUNCTION JUNCTION	STA-TUS CODE	TRANS. ERROR
A	NEW	1 01 11	ROBERTS	AB 1021 CRYSTAL COURT SAN FRANCISCO	(415) 555-0190 CA 99901 (415) 555-1387	RODNEY/CRYSTAL BAYFAIR PLAZA		D	
A	NEW	2 02 12	KAHN	DE 789 EMILY LANE SAN FRANCISCO	(415) 555-1890 CA 99921 (415) 555-2589	COYOTE 14TH STREET/166TH AVENUE		D	
P		4 88 99			(000) 555-7890 99 ASDFG (123) ASD-FGHJ	HOME CODE ' ' NOT FOUND. WORK CODE ' ' NOT FOUND.		T	TRANSACTION CODE SHIFT CODE HOME LOC. CODE WORK LOC. CODE LAST NAME INITIALS ADDRESS CITY STATE CODE ZIPCODE HOME PHONE WORK PHONE HOME JUNCTION WORK JUNCTION DRIVING STATUS
R	OLD	1 01 11	ROBERTS	AB 1021 CRYSTAL COURT SAN FRANCISCO	(415) 555-0190 CA 99901 (415) 555-1387	RODNEY/CRYSTAL BAYFAIR PLAZA		D	
	NEW	1 01 11	ROBERTS	AB 1221 CRYSTAL COURT SAN FRANCISCO	(415) 555-0190 CA 99901 (415) 555-1387	RODNEY/CRYSTAL BAYFAIR PLAZA		D	
A		2 02 12	KAHN	DE 789 EMILY LANE SAN FRANCISCO	(415) 555-1890 CA 99921 (415) 555-2589	COYOTE 14TH STREET/166TH AVENUE		D	DUPLICATE REC.
D	OLD	2 02 12	KAHN	DE 789 EMILY LANE SAN FRANCISCO	(415) 555-1890 CA 99921 (415) 555-2589	COYOTE 14TH STREET/166TH AVENUE		D	
D		2 02 12	KAHN	DE					REC. NOT FOUND
A	NEW	2 02 12	KAHN	DE 789 EMILY LANE SAN FRANCISCO	(415) 555-1890 CA 99921 (415) 555-2589	COYOTE 14TH STREET/166TH AVENUE		D	
A	NEW	1 01 11	BONNICK	FD 1025 FIFTH AVENUE SAN FRANCISCO	(415) 555-9590 CA 99905 (415) 555-7895	RODNEY 17TH FREEWAY SAN LEANDRO			
A	NEW	1 01 11	PETERSON	SW 435 THIRD AVENUE	(415) 555-4690	RODNEY/THIRD AVENUE			

## Running IGYTCARA

You can run IGYTCARA under OS/390 or CMS.

All the files required by the IGYTCARA program are supplied on the product installation tape. These files (IGYTCARA, IGYTCODE, and IGYTRANX) are located in the IGY.V2R2M0.SIGYSAMP data set.

Data set and procedure names can be changed at installation time. You should check with your system programmer to verify these names.

Do not change these options on the CBL statement in the source file for IGYTCARA:

```
NOADV
NOCMPR2
NODYNAM
NONAME
NONUMBER
QUOTE
SEQUENCE
```

With these options in effect, the program will not cause any diagnostic messages to be issued. You can use the sequence number string in the source file to search for the language elements used.

### RELATED CONCEPTS

"IGYTCARA: batch application" on page 587

### RELATED TASKS

"Running IGYTCARA under OS/390"

"Running IGYTCARA under CMS" on page 591

### RELATED REFERENCES

"Input data for IGYTCARA" on page 588

"Report produced by IGYTCARA" on page 589

"Language elements and concepts that are illustrated" on page 604

## Running IGYTCARA under OS/390

The procedure provided here does a combined compile, link-edit, and run of the IGYTCARA program. If you want only to compile or only to compile and link-edit the program, you need to change the IGYWCLG cataloged procedure.

To run IGYTCARA under OS/390, use JCL to define a VSAM cluster and compile the program. Insert the information specific to your system and installation in the fields that are shown in lowercase letters (accounting information, volume serial number, unit name, cluster prefix). We have used the name IGYTCAR.MASTFILE in these examples; you can use another name if you want.

1. Use this JCL to create the required VSAM cluster:

```
//CREATE JOB (acct-info), 'IGYTCAR CREATE VSAM',MSGLEVEL=(1,1),
// TIME=(0,29)
//CREATE EXEC PGM=IDCAMS
//VOL1 DD VOL=SER=your-volume-serial,UNIT=your-unit,DISP=SHR
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
DELETE your-prefix.IGYTCAR.MASTFILE -
FILE(VOL1) -
PURGE
DEFINE CLUSTER -
```

```

        (NAME(your-prefix.IGYTCAR.MASTFILE) -
        VOLUME(your-volume-serial) -
        FILE(VOL1) -
        INDEXED -
        RECSZ(80 80) -
        KEYS(16 0) -
        CYLINDERS(2))
/*

```

To remove any existing cluster, a DELETE is issued before the VSAM cluster is created.

2. Use the following JCL to compile, link-edit, and run the IGYTCARA program:

```

//IGYTCARA JOB (acct-info), 'IGYTCAR',MSGLEVEL=(1,1),TIME=(0,29)
//TEST EXEC IGYWCLG
//COBOL.SYSLIB DD DSN=IGY.V2R2M0.SIGYSAMP,DISP=SHR
//COBOL.SYSIN DD DSN=IGY.V2R2M0.SIGYSAMP(IGYTCARA),DISP=SHR
//GO.SYSOUT DD SYSOUT=A
//GO.COMMUTR DD DSN=your-prefix.IGYTCAR.MASTFILE,DISP=SHR
//GO.LOCCODE DD DSN=IGY.V2R2M0.SIGYSAMP(IGYTCODE),DISP=SHR
//GO.UPDTRANS DD DSN=IGY.V2R2M0.SIGYSAMP(IGYTRANX),DISP=SHR
//GO.UPDPRINT DD SYSOUT=A,DCB=BLKSIZE=133
//

```

#### RELATED TASKS

“Chapter 9. Processing VSAM files” on page 135

### Running IGYTCARA under CMS

The procedure provided here is for a combined compile, link-edit, and run of the IGYTCARA program. If you want only to compile or only to compile and link-edit the program, you need to change the procedure. Use the COBOL2, LOAD, START, GENMOD, LKED, and OSRUN commands to control how the program is compiled, link-edited, and run.

To run IGYTCARA under CMS, first access the COBOL for OS/390 & VM product using the CP LINK and ACCESS commands, and then do the following steps:

1. Access the Language Environment library:  
GLOBAL TXTLIB SCEELKED
2. Compile program by using the COBOL2 command:  
COBOL2 IGYTCARA

At the end of the compilation, LISTING and TEXT files are created and sent to your disk.

3. Identify the data sets that are used by issuing FILEDEF commands (where “\*” designates a disk):

```

FILEDEF SYSOUT DISK IGYTCARA SYSOUT A
FILEDEF UPDTRANS DISK IGYTRANX UPDATE A
FILEDEF LOCCODE DISK IGYTCODE INPUT *
FILEDEF UPDPRINT DISK IGYTCARA UPDPRINT A

```

4. Identify the catalog and your VSAM files by using the DLBL command:

```

DLBL * CLEAR
QUEUE '30 390'
QUEUE ''
DLBL IJSYSCT E DSN MASTCAT (EXTENT
QUEUE '60 360'
QUEUE ''
DLBL COMMUTR E DSN IGYTCARA (EXTENT

```

5. Set up a AMSERV file named IGYTCARA AMSERV to contain the following statements:

```

DEFINE MASTERCATALOG      -
      (NAME(MASTCAT)      -
       CYL(1)              -
       VOLUME(000001)     -
       FILE(IJSYSCT)      -
      )
DEFINE CLUSTER            -
      (NAME(IGYTCARA)     -
       VOLUME(000001)     -
       FILE(IJSYSCT)     -
       INDEXED            -
       RECSZ(80 80)       -
       KEYS(16 0)         -
       CYLINDERS(2)       -
      )

```

- Define VSAM data spaces, clusters, and more, using the AMSERV command to perform the necessary control statements in the AMSERV file:

```
AMSERV IGYTCARA
```

- Run the program by using the LOAD and START commands:

```
LOAD IGYTCARA
START *
```

#### RELATED TASKS

“Accessing the compiler (CP LINK and ACCESS)” on page 243

## IGYTCARB: interactive program

The IGYTCARB program contains an interactive program for entering the carpool data through a screen, using the IBM Interactive System Productivity Facility (ISPF) to invoke Dialog Manager and COBOL for OS/390 & VM. It creates a file that could be used as input for a carpool listing or matching program such as IGYTCARA.

The input data for IGYTCARB is the same as that for IGYTCARA. IGYTCARB lets you append to the information in your input file by using an ISPF panel. An example of the panel used by IGYTCARB is shown below:

```

----- CARPOOL DATA ENTRY -----
      New Data Entry
Type =====> -                               A, R, or D  A
Shift =====> -                               1, 2, or 3  1
Home Code ==> --                                2 Chars   01
Work Code ==> --                                2 Chars   11
Name =====> -----                          9 Chars   POPOWICH
Initials ==> --                                 2 Chars   AD
Address =====> -----                        18 Chars  134 SIXTH AVENUE
City =====> -----                           13 Chars  SAN FREANCISCO
State =====> --                               2 Chars   CA
Zip Code ==> -----                             5 Chars  99903
Home Phone => -----                             10 Chars  4155553390
Work Phone => -----                             10 Chars  4155557855
Home Jnc code > --                               2 Chars   H3
Work Jnc Code > --                               2 Chars   W7
Commuter Stat > -                                D, R or blank

```

#### RELATED TASKS

“Running IGYTCARB”

## Running IGYTCARB

You can run IGYTCARB using the Interactive System Productivity Facility (ISPF) under OS/390 or CMS.

All the files required by the IGYTCARB program are supplied on the product installation tape. These files (IGYTCARB, IGYTRANB, and IGYTPNL) are located in the IGY.V2R2M0.SIGYSAMP data set.

Data set and procedure names can be changed at installation time. Check with your system programmer to verify these names.

Do not change these options on the CBL card in the source file for IGYTCARB:

```
NOCMPR2
NONUMBER
QUOTE
SEQUENCE
```

With these options in effect, the program will not cause any diagnostic messages to be issued. You can use the sequence number string in the source file to search for language elements.

#### RELATED CONCEPTS

“IGYTCARB: interactive program” on page 592

#### RELATED TASKS

“Running IGYTCARB under OS/390”

“Running IGYTCARB under CMS” on page 595

#### RELATED REFERENCES

“Language elements and concepts that are illustrated” on page 604

## Running IGYTCARB under OS/390

The following procedure does a combined compile, link-edit, and run of the IGYTCARB program. If you want only to compile or only to compile and link-edit the program, you need to change the procedure.

To run IGYTCARB under OS/390, use the following steps:

1. Using the ISPF editor, change the ISPF/PDF Primary Option Panel (ISR@PRIM) or some other panel to include the IGYTCARB invocation. Panel ISR@PRIM is in your site’s PDF panel data set (normally ISRPLIB).

The following example shows an ISR@PRIM panel modified, in two identified locations, to include the IGYTCARB invocation. If you add or change an option in the upper portion of the panel definition, you must also add or change the corresponding line on the lower portion of the panel.

```
%----- ISPF/PDF PRIMARY OPTION PANEL -----+
%OPTION ==>_ZCMD
%
%                                +USERID  - &ZUSER
% 0 +ISPF PARMS - Specify terminal and user parameters +TIME    - &ZTIME
% 1 +BROWSE     - Display source data or output listings +TERMINAL - &ZTERM
% 2 +EDIT      - Create or change source data           +PF KEYS  - &ZKEYS
% 3 +UTILITIES - Perform utility functions
% 4 +FOREGROUND - Invoke language processors in foreground
% 5 +BATCH     - Submit to batch for language processing
% 6 +COMMAND   - Enter CMS command or EXEC
% 7 +DIALOG TEST - Perform dialog testing
% 8 +LM UTILITIES- Perform library management utility functions
% C +IGYTCARB  - Run IGYTCARB UPDATE TRANSACTION PROGRAM (1)
% T +TUTORIAL  - Display information about ISPF/PDF
% X +EXIT      - Terminate using console, log, and list defaults
%
%
+Enter%END+command to terminate ISPF.
%
```

```

)INIT
  .HELP = ISR00003
  &ZPRIM = YES          /* ALWAYS A PRIMARY OPTION MENU */
  &ZHTOP = ISR00003    /* TUTORIAL TABLE OF CONTENTS */
  &ZHINDEX = ISR91000 /* TUTORIAL INDEX - 1ST PAGE */
  VPUT (ZHTOP,ZHINDEX) PROFILE
)PROC
  &Z1 = TRUNC(&ZCMD,1)
  IF (&Z1 &notsym.= '.' )
    &ZSEL = TRANS( TRUNC (&ZCMD, '.' )
      0, 'PANEL(ISPOPTA)'
      1, 'PGM(ISRBRO) PARM(ISRBRO01)'
      2, 'PGM(ISREDIT) PARM(P,ISREDM01)'
      3, 'PANEL(ISRUTIL)'
      4, 'PANEL(ISRFPA)'
      5, 'PGM(ISRJB1) PARM(ISRJPA) NOCHECK'
      6, 'PGM(ISRPCC)'
      7, 'PGM(ISRYXDR) NOCHECK'
      8, 'PANEL(ISRLPRIM)'
      C, 'PGM(IGYTCARB)'
      T, 'PGM(ISPTUTOR) PARM(ISR00000)'
      ' ', ' '
      X, 'EXIT'
      *, '?' )
    &ZTRAIL = .TRAIL
  IF (&Z1 = '.' ) .msg = ISPD141
)END

```

As indicated by (1) in this example, you add IGYTCARB to the upper portion of the panel by entering:

```
% C +IGYTCARB - Run IGYTCARB UPDATE TRANSACTION PROGRAM
```

You add the corresponding line on the lower portion of the panel, indicated by (2), by entering:

```
C, 'PGM(IGYTCARB)'
```

2. Place ISR@PRIM (or your other modified panel) and IGYTPNL in a library and make this library the first library in the ISPLLIB concatenation.
3. If necessary, comment sequence line IB2200 and uncomment sequence line IB2210 in IGYTCARB. The OPEN EXTEND verb is supported under OS/390.
4. Compile and link-edit IGYTCARB and place the resulting load module in your LOADLIB.

5. Allocate ISPLLIB using the following command:

```
ALLOCATE FILE(ISPLLIB) DATASET(DSN1, SYS1.COBLIB, DSN2) SHR REUSE
```

Here *DSN1* is the library name of the LOADLIB from step 4. *DSN2* is your installed ISPLLIB.

6. Allocate the input and output data sets using the following command:

```
ALLOCATE FILE(UPDTRANS) DA('IGY.V2R2M0.SIGYSAMP(IGYTRANB)) SHR REUSE
```

7. Allocate ISPLLIB using the following command:

```
ALLOCATE FILE(ISPLLIB) DATASET(DSN3, DSN4) SHR REUSE
```

Here *DSN3* is the library containing the modified panels. *DSN4* is the ISPF panel library.

8. Invoke IGYTCARB using your modified panel.

#### RELATED REFERENCES

*Interactive System Productivity Facility Dialog Manager Guide*



## Running IGYTCARB under CMS

The following procedure is for a combined compile, link-edit, and run of the IGYTCARB program. If you want only to compile or only to compile and link-edit the program, you need to change this procedure.

To run IGYTCARB under CMS, use the following steps:

1. Using the ISPF editor, change the ISPF/PDF Primary Option Panel (ISR@PRIM) or some other panel to include the IGYTCARB invocation. Panel ISR@PRIM is in your site's PDF panel data set (normally ISRPLIB).

Running IGYTCARB under OS/390 shows an ISR@PRIM panel modified, in two identified locations, to include the IGYTCARB invocation. If you add or change an option in the upper portion of the panel definition, you must also add or change the corresponding line on the lower portion of the panel.

As indicated by **(1)** in Running IGYTCARB under OS/390, IGYTCARB is added to the upper portion of the panel by entering:

```
% C +IGYTCARB - Run IGYTCARB UPDATE TRANSACTION PROGRAM
```

The corresponding line on the lower portion of the panel, indicated by **(2)**, is added by entering:

```
C, 'PGM(IGYTCARB) CMS'
```

2. Place ISR@PRIM (or your other modified panel) and IGYTPANEL in a MACLIB using the following command:

```
MACLIB GEN IGYTCAR ISR@PRIM IGYTPNL
```

Make this MACLIB the first library in the ISPLIB concatenation by using the following FILEDEF command (where "\*" designates a disk):

```
FILEDEF ISPLIB DISK IGYTCAR MACLIB * (PERM CONCAT
```

3. Comment sequence line IB2210 and uncomment sequence line IB2200 in IGYTCARB. The OPEN EXTEND statement is not supported under CMS.
4. Concatenate SCEELKED TXTLIB with ISPLIB by issuing the following FILEDEF command (where "\*" designates a disk):  
FILEDEF ISPLIB DISK SCEELKED TXTLIB \* (PERM CONCAT
5. Concatenate SCEERUN LOADLIB with ISPLIB by issuing the following FILEDEF command (where "\*" designates a disk):  
FILEDEF ISPLIB DISK SCEERUN LOADLIB \* (PERM CONCAT
6. Access the COBOL for OS/390 & VM library:  
GLOBAL TXTLIB SCEELKED  
GLOBAL LOADLIB SCEERUN
7. Compile IGYTCARB and make IGYTCARB TEXT available on your system.
8. Identify the data sets that are used by issuing the following FILEDEF commands (where "\*" designates a disk):  
FILEDEF UPDTRANS DISK IGYTRANB OUTPUT \* (PERM DISP MOD
9. Invoke IGYTCARB using the modified panel.

### RELATED REFERENCES

*Interactive System Productivity Facility Dialog Manager Guide*

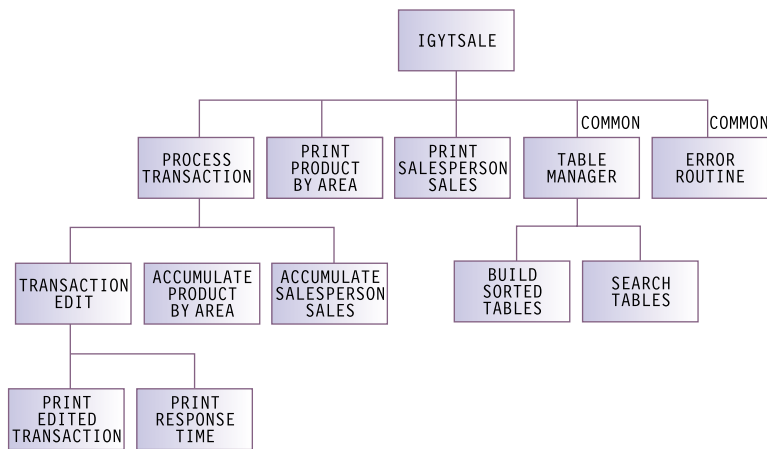
---

## IGYTSALE: nested program application

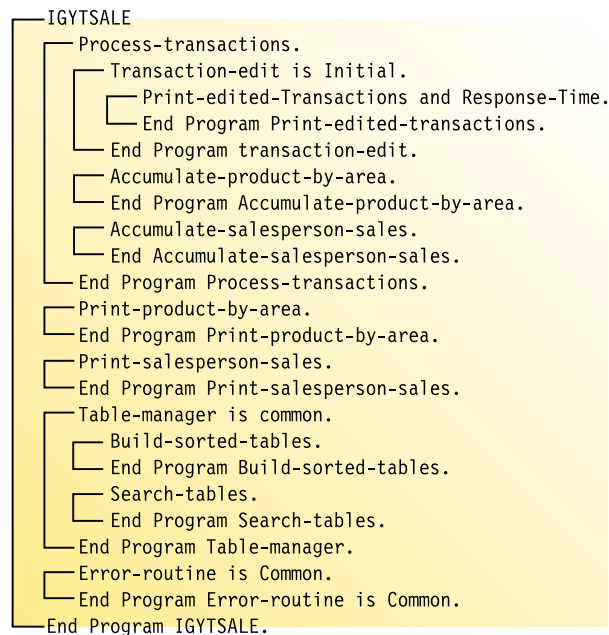
A sporting goods distributor wants to track product sales and sales commissions. This nested program application needs to perform the following tasks:

1. Keep a record of the product line, customers, and number of salespeople. This data is stored in a file called IGYTABLE.
2. Maintain a file that records valid transactions and transaction errors. All transactions that are not valid are flagged and the results are printed in a report. Transactions to be processed are in a file called IGYTRANA.
3. Process transactions and report sales by location.
4. Record an individual's sales performance and commission and print the results in a report.
5. Report the sale and shipment dates in local time and UTC (Universal Time Coordinate), respectively, and calculate the response time.

The following diagram shows the parts of the application as a hierarchy:



The following diagram shows how the parts are nested:



#### RELATED TASKS

“Running IGYTSALE” on page 602

RELATED REFERENCES

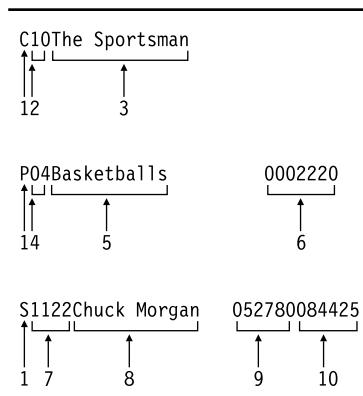
"Input data for IGYTSALE"

"Reports produced by IGYTSALE" on page 598

"Language elements and concepts that are illustrated" on page 604

## Input data for IGYTSALE

As input to our program, the distributor collected information about its customers, salespeople, and products, coded the information, and produced an input file. This input file, called IGYTABLE, is loaded into three separate tables for use during transaction processing. The format of the file is as follows, with an explanation of the items below:



1. Record type
2. Customer code
3. Customer name
4. Product code
5. Product description
6. Product unit price
7. Salesperson number
8. Salesperson name
9. Date of hire
10. Commission rate

The value of field 1 (C, P, or S) determines the format of the input record. The following sample shows a section of IGYTABLE:

```
S1111Edyth Phillips 062484042327
S1122Chuck Morgan 052780084425
S1133Art Tung 022882061728
S1144Billy Jim Bob 010272121150
S1155Chris Preston 122083053377
S1166Al Willie Roz 111276100000
P01Footballs 0000620
P02Football Equipment 0032080
P03Football Uniform 0004910
P04Basketballs 0002220
P05Basketball Rim/Board0008830
P06Basketball Uniform 0004220
C01L. A. Sports
C02Gear Up
C03Play Outdoors
C04Sports 4 You
C05Sports R US
```

C06Stay Active  
 C07Sport Shop  
 C08Stay Sporty  
 C09Hot Sports  
 C10The Sportsman  
 C11Playing Ball  
 C12Sports Play  
 . . .

In addition, the distributor collected information about sales transactions. Each transaction represents an individual salesperson's sales to a particular customer. The customer can purchase from one to five items during each transaction. The transaction information is coded and put into an input file, called IGYTRANA. The format of this file is as follows, with an explanation of the items below:

```

  B11123919901110123314SAN DIEGO 11660919901114235505260200270500110522250100140010
  ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
  1   2   3   4   5   6   7   8   9   8   9   8   9   8   9   8   9
  
```

1. Sales order number
2. Invoiced items (number of different items ordered)
3. Date of sale (year month day hour minutes seconds)
4. Sales area
5. Salesperson number
6. Customer code
7. Date of shipment (year month day hour minutes seconds)
8. Product code
9. Quantity sold

Fields 8 and 9 occur one to eight times depending on the number of different items ordered (field 2). The following sample shows a section of IGYTRANA:

```

A00001119900227010101CNTRL VALLEY11442019900228259999
A00004119900310100530CNTRL VALLEY11441019900403150099
A00005119900418222409CNTRL VALLEY11441219900419059900
A00006119900523151010CNTRL VALLEY11442019900623250004
    419990324591515SAN DIEGO    11615    60200132200110522045100
B11114419901111003301SAN DIEGO  11661519901114260200132200110522041100
A00007119901115003205CNTRL VALLEY11332019901117120023
C00125419900118101527SF BAY AREA 11331519900120160200112200250522145111
B11116419901201132013SF BAY AREA 11331519901203060200102200110522045102
B11117319901201070833SAN Diego  11656619901203330200132200120522041100
B11118419901221191544SAN DIEGO  11661419901223160200142200130522040300
B11119419901210211544SAN DIEGO  11221219901214060200152200160522050500
B11120419901212000816SAN DIEGO  11220419901213150200052200160522040100
B11121419901201131544SAN DIEGO  11330219901203120200112200140522250100
B11122419901112073312SAN DIEGO  11221019901113100200162200260522250100
B11123919901110123314SAN DIEGO  11660919901114260200270500110522250100140010
B11124219901313510000SAN DIEGO  116611    1 0200042200120a22141100
B11125419901215012510SAN DIEGO  11661519901216110200162200130522141111
B11126119901111000034SAN DIEGO  11331619901113260022
B11127119901110154100SAN DIEGO  11221219901113122000
B11128419901110175001SAN DIEGO  11661519901113260200132200160521041104
  . . .
  
```

## Reports produced by IGYTSALE

The following figures are samples of IGYTSALE output. The program records the following data in reports:

- Transaction errors
- Sales by product and area
- Individual sales performance and commissions
- Response time between the sale date and the date the sold products are shipped

Your output might vary slightly in appearance, depending on your system.

“Example: IGYTSALE transaction errors”

“Example: IGYTSALE sales analysis by product by area” on page 600

“Example: IGYTSALE sales and commissions” on page 601

“Example: IGYTSALE response time from sale to ship” on page 602

### Example: IGYTSALE transaction errors

The following sample of IGYTSALE output shows transaction errors in the last column.

Day of Report: Tuesday		C O B O L		S P O R T S		11/30/1999 03:12		Page: 1
Sales Order	Inv. Items	Sales Time Stamp	Sales Area	Sales Pers	Cust. Code	Product	And Quantity Sold	Ship Date Stamp
	4	19990324591515	SAN DIEGO	116	15	60200132200110522045100		Error Descriptions -Sales order number is missing -Date of sale time stamp is invalid -Salesperson number not numeric -Product code not in product-table -Date of ship time stamp is invalid
B11117	3	19901201070833	SAN Diego	1165	66	33020o132200120522041100		19901203 Error Descriptions -Sales area not in area-table -Salesperson not in sales-per-table -Customer code not in customer-table -Product code not in product-table -Quantity sold not numeric
B11123	9	19901110123314	SAN DIEGO	1166	09	260200270500110522250100140010		19901114 Error Descriptions -Invoiced items is invalid -Product and quantity not checked -Date of ship time stamp is invalid
B11124	2	19901313510000	SAN DIEGO	1166	11	1 0200042200120a22141100		Error Descriptions -Date of sale time stamp is invalid -Product code is invalid -Date of ship time stamp is invalid
	133	81119110000	LOS ANGELES	1166	10	040112110210160321251104		Error Descriptions -Sales order number is invalid -Invoiced items is invalid -Date of sale time stamp is invalid -Product and quantity not checked -Date of ship time stamp is invalid
C11133	4	1990111944		1166	10	040112110210160321251104		Error Descriptions -Date of sale time stamp is invalid -Sales area is missing -Date of ship time stamp is invalid
C11138	4	19901117091530	LOS ANGELES	1155		113200102010260321250004		19901119 Error Descriptions -Customer code is invalid
D00009	9	19901201222222	CNTRL COAST	115	19	141 1131221		19901202 Error Descriptions -Invoiced items is invalid

**Example: IGYTSALE sales analysis by product by area**  
 The following sample of IGYTSALE output shows sales by product and area.

Day of Report: Tuesday C O B O L S P O R T S 11/30/1999 03:12 Page: 1  
 Sales Analysis By Product By Area

Product Codes	Areas of Sale						Product Totals
	CNTRL COAST	CNTRL VALLEY	LOS ANGELES	NORTH COAST	SAN DIEGO	SF BAY AREA	
Product Number 04 Basketballs							
Units Sold			433		2604	5102	8139
Unit Price			22.20		22.20	22.20	
Amount of Sale			\$9,612.60		\$57,808.80	\$113,264.40	\$180,685.80
Product Number 05 Basketball Rim/Board							
Units Sold		9900	2120	11	2700		14731
Unit Price		88.30	88.30	88.30	88.30		
Amount of Sale		\$874,170.00	\$187,196.00	\$971.30	\$238,410.00		\$1,300,747.30
Product Number 06 Basketball Uniform							
Units Sold				990	200	200	1390
Unit Price				42.20	42.20	42.20	
Amount of Sale				\$41,778.00	\$8,440.00	\$8,440.00	\$58,658.00
Product Number 10 Baseball Cage							
Units Sold	45		3450	16	200	3320	7031
Unit Price	890.00		890.00	890.00	890.00	890.00	
Amount of Sale	\$40,050.00		\$3,070,500.00	\$14,240.00	\$178,000.00	\$2,954,800.00	\$6,257,590.00
Product Number 11 Baseball Uniform							
Units Sold	10003		3578		2922	2746	19249
Unit Price	45.70		45.70		45.70	45.70	
Amount of Sale	\$457,137.10		\$163,514.60		\$133,535.40	\$125,492.20	\$879,679.30
Product Number 12 Softballs							
Units Sold	10	137	2564	13	2200	22	4946
Unit Price	1.40	1.40	1.40	1.40	1.40	1.40	
Amount of Sale	\$14.00	\$191.80	\$3,589.60	\$18.20	\$3,080.00	\$30.80	\$6,924.40
Product Number 13 Softball Bats							
Units Sold	3227		3300	1998	5444	99	14068
Unit Price	12.60		12.60	12.60	12.60	12.60	
Amount of Sale	\$40,660.20		\$41,580.00	\$25,174.80	\$68,594.40	\$1,247.40	\$177,256.80
Product Number 14 Softball Gloves							
Units Sold	1155		136	3119	3833	5152	13395
Unit Price	12.00		12.00	12.00	12.00	12.00	
Amount of Sale	\$13,860.00		\$1,632.00	\$37,428.00	\$45,996.00	\$61,824.00	\$160,740.00
Product Number 15 Softball Cage							
Units Sold	997	99	2000		2400		5496
Unit Price	890.00	890.00	890.00		890.00		
Amount of Sale	\$887,330.00	\$88,110.00	\$1,780,000.00		\$2,136,000.00		\$4,891,440.00
Product Number 16 Softball Uniform							
Units Sold	44		465	16	6165	200	6890
Unit Price	45.70		45.70	45.70	45.70	45.70	
Amount of Sale	\$2,010.80		\$21,250.50	\$731.20	\$281,740.50	\$9,140.00	\$314,873.00
Product Number 25 RacketBalls							
Units Sold	1001	10003	1108	8989	200	522	21823
Unit Price	0.60	0.60	0.60	0.60	0.60	0.60	
Amount of Sale	\$600.60	\$6,001.80	\$664.80	\$5,393.40	\$120.00	\$313.20	\$13,093.80
Product Number 26 Racketball Rackets							
Units Sold	21		862	194	944	31	2052
Unit Price	12.70		12.70	12.70	12.70	12.70	
Amount of Sale	\$266.70		\$10,947.40	\$2,463.80	\$11,988.80	\$393.70	\$26,060.40
Total Units Sold	16503	20139	20016	15346	29812	17394 *	119210 *
Total Sales	\$1,441,929.40	\$968,473.60	\$5,290,487.50	\$128,198.70	\$3,163,713.90	\$3,274,945.70 *	\$14,267,748.80 *

## Example: IGYTSALE sales and commissions

The following sample of IGYTSALE output shows sales performance and commissions by salesperson.

```

Day of Report: Tuesday          C O B O L   S P O R T S   11/30/1999   03:12   Page: 1
                                Sales and Commission Report

Salesperson: Billy Jim Bob
Customers:      Number of  Products  Total for  Discount  Discount  Commission
                Orders    Ordered   Order     (if any)  Amount    Earned
-----
Sports Stop      3         10117    $6,161.40  2.25%     $138.63    $746.45
The Sportsman    1           99      $88,110.00  5.06%     $4,458.36  $10,674.52
Sports Play      1          9900    $874,170.00  7.59%     $66,349.50  $105,905.69
-----
Totals:         5         20116    $968,441.40                $70,946.49  $117,326.66

Salesperson: Willie Al Roz
Customers:      Number of  Products  Total for  Discount  Discount  Commission
                Orders    Ordered   Order     (if any)  Amount    Earned
-----
Winners Club     4         13998    $1,572,775.90  7.59%     $119,373.69  $157,277.59
Winning Sports   1          3222     $48,777.20  3.38%     $1,648.66    $4,877.72
The Sportsman    1          1747     $27,415.50  3.38%           $926.64    $2,741.55
Play Outdoors   1          2510     $18,579.60  3.38%           $627.99    $1,857.96
-----
Totals:         7         21477    $1,667,548.20                $122,576.98  $166,754.82

Salesperson: Art Tung
Customers:      Number of  Products  Total for  Discount  Discount  Commission
                Orders    Ordered   Order     (if any)  Amount    Earned
-----
Sports Stop      1           23         $32.20      2.25%           $ .72         $1.98
Winners Club     2         16057    $2,274,885.00  7.59%     $172,663.77  $140,424.10
Gear Up          1          3022     $107,144.00  7.59%     $8,132.22    $6,613.78
Sports Club      1           22         $279.40     2.25%           $6.28        $17.24
Sports Fans Shop 1          1044     $20,447.30  3.38%           $691.11     $1,262.17
L. A. Sports     1          1163     $979,198.10  7.59%     $74,321.13  $60,443.94
-----
Totals:         7         21331    $3,381,986.00                $255,815.23  $208,763.21

Salesperson: Chuck Morgan
Customers:      Number of  Products  Total for  Discount  Discount  Commission
                Orders    Ordered   Order     (if any)  Amount    Earned
-----
Sports Play      3          7422    $3,817,245.40  7.59%     $289,728.92  $322,270.94
Sports 4 You     1          3022     $398,335.40  7.59%     $30,233.65   $33,629.46
The Sportsman    1          3022     $285,229.40  7.59%     $21,648.91   $24,080.49
Sports 4 Winners 1          1100     $68,509.40  5.06%     $3,466.57    $5,783.90
Sports Club      1         12027    $1,324,256.10  7.59%     $100,511.03  $111,800.32
-----
Totals:         7         26593    $5,893,575.70                $445,589.08  $497,565.11

Salesperson: Chris Preston
Customers:      Number of  Products  Total for  Discount  Discount  Commission
                Orders    Ordered   Order     (if any)  Amount    Earned
-----
Playing Ball     1          5535    $1,939,219.10  7.59%     $147,186.72  $103,509.69
Play Sports     1          5675    $225,130.80  7.59%     $17,087.42   $12,016.80
Winners Club    1           631     $14,069.70  2.25%           $316.56     $750.99
The Jock Shop    1          2332     $28,716.60  3.38%           $970.62     $1,532.80
-----
Totals:         4         14173    $2,207,136.20                $165,561.32  $117,810.28

Salesperson: Edyth Phillips
Customers:      Number of  Products  Total for  Discount  Discount  Commission
                Orders    Ordered   Order     (if any)  Amount    Earned
-----
Sports Play      2          3575     $92,409.90  5.06%     $4,675.94    $3,911.43
Winning Sports   1         11945    $56,651.40  5.06%     $2,866.56    $2,397.88
-----
Totals:         3         15520    $149,061.30                $7,542.50     $6,309.31
Grand Totals:   33        119210    $14,267,748.80                $1,068,031.60  $1,114,529.39
    
```

## Example: IGYTSALE response time from sale to ship

The following sample of IGYTSALE output shows response time between the sale date in the United States and the date the sold products are shipped to Europe.

```
Day of Report: Tuesday          C O B O L   S P O R T S          11/30/1999   03:12   Page:  1
                                Response Time from USA Sale to European Ship
Prod  Units  Sale Date/Time(PST)  Ship Date  Ship  Response Time
Code  Sold    YYYYMMDD  HHMMSS    YYYYMMDD  Day    Days
-----
25    9999    19900226  010101    19900228  WED    .95
15     99     19900310  100530    19900403  TUE   23.57
05    9900    19900418  222409    19900419  THU    .06
25     4      19900523  151010    19900623  SAT   30.36
04    1100    19901110  003301    19901114  WED    2.97
12     23     19901114  003205    19901117  SAT    1.97
14    5111    19900118  101527    19900120  SAT    1.57
04    5102    19901201  132013    19901203  MON    1.44
04     300    19901221  191544    19901223  SUN    1.19
05     500    19901210  211544    19901214  FRI    3.11
04     100    19901211  000816    19901213  THU    .99
25     100    19901201  131544    19901203  MON    1.44
25     100    19901112  073312    19901113  TUE    .68
14    1111    19901214  012510    19901216  SUN    .94
26     22     19901110  000034    19901113  TUE    1.99
12    2000    19901110  154100    19901113  TUE    2.34
04    1104    19901110  175001    19901113  TUE    2.25
12     114    19901229  115522    19901230  SUN    .50
15    2000    19901110  190113    19901114  WED    3.20
10    1440    19901112  001500    19901115  THU    1.98
25    1104    19901118  120101    19901119  MON    .49
25     4      19901118  110030    19901119  MON    .54
12    144    19901114  010510    19901119  MON    3.95
14    112    19901119  010101    19901122  THU    1.95
26    321    19901117  173945    19901119  MON    1.26
13    1221    19901101  135133    19901102  FRI    .42
10     22     19901029  210000    19901030  TUE    .12
14     35     19901130  160500    19901201  SAT    .32
11    9005    19901211  050505    19901212  WED    .78
06     990    19900511  214409    19900515  TUE    3.09
13    1998    19900712  150100    19900716  MON    3.37
26     31     19901010  185559    19901011  THU    .21
14     30     19901210  195500    19901212  WED    1.17
```

## Running IGYTSALE

You can run IGYTSALE under OS/390 or CMS.

All the files required by the IGYTSALE program are supplied on the product installation tape. These files (IGYTSALE, IGYTCRC, IGYTPRC, IGYTSRC, IGYTABLE, IGYTRANA) are located in the IGY.V2R2M0.SIGYSAMP data set.

Data set and procedure names can be changed at installation time. Check with your system programmer to verify these names.

Do not change these options on the CBL card in the source file for IGYTSALE:

```
LIB
NOCMPR2
NONUMBER
SEQUENCE
NOFLAGMIG
NONUMBER
QUOTE
```

With these options in effect, the program might not cause any diagnostic messages to be issued. You can use the sequence number string in the source file to search for the language elements used.

When you run IGYTSALE, the following messages are printed to the SYSOUT data set:

```
Program IGYTSALE Begins
There were 00041 records processed in this program
Program IGYTSALE Normal End
```



#### RELATED CONCEPTS

"IGYTSALE: nested program application" on page 595

#### RELATED TASKS

"Running IGYTSALE under OS/390"

"Running IGYTSALE under CMS"

#### RELATED REFERENCES

"Input data for IGYTSALE" on page 597

"Reports produced by IGYTSALE" on page 598

"Language elements and concepts that are illustrated" on page 604

### Running IGYTSALE under OS/390

The following procedure does a combined compile, link-edit, and run of the IGYTSALE program. If you want only to compile or only to compile and link-edit the program, you need to change the IGYWCLG cataloged procedure.

Use the following JCL to compile, link-edit, and run the IGYTSALE program. Insert the information for your system or installation in the fields that are shown in lowercase letters (accounting information).

```
//IGYTSALE JOB (acct-info), 'IGYTSALE',MSGLEVEL=(1,1),TIME=(0,29)
//TEST EXEC IGYWCLG
//COBOL.SYSLIB DD DSN=IGY.V2R2M0.SIGYSAMP,DISP=SHR
//COBOL.SYSIN DD DSN=IGY.V2R2M0.SIGYSAMP(IGYTSALE),DISP=SHR
//GO.SYSOUT DD SYSOUT=A
//GO.IGYTABLE DD DSN=IGY.V2R2M0.SIGYSAMP(IGYTABLE),DISP=SHR
//GO.IGYTRANS DD DSN=IGY.V2R2M0.SIGYSAMP(IGYTRANA),DISP=SHR
//GO.IGYPRINT DD SYSOUT=A,DCB=BLKSIZE=133
//GO.IGYPRT2 DD SYSOUT=A,DCB=BLKSIZE=133
//
```

### Running IGYTSALE under CMS

The following procedure does a combined compile, link-edit, and run of the IGYTSALE program. If you want only to compile or only to compile and link-edit the program, you need to use the COBOL2, LOAD, START, GENMOD, LKED, and OSRUN commands to control how the program is compiled, link-edited, and run.

To run IGYTSALE under CMS, first access the COBOL for OS/390 & VM product using the CP LINK and ACCESS commands, and then do the following steps:

1. Issue the following FILEDEF and GLOBAL MACLIB commands:

```
FILEDEF * CLEAR
FILEDEF SYSLIB DISK SIGYSAMP MACLIB *
GLOBAL MACLIB SIGYSAMP
```

Here "\*" designates a disk. A library containing copy members needed by IGYTSALE is provided in SIGYSAMP MACLIB.

2. Access the COBOL for OS/390 & VM library:

```
GLOBAL TXTLIB SCEELKED
```

3. Compile the program by using the COBOL2 command:

```
COBOL2 IGYTSALE
```

At the end of the compilation, LISTING and TEXT files are created and sent to your disk.

4. Identify the data sets that are used by issuing FILEDEF commands (where "\*" designates a disk):

```

FILEDEF SYSOUT  DISK IGYTSALE SYSOUT  A
FILEDEF SYSIN   DISK IGYTSALE SYSIN   *
FILEDEF IGYTABLE DISK IGYTABLE INPUT  *
FILEDEF IGYTRANS DISK IGYTRANA INPUT  *
FILEDEF IGYPRINT DISK IGYPRINT OUTPUT *
FILEDEF IGYPRT2 DISK IGYPRT2  OUTPUT *

```

5. Run the program by using the LOAD and START commands:

```

LOAD IGYTSALE
START *

```

**RELATED TASKS**

“Accessing the compiler (CP LINK and ACCESS)” on page 243

## Language elements and concepts that are illustrated

To find the applicable language element for a sample program, locate the abbreviation for that program in the sequence string:

Sample program	Abbreviation
IGYTCARA	IA
IGYTCARB	IB
IGYTSALE	IS

The following table lists the language elements and programming concepts that the sample programs illustrate. The language element or concept is described, and the sequence string is shown. The sequence string is the special character string that appears in the sequence field of the source file. You can use this string as a search argument for locating the elements in the listing.

Language element or concept	Sequence string
ACCEPT . . . FROM DAY-OF-WEEK	IS0900
ACCEPT . . . FROM DATE	IS0901
ACCEPT . . . FROM TIME	IS0902
ADD . . . TO	IS4550
AFTER ADVANCING	IS2700
AFTER PAGE	IS2600
ALL	IS4200
ASSIGN	IS1101
AUTHOR	IA0040
CALL	IS0800
Callable services (Language Environment)	
CEEDATM - format date or time output	IS0875, IS2575
CEEDCOD - feedback code check from service call	IS0905
CEEGMTO - UTC offset from local time	IS0904
CEELOCT - local date and time	IS0850
CEESECS - convert timestamp to seconds	IS2350, IS2550
CLOSE files	IS1900
Comma, semicolon, and space interchangeable	IS3500, IS3600
COMMON statement for nested programs	IS4600
Complex OCCURS DEPENDING ON	IS0700, IS3700
COMPUTE	IS4501

Language element or concept	Sequence string
COMPUTE ROUNDED	IS4500
CONFIGURATION SECTION	IA0970
CONFIGURATION SECTION (optional)	IS0200
CONTINUE statement	IA5310, IA5380
COPY statement	IS0500
DATA DIVISION (optional)	IS5100
Data validation	IA5130-6190
Do-until (PERFORM . . . TEST AFTER)	IA4900-5010, IA7690-7770
Do-while (PERFORM . . . TEST BEFORE)	IS1660
END-ADD	IS2900
END-COMPUTE	IS4510
END-EVALUATE	IA6590, IS2450
END-IF	IS1680
END-MULTIPLY	IS3100
END-PERFORM	IS1700
END PROGRAM	IA9990
END-READ	IS1800
END-SEARCH	IS3400
ENVIRONMENT DIVISION (optional)	IS0200
Error handling, termination of program	IA4620, IA5080, IA7800-7980
EVALUATE statement	IA6270-6590
EVALUATE . . . ALSO	IS2400
EXIT PROGRAM not only statement in paragraph	IS2000
Exponentiation	IS4500
EXTERNAL clause	IS1200
FILE-CONTROL entry for sequential file	IA1190-1300
FILE-CONTROL entry for VSAM indexed file	IA1070-1180
FILE SECTION (optional)	IS0200
FILE STATUS code check	IA4600-4630, IA4760-4790
FILLER (optional)	IS0400
Flags, level-88, definition	IA1730-1800, IA2440-2480, IA2710
Flags, level-88, testing	IA4430, IA5200-5250
FLOATING POINT	IS4400
GLOBAL statement	IS0300
INITIAL statement for nested programs	IS2300
INITIALIZE	IS2500
Initializing a table in the DATA DIVISION	IA2920-4260
Inline PERFORM statement	IA4410-4520
I-O-CONTROL paragraphs (optional)	IS0200
INPUT-OUTPUT SECTION (optional)	IS0200

Language element or concept	Sequence string
Intrinsic functions:	
CURRENT-DATE	IA9005
MAX	IA9235
MEAN	IA9215
MEDIAN	IA9220
MIN	IA9240
STANDARD-DEVIATION	IA9230
UPPER-CASE	IA9015
VARIANCE	IA9225
WHEN-COMPILED	IA9000
IS (optional in all clauses)	IS0700
LABEL RECORDS (optional)	IS1150
LINKAGE SECTION	IS4900
Mixing of indexes and subscripts	IS3500
Mnemonic names	IA1000
MOVE	IS0903
MOVE CORRESPONDING statement	IA4810, IA4830
MULTIPLY . . . GIVING	IS3000
Nested IF statement, using END-IF	IA5460-5830
Nested program	IS1000
NEXT SENTENCE	IS4300
NOT AT END	IS1600
NULL	IS4800
OBJECT-COMPUTER (optional)	IS0200
OCCURS DEPENDING ON	IS0710
ODO uses maximum length for receiving item	IS1550
OPEN EXTEND	IB2210
OPEN INPUT	IS1400
OPEN OUTPUT	IS1500
ORGANIZATION (optional)	IS1100
Page eject	IA7180-7210
Parenthesis in abbreviated conditions	IS4850
PERFORM . . . WITH TEST AFTER ("Do-Until")	IA4900-5010, IA7690-7770
PERFORM . . . WITH TEST BEFORE ("Do-While")	IS1660
PERFORM . . . UNTIL	IS5000
PERFORM . . . VARYING statement	IA7690-7770
POINTER function	IS4700
Print file FD entry	IA1570-1620
Print report	IA7100-7360
PROCEDURE DIVISION . . . USING	IB1320-IB1650
PROGRAM-ID (30 characters allowed)	IS0120
READ .. INTO . . . AT END	IS1550
REDEFINES statement	IA1940, IA2060, IA2890, IA3320

Language element or concept	Sequence string
Reference modification	IS2425
Relational operator <= (less than or equal)	IS4400
Relational operator >= (greater than or equal)	IS2425
Relative subscripting	IS4000
REPLACE	IS4100
SEARCH statement	IS3300
SELECT	IS1100
Sequence number can contain any character	IA, IB, IS
Sequential file processing	IA4480-4510, IA4840-4870
Sequential table search, using PERFORM	IA7690-7770
Sequential table search, using SEARCH	IA5270-5320, IA5340-5390
SET INDEX	IS3200
SET . . . TO TRUE statement	IA4390, IA4500, IA4860, IA4980
SOURCE-COMPUTER (optional)	IS0200
SPECIAL-NAMES paragraph (optional)	IS0200
STRING statement	IA6950, IA7050
Support for lowercase letters	IS0100
TALLY	IS1650
TITLE statement for nested programs	IS0100
Update commuter record	IA6200-6610
Update transaction work value spaces	IB0790-IB1000
USAGE BINARY	IS1300
USAGE PACKED-DECIMAL	IS1301
Validate elements	IB0810, IB0860, IB1000
VALUE with OCCURS	IS0600
VALUE SPACE (S)	IS0601
VALUE ZERO (S) (ES)	IS0600
Variable-length table control variable	IA5100
Variable-length table definition	IA2090-2210
Variable-length table loading	IA4840-4990
VSAM indexed file key definition	IA1170
VSAM return-code display	IA7800-7900
WORKING-STORAGE SECTION	IS0250



---

## 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:

AIX	MVS
CICS	OpenEdition
CICS/ESA	OS/390
COBOL/370	RACF
DB2	SOM
DFSMS	SOMobjects
DFSORT	System Object Model
IBM	System/390
IMS	VisualAge
IMS/ESA	VM/ESA
Language Environment	

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 in 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 Btrieve Technologies, Inc.). 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.

**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.** All the valid characters for a programming language or a computer system.

**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.

\* **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 YYXXXX, 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 YYXXX is compatible with:

- Another windowed date field with DATE FORMAT YYXXX
- An expanded date field with DATE FORMAT YYYYXXX

\* **compile.** (1) To translate a program expressed in a high-level language into a program expressed in an intermediate language, assembly language, or a computer language. (2) To prepare a machine-language program from a computer program written in another programming language by making use of the overall logic structure of the program, or generating more than one computer instruction for each symbolic statement, or both, as well as performing the function of an assembler.

\* **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.** See *dynamic link library (DLL)*.

**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 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 (HFS).** A collection of files and directories that are organized in a hierarchical structure and can be accessed by using OS/390 UNIX System Services.

\* **high-order end.** The leftmost character of a string of characters.

## I

**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 COBOL for OS/390 & VM bootstrap routine. It must be link-edited with any module that contains a COBOL for OS/390 & VM 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.

\* **intra-record 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-0-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-0-CONTROL entry.** An entry in the I-0-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-0 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 COBOL for MVS & VM, COBOL for OS/390 & 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.

. 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.

**nodate.** 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-0, 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-0-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*.

**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).

**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.

**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.

\* **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*.

\* **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

<b>Character</b>	<b>Meaning</b>
IS NOT LESS THAN	Not less than
IS NOT <	Not less than
IS EQUAL TO	Equal to
IS =	Equal to
IS NOT EQUAL TO	Not equal to
IS NOT =	Not equal to
IS GREATER THAN OR EQUAL TO	Greater than or equal to
IS >=	Greater than or equal to
IS LESS THAN OR EQUAL TO	Less than or equal to
IS <=	Less than or equal to

\* **relation character.** A character that belongs to the following set:

<b>Character</b>	<b>Meaning</b>
>	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*.

\* **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)

Character	Meaning
;	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

---

### COBOL for OS/390 & VM

*Compiler and Run-Time Migration Guide*, GC26-4764  
*Customization under OS/390*, GC26-9045  
*Debug Tool User's Guide and Reference*, SC09-2137  
*Diagnosis Guide*, GC26-9047  
*Fact Sheet*, GC26-9048  
*Language Reference*, SC26-9046  
*Licensed Program Specifications*, GC26-9044  
*Programming Guide*, SC26-9049

---

### Related publications

#### VisualAge COBOL

*Fact Sheet*, GC26-9052  
*Getting Started*, GC26-8944  
*Language Reference*, SC26-9046  
*Programming Guide*, GC27-0812  
*Visual Builder User's Guide*, SC26-9053

#### 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

#### CICS/ESA

*Application Programming Guide*, GC33-1169  
*Application Programming Reference*, SC33-1170

*Sample Applications Guide*, SC33-1173

#### C/C++

*OS/390 C/C++ Run-Time Library Reference*, SC28-1663

#### DB/2 for OS/390

*Application Programming and SQL Guide*, SC26-3266

#### OS/390 DFSMS

*Access Method Services for Catalogs*, SC26-7326  
*Checkpoint/Restart*, SC26-7327  
*Macro Instructions for Data Sets*, SC26-7337

*Program Management*, SC27-0806

*Using Data Sets*, SC26-7339

*Utilities*, SC26-7343

#### DFSORT

*Application Programming Guide*, SC33-4035  
*Installation and Customization*, SC33-4034

#### IMS/ESA

*Application Programming: Client Server Object Manager Client/Server Application Programming Guide and Reference*, SC26-3483

*Application Programming: Client Server Object Manager Datastore Application Programming Guide and Reference*, SC26-3484

*Application Programming: Database Manager Summary*, SC26-8037

*Application Programming: Database Manager*, SC26-8015

*Application Programming: Design Guide*, SC26-8016

*Application Programming: EXEC DLI Commands for CICS and IMS Summary*, SC26-8036

*Application Programming: EXEC DLI Commands for CICS and IMS, SC26-8018*

*Application Programming: Transaction Manager Summary, SC26-8038*

*Application Programming: Transaction Manager, SC26-8017*

### **OS/390 ISPF**

*User's Guide, SC34-4791 & SC34-4792*

### **Language Environment for OS/390 & VM**

*Concepts Guide, GC28-1945*

*Debugging Guide and Run-Time Messages, SC28-1942*

*Installation & Customization, SC28-1941*

*Programming Guide, SC28-1939*

*Programming Reference, SC28-1940*

*Run-Time Migration Guide, SC28-1944*

*Writing Interlanguage Communication Applications, SC28-1943*

### **OS/390 MVS**

*JCL Reference, GC28-1757*

*JCL User's Guide, GC28-1758*

*System Codes, GC28-1780*

*System Commands, GC28-1781*

### **OS/390 SOMobjects**

*Configuration and Administration Guide, GC28-1851*

*Getting Started, GA22-7248*

*Messages, Codes, and Diagnosis, SC28-1996*

*Object Services, SC28-1995*

*Programmer's Guide, GC28-1859*

*Programmer's Reference Volume 1, SC28-1997*

*Programmer's Reference Volume 2, SC28-1998*

*Programmer's Reference Volume 3, SC28-1999*

*Reference Summary, SC28-1856*

### **OS/390 UNIX System Services**

*UNIX System Services Command Reference, SC28-1892*

*UNIX System Services Programming: Assembler Callable Services Reference, SC28-1899*

*UNIX System Services User's Guide, SC28-1891*

### **OS/390 TSO/E**

*Command Reference, SC28-1969*

*User's Guide, SC28-1968*

### **DB2 Server for VSE & VM**

*Application Programming for IBM VM Systems, SH09-8086*

*SQL Reference for IBM VM Systems, SH09-8087*

### **VM/ESA**

*CMS Application Development Guide, SC24-5450*

*CMS Application Development Reference, SC24-5451*

*CMS Command Reference, SC24-5461*

*CMS User's Guide, SC24-5460*

*CP Command and Utility Reference, SC24-5519*

*System Messages and Codes, SC24-5529*

*XEDIT Command and Macro Reference, SC24-5464*

*XEDIT User's Guide, SC24-5463*

### **Softcopy publications for OS/390 and VM**

The following collection kits contain IBM COBOL and related product publications.

*OS/390 Collection, SK2T-6700*

*VM Collection, SK2T-2067*



---

# Index

## Special Characters

- # cob2 option 240
- .asm file 281
- b cob2 option 239
- c cob2 command 239
- \*CBL statement 304
- \_CEE\_RUNOPTS environment variable
  - setting 363
- comprc\_ok cob2 option 239
- e cob2 option
  - specifying entry point 239
- g cob2 option
  - TEST option 239
- >>CALLINT statement 304
- I cob2 option
  - searching COPY files 239
- \_IGZ\_SYSOUT environment variable
  - setting 363
  - writing to stdout/stderr 29
- l cob2 option
  - specifying archive library name 240
- L cob2 option
  - specifying archive library path 240
- o cob2 option
  - specifying output file 240
- q cob2 option
  - specifying compiler options 240
- v cob2 option 240
- .wlist file 281

## Numerics

- 31-bit addressing mode
  - dynamic call 464
- 5203 - 5206 conditions 583

## A

- a extension with cob2 240
- a.out file from cob2 240
- abends
  - compile-time 271
- ACCEPT statement 28
  - reading from stdin 29
- access method services
  - build alternate indexes in advance 159
  - defining VSAM data sets, OS/390 153
  - loading a VSAM data set 146
  - under CMS 159
- ADATA compiler option 261
- adding records
  - to line-sequential files 167
  - to QSAM files 119
  - to VSAM files 149
- ADDRESS special register, CALL statement 476
- addresses
  - incrementing 480
  - NULL value 479

- addresses (*continued*)
  - passing entry point addresses 472
- ADEXIT suboption of EXIT compiler option 575
- adt extension with cob2 240
- ADV compiler option 262
- AIXBLD run-time option
  - affect on performance 549
- ALL subscript 45
- ALL31 run-time option 464
- ALLOCATE command (TSO) 215
  - with HFS files 215
- allocating data sets under TSO 215
- allocation
  - of files 105
  - of VSAM files 156
- ALPHABET clause, establishing collating sequence 8
- alphanumeric date fields, contracting 530
- alphanumeric literal
  - DBCS to alphanumeric conversion 92
  - with double-byte characters 90
- alternate collating sequence 179
- alternate entry point 473
- alternate index
  - creating 154
  - example of 155
  - password for 152
  - path 154
  - performance considerations 159
- ALTERNATE RECORD KEY 155
- alternate reserved-word table 301
  - CICS 351
- AMODE attribute
  - assigned for EXIT modules 576
- AMP parameter 156
- ANALYZE compiler option 261
- ANNUITY intrinsic function 48
- ANSI85 translator option 350
- APIs, UNIX and POSIX
  - calling 364
- APOST compiler option 289
- APPLY WRITE-ONLY clause 12
- arguments
  - describing in calling program 477
  - IDL passing conventions 435
- ARITH compiler option 262
- arithmetic
  - COMPUTE statement simpler to code 43
  - error handling 191
  - with intrinsic functions 44
- arithmetic comparisons 50
- arithmetic evaluation
  - data format conversion 39
  - examples 49
  - fixed-point versus floating-point 49
  - intermediate results 559
  - performance tips 537
  - precedence 561

- arithmetic evaluation (*continued*)
  - precision 559
- arithmetic expression
  - as reference modifier 88
  - description of 43
  - in nonarithmetic statement 568
  - in parentheses 43
  - with MLE 525
- arithmetic operation
  - with MLE 525
- arithmetic operations
  - with MLE 521
- ASCII
  - alphabet, QSAM 132
  - file labels 133
  - job control language (JCL) 132
  - record formats, QSAM 132
  - standard labels 133
  - tape files, QSAM 132
  - user labels 133
- ASCII files
  - CODE-SET clause 14
  - OPTCD= parameter in DCB 14
- asm file 281
- assembler expansion of PROCEDURE DIVISION 328
- assembler programs
  - calling COBOL programs 349
  - calls from (in CICS) 348
  - LE-conforming 349
  - listing of 542
  - non-LE-conforming 349
- ASSIGN clause
  - corresponds to ddname 10
- assigning values 25
- Associated Data File
  - creating 222
- assumed century window for nondates 520
- AT END (end-of-file) 193
- ATTACH macro 216
- automatic restart 505
- avoiding coding errors 535
- AWO compiler option
  - APPLY-WRITE ONLY clause
    - performance 12
  - description 263
  - performance considerations 545

## B

- backward branches 536
- base cluster name 155
- base locator 324
- basis libraries 221
- BASIS statement 304
- batch compile 226
  - compiler option hierarchy
    - example 228
- BINARY
  - general description 37

BINARY (*continued*)  
 synonyms 35  
 using efficiently 37

binary data item  
 general description 37  
 intermediate results 564  
 using efficiently 37

binary search of a table 66

blank characters, use in CMS 246

BLOCK CONTAINS clause  
 FILE SECTION entry 14  
 no meaning for VSAM files 141  
 QSAM files 109

block size  
 ASCII files 132  
 QSAM files 111  
 system-determined 220

blocking factor 109

blocking QSAM files 115

blocking records 115

BPXBATCH utility 361

buffer, best use of 12

buffers, obtaining on QSAM 267

BUFOFF= 132

BUFSIZE compiler option 263

byte-stream files  
 processing with QSAM 127

**C**

C/C++ programs  
 with COBOL DLLs 498

CALL command (TSO) 215

CALL identifier  
 making from DLLs 493

CALL statement  
 . . . USING 477  
 AMODE processing 464  
 BY CONTENT 475  
 BY REFERENCE 476  
 BY VALUE 475  
 effect of DYNAM compiler option 272  
 effect of EXIT option on registers 576  
 exception condition 199  
 for error handling 199  
 handling of programs name in 287  
 identifier 464  
 overflow condition 199  
 procedure-pointer 473  
 static 467  
 to alternate entry points 473  
 to invoke Language Environment callable services 555  
 using NODYNAM compiler option 464  
 with CANCEL 464  
 with ON EXCEPTION 199  
 with ON OVERFLOW 19

CALLINT statement 304

calls  
 31-bit addressing mode 464  
 AMODE switching for 24-bit programs 464  
 between COBOL and non-COBOL programs 459  
 between COBOL programs 459

calls (*continued*)  
 CICS restrictions 348  
 dynamic 462  
 dynamic performance 464  
 interlanguage 459  
 Linkage Section 477  
 overflow condition 199  
 passing arguments 477  
 passing data 475  
 receiving parameters 477  
 recursive 472  
 static 462  
 static performance 464  
 to Language Environment callable services 555

CANCEL statement  
 handling of programs name in 287

case structure 71

cataloged procedure  
 JCL for compiling 204  
 to compile (IGYWC) 205  
 to compile, link-edit, run (IGYWCLG) 207  
 to compile, load, run (IGYWCG) 208  
 to compile, prelink, link-edit (IGYWCP) 209  
 to compile, prelink, link-edit, run (IGYWCP) 210  
 to compile, prelink, load, run (IGYWCPG) 212  
 to compile and link-edit (IGYWCL) 206  
 to prelink and link-edit (IGYWPL) 211

cbl extension with cob2 240

CBL statement 304

CBLPSHPOP run-time option 352

CBLQDA run-time option 118

century window  
 assumed for nondates 520  
 fixed 512  
 sliding 512

chained list processing 479

changing  
 characters to numbers 95  
 file-name 11  
 title on source listing 6

CHAR intrinsic function 96

CHECK(OFF) run-time option 545

checking for valid data 73  
 numeric 42

checkpoint  
 COBOL 85 Standard 502  
 designing 502  
 messages generated during 504  
 methods 502  
 multiple 504  
 record data set 503  
 restart data sets 507  
 restart during DFSORT 186  
 restart job control sample 507  
 restrictions 502  
 single 502  
 disk 503  
 tape 503

CHKPT keyword 186

CICS  
 CALL statement 348  
 calls 348  
 CICS HANDLE  
 example 352  
 LABEL value 352  
 using 352

COBOL 85 Standard  
 considerations 350  
 coding output 348  
 coding programs to run under 347  
 coding restrictions 347  
 command level interface 347  
 commands and the PROCEDURE DIVISION 347  
 compiler options for 350  
 compiler restrictions 347  
 compiling under 350  
 developing programs for 347  
 link-editing under 350  
 performance considerations 548  
 reserved-word table 351  
 restrictions under OS/390 349  
 sorting 187  
 system date 348  
 translator 347

CICS ECI  
 return code 349

CISZ (control interval size), performance considerations 549

CKPT keyword 186

class condition 73  
 numeric 42

class definition 378

class test 311  
 numeric 42

client definition 389

CLOSE statement 117

closing files  
 line-sequential 167  
 QSAM 120  
 VSAM 150

closing files, automatic  
 line-sequential 167  
 QSAM 120  
 VSAM 150

cluster, VSAM 153

CMPR2 compiler option 349  
 mutually exclusive with 259  
 no support for some data comparisons 264  
 NOCMPR2 for dynamic allocation 106

CMS  
 compiling under 243  
 DISK compiler option 270  
 PRINT compiler option 270  
 restrictions 369  
 run-time restrictions 369  
 running programs under 369

cob2  
 example 238  
 input and output 240  
 options 239

cob2 command  
 compiling and linking with 237

- COBOL 85 Standard
  - checkpoints 502
  - considerations for CICS 350
  - required compiler options 259
  - required run-time options 259
- COBOL language usage with SQL statements 355
- COBOL terms 23
- COBOL2 command
  - entering options 246
  - error messages 252
  - file name 244
  - format 244
  - GRAPHIC option 256
  - indicating source 244
  - invoking compiler 244
  - issue FILEDEF for SYSIN 244
  - options 244
  - syntax 244
  - with file name of source 244
  - with the EXIT option 244
- COBOPT environment variable
  - COBOPT 235
  - library-name 236
  - SYSLIB 235
  - text-name 236
- code
  - copy 551
  - optimized 542
- code 39 370
- CODE-SET clause
  - description 14
- coding
  - class definition 378
  - client definition 389
  - condition tests 74
  - DATA DIVISION 12
  - decisions 69
  - efficient 535
  - ENVIRONMENT DIVISION 7
  - EVALUATE statement 71
  - file input/output overview 101
  - IDENTIFICATION DIVISION 5
  - IF statement 69
  - input 348
  - input/output overview 103
  - input/output statements
    - for line-sequential files 165
    - for QSAM files 117
    - for VSAM files 143
  - loops 73
  - metaclass definition 405
  - method definition 381
  - OO programs 375
  - output 348
  - PROCEDURE DIVISION 17
  - programs to run under IMS 359
  - programs to run under ISPF 371
  - restrictions for programs for
    - CICS 347
  - restrictions for programs for IMS 359
  - subclass definition 393
  - tables 53
  - techniques 535
  - test conditions 74
- collating sequence
  - alternate 8
- collating sequence (*continued*)
  - EBCDIC 8
  - HIGH-VALUE 8
  - LOW-VALUE 8
  - MERGE 8
  - nonnumeric comparisons 8
  - SEARCH ALL 8
  - SORT 8
  - specifying 8
  - symbolic character in the
    - the ordinal position of a character 96
- columns in tables 53
- comma as a separator 246
- COMMON attribute 6
- COMP (COMPUTATIONAL) 37
- COMP-1 (COMPUTATIONAL-1) 38
- COMP-2 (COMPUTATIONAL-2) 38
- COMP-3 (COMPUTATIONAL-3) 38
- COMP-4 (COMPUTATIONAL-4) 37
- COMP-5 (COMPUTATIONAL-5) 37
- comparison of date fields 516
- COMPAT suboption of PGMNAME 287
- compatibility mode 559
- compatible dates
  - in comparisons 516
  - with MLE 517
- compilation
  - CICS 350
  - COBOL 85 Standard 259
  - control
    - using compiler-directing statements under CMS 245
    - with COBOL2 command 246
  - controlling under CMS 243
  - results 224
  - results, work and default files (CMS) 248
  - statistics 321
  - using the CBL statement 246
  - using the PROCESS statement 246
  - with HFS files 205
- COMPILE compiler option 265
  - use NOCOMPILE to find syntax errors 314
- compile-time considerations
  - compiler directed errors 231
  - display compile and link steps 240
  - error message severity 232
  - executing compile and link steps after display 240
- compile-time dump, generating 271
- compile-time error messages
  - choosing severity to be flagged 316
  - determining what severity level to produce 274
  - embedding in source listing 316
- compiler
  - accessing under CMS 243
  - calculation of intermediate results 560
  - generating list of error messages 230
  - invoking in the OS/390 UNIX shell 237
  - invoking with COBOL2 command under CMS 244
  - limits 12
- compiler data sets
  - in the HFS 204
  - input and output 218
  - SYSADATA (ADATA records) 222
  - SYSDEBUG (debug records) 222
  - SYSIDL 222
  - SYSIN (source code) 220
  - SYSLIB (libraries) 220
  - SYSOUT (listing) 221
  - SYSPUNCH (object code) 221
  - SYSTEM (messages) 221
  - with cob2 240
- compiler-directing statements 304
  - list 20
  - overview 20
  - under CMS 245
- compiler error messages
  - from exit modules 583
  - sending to terminal 221
  - under CMS 252
- compiler listings
  - getting 319
- compiler messages
  - analyzing 528
- compiler options
  - abbreviations 259
  - ADATA 261
  - ANALYZE 261
  - APOST 289
  - ARITH 262
  - batch hierarchy example 228
  - BUFSIZE 263
  - CMPR2 264
  - CMS only 247
  - COBOL for OS/390 & VM
    - AWO 545
    - RMODE 545
  - COMPILE 265
  - conflicting 259
  - CURRENCY 265
  - DATA(24|31) 266
  - DATEPROC 267
  - DBCS 269
  - DECK 269
  - DIAGTRUNC 269
  - DISK 247
  - DLL 270
  - DUMP 271
  - DYNAM 545
  - EXIT 575
  - EXPORTALL 272
  - FASTSRT 545
  - FLAG 316
  - FLAGMIG 275
  - FLAGSTD 275
  - for debugging 313
  - IDLGEN 277
  - IMS, recommended for 359
  - in effect 331
  - INTDATE 278
  - LANGUAGE 279
  - LIB 321
  - LINECOUNT 280
  - LIST 319
  - MAP 318
  - NAME 282
  - NOCOMPILE 314

- compiler options (*continued*)
    - NODYNAM 467
    - NOFASTSRT 183
    - NOSOURCE 321
    - NUMBER 320
    - NUMPROC 283
    - NUMPROC(PFD) 545
    - NUMPROC(PFD|NOPFD|MIG) 41
    - OBJECT 284
    - on compiler invocation 321
    - OPTIMIZE 545
    - OUTDD 286
    - PGMNAME 287
    - precedence of 248
    - PRINT 247
    - QUOTE 289
    - RENT 545
    - RMODE 290
    - SEQUENCE 315
    - settings for standard compilation 259
    - signature bytes 331
    - SIZE 291
    - SOURCE 319
    - SPACE 293
    - specifying 223
      - PROCESS (CBL) statement 223
    - specifying under CMS 246
    - specifying under OS/390 224
    - specifying under OS/390 UNIX 236
    - specifying under TSO 224
    - SQL 356
    - SSRANGE 545
    - status 321
    - TERMINAL 248
    - TEST 545
    - TRUNC 297
    - TRUNC(STD|OPT|BIN) 545
    - TYPECHK 300
    - under CICS 350
    - under IMS 349
    - VBREF 319
    - when coding for CICS 350
    - WORD 301
    - XREF 317
    - YEARWINDOW 303
    - ZWB 304
  - compiler output, CMS 249
  - compiling
    - accessing compiler under CMS 243
    - batch 226
    - CMS 243
    - CMS work files 248
    - compile, link-edit, run cataloged procedure 207
    - compile, load, run cataloged procedure 208
    - compile, prelink, link-edit, run cataloged procedure 210
    - compile, prelink, link-edit cataloged procedure 209
    - compile, prelink, load, run cataloged procedure 212
    - compile and link-edit cataloged procedure 206
    - compile cataloged procedure 205
    - control of 223
    - data sets for 218
  - compiling (*continued*)
    - from an assembler program 216
    - invoking with COBOL2 command
      - under CMS 244
    - under OS/390 203
    - under OS/390 UNIX 235
    - under TSO 215
    - using shell script 241
    - using the cob2 command 237
    - with cataloged procedures 204
    - with JCL (job control language) 203
  - compiling and linking in the OS/390 UNIX shell 237
  - completion code, sort 180
  - complex OCCURS DEPENDING ON
    - basic forms of 569
    - complex ODO item 569
    - variably located data item 570
    - variably located group 570
  - computation
    - constant data items 536
    - duplicate 537
    - subscript 540
  - COMPUTATIONAL (COMP) 37
  - COMPUTATIONAL-1 (COMP-1) 38
  - COMPUTATIONAL-2 (COMP-2) 38
  - COMPUTATIONAL-3 (COMP-3) 38
  - COMPUTATIONAL-3 date fields, potential problems 530
  - COMPUTATIONAL-4 (COMP-4) 37
  - COMPUTATIONAL-5 (COMP-5) 37
  - computer, describing 7
  - concatenating data items 81
  - condition handling 175
  - condition-name 519
  - condition testing 74
  - conditional expression
    - IF statement 69
    - PERFORM statement 78
  - conditional statement
    - in EVALUATE statement 70
    - list of 19
    - overview 19
    - with NOT phrase 19
  - CONFIGURATION SECTION 7
  - conflicting compiler options 259
  - constant
    - computations 536
    - data items 536
    - figurative 24
  - constructor method 407
  - contained program integration 543
  - continuation
    - entry 185
  - continuation, syntax checking 265
  - CONTINUE statement 70
  - contracting alphanumeric dates 530
  - control
    - in nested programs 468
    - program flow 69
    - transfer 459
  - control interval size (CISZ), performance considerations 549
  - CONTROL statement 304
  - controlling program compilation under CMS 243
  - conversion of data formats 39
  - converting data items
    - characters to numbers 95
    - INSPECT statement 89
    - reversing order of characters 95
    - to integers 88
    - to uppercase or lowercase 95
    - with intrinsic functions 94
  - converting files to expanded date form, example 515
  - copy
    - libraries 552
    - OS/390 UNIX search order 239
    - search order 305
    - SYSLIB 220
  - copy code, obtaining from user-supplied module 575
  - COPY statement
    - CMS considerations 245
    - example 552
    - nested 552
    - OS/390 considerations 220
    - OS/390 UNIX considerations 305
  - copybook 304
  - copying, code 551
  - counting data items 89
  - creating
    - Associated Data File 222
    - IDL data set 222
    - line-sequential files, OS/390 165
    - object code 221
    - QSAM files, CMS 128
    - QSAM files, OS/390 123
  - cross-reference
    - data and procedure names 317
    - embedded 319
    - program-name 341
    - special definition symbols 341
    - verbs 319
  - cross-reference list 301
  - CRP (file position indicator) 144
  - CURRENCY compiler option 265
  - currency signs
    - euro 51
    - hex literals 51
    - multiple-character 51
    - using 51
  - CURRENT-DATE intrinsic function 47
- ## D
- D format record 111
  - DASD (direct-access storage device) 159
  - data
    - concatenating 81
    - conversion, DBCS to nonnumeric 90
    - conversion, nonnumeric to DBCS 90
    - efficient execution 535
    - format, numeric types 35
    - format conversion 39
    - grouping 478
    - incompatible 42
    - joining 81
    - naming 14
    - numeric 33
    - passing 475
    - record size 14
    - splitting 83
    - validation 42

DATA(24|31) compiler option 266  
data and procedure name cross-reference, description 317  
data areas, dynamic 272  
data definition 324  
data-definition attribute codes 324  
data description entry 13  
DATA DIVISION  
  client 390  
  coding 12  
  description 12  
  entries for line-sequential files 164  
  entries for QSAM files 108  
  entries for VSAM files 141  
  FD entry 12  
  FILE SECTION 12  
  items present in 333  
  limits 12  
  LINKAGE SECTION 16  
  listing 319  
  mapping of items 319  
  method 382  
  OCCURS clause 53  
  restrictions 12  
  signature bytes 333  
  WORKING-STORAGE SECTION 12  
DATA DIVISION items, mapping 281  
data item  
  alphanumeric with double-byte characters 90  
  common, in subprogram linkage 477  
  concatenating 81  
  converting characters to numbers 95  
  converting to  
  uppercase/lowercase 95  
  converting with INSPECT 89  
  converting with intrinsic functions 94  
  DBCS 90  
  evaluating with intrinsic functions 96  
  finding the smallest/largest in group 97  
  index 56  
  map 224  
  nonnumeric with double-byte characters 90  
  numeric 33  
  reference modification 86  
  referencing substrings 86  
  replacing 89  
  reversing characters 95  
  splitting 83  
  unused 324  
  variably located 570  
data-manipulation  
  DBCS data 90  
  nonnumeric data 81  
data-name  
  cross-reference 340  
  in MAP listing 324  
  OMITTED 14  
  password for VSAM files 151  
DATA RECORDS clause 14  
data set  
  alternate data set names 216  
  checkpoint record 503  
  checkpoint/restart 507  
  data set (*continued*)  
  defining with environment variable 105  
  IDL 222  
  names, alternate 217  
  output 221  
  source code 220  
  SYSADATA 222  
  SYSDEBUG 222  
  SYSIN 220  
  SYSLIN 221  
  SYSPRINT 221  
  SYSPUNCH 221  
  SYSTEM 221  
  used interchangeably for file 7  
  data sets  
  required for compilation, OS/390 218  
  used for compilation 218  
date and time operations  
  Language Environment callable services 553  
date arithmetic 525  
date comparisons 516  
Date-Compiled paragraph 5  
date field expansion 514  
date fields  
  potential problems 530  
DATE-OF-INTEG intrinsic function 47  
date operations  
  intrinsic functions 31  
date processing with internal bridges  
  advantages 512  
date windowing  
  advantages 512  
  example 518  
  how to control 527  
  MLE approach 512  
  when not supported 518  
DATEPROC compiler option 267  
  analyzing warning-level diagnostic messages 528  
DATEVAL intrinsic function 527  
DB2  
  COBOL language usage with SQL statements 355  
  coding considerations 355  
  coprocessor 358  
  options 356  
  return codes 356  
  SQL INCLUDE statement 355  
  SQL statements 355  
DBCS (Double-Byte Character Data) 90  
DBCS compiler option 259  
DBCS user-defined words, listed in XREF output 317  
DBCSXREF 318  
dbg extension with cob2 240  
DCB 116  
DD control statement  
  AMP parameter 156  
  ASCII tape files 132  
  creating line-sequential files 165  
  creating QSAM files 123  
  DCB overrides data set label 122  
  define file 10  
DD control statement (*continued*)  
  defining sort data sets 175  
  RLS parameter 157  
  SYSADATA 222  
  SYSDEBUG 222  
  SYSIDL 222  
  SYSIN 220  
  SYSLIB 220  
  SYSLIN 221  
  SYSPRINT 221  
  SYSPUNCH 221  
ddname definition 10  
Debug Tool  
  compiler options for maximum support 343  
  description 309  
debugging  
  defining data set 222  
  production 296  
  useful compiler options 313  
  using COBOL language features 310  
debugging, language features  
  class test 311  
  debugging declaratives 312  
  file status keys 311  
  INITIALIZE statements 311  
  scope terminators 310  
  SET statements 311  
DECK compiler option 269  
declarative procedures  
  LABEL 130  
  USE FOR DEBUGGING 312  
deferred restart 505  
defining  
  debug data set 222  
  files, overview 101  
  libraries 220  
  line-sequential files to OS/390 165  
  QSAM files to CMS 128 to OS/390 123  
  sort files under CMS 176 under OS/390 175  
  VSAM files 153 to CMS 158 to OS/390 153  
DELETE statement 304  
deleting records from VSAM file 150  
delimited scope statement  
  description of 19  
  nested 21  
depth in tables 54  
developing programs for CICS 347  
device  
  requirements 218  
DFCOMMAREA parameter for CALL 348  
DFHEIBLK parameter for CALL 348  
DFSORT 175  
DFSORT/CMS  
  FILEDEF required 176  
  sort work files 176  
  TXTLIB required 176

- diagnostic messages
  - from millennium language
  - extensions 528
- diagnostics, program 321
- DIAGTRUNC compiler option 269
- direct-access
  - direct indexing 57
  - file organization 102
  - storage device (DASD) 159
- directories
  - adding a path to 239
  - where error listing file is written 231
- DISK compiler option 247
- DISPLAY (USAGE IS) 36
- DISPLAY statement
  - directing output 286
  - displaying data values 29
  - interaction with OUTDD 29
  - using in debugging 310
  - writing to stdout/stderr 29
- DLBL command 158
- DLL compiler option 270
  - mutually exclusive with 259
- DLLs (see dynamic link libraries) 489
- do loop 77
- do-until 77
- do-while 77
- documentation of program 7
- Double-Byte Character Data (DBCS) 90
- dump
  - with DUMP compiler option 224
- DUMP compiler option 248
- duplicate computations 537
- DYNAM compiler option 272
  - description 272
  - performance considerations 545
- dynamic calls
  - using with DLL linkage 494
- dynamic data areas, allocating
  - storage 266
- dynamic file allocation 118
- dynamic link libraries
  - about 489
  - compiling 490
  - creating 489
  - in OO COBOL applicaitons 498
  - linking 491
  - prelinking 492
  - search order for in HFS 494
  - using CALL indentifer with 493
  - using with C/C++ programs 498
  - using with dynamic calls 494

## E

- E-level error message 232
- EJECT statement 304
- embedded cross-reference 319
  - example 341
- embedded error messages 316
- embedded MAP summary 318
- empty VSAM file, opening 145
- enclave 459
- end-of-file phrase (AT END) 193
- entry point
  - alternate 473
  - ENTRY label 460

- entry point (*continued*)
  - passing entry addresses of 472
  - procedure-pointer data item 472
- ENTRY statement
  - handling of programs name in 287
- ENVIRONMENT DIVISION
  - class 379
  - client 389
  - collating sequence coding 8
  - CONFIGURATION SECTION 7
  - description 7
  - entries for line-sequential files 163
  - entries for QSAM files 107
  - entries for VSAM files 137
  - INPUT-OUTPUT SECTION 7
  - items present in, program
    - initialization code 333
  - method 382
  - signature bytes 333
  - subclass 394
- environment variable
  - defining line-sequential files 165
  - defining QSAM files 121
  - example of defining files 10
- environment variables
  - \_CEE\_RUNOPTS 362
  - and COPY files 305
  - COBOPT 235
  - example of setting and accessing 363
  - LIBPATH 362
  - library-name 304
  - setting 235
  - setting and accessing 362
  - SYSLIB 235
  - System Object Model (SOM) 417
  - using to allocate files 105
- ERRMSG, for generating list of error
  - messages 230
- error
  - arithmetic 191
  - compiler options, conflicting 259
  - example of message table 60
  - handling 189
  - handling for input-output 106
  - listing 224
  - messages, compiler
    - embedding in source listing 316
    - sending to terminal 221
    - under CMS 252
  - messages, from COBOL2 command
    - identified by DMSIGY 252
    - list of 256
  - processing, line-sequential files 168
  - processing, QSAM files 120
  - processing, VSAM files 151
- error messages
  - compiler-directed 231
  - determining what severity level to
    - produce 274
  - format 231
  - from exit modules 583
  - generating a list of 230
  - severity levels 232
- errors
  - compiler-directed 231
- ESDS (entry-sequenced data sets)
  - file access mode 141

- euro currency sign 51
- EVALUATE statement
  - case structure 71
  - structured programming 535
- evaluating data item contents
  - class test 42
  - INSPECT statement 89
  - intrinsic functions 96
- exception condition 199
- EXCEPTION/ERROR declarative
  - description 194
  - file status key 195
  - line-sequential error processing 168
  - QSAM error processing 120
  - VSAM error processing 151
- EXEC control statement 504
- EXIT compiler option 575
  - with the COBOL2 command 244
- exit modules
  - called for SYSADATA data set 582
  - error messages generated 583
  - loading and invoking 577
  - when used in place of
    - library-name 578
  - when used in place of SYSLIB 578
  - when used in place of
    - SYSPRINT 580
- EXIT PROGRAM statement
  - in subprogram 460
- expanded IF statement 69
- explicit scope terminator 20
- exponentiation
  - evaluated in fixed-point
    - arithmetic 562
  - evaluated in floating-point
    - arithmetic 567
  - performance tips 538
- EXPORTALL compiler option 272
- extended mode 559
- EXTERNAL clause
  - example for files 484
  - for data items 483
  - for files 13
  - used for input/output 483
- external data
  - obtaining storage for 267
  - sharing 483
  - storage location of 266
- external decimal data item 36
- external file 13
- external floating-point data item 36

## F

- F format record 109
- factoring expressions 536
- FASTSRT compiler option 273
  - improves sort performance 545
  - information message 181
  - requirements 181
- FD (file description) entry 14
- figurative constant 24
- file access mode
  - dynamic 141
  - example 141
  - for indexed files (KSDS) 141
  - for relative files (RRDS) 141
  - for sequential files (ESDS) 141

- file access mode (*continued*)
  - performance considerations 159
  - random 141
  - sequential 141
- file allocation 105
- file availability
  - QSAM files under CMS 118
  - QSAM files under OS/390 118
  - VSAM files under OS/390 152
- file conversion
  - with millennium language
    - extensions 515
- file description (FD) entry 14
- file name
  - change 11
  - with COBOL2 command 244
- file-name
  - specification 14
- file organization
  - comparison of ESDS, KSDS, RRDS 136
  - line-sequential 163
  - overview 101
  - relative 101
  - relative-record 139
  - sequential 101
  - VSAM 136
- file position indicator (CRP) 144
- FILE SECTION
  - BLOCK CONTAINS clause 14
  - CODE-SET clause 14
  - DATA RECORDS clause 14
  - description 13
  - EXTERNAL clause 13
  - FD entry 14
  - GLOBAL clause 14
  - LABEL RECORDS clause 14
  - LINAGE clause 14
  - OMITTED 14
  - RECORD CONTAINS clause 14
  - record description 13
  - RECORD IS VARYING 14
  - RECORDING MODE clause 14
  - VALUE OF 14
- FILE STATUS clause
  - description 106
  - line-sequential error processing 168
  - NOFASTSRT error processing 183
  - QSAM error processing 120
  - using 194
  - VSAM error processing 151
  - VSAM file loading 146
  - with VSAM return code 196
- file status code 39 370
- file status key
  - checking for successful OPEN 194
  - set for error handling 311
  - to check for I/O errors 194
  - used with VSAM return code 196
- FILEDEF command
  - ASCII tape files 132
  - defining sort files 176
  - example of defining files 10
  - QSAM files 128
  - with the COBOL2 command 244
- filemode number 4 129

- files
  - associating program files to external files 7
  - COBOL coding
    - DATA DIVISION entries 164
    - ENVIRONMENT DIVISION entries 163
    - input/output statements 165
    - overview 103
  - defining to operating system 10
  - defining under CMS 128
  - describing 13
  - description of optional 118
  - identifying to CMS 128
  - identifying to OS/390 123
  - improving sort performance 181
  - labels 133
  - LISTING 248
  - processing
    - line-sequential 163
    - QSAM 107
    - VSAM 135
  - TEXT 248
  - usage explanation 11
  - work 248
- finding the largest or smallest data item 97
- finding the length of data items 98
- fixed century window 512
- fixed-length record format 142
- fixed-length records
  - QSAM 109
  - VSAM 136
- fixed-point arithmetic
  - comparisons 50
  - evaluation 49
  - example evaluations 50
  - exponentiation 562
- fixed-point data
  - binary 37
  - conversions between fixed- and floating-point data 39
  - external decimal 36
  - intermediate results 561
  - packed-decimal 38
  - planning use of 537
- FLAG compiler option 274
  - compiler output 316
  - description 316
- FLAGMIG compiler option
  - description 275
  - mutually exclusive with 259
- flags 74
- FLAGSTD compiler option 275
  - mutually exclusive with 259
- floating-point arithmetic
  - comparisons 50
  - evaluation 49
  - example evaluations 50
  - exponentiation 567
- floating-point data
  - conversions between fixed- and floating-point data 40
  - external floating point 36
  - intermediate results 566
  - internal 38
  - planning use of 537

- format notation, rules for 90
- format of record
  - for QSAM ASCII tape 132
  - format D 111
  - format F 109
  - format S 113
  - format U 114
  - format V 111
  - spanned 113
  - undefined 114
  - variable-length 111
- formatted dump 189
- full date field expansion
  - advantages 512

## G

- GETMAIN, saving address of 576
- GLOBAL clause for files 13
- GLOBAL command (CMS) 245
- global names 471
- GO TO MORE-LABELS 130
- GOBACK statement
  - in main program 460
  - in subprogram 460
- GRAPHIC option error message 256
- group item
  - variably located 570
- grouping data 478

## H

- header on listing 6
- hex literal as currency sign 51
- hierarchical file system (HFS)
  - compiler data sets 205
  - processing files with QSAM 127
  - reading file with ACCEPT 29
  - search order for DLLs in 494
  - writing file with DISPLAY 29
- hierarchy of compiler options under CMS 248
- hierarchy of compiler options under OS/390 223

## I

- I-level error message 232
- I-O, controlling with FASTSRT option 273
- IDENTIFICATION DIVISION
  - class 378
  - client 389
  - coding 5
  - Date-Compiled paragraph 5
  - errors 5
  - listing header example 6
  - method 381
  - Program-ID paragraph 5
  - required paragraphs 5
  - TITLE statement 6
- identifying QSAM files to CMS
  - FILEDEF command 128
  - LABELDEF command 129
- identifying VSAM files to CMS
  - DLBL command 158
- IDL 425

- IDL 425 (*continued*)
    - access intent specifiers 435
    - attributes 428
    - common types 429
    - complex types 432
    - files 446
    - identifiers 427
    - literal arguments 435
    - mapping to COBOL 427
    - operations 427
    - parameter-passing conventions 435
    - passing complex types 435
  - IDL data set
    - creating 222
  - idl extension with cob2 240
  - IDL type
    - any 432
    - array 433
    - enum 430
    - interface 430
    - long 430
    - sequence 433
    - short 431
    - string 431
    - struct 434
    - union 434
  - IDLGEN compiler option 277
  - IDLStringFromCOBOL 450
  - IDLStringToCOBOL 450
  - IF statement
    - coding 69
    - nested 70
    - with null branch 69
  - IGZCA2D service routine 90
  - IGZCD2A service routine 92
  - IGZSRICD data set 185
  - ILC (interlanguage communication) 459
  - imperative statement, list 18
  - implicit scope terminator 20
  - IMS
    - coding programs under 359
    - coding restrictions 359
    - mixed COBOL for OS/390 & VM, COBOL for MVS & VM, VS COBOL II, and OS/VS COBOL applications 359
    - performance considerations 549
    - recommended compiler options 359
  - incrementing addresses 480
  - index, table 56
  - index data item 57
  - index key, detecting faulty 198
  - index-name subscripting 57
  - index range checking 315
  - indexed file organization 101
  - indexing
    - example 61
    - preferred to subscripting 539
    - tables 57
  - INEXIT suboption of EXIT option 575
  - INITIAL attribute 6
  - INITIALIZE statement
    - examples 25
    - loading table values 58
    - using for debugging 311
  - initializing
    - a table 58
  - inline PERFORM 76
  - input
    - coding for line-sequential files 165
    - coding for QSAM files 117
    - coding for VSAM files 143
    - coding in CICS 348
    - overview 101
    - to compiler, under CMS 244
    - to compiler, under OS/390 218
  - input/output
    - checking for errors 194
    - coding overview 103
    - logic flow after error 191
    - overview 101
    - processing errors for line-sequential files 168
    - processing errors for QSAM files 191
    - processing errors for VSAM files 191
  - input/output coding
    - AT END (end-of-file) phrase 193
    - checking for successful operation 194
    - checking VSAM return codes 196
    - detecting faulty index key 198
    - error handling techniques 191
    - EXCEPTION/ERROR declaratives 194
  - INPUT-OUTPUT SECTION 7
  - input procedure
    - FASTSRT option not effective 182
    - requires RELEASE or RELEASE FROM 172
    - restrictions 175
    - using 172
  - INSERT statement 304
  - INSPECT statement 89
  - inspecting data 89
  - INTDATE compiler option 278
  - INTEGER intrinsic function 88
  - INTEGER-OF-DATE intrinsic function 47
  - interactive program, example 592
  - Interactive System Productivity Facility (ISPF) 592
  - Interface Repository (IR)
    - accessing 415
    - definition 415
    - populating 416
  - interlanguage communication (ILC) 349
  - intermediate results 559
  - internal bridges
    - advantages 512
    - example 514
    - for date processing 513
  - internal floating-point data
    - bytes required 38
    - defining 38
    - uses for 38
  - intrinsic functions
    - as reference modifier 88
    - converting character data items 94
    - DATEVAL 527
    - evaluating data items 96
    - example of
      - ANNUITY 48
      - CHAR 96
      - CURRENT-DATE 47
      - INTEGER 88
  - intrinsic functions (*continued*)
    - example of (*continued*)
      - INTEGER-OF-DATE 47
      - LENGTH 47
      - LOG 48
      - LOWER-CASE 95
      - MAX 47
      - MEAN 48
      - MEDIAN 48
      - MIN 88
      - NUMVAL 95
      - NUMVAL-C 47
      - ORD 96
      - ORD-MAX 97
      - PRESENT-VALUE 48
      - RANGE 48
      - REM 48
      - REVERSE 95
      - SQRT 48
      - SUM 68
      - UPPER-CASE 95
      - WHEN-COMPILED 99
    - intermediate results 565
    - introduction to 31
    - nesting 32
    - numeric functions
      - differences from Language Environment callable services 46
      - equivalent Language Environment callable services 45
      - examples of 44
      - nested 45
      - special registers as arguments 45
      - table elements as arguments 45
      - type of—integer, floating-point, mixed 44
      - uses for 44
    - processing table elements 67
    - simplifying coding 551
    - UNDATE 527
  - INVALID KEY phrase 198
  - INVOKE statement
    - use with PROCEDURE DIVISION RETURNING 482
  - invoking
    - COBOL UNIX programs 361
    - Language Environment callable services 555
    - the compiler under CMS 243
  - ISAM data set 135
  - ISPF (Interactive System Productivity Facility) 592
- ## J
- JCL (job control language)
    - ASCII tape files 132
    - cataloged procedures 204
    - for compiling 203
    - for compiling with HFS 205
    - for line-sequential files 165
    - for QSAM files 122
    - for VSAM data sets 156
    - SOM 418
  - JCL (Job Control Language)
    - FASTSRT requirement 181
  - job control statement
    - checkpoint/restart sample 507



job resubmission 507  
job stream 459

## K

KSDS (key-sequenced data sets)  
file access mode 141  
organization 138

## L

LABEL= 132  
LABEL declarative 304  
GO TO MORE-LABELS 130  
LABEL RECORDS clause  
FILE SECTION entry 14  
LABELDEF tape volume identification  
under CMS 129  
labels  
ASCII file 133  
format, standard 131  
processing, QSAM files 129  
standard user 131  
LANGUAGE compiler option 279  
Language Environment callable services  
condition handling 553  
corresponding intrinsic functions 45  
date and time computations 553  
differences from intrinsic  
functions 46  
dynamic storage services 553  
equivalent intrinsic functions 45  
example of using 556  
feedback code 555  
for date and time computations 45  
for mathematics 45  
invoking with a CALL statement 555  
mathematics 553  
message handling 553  
national language support 553  
overview 553  
return code 555  
RETURN-CODE special register 555  
sample list of 554  
types of 553  
language features for debugging  
DISPLAY statements 310  
large block interface 116  
last-used state 460  
LE-conforming assembler programs 349  
LENGTH intrinsic function  
example 47  
variable length results 97  
versus LENGTH OF special  
register 98  
length of data items, finding 98  
LENGTH OF special register 476  
level  
88 item 73  
level-88 item  
for windowed date fields 519  
restriction 519  
switches and flags 74  
level definition 324  
LIB compiler option  
description and syntax 280  
LIBEXIT suboption of EXIT option 575

LIBPATH environment variable  
setting 363  
library  
BASIS 221  
COPY 221  
defining 220  
directory entry 216  
library-name  
alternative if not specified 239  
library text  
specifying path for 304  
limits of the compiler 12  
LINAGE clause 14  
line number 323  
line-sequential files  
adding records to 167  
allowable control characters 164  
blocking 13  
closing 167  
closing to prevent reopening 166  
DATA DIVISION entries 164  
ENVIRONMENT DIVISION  
entries 163  
input/output error processing 168  
input/output statements for 165  
opening 166  
processing files 163  
reading from 165  
reading records from 166  
sort and merge 169  
under OS/390  
creating files 165  
DD statement for 165  
defining 165  
environment variable for 165  
job control language (JCL) 165  
writing to 166  
LINECOUNT compiler option 280  
link-edit  
CICS 350  
LINK macro 216  
LINKAGE SECTION  
description 477  
GLOBAL clause 17  
run unit 16  
with recursive calls 16  
with the THREAD option 16  
linker  
passing information to 239  
LIST compiler option 339  
assembler code for source  
program 328  
compiler output 329  
conflict with OFFSET option 319  
DSA memory map 328  
getting output 319  
location and size of  
WORKING-STORAGE 339  
mutually exclusive with 259  
reading output 328  
symbols used in output 326  
LISTING files 248  
listings  
assembler expansion of procedure  
division 319  
assembler expansion of PROCEDURE  
DIVISION 328

listings (*continued*)  
compiler options affecting 259  
data- and procedure-name cross  
reference 317  
embedded cross-reference 319  
embedded MAP summary 319  
generating a short listing 319  
including your source code 319  
line numbers, user-supplied 320  
mapping DATA DIVISION items 319  
sorted cross reference of program  
names 341  
terms used in MAP output 326  
verb cross-reference 319  
with error messages embedded 316  
loading a table dynamically 58  
local names 471  
LOCAL-STORAGE section 15  
LOG intrinsic function 48  
logical record  
description 101  
variable-length format 111  
LONGMIXED suboption of  
PGMNAME 288  
LONGUPPER suboption of  
PGMNAME 288  
loops  
coding 76  
conditional 78  
do 77  
in a table 78  
performed a definite number of  
times 77  
LOWER-CASE intrinsic function 95  
lowercase 95  
lst extension with cob2 240  
LST file extension 231

## M

MACLIB 245  
main program  
and subprograms 459  
dynamic CALL 462  
parameter list in UNIX 365  
main storage, allocating to buffers 263  
making compiler available under  
CMS 243  
map  
data items and relative addresses 224  
MAP compiler option 318  
embedded MAP summary 319  
example 324  
nested program map 319  
nested program map, example 328  
symbols used in output 326  
terms used in output 326  
mapping of DATA DIVISION items 319  
maps and listings 259  
mathematics  
intrinsic functions 44  
Language Environment callable  
services 46  
MAX intrinsic function 97  
example 47  
MEAN intrinsic function 48  
MEDIAN intrinsic function 48  
memory map, DSA 328

- memory map, TGT 328
  - example 338
- merge
  - concepts 170
  - description 169
  - files, describing 171
  - line-sequential files 169
  - pass control statements to 185
  - storage use 186
  - successful 180
- MERGE statement
  - description 170
- message handling, Language Environment callable services 553
- messages
  - COBOL2 command error 252
  - compile-time error
    - embedding in source listing 316
  - compiler-directed 231
  - compiler error
    - sending to terminal 221
    - under CMS 252
  - determining what severity level to produce 274
  - from exit modules 583
  - severity levels 232
- messages, error
  - generating a list of 230
- metaclass definition 405
- method definition 381
- METHOD-ID paragraph 381
- methods 391
  - PROCEDURE DIVISION RETURNING 482
- migration
  - FLAGMIG compiler option 275
- millennium language extensions 510
  - assumed century window 520
  - compatible dates 517
  - date windowing 509
  - DATEPROC compiler option 267
  - nondates 521
  - objectives 511
  - principles 510
  - YEARWINDOW compiler option 303
- MIN intrinsic function 88
- mixed COBOL for OS/390 & VM, COBOL for MVS & VM, VS COBOL II, and OS/VS COBOL applications
  - coding under IMS 359
- mixed DBCS/EBCDIC literal 90
  - alphanumeric to DBCS conversion 90
  - conversion 90
- mixed literal
  - alphanumeric to DBCS data conversion 90
  - conversion 90
  - DBCS to alphanumeric conversion 92
- MLE 510
- mnemonic-name
  - SPECIAL-NAMES paragraph 7
- MOVE statement 27
- MSGFILE run-time option 286
- multiple-character currency signs 51
- multiple currency signs 52
- multivolume tape files, identifying to CMS 129

## N

- NAME compiler option 5
- name declaration
  - searching for 471
- naming
  - files 10
  - programs 5
- National Language Support 279
- nested COPY statement 578
- nested delimited scope statements 21
- nested IF statement
  - CONTINUE statement 70
  - description 70
  - EVALUATE statement preferred 70
  - with null branches 70
- nested intrinsic functions 45
- nested program integration 543
- nested program map
  - about 319
  - example 328
- nested programs
  - calling 468
  - conventions for using 468
  - description 469
  - map 319
  - scope of names 471
  - transfer of control 468
- nesting level
  - program 323
  - statement 323
- NOCOMPILER compiler option
  - use of to find syntax errors 314
- NODYNAM compiler option
  - mixing static and dynamic calls 467
- NODYNAM parameter option 349
- NOFASTSORT compiler option 183
- nondates
  - with MLE 521
- nonnumeric literal
  - alphanumeric to DBCS conversion 90
  - conversion of mixed DBCS/EBCDIC 90
  - DBCS to alphanumeric conversion 92
  - with double-byte characters 90
- NOPRINT compiler option 247
- Notices 609
- null branch 70
- null-terminated strings 85
- NUMBER compiler option 320
  - syntax and description 283
- NUMCLS installation option 42
- numeric class test 42
- numeric condition 73
- numeric data
  - binary
    - USAGE IS BINARY 37
    - USAGE IS COMPUTATIONAL (COMP) 37
    - USAGE IS COMPUTATIONAL-4 (COMP-4) 37
    - USAGE IS COMPUTATIONAL-5 (COMP-5) 37
  - conversions between fixed- and floating-point data 40
  - editing symbols 34
  - external decimal
    - USAGE IS DISPLAY 36

- numeric data (*continued*)
  - external floating-point
    - USAGE IS DISPLAY 36
  - format conversions between fixed- and floating-point 39
  - internal floating-point
    - USAGE IS COMPUTATIONAL-1 (COMP-1) 38
    - USAGE IS COMPUTATIONAL-2 (COMP-2) 38
  - overview 33
  - packed-decimal
    - USAGE IS COMPUTATIONAL-3 (COMP-3) 38
    - USAGE IS PACKED-DECIMAL 38
  - PICTURE clause 33
  - storage formats 35
  - zoned decimal
    - USAGE IS DISPLAY 36
- numeric-edited data item 34
- numeric editing symbol 34
- numeric intrinsic functions
  - differences from Language Environment callable services 46
  - equivalent Language Environment callable services 45
  - example of
    - ANNUITY 48
    - CURRENT-DATE 47
    - INTEGER 88
    - INTEGER-OF-DATE 47
    - LENGTH 47
    - LOG 48
    - MAX 47
    - MEAN 48
    - MEDIAN 48
    - MIN 88
    - NUMVAL 95
    - NUMVAL-C 47
    - ORD 96
    - ORD-MAX 68
    - PRESENT-VALUE 48
    - RANGE 48
    - REM 48
    - SQRT 48
    - SUM 68
  - nested 45
  - special registers as arguments 45
  - table elements as arguments 45
  - types of—integer, floating-point, mixed 44
  - uses for 44
- NUMPROC compiler option 283
- NUMPROC(PFD) compiler option
  - performance considerations 545
- NUMPROC(PFD|NOPFD|MIG) compiler option
  - affected by NUMCLS 42
  - effect on sign processing 41
- NUMVAL-C intrinsic function 95
  - example 47
- NUMVAL intrinsic function 95

## O

- o extension with cob2 240
- object code
  - compilation and listing 224

- object code (*continued*)
    - controlling 259
    - creating 221
    - generating 265
    - producing in 80-column card 269
  - OBJECT compiler option 284
  - OBJECT-COMPUTER paragraph 7
  - object deck generation 259
  - object-oriented COBOL
    - compiler options not supported 264
    - DLLs in 498
    - restrictions for DYNAM compiler option 272
  - object references 391
  - objectives of millennium language
    - extensions 511
  - OCCURS clause 540
  - OCCURS DEPENDING ON (ODO) clause
    - complex 569
    - initializing ODO elements 64
    - optimization 540
    - simple 62
    - variable-length records 110
    - variable-length tables 62
  - OFFSET compiler option 259
    - output 342
  - OMITTED clause 14
  - OMITTED parameters 555
  - OMMAlloc 450
  - OMMFree 450
  - ON EXCEPTION phrase 349
  - ON OVERFLOW phrase 349
  - ON SIZE ERROR
    - intermediate and final results 567
    - with windowed date fields 525
  - OO COBOL
    - generating IDL definitions 277
  - OPEN operation code 577
  - OPEN statement
    - file availability 166
    - file status key 194
    - line-sequential files 165
    - QSAM files 117
    - VSAM files 143
  - opening files
    - line-sequential 166
    - QSAM 117
    - VSAM 145
  - optimization
    - avoid ALTER statement 536
    - avoid backward branches 536
    - consistent data 538
    - constant computations 536
    - constant data items 536
    - contained program integration 543
    - duplicate computations 537
    - effect of compiler options on 544
    - effect on performance 535
    - factor expressions 536
    - index computations 540
    - indexing 539
    - nested program integration 543
    - OCCURS DEPENDING ON 540
    - out-of-line PERFORM 536
    - PACKED-DECIMAL data items 538
    - performance implications 540
  - optimization (*continued*)
    - procedure integration 543
    - structured programming 535
    - subscript computations 540
    - subscripting 539
    - table elements 539
    - top-down programming 536
    - unreachable code 543
    - unused data items 324
  - OPTIMIZE compiler option 285
    - description 542
    - effect on performance 542
    - mutually exclusive with 259
    - performance considerations 545
  - optimizer 542
  - optional files 118
  - ORD intrinsic function 96
  - ORD-MAX intrinsic function 97
  - ORD-MIN intrinsic function 97
  - order of evaluation
    - arithmetic operators 561
    - compiler options 259
  - OS/390
    - compiling under 203
  - OS/390 UNIX
    - accessing environment variables 362
      - example 363
    - accessing main parms 365
    - calling APIs 364
    - compiler environment variables 235
    - compiling from script 241
    - compiling under 235
    - COPY files 305
    - copy search order 305
    - developing programs 361
    - execution environments 361
    - setting environment variables 362
      - example 363
    - sort and merge 169
    - specifying compiler options 236
  - OS data sets, use under CMS 129
  - out-of-line PERFORM 76
  - OUTDD compiler option 286
    - DD not allocated 29
    - interaction with DISPLAY 29
  - output
    - coding for line-sequential files 165
    - coding for QSAM files 117
    - coding for VSAM files 143
    - coding in CICS 348
    - data set 221
    - from compiler, under CMS 249
    - from compiler, under OS/390 218
    - overview 101
  - output file with cob2 240
  - output procedure
    - FASTSRT option not effective 182
    - requires RETURN or RETURN INTO statement 173
    - restrictions 175
    - using 173
  - overflow condition 190
- P**
- PACKED-DECIMAL
    - general description 38
    - synonym 35
  - PACKED-DECIMAL (*continued*)
    - using efficiently 38
  - packed-decimal data item
    - date fields, potential problems 530
    - general description 38
    - using efficiently 38
  - page
    - depth 14
  - page control 119
  - page header 321
  - paragraph
    - grouping 79
    - introduction 18
  - parameter
    - describing in called program 477
  - parameter list
    - address of with INEXIT 577
    - for ADEXIT 582
    - for PRTEXIT 580
    - main program in UNIX 365
  - parentheses in COBOL2 246
  - passing addresses between programs 479
  - passing data between programs
    - BY CONTENT 475
    - BY REFERENCE 475
    - BY VALUE 475
    - called program 477
    - calling program 477
    - EXTERNAL data 483
    - language used 477
  - password
    - alternate index 152
    - example 152
    - VSAM files 151
  - PASSWORD clause 151
  - path name
    - for COPY files search 304
  - PERFORM statement
    - ...THRU 79
    - coding loops 76
    - for a table 60
    - indexing 57
    - inline 76
    - out-of-line 76
    - performed a definite number of times 77
    - TEST AFTER 77
    - TEST BEFORE 77
    - TIMES 77
    - UNTIL 78
    - VARYING 78
    - VARYING WITH TEST AFTER 78
    - WITH TEST AFTER ... UNTIL 78
    - WITH TEST BEFORE ... UNTIL 78
  - performance
    - AIXBLD run-time option 549
    - APPLY WRITE-ONLY clause 12
    - AWO compiler option 545
    - blocking QSAM files 115
    - coding 535
    - coding tables 539
    - data usage 537
    - DYNAM compiler option 545
    - effect of compiler options on 544
    - effects of buffer size 263
    - FASTSRT compiler option 545

performance (*continued*)  
 in a CICS environment 548  
 in IMS environment 549  
 mixed-level COBOL applications 359  
 NUMPROC compiler option 41  
 NUMPROC(PFD) compiler option 545  
 OCCURS DEPENDING ON 540  
 of calls 464  
 of tape data sets 116  
 OPTIMIZE compiler option 545  
 optimizer 542  
 planning arithmetic evaluations 537  
 programming style 535  
 RENT compiler option 545  
 RMODE compiler option 545  
 run-time considerations 535  
 SSRANGE compiler option 545  
 table handling 540  
 TEST compiler option 545  
 TRUNC compiler option 297  
 TRUNC(STD|OPT|BIN) compiler option 545  
 use of arithmetic expressions 538  
 variable subscript data format 56  
 VSAM file considerations 159  
 worksheet 548

performance considerations  
 compiler options  
 AWO 545  
 RMODE 545

period, as scope terminator 20  
 PGMNAME compiler option 287

physical  
 block 101  
 record 14

PICTURE clause  
 determining symbol used 265  
 numeric data 33

pointer data item  
 incrementing addresses with 480  
 NULL value 479  
 used to pass addresses 479  
 used to process chained list 479

porting your program 34

POSIX APIs  
 calling 364

potential problems with date fields 530

precedence  
 arithmetic operators 561  
 compiler options under CMS 248  
 compiler options under OS/390 223

preferred sign 41

prelinking  
 compile, prelink, link-edit, run cataloged procedure 210  
 prelink and link-edit cataloged procedure 211

PRESENT-VALUE intrinsic function 48

preserving original sequence in a sort 180

PRINT compiler option 247

priority numbers, segmentation 545

PROCEDURE DIVISION  
 additional information 335  
 client 389  
 description 17

PROCEDURE DIVISION (*continued*)  
 in subprograms 478  
 method 383  
 RETURNING 17  
 signature bytes 333  
 statements  
 compiler-directing 20  
 conditional 19  
 delimited scope 19  
 imperative 18  
 terminology 17  
 USING 17  
 verbs present in 333

PROCEDURE DIVISION RETURNING  
 methods, use of 482

procedure integration 543

procedure-pointer data item  
 entry address for entry point 472  
 SET procedure-pointer 472  
 with DLLs 495

PROCESS (CBL) statement  
 batch compiling 227  
 conflicting options in 259  
 precedence 248  
 specifying compiler options 246

PROCESS statement 223

processing  
 chained list 479  
 labels for QSAM files 129  
 tables 60  
 using indexing 61  
 using subscripting 60

production debugging 296

program  
 attribute codes 328  
 compiling under CMS 243  
 compiling under OS/390 203  
 compiling under OS/390 UNIX 235  
 controlling compilation under CMS 243  
 decisions  
 EVALUATE statement 70  
 IF statement 69  
 loops 77  
 PERFORM statement 77  
 switches and flags 74  
 diagnostics 321  
 initialization code 329  
 limitations 535  
 main 459  
 nesting level 323  
 reentrant 474  
 restarting 504  
 signature information bytes 331  
 statistics 321  
 structure 5  
 sub 459

PROGRAM COLLATING SEQUENCE  
 clause 8

Program-ID paragraph  
 COMMON attribute 6  
 description 5  
 INITIAL attribute 6

program-name cross-reference 341

program names 5  
 handling of case 287

program processing table 349

program termination  
 statements 460

programs  
 developing for OS/390 UNIX 361

protecting VSAM files 151

PRTEXIT suboption of EXIT option 575

## Q

QSAM (queued sequential access method) 107

QSAM files  
 adding records to 119  
 ASCII tape file 132  
 ASSIGN clause 108  
 BLOCK CONTAINS clause 115  
 block size 115  
 blocking records 127  
 closing 120  
 closing to prevent reopening 118  
 DATA DIVISION entries 108  
 ENVIRONMENT DIVISION entries 107  
 input/output error processing 191  
 input/output statements for 117  
 label processing 129  
 obtaining buffers for 267  
 opening 117  
 processing files 107  
 processing files in reverse order 118  
 processing HFS files 127  
 replacing records 119  
 retrieving 123  
 striped extended-format 126  
 tape performance 116  
 under CMS  
 defining 128  
 dynamic allocation 118  
 file availability 118  
 FILEDEF command for 128  
 identifying files, FILEDEF command 128  
 LABELDEF command 129  
 tape file identification 129  
 under OS/390  
 creating files 123  
 DD statement for 123  
 defining 123  
 environment variable for 121  
 file availability 118  
 job control language (JCL) 122  
 updating files 119  
 writing to a printer 119

QSAM unblocked files 369

queued sequential access method (QSAM) 107

QUOTE compiler option 289

## R

random numbers, generating 46

RANGE intrinsic function 48

RD parameter 504

read a block of records 115

READ INTO. . . 142

READ NEXT statement 143

READ statement 117

READ statement 117 (*continued*)  
     line-sequential files 165  
 reading records  
     to line-sequential files 166  
 reading records from VSAM files  
     dynamically 147  
     randomly 147  
     sequentially 147  
 receiving field 83  
 record  
     description 13  
     format 101  
         fixed-length 109  
         format D 111  
         format F 109  
         format S 113  
         format U 114  
         format V 111  
         QSAM ASCII tape 132  
         spanned 113  
         undefined 114  
         variable-length 111  
 RECORD CONTAINS clause  
     FILE SECTION entry 14  
 RECORDING MODE clause  
     fixed-length records, QSAM 109  
     QSAM files 14  
     to specify record format 108  
     variable-length records, QSAM 111  
 recursive calls 6  
     and the LINKAGE SECTION 16  
 reentrant programs 474  
 reference modification  
     example 87  
     out-of-range values 87  
     tables 87  
 reference modifier  
     arithmetic expression as 88  
     intrinsic function as 88  
     variables as 87  
 register 15, and CICS 349  
 registers, affected by EXIT compiler  
     option 576  
 relation condition 73  
 relative file organization 101  
 RELEASE FROM statement  
     compared to RELEASE 172  
     example 172  
 RELEASE statement  
     compared to RELEASE FROM 172  
     with SORT 172  
 REM intrinsic function 48  
 RENT compiler option 289  
     description 289  
     performance considerations 545  
 REPLACE statement 304  
 replacing  
     records in QSAM file 119  
     records in VSAM file 149  
 REPOSITORY paragraph 379  
 representation  
     data 42  
     sign 41  
 RERUN clause  
     checkpoint/restart 186  
 reserved-word table  
     alternate, CICS 351

reserved-word table (*continued*)  
     alternate, OO 301  
 residency mode 290  
 restart  
     automatic 505  
     deferred 505  
     routine 501  
 restarting a program 504  
 restrictions  
     CICS coding 7  
     CMS run-time 369  
     coding programs for CICS 347  
     coding programs for IMS 359  
     IMS coding 349  
     input/output procedures 175  
     subscripting 56  
 resubmitting a job 507  
 return code  
     compiler 232  
     feedback code from Language  
         Environment services 555  
     from CICS ECI 349  
     from DB2 356  
     RETURN-CODE special register 555  
     VSAM files 196  
     when control returns to operating  
         system 482  
 RETURN-CODE special register  
     value after call to Language  
         Environment service 555  
     when control returns to operating  
         system 482  
 RETURN INTO statement 173  
 RETURN statement 173  
 RETURNING phrase  
     methods, use of 482  
 REVERSE intrinsic function 95  
 reverse order of tape files 118  
 reversing characters 95  
 RLS parameter 157  
 RMODE attribute  
     assigned for EXIT modules 576  
 RMODE compiler option 290  
     performance considerations 545  
 Rotational Position Sensing feature 116  
 rows in tables 54  
 RRDS (relative-record data sets)  
     file access mode 141  
     fixed-length records 136  
     organization 138  
     performance considerations 159  
     simulating variable-length  
         records 140  
     variable-length records 136  
 run time  
     changing file-name 11  
 run-time options  
     affecting DATA compiler option 266  
     ALL31 464  
     CBLPSHPOP 352  
     CHECK(OFF) 545  
     COBOL 85 Standard  
         conformance 259  
     MSGFILE 286  
     SIMVRD 140  
     TRAP  
         closing files in line-sequential 167

run-time options (*continued*)  
     TRAP (*continued*)  
         closing files in QSAM 120  
         closing files in VSAM 150  
         ON SIZE ERROR 191  
 run unit 459  
**S**  
 S format record 113  
 S-level error message 232  
 sample programs 587  
 scope of names 471  
 scope terminator  
     aids in debugging 310  
     explicit 19  
     implicit 20  
 SEARCH ALL statement  
     binary search 66  
     indexing 57  
     ordered table 66  
 SEARCH statement  
     examples 66  
     indexing 57  
     nesting 65  
     serial search 65  
 searching a table 65  
 searching for name declarations 471  
 section  
     declarative 21  
     description of 17  
     grouping 79  
 segmentation 545  
 SELECT clause  
     ASSIGN clause 10  
     naming files 10  
     vary input-output file 11  
 SELECT OPTIONAL 118  
 sending field 83  
 sentence 18  
 separate digit sign 34  
 SEQUENCE compiler option 315  
 sequential file organization 101  
 sequential storage device 102  
 serial search 65  
 SERVICE LABEL statement 304  
 SET condition-name TO TRUE statement  
     description 75  
     example 77  
 SET statement  
     for procedure-pointer data items 472  
     handling of programs name in 287  
     using for debugging 311  
 setting  
     switches and flags 75  
 sharing  
     data 471  
     files 471  
 short listing, example 320  
 sign condition 73  
 sign representation 41  
 signature bytes  
     compiler options in effect 331  
     DATA DIVISION 333  
     ENVIRONMENT DIVISION 333  
     PROCEDURE DIVISION 333  
 SIMVRD run-time option 140  
 size  
     of printed page, control 119

SIZE compiler option 291  
 skip a block of records 115  
 sliding century window 512  
 SOM 425  
   CORBA-style exceptions 442  
   environment arguments 442  
   errors and exceptions 442  
     COBOL example 443  
   initializers 445  
   JCL for 418  
   memory management with 447  
   SOMerror-style exceptions 442  
   somFree 390  
   somNew 390  
 SOMENV data set 218  
 sort  
   alternate collating sequence 179  
   checkpoint/restart 186  
   concepts 170  
   criteria 177  
   data sets needed, OS/390 175  
   DD statements, defining OS/390 data sets 175  
   description 169  
   FASTSORT compiler option 181  
   files, describing 171  
   files needed, CMS 176  
   line-sequential files 169  
   more than one 170  
   NOFASTSORT compiler option 183  
   pass control statements to 185  
   performance 181  
   preserving original sequence 180  
   restrictions on input/output procedures 175  
   special registers 184  
   storage use 186  
   successful 180  
   terminating 181  
   under CICS 187  
   under CMS 176  
   under OS/390 175  
   using input procedures 172  
   using output procedures 173  
   variable-length records 176  
   windowed date fields 179  
 SORT-CONTROL special register 184  
 SORT-CORE-SIZE special register 184  
 Sort File Description (SD) entry  
   example 171  
 SORT-FILE-SIZE special register 184  
 SORT-MESSAGE special register 184  
 SORT-MODE-SIZE special register 184  
 SORT-RETURN special register 181  
 SORT statement  
   description 177  
   restrictions for CICS applications 187  
   under CICS 187  
 SORTCKPT DD statement 186  
 source  
   program listing 224  
 SOURCE and NUMBER output, example 323  
 source code  
   compiler data set (CMS) 244  
   compiler data set (OS/390) 220  
   line number 324  
 source code (*continued*)  
   listing, description 319  
 SOURCE compiler option 319  
 SOURCE-COMPUTER paragraph 7  
 SPACE compiler option 293  
 spanned record format 113  
 spanned records 113  
 special feature specification 7  
 SPECIAL-NAMES paragraph  
   QSAM files 132  
 special register  
   ADDRESS 476  
   arguments in intrinsic functions 45  
   LENGTH OF 476  
   SORT-RETURN 180  
   WHEN-COMPILED 99  
 specifying the source program under CMS  
   by file name 244  
 splitting data items 83  
 SQL compiler option 356  
   mutually exclusive with 259  
 SQL INCLUDE statement 355  
 SQL statements 355  
 SQLCA 355  
 SQLCODE 356  
 SQLSTATE 356  
 SQRT intrinsic function 48  
 SSRANGE compiler option 294  
   CHECK(OFF) run-time option 545  
   description 315  
   performance considerations 545  
 STANDARD clause, FD entry 14  
 standard label, QSAM 133  
 standard label format 131  
 START statement 143  
 statement  
   compiler-directing 20  
   conditional 19  
   definition 18  
   delimited scope 19  
   explicit scope terminator 20  
   imperative 18  
   implicit scope terminator 20  
 statement nesting level 323  
 static call  
   statement 462  
 statistics  
   intrinsic functions 48  
 status key  
   importance of in VSAM 151  
 stderr  
   directing DISPLAY 29  
   setting DISPLAY to 363  
 stdin  
   reading with ACCEPT 29  
 stdout  
   directing DISPLAY 29  
   setting DISPLAY to 363  
 STEPLIB environment variable  
   setting 363  
 STOP RUN statement  
   in main program 460  
 storage  
   device  
     direct-access 102  
     sequential 102  
   storage (*continued*)  
     management, Language Environment  
       callable services 553  
     mapping 319  
     use during sort 186  
 STRING statement  
   description 81  
   example of 81  
   overflow condition 190  
   with DBCS data 90  
 strings  
   null-terminated 478  
 striped extended-format QSAM file 126  
 structured programming 536  
 subclass definition 393  
 subprogram  
   and main program 459  
   definition of 459  
   linkage 459  
     common data items 477  
   Procedure Division in 478  
   termination  
     effects 460  
 subscript  
   computations 540  
   range checking 315  
 subscripting  
   example of processing a table 60  
   index-names 57  
   literal 54  
   reference modification 56  
   relative 56  
   variable 55  
 substrings  
   of data 86  
   referencing table items 87  
 SUM intrinsic function 68  
 switch-status condition 73  
 switches 74  
 switches and flags  
   about 74  
   defining 74  
   resetting 75  
 SYMBOLIC CHARACTER clause 9  
 symbolic constant 536  
 symbols used in LIST and MAP  
   output 326  
 SYNAD error 370  
 syntax errors  
   finding with NOCOMPILE compiler  
     option 314  
 SYSADATA  
   SYSADATA data set 222  
   SYSADATA file 248  
 SYSADATA file 261  
 SYSADATA records  
   exit module called 582  
 SYSIDL data set 218  
 SYSIN  
   user exit error message 583  
 SYSLIB  
   when not used 578  
 SYSLIB data set 218  
 SYSLIN data set 221  
 SYSPRINT  
   when not used 580  
 SYSPRINT data set 218

SYSPUNCH data set 218  
 SYSPUNCH requirements for DECK  
   compiler option 269  
   system date  
     under CICS 348  
   system-determined block size 220  
   system dump 190  
   system-name 7  
 System Object Model 425  
   configuration file 417  
   environment variables 417  
   Interface Repository (IR) 415  
   methods and functions 420  
   services 420  
 SYSTEM data set 218  
   sending messages to 295

**T**

table  
   assigning values 58  
   columns 53  
   defining 53  
   depth 54  
   dynamically loading 58  
   efficient coding 540  
   handling 53  
   identical element specifications 539  
   index 56  
   initialize 58  
   intrinsic functions 67  
   loading values in 58  
   looping through 78  
   making reference 55  
   one-dimensional 53  
   reference modification 56  
   referencing table entry substrings 87  
   rows 54  
   searching 65  
   subscripts 54  
   three-dimensional 54  
   two-dimensional 54  
   variable-length 62  
 TALLYING option 89  
 tape file identification under CMS 129  
 tape files  
   performance 116  
   reverse order 118  
 terminal, sending messages to 295  
 TERMINAL compiler option 295  
 termination 460  
 terminology  
   VSAM 135  
 terms used in MAP output 326  
 test  
   conditions 77  
   data 73  
   numeric operand 73  
   UPSI switch 73  
 TEST AFTER 77  
 TEST BEFORE 77  
 TEST compiler option 295  
   conflict with other options 259  
   for full advantage of Debug Tool 343  
   performance considerations 545  
 TEXT files 248  
 TGT memory map 328

TGT memory map 328 (*continued*)  
   example 338  
 THREAD compiler option  
   and the LINKAGE SECTION 16  
 threading consideration 361  
 TITLE statement  
   controlling header on listing 6  
 top-down programming  
   constructs to avoid 536  
 TRACK OVERFLOW option 116  
 transferring control  
   between COBOL and non-COBOL  
   programs 459  
   between COBOL programs 461  
   called program 459  
   calling program 459  
   main and subprograms 459  
   nested programs 469  
 translating CICS into COBOL 347  
 TRAP run-time option  
   closing line-sequential files 167  
   closing QSAM files 120  
   closing VSAM files 150  
   ON SIZE ERROR 191  
 TRUNC compiler option 297  
 TRUNC(STD|OPT|BIN) compiler  
   option 545  
 TSO  
   CALL command 215  
   compiling under 215  
   SYSTEM for compiler messages 221  
 tuning considerations, performance 545  
 TYPECHK compiler option 300

**U**

U format record 114  
 U-level error message 232  
 ull-terminated strings 478  
 unblocked files 369  
 UNDATE intrinsic function 527  
 undefined record format 114  
 undefined records 114  
 unfilled tracks 116  
 UNIX APIs  
   calling 364  
 UNSTRING statement  
   description 83  
   example 83  
   overflow condition 190  
   with DBCS data 90  
 updating VSAM records 148  
 UPPER-CASE intrinsic function 95  
 uppercase 95  
 USAGE clause  
   incompatible data 42  
   IS INDEX 57  
 USE . . . LABEL declarative 130  
 USE AFTER STANDARD LABEL 133  
 USE FOR DEBUGGING declaratives 312  
 user-defined condition 73  
 user-exit work area 576  
 user label  
   exits 133  
   QSAM 133  
   standard 131  
 user-label track 130

**V**

V format record 111  
 valid data  
   numeric 42  
 VALUE clause  
   assigning table values 58  
   Data Description entry 59  
 VALUE OF clause 14  
 variable  
   as reference modifier 87  
   COBOL term for 23  
 variable-length records  
   OCCURS DEPENDING ON (ODO)  
   clause 540  
   QSAM 111  
   sorting 176  
   VSAM 136  
 variable-length table 62  
 variables, environment  
   library-name 304  
 variably located data item 570  
 variably located group 570  
 VBREF compiler option 319  
 VBREF compiler output, example 343  
 verb cross-reference listing  
   description 319  
 verbs used in program 319  
 VSAM files  
   adding records to 149  
   allocating with environment  
   variable 156  
   closing 150  
   coding input/output statements 143  
   comparison of file organizations 136  
   creating alternate indexes 154  
   DATA DIVISION entries 141  
   deleting records from 150  
   dynamically loading 146  
   ENVIRONMENT DIVISION  
   entries 137  
   error processing 191  
   file position indicator (CRP) 144  
   file status key 151  
   input/output error processing 151  
   loading randomly 146  
   loading records into 146  
   opening 145  
   performance considerations 159  
   processing files 135  
   protecting with password 151  
   reading records from 147  
   replacing records in 149  
   return codes 196  
   under CMS  
     file availability 158  
     identifying files, DLBL  
     command 158  
   under OS/390  
     defining data sets 153  
     file availability 152  
     JCL 156  
     RLS mode 157  
   updating records 148  
   with Access Method Services 146  
 VSAM terminology  
   BDAM data set 135  
   comparison to non-VSAM terms 135

VSAM terminology (*continued*)

- ESDS for QSAM 135
- KSDS for ISAM 135
- RRDS for BDAM 135

## W

- W-level error message 232
- WHEN-COMPILED intrinsic function
  - example 99
  - versus WHEN-COMPILED special register 99
- WHEN-COMPILED special register 99
- WHEN phrase
  - EVALUATE statement 71
  - SEARCH statement 65
- windowed date fields 179
- wlist file 281
- WORD compiler option 301
- work data sets 218
- work files 248
- WORKING-STORAGE
  - finding location and size of in storage 339
  - storage location for data 266
- WORKING-STORAGE SECTION
  - class 379
  - description 15
  - method 382
- write a block of records 115
- WRITE ADVANCING statement 119
- WRITE statement 117
  - line-sequential files 165

## X

- x extension with cob2 240
- XREF compiler option 317
- XREF output
  - data-name cross-references 340
  - program-name cross-references 341

## Y

- year field expansion 514
- year-last date fields 516
- year windowing
  - advantages 512
  - how to control 527
  - MLE approach 512
  - when not supported 518
- YEARWINDOW compiler option 303
  - effect on sort/merge 184

## Z

- zero comparison 523
- zoned decimal 36
- ZWB compiler option 304



---

# Readers' Comments — We'd Like to Hear from You

COBOL for OS/390 & VM  
Programming Guide  
Version 2 Release 2

Publication No. SC26-9049-05

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.

\_\_\_\_\_



Fold and Tape

Please do not staple

Fold and Tape



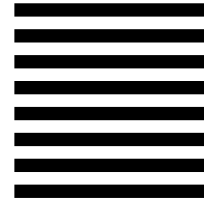
NO POSTAGE  
NECESSARY  
IF MAILED IN THE  
UNITED STATES

# 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





Printed in the United States of America  
on recycled paper containing 10%  
recovered post-consumer fiber.

SC26-9049-05

