

Enterprise COBOL for z/OS
6.4

Programming Guide



Note

Before using this information and the product it supports, be sure to read the general information under [“Notices” on page 959](#).

Fourth edition (26 February 2024 update)

This edition applies to Version 6.4 of IBM® Enterprise COBOL for z/OS® (program number 5655-EC6) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure that you are using the correct edition for the level of the product.

You can view or download softcopy publications free of charge in the [Enterprise COBOL for z/OS library](#). Because Enterprise COBOL for z/OS supports the continuous delivery (CD) model and publications are updated to document the features delivered under the CD model, it is a good idea to check for updates once every two months.

It is our intention to update the product documentation for this release periodically, without updating the order number. If you need to uniquely refer to the version of your product documentation, refer to the order number with the date of update.

© **Copyright International Business Machines Corporation 1991, 2024.**

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

Contents

Tables.....	xxi
Preface.....	xxix
About this information.....	xxix
How this information will help you.....	xxix
Abbreviated terms.....	xxix
Comparison of commonly used terms.....	xxx
How to read syntax diagrams.....	xxx
How examples are shown.....	xxx
Additional documentation and support.....	xxx
Summary of changes.....	xxx
Enterprise COBOL for z/OS 6.4 with PTFs installed.....	xxx
Enterprise COBOL for z/OS 6.4.....	xxx
How to send your comments.....	xxx
Part 1. Coding your program.....	1
Chapter 1. Structuring your program.....	3
Identifying a program.....	3
Identifying a program as recursive.....	4
Marking a program as callable by containing programs.....	4
Setting a program to an initial state.....	4
Changing the header of a source listing.....	5
Describing the computing environment.....	5
Example: FILE-CONTROL entries.....	6
Specifying the collating sequence.....	7
Defining symbolic characters.....	8
Defining a user-defined class.....	8
Defining files to the operating system.....	8
Describing the data.....	11
Using data in input and output operations.....	11
Comparison of WORKING-STORAGE and LOCAL-STORAGE.....	14
Using data from another program.....	16
Processing the data.....	17
How logic is divided in the PROCEDURE DIVISION.....	18
Declaratives.....	21
Chapter 2. Using data.....	23
Using variables, structures, literals, and constants.....	23
Using variables.....	23
Using data items and group items.....	24
Using literals.....	25
Using constants.....	26
Using figurative constants.....	26
Assigning values to data items.....	26
Examples: initializing data items.....	27
Initializing a structure (INITIALIZE).....	31
Assigning values to elementary data items (MOVE).....	32
Assigning values to group data items (MOVE).....	33
Assigning arithmetic results (MOVE or COMPUTE).....	34

Assigning input from a screen or file (ACCEPT).....	35
Displaying values on a screen or in a file (DISPLAY).....	35
Displaying data on the system logical output device.....	36
Using WITH NO ADVANCING.....	37
Using intrinsic functions (built-in functions).....	38
Using tables (arrays) and pointers.....	39
Storage and its addressability.....	39
Chapter 3. Working with numbers and arithmetic.....	43
Defining numeric data.....	43
Displaying numeric data.....	45
Controlling how numeric data is stored.....	46
Formats for numeric data.....	47
Examples: numeric data and internal representation.....	49
Data format conversions.....	51
Conversions and precision.....	52
Sign representation of zoned and packed-decimal data.....	53
Checking for incompatible data (numeric class test).....	53
Performing arithmetic.....	55
Using COMPUTE and other arithmetic statements.....	55
Using arithmetic expressions.....	56
Using numeric intrinsic functions.....	56
Using math-oriented callable services.....	58
Using date callable services.....	59
Examples: numeric intrinsic functions.....	60
Fixed-point contrasted with floating-point arithmetic.....	62
Examples: fixed-point and floating-point evaluations.....	63
Using currency signs.....	64
Example: multiple currency signs.....	65
Chapter 4. Handling tables.....	67
Defining a table (OCCURS).....	67
Nesting tables.....	69
Example: subscripting.....	70
Example: indexing.....	70
Referring to an item in a table.....	70
Subscripting.....	71
Indexing.....	72
Putting values into a table.....	73
Loading a table dynamically.....	73
Initializing a table (INITIALIZE).....	73
Assigning values when you define a table (VALUE).....	74
Example: PERFORM and subscripting.....	76
Example: PERFORM and indexing.....	77
Creating variable-length tables (DEPENDING ON).....	78
Loading a variable-length table.....	80
Assigning values to a variable-length table.....	80
Complex OCCURS DEPENDING ON.....	81
Example: complex ODO.....	81
Effects of change in ODO object value.....	82
Searching a table.....	84
Doing a serial search (SEARCH).....	85
Doing a binary search (SEARCH ALL).....	86
Sorting a table.....	87
Processing table items using intrinsic functions.....	87
Example: processing tables using intrinsic functions.....	88
Working with unbounded tables and groups.....	88
Example: Using unbounded tables for parsing XML documents.....	89

Chapter 5. Selecting and repeating program actions.....	91
Selecting program actions.....	91
Coding a choice of actions.....	91
Coding conditional expressions.....	95
Repeating program actions.....	99
Choosing inline or out-of-line PERFORM.....	99
Coding a loop.....	100
Looping through a table.....	101
Executing multiple paragraphs or sections.....	101
 Chapter 6. Handling strings.....	 103
Joining data items (STRING).....	103
Example: STRING statement.....	104
Splitting data items (UNSTRING).....	105
Example: UNSTRING statement.....	106
Manipulating null-terminated strings.....	108
Example: null-terminated strings.....	108
Referring to substrings of data items.....	109
Reference modifiers.....	110
Example: arithmetic expressions as reference modifiers.....	111
Example: intrinsic functions as reference modifiers.....	111
Tallying and replacing data items (INSPECT).....	112
Examples: INSPECT statement.....	112
Converting data items (intrinsic functions).....	113
Changing case (UPPER-CASE, LOWER-CASE).....	114
Transforming to reverse order (REVERSE).....	114
Converting to numbers (NUMVAL, NUMVAL-C, NUMVAL-F).....	115
Converting from one code page to another.....	116
Converting to hexadecimal or bit data (HEX-OF, BIT-OF).....	116
Converting from hexadecimal or bit data (HEX-TO-CHAR, BIT-TO-CHAR).....	117
Evaluating data items (intrinsic functions).....	117
Evaluating single characters for collating sequence.....	118
Finding the largest or smallest data item.....	118
Finding the length of data items.....	120
Finding the date of compilation.....	121
 Chapter 7. Processing data in an international environment.....	 123
Unicode and the encoding of language characters.....	124
Using national data (Unicode) in COBOL.....	125
Defining national data items.....	126
Using national literals.....	126
COBOL statements and national data.....	127
Intrinsic functions and national data.....	130
Using national-character figurative constants.....	132
Defining national numeric data items.....	132
National groups.....	133
Converting to or from national (Unicode) representation.....	134
Using national groups.....	137
Storage of character data.....	139
Comparing national (UTF-16) data.....	140
Using UTF-8 data (Unicode) in COBOL.....	142
Defining UTF-8 data items.....	142
Using UTF-8 literals.....	144
COBOL statements and UTF-8 data items.....	145
Intrinsic functions support for UTF-8 data items.....	146
Db2 considerations.....	147
Using UTF-8-character figurative constants.....	147

Reference modifiers and UTF-8 data items.....	148
UTF-8 groups.....	149
Converting to or from UTF-8 (Unicode) representation.....	150
Comparing UTF-8 data.....	151
Processing UTF-8 data using UTF-16 (national) data types.....	153
Using intrinsic functions to process UTF-8 encoded data.....	153
Processing Chinese GB 18030 data.....	157
Coding for use of DBCS support.....	158
Defining DBCS data.....	158
Using DBCS literals.....	159
Testing for valid DBCS characters.....	159
Processing alphanumeric data items that contain DBCS data.....	160
Chapter 8. Processing files.....	161
File organization and input-output devices.....	161
Choosing file organization and access mode.....	162
Format for coding input and output.....	163
Allocating files.....	164
Checking for input or output errors.....	165
Chapter 9. Processing QSAM files.....	167
Defining QSAM files and records in COBOL.....	167
Establishing record formats.....	168
Setting block sizes.....	174
Coding input and output statements for QSAM files.....	176
Opening QSAM files.....	177
Dynamically creating QSAM files.....	177
Adding records to QSAM files.....	178
Updating QSAM files.....	178
Writing QSAM files to a printer or spooled data set.....	178
Closing QSAM files.....	179
Handling errors in QSAM files.....	179
Working with QSAM files.....	180
Defining and allocating QSAM files.....	180
Retrieving QSAM files.....	182
Ensuring that file attributes match your program.....	183
Using striped extended-format QSAM data sets.....	185
Allocation of buffers for QSAM files.....	186
Accessing z/OS UNIX files using QSAM.....	186
Processing QSAM ASCII files on tape.....	187
Chapter 10. Processing VSAM files.....	189
VSAM files.....	190
Defining VSAM file organization and records.....	191
Specifying sequential organization for VSAM files.....	192
Specifying indexed organization for VSAM files.....	192
Specifying relative organization for VSAM files.....	193
Specifying access modes for VSAM files.....	194
Defining record lengths for VSAM files.....	195
Coding input and output statements for VSAM files.....	196
File position indicator.....	198
Opening a file (ESDS, KSDS, or RRDS).....	198
Reading records from a VSAM file.....	201
Updating records in a VSAM file.....	201
Adding records to a VSAM file.....	202
Replacing records in a VSAM file.....	203
Deleting records from a VSAM file.....	203
Closing VSAM files.....	203

Handling errors in VSAM files.....	204
Protecting VSAM files with a password.....	204
Example: password protection for a VSAM indexed file.....	205
Working with VSAM data sets under z/OS and z/OS UNIX.....	205
Defining VSAM files.....	205
Creating alternate indexes.....	206
Allocating VSAM files.....	208
Sharing VSAM files through RLS.....	209
Allocation of record areas for VSAM files.....	211
Improving VSAM performance.....	211
Extended addressability support.....	212
Chapter 11. Processing line-sequential files.....	215
Defining line-sequential files and records in COBOL.....	215
Describing the structure of a line-sequential file.....	215
Control characters in line-sequential files.....	216
Allocating line-sequential files.....	216
Coding input-output statements for line-sequential files.....	217
Opening line-sequential files.....	217
Reading records from line-sequential files.....	218
Adding records to line-sequential files.....	218
Closing line-sequential files.....	219
Handling errors in line-sequential files.....	219
Chapter 12. Sorting and merging files.....	221
Sort and merge process.....	222
Describing the sort or merge file.....	222
Describing the input to sorting or merging.....	223
Example: describing sort and input files for SORT.....	223
Coding the input procedure.....	224
Describing the output from sorting or merging.....	225
Coding the output procedure.....	225
Example: coding the output procedure when using DFSORT.....	226
Restrictions on input and output procedures.....	227
Defining sort and merge data sets.....	227
Sorting variable-length records.....	227
Requesting the sort or merge.....	228
Setting sort or merge criteria.....	229
Example: sorting with input and output procedures.....	230
Choosing alternate collating sequences.....	231
Preserving the original sequence of records with equal keys.....	231
Determining whether the sort or merge was successful.....	231
Stopping a sort or merge operation prematurely.....	232
Improving sort performance with FASTSRT.....	232
FASTSRT requirements for JCL.....	232
FASTSRT requirements for sort input and output files.....	233
Checking for sort errors with NOFASTSRT.....	234
Controlling sort behavior.....	235
Changing DFSORT defaults with control statements.....	236
Allocating storage for sort or merge operations.....	236
Allocating space for sort files.....	237
Using checkpoint/restart with DFSORT.....	237
Using COBOL user exit routines with DFSORT.....	237
Sorting under CICS.....	238
CICS SORT application restrictions.....	238
Chapter 13. Handling errors.....	241
Requesting dumps.....	241

Handling errors in joining and splitting strings.....	242
Handling errors in arithmetic operations.....	242
Example: checking for division by zero.....	243
Handling errors in input and output operations.....	243
Using the end-of-file condition (AT END).....	246
Coding ERROR declaratives.....	246
Using file status keys.....	247
Example: file status key.....	248
Using VSAM status codes (VSAM files only).....	248
Example: checking VSAM status codes.....	249
Coding INVALID KEY phrases.....	250
Example: FILE STATUS and INVALID KEY.....	251
Handling errors when calling programs.....	251
Writing routines for handling errors.....	252

Part 2. Compiling and debugging your program.....253

Chapter 14. Compiling under z/OS.....	255
Compiling with JCL.....	255
Using a cataloged procedure.....	256
Writing JCL to compile programs.....	260
Compiling under TSO.....	262
Example: ALLOCATE and CALL for compiling under TSO.....	263
Example: CLIST for compiling under TSO.....	264
Starting the compiler from an assembler program.....	264
Defining compiler input and output.....	266
Data sets used by the compiler under z/OS.....	267
Defining the source code data set (SYSIN).....	269
Defining a compiler-option data set (SYSOPTF).....	270
Specifying source libraries (SYSLIB).....	270
Defining the output data set (SYSPRINT).....	271
Directing compiler messages to your terminal (SYSTEM).....	271
Creating object code (SYSLIN or SYSPUNCH).....	271
Defining an associated-data file (SYSADATA).....	272
Defining the Java-source output file (SYSJAVA).....	272
Defining the debug data set (SYSDEBUG).....	272
Defining the library-processing output file (SYSMDECK).....	273
Specifying compiler options under z/OS.....	273
Specifying compiler options in the PROCESS (CBL) statement.....	274
Example: specifying compiler options using JCL.....	275
Example: specifying compiler options under TSO.....	275
Compiler options and compiler output under z/OS.....	275
Compiling multiple programs (batch compilation).....	276
Example: batch compilation.....	277
Specifying compiler options in a batch compilation.....	278
Example: precedence of options in a batch compilation.....	278
Example: LANGUAGE option in a batch compilation.....	279
Correcting errors in your source program.....	280
Generating a list of compiler messages.....	281
Messages and listings for compiler-detected errors.....	281
Format of compiler diagnostic messages.....	281
Severity codes for compiler diagnostic messages.....	282
Chapter 15. Compiling under z/OS UNIX.....	283
Setting environment variables under z/OS UNIX.....	283
Specifying compiler options under z/OS UNIX.....	284
Compiling and linking with the cob2 command.....	285

Creating a DLL under z/OS UNIX.....	286
Example: using cob2 to compile and link under z/OS UNIX.....	286
cob2 syntax and options.....	287
cob2 input and output files.....	289
Compiling using scripts.....	290
Chapter 16. Compiling, linking, and running OO applications.....	291
Compiling, linking, and running OO applications under z/OS UNIX.....	291
Compiling OO applications under z/OS UNIX.....	291
Preparing OO applications under z/OS UNIX.....	292
Example: compiling and linking a COBOL class definition under z/OS UNIX.....	293
Running OO applications under z/OS UNIX.....	293
Compiling, linking, and running OO applications in JCL or TSO/E.....	295
Compiling OO applications in JCL or TSO/E.....	295
Preparing and running OO applications in JCL or TSO/E.....	296
Example: compiling, linking, and running an OO application using JCL.....	297
Using Java SDKs for z/OS.....	298
Object-oriented syntax, and Java 6 or later.....	299
Chapter 17. Compiling, linking, and running non-OO COBOL applications that interoperate with Java.....	301
Building non-OO applications that interoperate with Java.....	301
Introducing the COBPROD application.....	302
Preparing to compile, bind, and run non-OO COBOL applications that interoperate with Java.....	303
Step 1: Compiling user COBOL programs.....	303
Step 2: Running the cjbuild utility to build a DLL of Java stub programs.....	304
Step 3: Linking COBOL programs.....	308
Step 4: Building Java programs.....	308
Troubleshooting.....	308
Example: COBPROD application - building and running.....	309
Sample JCL for building and running the COBPROD application.....	314
AMODE considerations for running non-OO COBOL/Java interoperable applications.....	321
cjbuild command reference	321
Chapter 18. Compiling, binding, and running COBOL AMODE 64 applications.....	325
Compiling and binding AMODE 64 programs under z/OS.....	325
DLL considerations.....	326
Compiling and binding AMODE 64 programs under z/OS UNIX.....	326
Running COBOL AMODE 64 applications.....	327
Specifying COBOL only runtime options.....	327
Chapter 19. COBOL compiler and runtime options.....	329
COBOL compiler options.....	329
Option settings for 85 COBOL Standard conformance.....	333
Conflicting compiler options.....	334
ADATA.....	335
ADV.....	336
AFP.....	336
APOST/QUOTE.....	337
ARCH.....	337
ARITH.....	339
AWO.....	340
BLOCK0.....	340
BUFSIZE.....	341
CICS.....	342
CODEPAGE.....	343
COMPILE.....	345
CONDCOMP.....	346

COPYLOC.....	346
COPYRIGHT.....	347
CURRENCY.....	348
DATA.....	349
DBCS.....	350
DECK.....	350
DEFINE.....	350
DIAGTRUNC.....	352
DISPSIGN.....	352
DLL.....	353
DUMP.....	355
DYNAM.....	356
EXIT.....	357
EXPORTALL.....	359
FASTSRT.....	359
FLAG.....	359
FLAGSTD.....	360
HGPR.....	362
INITCHECK.....	363
INITIAL.....	364
INLINE.....	365
INTDATE.....	365
INVDATA.....	366
JAVAIOP.....	369
LANGUAGE.....	370
LINECOUNT.....	371
LIST.....	372
LP.....	373
MAP.....	374
MAXPCF.....	375
MDECK.....	376
NAME.....	377
NSYMBOL.....	377
NUMBER.....	378
NUMCHECK.....	378
NUMPROC.....	382
OBJECT.....	383
OFFSET.....	383
OPTFILE.....	384
OPTIMIZE.....	385
OUTDD.....	386
PARMCHECK.....	387
PGMNAME.....	388
QUALIFY.....	390
RENT.....	391
RMODE.....	392
RULES.....	393
SEQUENCE.....	395
SERVICE.....	395
SMARTBIN.....	396
SOURCE.....	397
SPACE.....	398
SQL.....	398
SQLCCSID.....	399
SQLIMS.....	400
SSRANGE.....	401
STGOPT.....	402
SUPPRESS.....	403

	TERMINAL.....	403
	TEST.....	404
	THREAD.....	408
	TRUNC.....	409
	TUNE.....	411
	VBREF.....	412
	VLR.....	412
	VSAMOPENFS.....	414
	WORD.....	415
	XMLPARSE.....	415
	XREF.....	416
	ZONECHECK.....	417
	ZONEDATA.....	418
	ZWB.....	421
	COBOL runtime options.....	421
	Specifying COBOL runtime options.....	422
	VSAM dynamic access read option VSAMDYNAMICDIR.....	424
	Disabling COBOL runtime options report option DISABLEUOPTREPORT.....	424
	Building IGZUOPT and IGZQUOPT runtime option control block.....	425
	JCL samples for COBOL IGZUOPT runtime options.....	425
	COBOL runtime options report.....	425
	Chapter 20. Compiler-directing statements.....	429
	Chapter 21. Debugging.....	433
	Debugging with source language.....	434
	Tracing program logic.....	434
	Finding and handling input-output errors.....	435
	Validating data.....	435
	Moving, initializing or setting uninitialized data.....	435
	Generating information about procedures.....	436
	Debugging using compiler options.....	437
	Finding coding errors.....	438
	Finding line sequence problems.....	438
	Checking for invalid COBOL data or invalid COBOL programs.....	439
	Checking for valid ranges.....	439
	Selecting the level of error to be diagnosed.....	440
	Finding program entity definitions and references.....	442
	Listing data items.....	442
	Using the debugger.....	443
	Getting listings.....	443
	Example: short listing.....	445
	Example: SOURCE and NUMBER output.....	448
	Example: MAP output.....	449
	Reading LIST output.....	454
	Example: XREF output: data-name cross-references.....	473
	Example: OFFSET compiler output.....	477
	Example: VBREF compiler output.....	478
	Example: conditional compilation output.....	478
	Suppressing information in CEEDUMP processing (IGZ1OPT).....	479
	Generating a report of dynamically called COBOL programs.....	480

Part 3. Targeting COBOL programs for certain environments..... 481

	Chapter 22. Developing COBOL programs for CICS.....	483
	Coding COBOL programs to run under CICS.....	483
	Getting the system date under CICS.....	485

Calling to or from COBOL programs.....	485
Determining the success of ECI calls.....	486
Compiling with the CICS option.....	487
Separating CICS suboptions.....	488
Integrated CICS translator.....	488
Using the separate CICS translator.....	489
CICS reserved-word table.....	490
Handling errors by using CICS HANDLE.....	491
Example: handling errors by using CICS HANDLE.....	492
Chapter 23. Programming for a Db2 environment.....	493
Db2 coprocessor.....	493
Using the separate Db2 precompiler.....	494
Coding SQL statements.....	494
Using SQL INCLUDE with the Db2 coprocessor.....	495
Using character data in SQL statements.....	495
Using national decimal data in SQL statements.....	496
Using national group items in SQL statements.....	496
Using binary items in SQL statements.....	497
Determining the success of SQL statements.....	497
Compiling with the SQL option.....	497
Separating Db2 suboptions.....	498
COBOL and Db2 CCSID determination.....	498
Code-page determination for string host variables in SQL statements.....	499
Programming with the SQLCCSID or NOSQLCCSID option.....	500
Differences in how the Db2 precompiler and coprocessor behave.....	500
Choosing the DYNAM or NODYNAM compiler option.....	502
Chapter 24. Developing COBOL programs for IMS.....	505
IMS SQL coprocessor.....	505
Coding SQLIMS statements.....	506
Using SQLIMS INCLUDE with the IMS SQL coprocessor.....	506
Using character data in SQLIMS statements.....	506
Using binary items in SQLIMS statements.....	507
Determining the success of SQLIMS statements.....	507
Compiling with the SQLIMS option.....	507
Separating IMS suboptions.....	508
Compiling and linking COBOL programs for running under IMS.....	508
Using object-oriented COBOL and Java under IMS.....	509
Calling a COBOL method from a Java application under IMS.....	509
Building a mixed COBOL-Java application that starts with COBOL.....	510
Writing mixed-language IMS applications.....	510
Chapter 25. Running COBOL programs under z/OS UNIX.....	513
Running in z/OS UNIX environments.....	513
Setting and accessing environment variables.....	514
Setting environment variables that affect execution.....	515
Runtime environment variables.....	515
Example: setting and accessing environment variables.....	516
Calling UNIX/POSIX APIs.....	516
Accessing main program parameters under z/OS UNIX.....	518
Example: accessing main program parameters under z/OS UNIX.....	518
Chapter 26. Developing AMODE 64 programs.....	521
Pointer data items with AMODE 64 programs.....	521
Compiler limits with AMODE 64 support.....	522
CALL statement with AMODE 64 support.....	522
Using compiler options to compile AMODE 64 programs.....	522

Other AMODE 64 considerations.....	523
Chapter 27. Developing COBOL programs in the mixed AMODE Language Environment.....	525
COBOL programs in AMODE 31 interacting with Java programs in AMODE 64.....	525
Application calls between COBOL programs in AMODE 31 and Java programs in AMODE 64..	525
Building the mixed AMODE COBOL/Java interoperability application	526
Running the mixed AMODE COBOL/Java interoperability program.....	526
Examples for mixed AMODE COBOL/Java interoperability application	527
Other considerations and restrictions.....	528
COBOL applications with mixed AMODE 31 and AMODE 64 programs.....	528
Dynamic call between AMODE 31 and AMODE 64 programs.....	528
COBOL runtime option AMODE3164.....	529
Other considerations and restrictions.....	530

Part 4. Structuring complex applications..... 531

Chapter 28. Using subprograms.....	533
Main programs, subprograms, and calls.....	533
Ending and reentering main programs or subprograms.....	534
Transferring control to another program.....	535
Making static calls.....	536
Making dynamic calls.....	536
AMODE switching.....	538
Performance considerations of static and dynamic calls.....	540
Making both static and dynamic calls.....	540
Examples: static and dynamic CALL statements.....	541
Calling nested COBOL programs.....	542
Making recursive calls.....	545
Calling to and from object-oriented programs.....	545
Using procedure and function pointers.....	545
Deciding which type of pointer to use.....	547
Calling alternate entry points.....	548
Making programs reentrant.....	548
Chapter 29. Sharing data.....	551
Passing data.....	551
Describing arguments in the calling program.....	553
Describing parameters in the called program.....	554
Testing for OMITTED arguments.....	555
Coding the LINKAGE SECTION.....	555
Coding the PROCEDURE DIVISION for passing arguments.....	556
Grouping data to be passed.....	556
Handling null-terminated strings.....	556
Using pointers to process a chained list.....	557
Passing return-code information.....	559
Using the RETURN-CODE special register.....	560
Using PROCEDURE DIVISION RETURNING . . .	560
Specifying CALL . . . RETURNING.....	560
Sharing data by using the EXTERNAL clause.....	560
Sharing files between programs (external files).....	561
Example: using external files.....	561
Accessing main program parameters under z/OS.....	564
Example: accessing main program parameters under z/OS.....	564
Chapter 30. Creating a DLL or a DLL application.....	567
Dynamic link libraries (DLLs).....	567
Compiling programs to create DLLs.....	568

Linking DLLs.....	569
Example: sample JCL for a procedural DLL application.....	569
Using CALL identifier with DLLs.....	570
Search order for DLLs in the z/OS UNIX file system.....	571
Using DLL linkage and dynamic calls together.....	571
Using procedure or function pointers with DLLs.....	572
Calling DLLs from non-DLLs.....	572
Example: calling DLLs from non-DLLs.....	573
Using COBOL DLLs with C/C++ programs.....	574
Using DLLs in OO COBOL applications.....	575
Chapter 31. Preparing COBOL programs for multithreading.....	577
Multithreading.....	577
Choosing THREAD to support multithreading.....	578
Transferring control to multithreaded programs.....	579
Ending multithreaded programs.....	579
Processing files with multithreading.....	580
File-definition (FD) storage.....	580
Serializing file access with multithreading.....	581
Example: usage patterns of file input and output with multithreading.....	581
Handling COBOL limitations with multithreading.....	582
Chapter 32. Using user-defined functions.....	585
Structuring user-defined functions.....	585
Coding user-defined function definitions.....	585
Compiling user-defined function definitions.....	586
Invoking user-defined functions.....	586
Specifying external names for user-defined functions.....	587
Link-editing user-defined functions.....	587
Passing arguments BY CONTENT to user-defined functions.....	587
Chapter 33. Using function prototypes.....	589
Function prototype example.....	589
Function prototype with external name.....	589
Function prototype best practices.....	590
Language standard considerations.....	591
Part 5. Using COBOL for Web Services.....	593
Chapter 34. Web Services interface	595
Chapter 35. Processing JSON input	597
Parsing JSON documents	597
How to match JSON names that are not valid COBOL data names to data items	598
Preventing data items from being populated by the JSON PARSE statement	598
Handling JSON arrays.....	599
Parsing JSON anonymous arrays.....	600
Handling JSON null values.....	602
JSON PARSE example.....	603
Chapter 36. Producing JSON output	607
Generating JSON anonymous arrays.....	607
Generating JSON null values.....	609
Chapter 37. Processing XML input.....	611
XML parser in COBOL.....	612
Accessing XML documents.....	613

Parsing XML documents.....	613
Writing procedures to process XML.....	615
XML events.....	617
Transforming XML text to COBOL data items.....	622
Parsing XML documents with validation.....	622
Parsing XML documents one segment at a time.....	625
Handling splits using the XML-INFORMATION special register.....	627
The encoding of XML documents.....	628
XML input document encoding.....	629
Parsing XML documents encoded in UTF-8.....	632
Handling XML PARSE exceptions.....	633
How the XML parser handles errors.....	635
Handling encoding conflicts.....	636
Terminating XML parsing.....	637
XML PARSE examples.....	637
Example: parsing a simple document.....	637
Example: program for processing XML.....	638
Example: parsing an XML document that uses namespaces.....	642
Example: parsing an XML document one segment at a time.....	646
Example: parsing XML documents with validation.....	648
 Chapter 38. Producing XML output.....	 651
Generating XML output.....	651
Controlling the encoding of generated XML output.....	656
Handling XML GENERATE exceptions.....	656
Example: generating XML.....	657
Enhancing XML output.....	661
Example: enhancing XML output.....	662

Part 6. Developing object-oriented programs.....665

Chapter 39. Writing object-oriented programs.....	667
Example: accounts.....	668
Defining a class.....	670
CLASS-ID paragraph for defining a class.....	671
REPOSITORY paragraph for defining a class.....	672
WORKING-STORAGE SECTION for defining class instance data.....	673
Example: defining a class.....	674
Defining a class instance method.....	674
METHOD-ID paragraph for defining a class instance method.....	675
INPUT-OUTPUT SECTION for defining a class instance method.....	676
DATA DIVISION for defining a class instance method.....	676
PROCEDURE DIVISION for defining a class instance method.....	677
Overriding an instance method.....	678
Overloading an instance method.....	678
Coding attribute (get and set) methods.....	679
Example: defining a method.....	680
Defining a client.....	682
REPOSITORY paragraph for defining a client.....	683
DATA DIVISION for defining a client.....	683
Comparing and setting object references.....	684
Invoking methods (INVOKE).....	685
Creating and initializing instances of classes.....	689
Freeing instances of classes.....	691
Example: defining a client.....	691
Defining a subclass.....	692
CLASS-ID paragraph for defining a subclass.....	693

REPOSITORY paragraph for defining a subclass.....	693
WORKING-STORAGE SECTION for defining subclass instance data.....	694
Defining a subclass instance method.....	694
Example: defining a subclass (with methods).....	694
Defining a factory section.....	695
WORKING-STORAGE SECTION for defining factory data.....	696
Defining a factory method.....	697
Example: defining a factory (with methods).....	699
Wrapping procedure-oriented COBOL programs.....	703
Structuring OO applications.....	703
Examples: COBOL applications that run using the java command.....	704
Chapter 40. Communicating with Java methods.....	707
Accessing JNI services.....	707
Handling Java exceptions.....	708
Managing local and global references.....	709
Java access controls.....	711
Sharing data with Java.....	711
Coding interoperable data types in OO COBOL and Java.....	711
Declaring arrays and strings for Java.....	712
Manipulating Java arrays.....	714
Manipulating Java strings.....	716
Example: J2EE client written in COBOL.....	719
Example: invoking Java from a batch COBOL program.....	722

Part 7. COBOL/Java interoperability outside of the object-oriented (OO) COBOL framework.....727

Chapter 41. Making an existing program callable from Java.....	729
Chapter 42. Calling static Java methods from COBOL.....	731
Chapter 43. Using Java-compatible array types in COBOL.....	733
Byte arrays and general group parameters.....	733
Array update semantics and restrictions in Java-interoperable applications.....	734
Chapter 44. Accessing COBOL WORKING-STORAGE data items from Java.....	735
Chapter 45. Handling errors in Java-interoperable COBOL applications.....	737
Chapter 46. AMODE considerations.....	739

Part 8. Specialized processing.....741

Chapter 47. Interrupts and checkpoint/restart.....	743
Setting checkpoints.....	743
Designing checkpoints.....	744
Testing for a successful checkpoint.....	744
DD statements for defining checkpoint data sets.....	744
Messages generated during checkpoint.....	746
Restarting programs.....	746
Requesting automatic restart.....	746
Requesting deferred restart.....	747
Formats for requesting deferred restart.....	747
Resubmitting jobs for restart.....	748
Example: restarting a job at a specific checkpoint step.....	748
Example: requesting a step restart.....	748

Example: resubmitting a job for a step restart.....	749
Example: resubmitting a job for a checkpoint restart.....	749
I Chapter 48. Using zlib compression from a COBOL program.....	751
Part 9. Improving performance and productivity.....	755
Chapter 49. Tuning your program.....	757
Using an optimal programming style.....	757
Using structured programming.....	758
Factoring expressions.....	758
Using symbolic constants.....	758
Choosing efficient data types.....	758
Choosing efficient computational data items.....	759
Using consistent data types.....	759
Making arithmetic expressions efficient.....	759
Making exponentiations efficient.....	760
Using VOLATILE clauses efficiently.....	760
Handling tables efficiently.....	760
Optimization of table references.....	761
Optimizing your code.....	762
Optimization.....	762
Choosing compiler features to enhance performance.....	763
Performance-related compiler options.....	764
Evaluating performance.....	768
Running efficiently with CICS, IMS, or VSAM.....	769
Choosing static or dynamic calls.....	769
Chapter 50. Simplifying coding.....	771
Eliminating repetitive coding.....	772
Example: using the COPY statement.....	772
Using Language Environment callable services.....	773
Sample list of Language Environment callable services.....	774
Calling Language Environment services.....	775
Example: Language Environment callable services.....	776
Using the format 2 SORT statement to sort a table.....	777
Appendix A. Intermediate results and arithmetic precision.....	779
Terminology used for intermediate results.....	780
Example: calculation of intermediate results.....	780
Fixed-point data and intermediate results.....	781
Addition, subtraction, multiplication, and division.....	781
Exponentiation.....	782
Example: exponentiation in fixed-point arithmetic.....	783
Truncated intermediate results.....	783
Binary data and intermediate results.....	784
Intrinsic functions evaluated in fixed-point arithmetic.....	784
Integer functions.....	784
Mixed functions.....	785
Floating-point data and intermediate results.....	786
Exponentiations evaluated in floating-point arithmetic.....	786
Intrinsic functions evaluated in floating-point arithmetic.....	787
Arithmetic expressions in nonarithmetic statements.....	787
Appendix B. Converting double-byte character set (DBCS) data.....	789
DBCS notation.....	789
Alphanumeric to DBCS data conversion (IGZCA2D).....	789

IGZCA2D syntax.....	789
IGZCA2D return codes.....	790
Example: IGZCA2D.....	791
DBCS to alphanumeric data conversion (IGZCD2A).....	791
IGZCD2A syntax.....	791
IGZCD2A return codes.....	792
Example: IGZCD2A.....	793
Appendix C. XML reference material.....	795
XML PARSE exceptions with XMLPARSE(XMLSS) in effect.....	795
XML PARSE exceptions with XMLPARSE(COMPAT) in effect.....	797
XML PARSE exceptions that allow continuation.....	797
XML PARSE exceptions that do not allow continuation.....	800
XML GENERATE exceptions.....	803
Appendix D. JSON reference material.....	805
JSON GENERATE exceptions.....	805
JSON PARSE conditions and associated codes and runtime messages.....	805
Nonexception conditions and corresponding values of JSON-STATUS.....	806
Exception conditions, and corresponding values of JSON-CODE.....	806
Nonexception condition runtime messages.....	807
Exception condition runtime messages.....	808
Appendix E. EXIT compiler option.....	811
Using the user-exit work area.....	811
Calling from exit modules.....	812
Processing of INEXIT.....	812
INEXIT parameters.....	812
Processing of LIBEXIT.....	813
Processing of LIBEXIT with nested COPY statements.....	814
LIBEXIT parameters.....	815
Processing of PRTEXTIT.....	816
PRTEXTIT parameters.....	817
Processing of ADEXIT.....	818
ADEXIT parameters.....	818
Processing of MSGEXIT.....	819
MSGEXIT parameters.....	820
Customizing compiler-message severities.....	821
Example: MSGEXIT user exit.....	823
Error handling for exit modules.....	828
Using the EXIT compiler option with CICS, SQL and SQLIMS statements.....	830
Appendix F. JNI.cpy copybook.....	833
Appendix G. COBOL SYSADATA file contents.....	837
Compiler options that affect the SYSADATA file.....	837
SYSADATA record types.....	838
Example: SYSADATA.....	839
SYSADATA record descriptions.....	840
Common header section.....	841
Job identification record: X'0000'.....	843
ADATA identification record: X'0001'.....	844
Compilation unit start end record: X'0002'.....	844
Options record: X'0010'.....	845
External symbol record: X'0020'.....	857
Parse tree record: X'0024'.....	858
Token record: X'0030'.....	885

Source error record: X'0032'.....	913
Source record: X'0038'.....	913
COPY REPLACING record: X'0039'.....	914
Symbol record: X'0042'.....	915
Symbol cross-reference record: X'0044'.....	927
Nested program record: X'0046'.....	929
Library record: X'0060'.....	929
Statistics record: X'0090'.....	930
EVENTS record: X'0120'.....	930
Appendix H. Using sample programs.....	935
IGYTCARA: batch application.....	935
Input data for IGYTCARA.....	936
Report produced by IGYTCARA.....	937
Preparing to run IGYTCARA.....	938
IGYTCARB: interactive program.....	939
Preparing to run IGYTCARB.....	939
IGYTSALE: nested program application.....	941
Input data for IGYTSALE.....	942
Reports produced by IGYTSALE.....	944
Preparing to run IGYTSALE.....	948
Language elements and concepts that are illustrated.....	949
Appendix I. Accessing JCL parameters.....	955
Appendix J. Accessibility features for Enterprise COBOL for z/OS.....	957
Notices.....	959
Trademarks.....	961
Glossary.....	963
List of resources.....	1007
Enterprise COBOL for z/OS publications.....	1007
Related publications.....	1007
Index.....	1011

Tables

1. FILE-CONTROL entries.....	6
2. FILE SECTION entries.....	12
3. Assignment to data items in a program.....	27
4. Effect of RMODE and RENT compiler options on the RMODE attribute.....	41
5. Ranges in value of COMP-5 data items.....	49
6. Internal representation of numeric items.....	50
7. NUMCLS(PRIM) and valid signs.....	54
8. NUMCLS(ALT) and valid signs.....	54
9. Order of evaluation of arithmetic operators.....	56
10. Numeric intrinsic functions.....	57
11. Compatibility of math intrinsic functions and callable services.....	58
12. INTDATE(LILIAN) and compatibility of date intrinsic functions and callable services.....	59
13. INTDATE(ANSI) and compatibility of date intrinsic functions and callable services.....	59
14. Hexadecimal values of the euro sign.....	65
15. COBOL statements and national data.....	127
16. Intrinsic functions and national character data.....	130
17. National group items that are processed with group semantics.....	138
18. Encoding and size of alphanumeric, DBCS, and national data.....	139
19. Summary of file organizations, access modes, and record formats of COBOL files.....	163
20. QSAM file allocation.....	181
21. Maximum record length of QSAM files.....	184
22. Comparison of VSAM, COBOL, and non-VSAM terminology.....	189
23. Comparison of VSAM data-set types.....	190

24. VSAM file organization, access mode, and record format.....	191
25. Definition of VSAM fixed-length records.....	195
26. Definition of VSAM variable-length records.....	196
27. I/O statements for VSAM sequential files.....	197
28. I/O statements for VSAM relative and indexed files.....	197
29. Statements to load records into a VSAM file.....	200
30. Statements to update records in a VSAM file.....	202
31. Methods for improving VSAM performance.....	211
32. Methods for checking for sort errors with NOFASTSRT.....	234
33. Methods for controlling sort behavior.....	235
34. Compiler data sets.....	267
35. Block size of fixed-length compiler data sets.....	269
36. Block size of variable-length compiler data sets.....	269
37. Types of compiler output under z/OS.....	275
38. Severity codes for compiler diagnostic messages.....	282
39. Input files to the cob2 command.....	289
40. Output files from the cob2 command.....	290
41. Commands for compiling and linking a class definition.....	292
42. java command options for customizing the JVM.....	294
43.	305
44. Options and descriptions.....	322
45. Compiler options.....	329
46. Mutually exclusive compiler options.....	334
47. EBCDIC multibyte coded character set identifiers.....	344
48. DISPLAY output with the DISPSIGN(COMPAT) option or the DISPSIGN(SEP) option specified:.....	353

49. Setting INVDATA and NUMPROC options when migrating from earlier COBOL versions.....	366
50. Values of the LANGUAGE compiler option.....	371
51. Mapping of removed options to new options.....	385
52. Length of record read and file status.....	414
53. COBOL runtime options.....	421
54. AMODE 31 COBOL runtime options.....	423
55. AMODE 64 COBOL runtime options.....	423
56. COBOL runtime options and corresponding JCL samples.....	425
57. Examples for AMODE 31 runtime scenarios.....	426
58. Examples for AMODE 64 runtime scenarios.....	427
59. Severity levels of compiler messages.....	440
60. Using compiler options to get listings.....	443
61. Terms used in MAP output.....	452
62. Symbols used in LIST and MAP output.....	453
63. Compiler options in the INFO BYTE section.....	456
64. Signature information bytes.....	457
65. Calls between COBOL and assembler under CICS.....	486
66. Compiler options required for the integrated CICS translator.....	487
67. Compiler options required for the separate CICS translator.....	489
68. TRUNC compiler options recommended for the separate CICS translator.....	490
69. Samples with POSIX function calls.....	517
70. Effects of termination statements.....	534
71. Methods for passing data in the CALL statement.....	552
72. Compiler options for DLL applications.....	568
73. Binder options for DLL applications.....	569

74. Special registers used by the XML parser.....	615
75. Results of processing-procedure changes to XML-CODE with XMLPARSE(XMLSS) in effect.....	618
76. Results of processing-procedure changes to XML-CODE with XMLPARSE(COMPAT) in effect.....	619
77. Coded character sets for XML documents.....	628
78. Hexadecimal values of white-space characters.....	629
79. Aliases for XML encoding declarations.....	631
80. Hexadecimal values of special characters for various EBCDIC CCSIDs.....	632
81. XML events and special registers.....	638
82. XML events and special registers.....	643
83. XML events and special registers from parsing XML document with an undeclared namespace prefix.....	645
84. Encoding of generated XML if the ENCODING phrase is omitted.....	656
85. Structure of class definitions.....	670
86. Structure of instance method definitions.....	674
87. Structure of COBOL clients.....	682
88. Conformance of arguments in a COBOL client.....	687
89. Conformance of the returned data item in a COBOL client.....	688
90. Structure of factory definitions.....	696
91. Structure of factory method definitions.....	697
92. JNI services for local and global references.....	710
93. Interoperable data types in COBOL and Java.....	712
94. Interoperable arrays and strings in COBOL and Java.....	712
95. Noninteroperable array types in COBOL and Java.....	713
96. JNI array services.....	714
97. Services that convert between jstring references and national data.....	716
98. Services that convert between jstring references and alphanumeric data.....	717

99. Performance-related compiler options.....	764
100. Performance-tuning worksheet.....	768
101. Language Environment callable services.....	775
102. Comparison of format 1 and format 2 SORT statements.....	777
103. IGZCA2D return codes.....	790
104. IGZCD2A return codes.....	792
105. Reason codes for XML PARSE exceptions that are unique to Enterprise COBOL.....	795
106. XML PARSE exceptions that allow continuation.....	797
107. XML PARSE exceptions that do not allow continuation (for XMLPARSE(COMPAT)).....	800
108. XML GENERATE exceptions.....	803
109. JSON GENERATE exceptions.....	805
110. Reason codes for JSON nonexception conditions.....	806
111. Reason codes for JSON exception conditions.....	806
112. Layout of the user-exit work area.....	811
113. INEXIT processing.....	812
114. INEXIT parameters.....	812
115. LIBEXIT processing.....	813
116. LIBEXIT processing with nonnested COPY statements.....	814
117. LIBEXIT processing with nested COPY statements.....	815
118. LIBEXIT parameters.....	815
119. PRTEXT processing.....	816
120. PRTEXT parameters.....	817
121. ADEXIT processing.....	818
122. ADEXIT parameters.....	818
123. MSGEXIT processing.....	819

124. MSGEXIT parameters.....	820
125. FIPS (FLAGSTD) message categories.....	822
126. Actions possible in exit modules for CICS, SQL and SQLIMS statements.....	830
127. SYSADATA record types.....	838
128. SYSADATA common header section.....	841
129. SYSADATA job identification record.....	843
130. ADATA identification record.....	844
131. SYSADATA compilation unit start end record.....	844
132. SYSADATA options record.....	845
133. SYSADATA external symbol record.....	857
134. SYSADATA parse tree record.....	858
135. SYSADATA token record.....	886
136. SYSADATA source error record.....	913
137. SYSADATA source record.....	913
138. SYSADATA COPY REPLACING record.....	914
139. SYSADATA symbol record.....	915
140. SYSADATA symbol cross-reference record.....	927
141. SYSADATA nested program record.....	929
142. SYSADATA library record.....	929
143. SYSADATA statistics record.....	930
144. SYSADATA EVENTS TIMESTAMP record layout.....	931
145. SYSADATA EVENTS PROCESSOR record layout.....	931
146. SYSADATA EVENTS FILE END record layout.....	931
147. SYSADATA EVENTS PROGRAM record layout.....	932
148. SYSADATA EVENTS FILE ID record layout.....	932

149. SYSADATA EVENTS ERROR record layout..... 933

Preface

About this information

This information is for COBOL programmers and system programmers. It helps you understand how to use Enterprise COBOL for z/OS to compile COBOL programs. It also describes the operating system features that you might need to optimize program performance or handle errors.

For information about COBOL language, and for references needed to write a program for an IBM COBOL compiler, see the *Enterprise COBOL for z/OS Language Reference*.

Throughout this information, "COBOL" or "Enterprise COBOL" refers to "IBM Enterprise COBOL for z/OS" or "IBM Enterprise COBOL Value Unit Edition for z/OS".

How this information will help you

This information will help you write and compile Enterprise COBOL programs.

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

For information about migrating programs to Enterprise COBOL, see the *IBM Enterprise COBOL for z/OS Migration Guide*.

IBM z/OS Language Environment® provides the runtime environment and runtime services that are required to run Enterprise COBOL programs. You can find information about link-editing and running programs in the *IBM z/OS Language Environment Programming Guide* and *IBM z/OS Language Environment Programming Reference*.

For a comparison of commonly used Enterprise COBOL and Language Environment terms, see [“Comparison of commonly used terms” on page xxx](#).

Abbreviated terms

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

Term used	Long form
CICS®	CICS Transaction Server
z/OS Debugger	IBM z/OS Debugger (formerly known as Debug Tool) ¹
Enterprise COBOL	IBM Enterprise COBOL for z/OS or IBM Enterprise COBOL Value Unit Edition for z/OS
Language Environment or LE	IBM z/OS Language Environment
MVS™	MVS/ESA
z/OS UNIX	z/OS UNIX System Services

Term used	Long form
<p>Note:</p> <p>1. IBM z/OS Debugger is a common component that is provided in the following products:</p> <ul style="list-style-type: none"> • IBM Debug for z/OS, previously known as IBM Debug Tool for z/OS • IBM Developer for z/OS, previously known as IBM Developer for z Systems® and Rational® Developer for z Systems • IBM Developer for z/OS (Enterprise Edition), which is also provided with the IBM Application Delivery Foundation for z/OS bundle • IBM Z® and Cloud Modernization Stack <p>The debug features provided in different products vary. For more information, see Overview of IBM z/OS Debugger.</p> <p>Depending on the level of Enterprise COBOL that you are using to create the COBOL application, update your debugger to a certain version:</p> <ul style="list-style-type: none"> • z/OS Debugger 15.0.5 and later supports Enterprise COBOL 6.4 and earlier versions • z/OS Debugger 14.2 and later supports Enterprise COBOL 6.3 and earlier versions • z/OS Debugger 14.1 supports Enterprise COBOL 6.2 and earlier versions • z/OS Debugger 14.0 supports Enterprise COBOL 6.1 and earlier versions • Debug Tool 13.1 supports Enterprise COBOL 5.1 and earlier versions 	

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

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

The term "2002 COBOL Standard" is used to refer to the following standard:

- *INCITS/ISO/IEC 1989-2002, Information technology - Programming languages - COBOL*

The term "2014 COBOL Standard" is used to refer to the following standard:

- *INCITS/ISO/IEC 1989:2014, Information technology - Programming languages, their environments and system software interfaces - Programming language COBOL*

The ISO standards are identical to the American National standards.

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

Comparison of commonly used terms

To better understand the terms used throughout the IBM z/OS Language Environment and IBM Enterprise COBOL for z/OS information, and to understand which terms are meant to be equivalent, see the following table.

Language Environment term	Enterprise COBOL equivalent
Aggregate	Group item

Language Environment term	Enterprise COBOL equivalent
Array	A table created using the OCCURS clause
Array element	Table element
Enclave	Run unit
External data	WORKING-STORAGE data defined using the 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

How to read syntax diagrams

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

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

The **▶▶**— symbol indicates the beginning of a syntax diagram.

The —**▶** symbol indicates that the syntax diagram is continued on the next line.

The **▶**— symbol indicates that the syntax diagram is continued from the previous line.

The —**▶◀** symbol indicates the end of a syntax diagram.

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

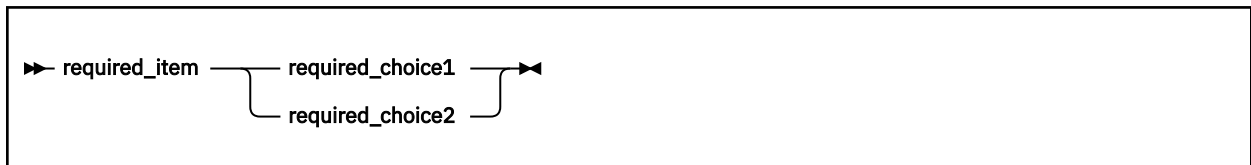
- Required items appear on the horizontal line (the main path):



- Optional items appear below the main path:



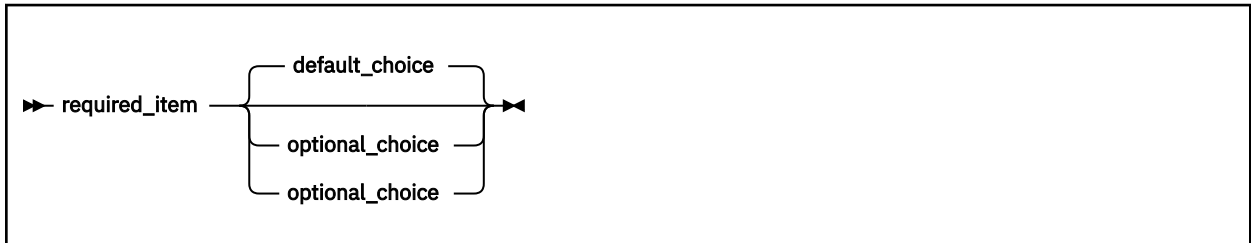
- If you can choose from two or more items, they appear vertically, in a stack. If you must choose one of the items, one item of the stack appears on the main path:



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



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



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



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



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

How examples are shown

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

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

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

If you copy and paste examples from the PDF format documentation, make sure that the spaces in the examples (if any) are in place; you might need to manually add some missing spaces to ensure that COBOL source text aligns to the required columns per the COBOL reference format in the *Enterprise COBOL for z/OS Language Reference*. Alternatively, you can copy and paste examples from the HTML format documentation and the spaces should be already in place.

Additional documentation and support

IBM Enterprise COBOL for z/OS provides Portable Document Format (PDF) versions of the entire library for this version and for previous versions on the library page at <https://www.ibm.com/support/pages/enterprise-cobol-zos-documentation-library>. These documents are also available in Japanese.

Support information is also available at <https://www.ibm.com/support/pages/node/6560933>.

Summary of changes

This topic lists the major changes that have been made to this document since Enterprise COBOL for z/OS 6.4. The changes that are described in this information have an associated cross-reference for your convenience. The latest technical changes are marked within >| and |< in the HTML version, or marked by vertical bars (|) in the left margin in the PDF version.

For a complete list of new and improved features in Enterprise COBOL for z/OS 6.4 and COBOL 6.4 with PTFs installed, see *What is new in Enterprise COBOL for z/OS 6.4 and COBOL 6.4 with PTFs installed* in the *Enterprise COBOL for z/OS What's New*.

Enterprise COBOL for z/OS 6.4 with PTFs installed

- PH48453: New sample files demonstrating a COBOL/Java™ interoperable application and how to build it are provided in the demo subdirectory of the COBOL install directory in the z/OS UNIX file system. ([“Example: COBPROD application - building and running” on page 309](#))
 - PH48667: A problem is fixed for using figurative constant HIGH-VALUES with fixed byte-length UTF-8 data items of a length not a multiple of 4 bytes. ([“Using UTF-8-character figurative constants” on page 147](#))
 - PH49715: A new cjbuild command reference section is added to address various usability and stability issues relating to the non-OO Java/COBOL interoperability. ([“cjbuild command reference ” on page 321](#))
 - PH50295: The NUMCHECK compiler option is updated to avoid generating runtime checking code of zoned-decimal senders in MOVE statements when the receiver is an alphanumeric data item and NUMCHECK(ZON(LAX)) is in effect. ([“NUMCHECK” on page 378](#))
 - PH50296: CONDCOMP: The new CONDCOMP compiler option is introduced to control how conditional code will be displayed in the listing. ([“CONDCOMP” on page 346](#))
 - PH51752: A new sample JCL is provided to demonstrate how a non-OO COBOL/Java interoperable application can be built and run entirely using JCL. ([“Sample JCL for building and running the COBPROD application” on page 314](#))
 - PH53631: Enhanced the ON EXCEPTION phrase support to deal with exceptions in the non-OO COBOL/Java interoperability framework. (Chapter 45, [“Handling errors in Java-interoperable COBOL applications,” on page 737](#))
 - PH56036 and PH56037: An optional alternate logic path is introduced for VSAM files that use the ACCESS IS DYNAMIC mode. The alternate logic path uses a direct read-by-key request instead of a point to a record by key. ([“Specifying access modes for VSAM files” on page 194](#))
 - PH56142: When compiling under z/OS UNIX, you can use the cob2 -M option to generate a make dependency file, file.u. This file contains entries for each copybook file that resides in a z/OS UNIX file system and is referenced in your COBOL source file. ([“cob2 syntax and options” on page 287](#))
 - PH57297: You can use UTF-8 (PIC U) data items as the arguments to the STRING and UNSTRING statements. ([“COBOL statements and UTF-8 data items” on page 145](#))
- Note:** COBOL Runtime LE APAR PH57264 (for AMODE 31) or APAR PH57265 (for AMODE 64) must also be applied on all systems where programs that make use of this new feature are linked or run.
- PH57397: You can use function prototypes to invoke COBOL user-defined functions. This is part of the 2014 COBOL Standard. (Chapter 33, [“Using function prototypes,” on page 589](#))

- PH57400: You can use dynamic-length and UTF-8 (PIC U) data items as the arguments to the JSON GENERATE and JSON PARSE statements. ([Chapter 35, “Processing JSON input ,” on page 597](#) and [Chapter 36, “Producing JSON output ,” on page 607](#))

Note: COBOL Runtime LE APAR PH57152 must also be applied on all systems where programs that make use of this new feature are linked or run.

- PH59733: You can generate and parse JSON null values by using the JSON GENERATE and JSON PARSE statements. ([“Generating JSON null values” on page 609](#) and [“Handling JSON null values” on page 602](#))

Enterprise COBOL for z/OS 6.4

Compiler option changes

- The following compiler options are new:
 - SMARTBIN: Use SMARTBIN to instruct the compiler to generate modules containing additional binary metadata that enables them to be optimized by IBM Automatic Binary Optimizer (ABO) for z/OS 2.2. ([“SMARTBIN” on page 396](#))
 - JAVAIOPI: Use JAVAIOPI to control the behavior of COBOL programs that interoperate with Java through the JAVA-CALLABLE or JAVA-SHAREABLE directives or by calling Java static methods using the CALL statement. ([“JAVAIOPI” on page 369](#))
- The following compiler options are modified:
 - ARCH: ARCH(8) and ARCH(9) are no longer accepted. A new higher level of ARCH(14) is accepted. ARCH(10) is the default. ([“ARCH” on page 337](#))
 - RULES: If there are multiple RULES specifications for a compilation, the suboptions are additive, which means they are accumulated. ([“RULES” on page 393](#))
 - TUNE: TUNE(8) and TUNE(9) are no longer accepted. A new higher level of TUNE(14) is accepted. TUNE(10) is the default if ARCH is not specified. ([“TUNE” on page 411](#))

Improved COBOL/Java interoperability

You can easily extend your COBOL applications with Java using a more comprehensive and flexible form without the need to write object-oriented (OO) COBOL programs. ([Part 7, “COBOL/Java interoperability outside of the object-oriented \(OO\) COBOL framework,” on page 727](#) and [Chapter 17, “Compiling, linking, and running non-OO COBOL applications that interoperate with Java,” on page 301](#))

Interoperability between AMODE 31 (31-bit) and AMODE 64 (64-bit) COBOL programs

AMODE 64 (64-bit) COBOL applications can interoperate with your existing AMODE 31 (31-bit) COBOL applications. Dynamic call is supported in a mixed AMODE 31/AMODE 64 environment. ([“COBOL applications with mixed AMODE 31 and AMODE 64 programs” on page 528](#))

User-defined functions support

You can write your own function definitions and invoke them by using a reference to a function identifier. This is part of the 2002 COBOL Standard. ([Chapter 32, “Using user-defined functions,” on page 585](#))

Debugging enhancement

You can use ddname IGZPROUT at the run step of your JCL to generate a report of all dynamically called programs that are compiled with Enterprise COBOL 5 or later. ([“Generating a report of dynamically called COBOL programs” on page 480](#))

How to send your comments

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

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

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

Part 1. Coding your program

Chapter 1. Structuring your program

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

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

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

Related tasks

[“Identifying a program” on page 3](#)

[“Describing the computing environment” on page 5](#)

[“Describing the data” on page 11](#)

[“Processing the data” on page 17](#)

[“Defining a class” on page 670](#)

[“Defining a class instance method” on page 674](#)

[“Structuring OO applications” on page 703](#)

Identifying a program

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

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

```
IDENTIFICATION DIVISION.  
Program-ID.      Helloprog.  
Author.         A. Programmer.  
Installation.   Computing Laboratories.  
Date-Written.   05/31/2022.  
Date-Compiled.  05/31/2022.
```

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

- Other programs use that name to call your program.
- The name appears in the header on each page, except the first, of the program listing that is generated when you compile the program.
- If you use the NAME compiler option, the name is placed on the NAME binder (linkage-editor) control statement to identify the object module that the compilation creates.

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

- AFB
- AFH
- CBC
- CEE
- CEH
- CEL
- CEQ
- CEU

- DFH
- DSN
- EDC
- FOR
- IBM
- IFY
- IGY
- IGZ
- ILB

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

Related tasks

[“Changing the header of a source listing” on page 5](#)

[“Identifying a program as recursive” on page 4](#)

[“Marking a program as callable by containing programs” on page 4](#)

[“Setting a program to an initial state” on page 4](#)

Related references

Compiler limits (*Enterprise COBOL for z/OS Language Reference*)

Conventions for program-names (*Enterprise COBOL for z/OS Language Reference*)

Identifying a program as recursive

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

You can code RECURSIVE only on the outermost program of a compilation unit. Neither nested subprograms nor programs that contain nested subprograms can be recursive. You must code RECURSIVE for programs that you compile with the THREAD option.

Refer to the [“Example: storage sections” on page 14](#), which shows that a recursive program uses both WORKING-STORAGE and LOCAL-STORAGE.

Related tasks

[“Sharing data in recursive or multithreaded programs” on page 16](#)

[“Making recursive calls” on page 545](#)

Marking a program as callable by containing programs

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

Only contained programs can have the COMMON attribute.

Related concepts

[“Nested programs” on page 543](#)

Setting a program to an initial state

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

When a program is set to its initial state:

- Data items that have VALUE clauses are set to the specified values.

- Changed GO TO statements and PERFORM statements are in their initial states.
- Non-EXTERNAL files are closed.

Related tasks

[“Ending and reentering main programs or subprograms” on page 534](#)

[“Making static calls” on page 536](#)

[“Making dynamic calls” on page 536](#)

Related references

[“INITIAL” on page 364](#)

Changing the header of a source listing

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

The following example shows these five elements:

```
PP 5655-EC6 IBM Enterprise COBOL for z/OS 6.4.0 PXXXXXX   Date 05/31/2022   Time 15:05:19
Page      1
```

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

Note: If you are using the IBM Enterprise COBOL Value Unit Edition for z/OS product, the header on the first page of a source listing is the same as that for IBM Enterprise COBOL for z/OS. The product number will show 5655-EC6 rather than 5697-V61. The 5697-V61 product number for the Value Unit Edition product is only significant for product ordering purposes and product registration at installation time.

Related references

TITLE statement (*Enterprise COBOL for z/OS Language Reference*)

Describing the computing environment

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

Use the CONFIGURATION SECTION to specify the following items:

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

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

- Identify and describe the characteristics of the files in the program.
- Associate your files with the external QSAM, VSAM, or z/OS UNIX file system data sets where they physically reside.

The terms *file* in COBOL terminology and *data set* in operating-system terminology have essentially the same meaning and are used interchangeably in this information.

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. Do not code file definitions in your COBOL programs that will run under CICS. 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” on page 6](#)

Related tasks

[“Specifying the collating sequence” on page 7](#)

[“Defining symbolic characters” on page 8](#)

[“Defining a user-defined class” on page 8](#)

[“Defining files to the operating system” on page 8](#)

Related references

Sections and paragraphs (*Enterprise COBOL for z/OS Language Reference*)

Example: FILE-CONTROL entries

The following table shows example 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 ¹ ASSIGN TO UPDPRINT ² ORGANIZATION IS SEQUENTIAL ³ ACCESS IS SEQUENTIAL. ⁴	SELECT COMMUTER-FILE ¹ ASSIGN TO COMMUTER ² ORGANIZATION IS INDEXED ³ ACCESS IS RANDOM ⁴ RECORD KEY IS COMMUTER-KEY ⁵ FILE STATUS IS ⁵ COMMUTER-FILE- STATUS COMMUTER-VSAM- STATUS.	SELECT PRINTFILE ¹ ASSIGN TO UPDPRINT ² ORGANIZATION IS LINE SEQUENTIAL ³ ACCESS IS SEQUENTIAL. ⁴

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 9, “Processing QSAM files,” on page 167](#)

[Chapter 10, “Processing VSAM files,” on page 189](#)

[Chapter 11, “Processing line-sequential files,” on page 215](#)

[“Describing the computing environment” on page 5](#)

Specifying the collating sequence

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

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

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

[“Example: specifying the collating sequence” on page 7](#)

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

- EBCDIC: references the collating sequence associated with the EBCDIC character set
- NATIVE: references the same collating sequence as EBCDIC
- STANDARD-1: references the collating sequence associated with the ASCII character set defined by *ANSI INCITS X3.4, Coded Character Sets - 7-bit American National Standard Code for Information Interchange (7-bit ASCII)*
- STANDARD-2: references the collating sequence associated with the character set defined by *ISO/IEC 646 -- Information technology -- ISO 7-bit coded character set for information interchange, International Reference Version*
- An alteration of the EBCDIC sequence that you define in the SPECIAL - NAMES paragraph

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

Related tasks

[“Choosing alternate collating sequences” on page 231](#)

[“Comparing national \(UTF-16\) data” on page 140](#)

Example: specifying the collating sequence

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

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

```
IDENTIFICATION DIVISION.  
.  
ENVIRONMENT DIVISION.  
CONFIGURATION SECTION.  
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 7](#)

Defining symbolic characters

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

For example, to give a name to the 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 that you list in the clause.

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

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

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

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.

Depending on the operating system, this system data definition can take any of the following forms:

- DD statement for MVS JCL.
- ALLOCATE command under TSO.
- Environment variable for z/OS or z/OS UNIX. The contents can define either an MVS data set or a file in the z/OS UNIX file system.

The following examples show 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 DSN=MY.OUT171,UNIT=SYSDA,SPACE=(TRK,(50,5))  
/*
```

- Environment variable (export command):

```
export OUTFILE=(1)DSN(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 *assignment-name* in the ASSIGN clause points to the *ddname* OUTFILE in the DD statement or the environment variable OUTFILE in the export command:

- //OUTFILE DD DSN=OUT171 . . ., or
- export OUTFILE= . . .

(2)

When you specify a file *file-name* in a FILE-CONTROL entry, you must describe the file in an FD entry:

```
SELECT CARPOOL
FD CARPOOL
```

Related tasks

[“Optimizing buffer and device space” on page 10](#)

Related references

[“FILE SECTION entries” on page 12](#)

FILE SECTION (*Enterprise COBOL for z/OS Language Reference*)

Varying the input or output file at run time

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

Changing a file-name within a COBOL program would require changing the input statements and output statements and recompiling the program. Alternatively, you can change the DSN value in the DD statement or the DSN or PATH value in the export command to use a different file at run time.

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

The name that you use in the *assignment-name* of the ASSIGN clause must be the same as the *ddname* in the DD statement or the environment variable in the export command.

The *file-name* that you use in the SELECT clause (such as SELECT MAINFILE) 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 output is directed to SYSOUT, do not use SYSOUT as the *ddname* or environment variable name in the SELECT clause for a file.

Example: using different input files

This example shows that you use the same COBOL program to access different files by coding a DD statement or an `export` command before the program runs.

Consider a COBOL program that contains the following `SELECT` clause:

```
SELECT MAINFILE ASSIGN TO DA-3330-S-MAINA
```

Assume the three possible input files are MAIN1, MAIN2, and MAIN3. Before running the program, 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:

```
//MAINA DD DSN=MY.MAIN1, . . .  
export MAINA=DSN(MY.MAIN1), . . .  
  
//MAINA DD DSN=MY.MAIN2, . . .  
export MAINA=DSN(MY.MAIN2), . . .  
  
//MAINA DD DSN=MY.MAIN3, . . .  
export MAINA=DSN(MY.MAIN3), . . .
```

Any reference in the program to MAIN will therefore be a reference to the file that is currently assigned to the ddname or environment-variable name MAINA.

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

Optimizing buffer and device space

Use the `APPLY WRITE-ONLY` clause to make optimum use of buffer and device space when you create 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.

There is a record area allocated for file processing. However, due to buffer and device space optimization, the record area may or may not contain the actual record data for processing by the program:

- The file processing occurs in the record area and the data in the record area can be accessed by the program if one of the following is true:
 - The file is `EXTERNAL`
 - The file is contained in a "`SAME RECORD AREA`" statement in the `I-O-CONTROL` section
 - The program is compiled with the `AWO` compiler option
- The file processing typically occurs in a buffer and the data in the record area is not guaranteed to be accessible by the program (storage undefined) if all of the following is true:
 - The file is not `EXTERNAL`
 - The file is not contained in a "`SAME RECORD AREA`" statement in the `I-O-CONTROL` section
 - The program is compiled with the `NOAWO` compiler option

Therefore, programs should not reference data directly in the file's record area. Instead, when a file record is read in, it should be read into a data item defined in `WORKING-STORAGE` or if read into the

record area, should be immediately moved to a data item in WORKING-STORAGE for use by the program. For example:

```
FD MY-INPUT-FILE
  LABEL RECORDS ARE STANDARD
  RECORDING MODE IS V
  BLOCK CONTAINS 0 RECORDS
  DATA RECORD IS IN-RECORD-AREA.
01 IN-RECORD-AREA      PIC X(100).

WORKING-STORAGE.
01 WS-IN-RECORD-AREA  PIC X(100).

PROCEDURE DIVISION.
...
READ MY-INPUT-FILE INTO WS-IN-RECORD-AREA.
```

Related references

[“AWO” on page 340](#)

Describing the data

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

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

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

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

Related concepts

[“Comparison of WORKING-STORAGE and LOCAL-STORAGE” on page 14](#)

Related tasks

[“Using data in input and output operations” on page 11](#)

[“Using data from another program” on page 16](#)

Related references

Compiler limits (*Enterprise COBOL for z/OS Language Reference*)

Using data in input and output operations

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

Provide the following information about the data:

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

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

- In the record description that follows the FD entry, describe the 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 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 z/OS data management facilities (including a DD file job-control statement).
- 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 in bytes, 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 543](#)

Related tasks

[“Sharing files between programs \(external files\)” on page 561](#)

Related references

[“FILE SECTION entries” on page 12](#)

FILE SECTION entries

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

<i>Table 2. FILE SECTION entries</i>		
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> .

Table 2. **FILE SECTION entries** (continued)

Clause	To define	Notes
BLOCK CONTAINS	Size of physical records	<p>If the CHARACTERS phrase is specified, size indicates the number of bytes in a record regardless of the USAGE of the data items in the record.</p> <p>QSAM: If provided, must match information on JCL or data-set label. If specified as BLOCK CONTAINS 0, or not provided, the system determines the optimal block size for you.</p> <p>Line sequential: Can be specified to control buffering for WRITE statements.</p> <p>VSAM: Syntax-checked, but has no effect on execution.</p>
RECORD CONTAINS <i>n</i>	Size of logical records (fixed length)	Integer size indicates the number of bytes in a record regardless of the USAGE of the data items in the record. If the clause is provided, it 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)	Integer size or sizes, if specified, indicate the number of bytes in a record regardless of the USAGE of the data items in the record. If the clause is provided, it must match information on JCL or data-set label; compiler checks that record descriptions match.
RECORD CONTAINS <i>n</i> TO <i>m</i>	Size of logical records (variable length)	The integer sizes indicate the number of bytes in a record regardless of the USAGE of the data items in the record. If the clause is provided, it must match information on JCL or data-set label; compiler checks that record descriptions match.
LABEL RECORDS	Labels for QSAM files	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
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, COBOL for OS/390® & VM, or IBM Enterprise COBOL for z/OS.</p>

Table 2. FILE SECTION entries (continued)		
Clause	To define	Notes
RECORDING MODE	Physical record description	QSAM only

Related references

FILE SECTION (*Enterprise COBOL for z/OS Language Reference*)

Comparison of WORKING-STORAGE and LOCAL-STORAGE

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

WORKING-STORAGE for programs is allocated when the run unit is started.

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

- A program with INITIAL specified in the PROGRAM-ID paragraph
In this case, WORKING-STORAGE data items are reinitialized each time that the program is entered.
- A subprogram that is dynamically called and then canceled
In this case, WORKING-STORAGE data items are reinitialized on the first reentry into the program following the CANCEL.

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

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

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

Threading: Each invocation of a program that runs simultaneously on multiple threads shares access to a single copy of WORKING-STORAGE data. Each invocation has a separate copy of LOCAL-STORAGE data.

[“Example: storage sections” on page 14](#)

Related tasks

[“Ending and reentering main programs or subprograms” on page 534](#)

[Chapter 31, “Preparing COBOL programs for multithreading,” on page 577](#)

[“WORKING-STORAGE SECTION for defining class instance data” on page 673](#)

Related references

WORKING-STORAGE SECTION (*Enterprise COBOL for z/OS Language Reference*)

LOCAL-STORAGE SECTION (*Enterprise COBOL for z/OS Language Reference*)

Example: storage sections

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

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

```

01 num pic 9(4).
PROCEDURE DIVISION.
  move numb to num.

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

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

```

The program produces the following output:

```

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

```

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

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

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

Related concepts

[“Comparison of WORKING-STORAGE and LOCAL-STORAGE” on page 14](#)

Using data from another program

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

Related tasks

[“Sharing data in separately compiled programs” on page 16](#)

[“Sharing data in nested programs” on page 16](#)

[“Sharing data in recursive or multithreaded programs” on page 16](#)

[“Passing data” on page 551](#)

Sharing data in separately compiled programs

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

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

Related tasks

[“Passing data” on page 551](#)

[“Coding the LINKAGE SECTION” on page 555](#)

Sharing data in nested programs

Some applications consist of nested programs, that is, programs that are contained in other programs. Level-01 data items can include the GLOBAL clause. The GLOBAL clause specifies that a data-name is available to every program contained within the program that defines it, as long as the contained program does not itself have a definition for that name.

A nested program cannot access data items in a sibling program (one at the same nesting level in the same containing program), but they can both refer to data items defined with the EXTERNAL clause or data items in the containing program if they are defined with the GLOBAL clause.

Related concepts

[“Nested programs” on page 543](#)

Sharing data in recursive or multithreaded programs

If your program has the RECURSIVE attribute or is compiled with the THREAD compiler option, data that is defined in the LINKAGE SECTION is not accessible on subsequent invocations of the program.

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

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

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

Related concepts

[“Multithreading” on page 577](#)

Related tasks

[“Making recursive calls” on page 545](#)

[“Processing files with multithreading” on page 580](#)

Related references

[“THREAD” on page 408](#)

SET statement (*Enterprise COBOL for z/OS Language Reference*)

Processing the data

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

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

```
PROCEDURE DIVISION.
```

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

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

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

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

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

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

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

```
PROCEDURE DIVISION RETURNING dataname2
```

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

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

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

Related concepts

[“How logic is divided in the PROCEDURE DIVISION” on page 18](#)

Related tasks

[“Coding the LINKAGE SECTION” on page 555](#)

[“Coding the PROCEDURE DIVISION for passing arguments” on page 556](#)

[“Using PROCEDURE DIVISION RETURNING . . .” on page 560](#)

[“Eliminating repetitive coding” on page 772](#)

Related references

The procedure division header (*Enterprise COBOL for z/OS Language Reference*)

The USING phrase (*Enterprise COBOL for z/OS Language Reference*)

CALL statement (*Enterprise COBOL for z/OS Language Reference*)

How logic is divided in the PROCEDURE DIVISION

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

Section

Logical subdivision of your processing logic.

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

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

Paragraph

Subdivision of a section, procedure, or program.

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

A paragraph can be the subject of a statement.

Sentence

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

Statement

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

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

Phrase

A subdivision of a statement.

Related concepts

[“Compiler-directing statements” on page 19](#)

[“Scope terminators” on page 20](#)

[“Imperative statements” on page 18](#)

[“Conditional statements” on page 18](#)

[“Declaratives” on page 21](#)

Related references

PROCEDURE DIVISION structure (*Enterprise COBOL for z/OS Language Reference*)

Imperative statements

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

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

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

Related concepts

[“Conditional statements” on page 18](#)

[“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 phrase instead of a period to end the scope of an IF statement within a nested IF.

- To code a conditional statement where the COBOL syntax calls for an imperative statement

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

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

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

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

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

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

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

Related concepts

[“Imperative statements” on page 18](#)

[“Scope terminators” on page 20](#)

Related tasks

[“Selecting program actions” on page 91](#)

Related references

Conditional statements (*Enterprise COBOL for z/OS Language Reference*)

Compiler-directing statements

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

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

Related references

[Chapter 20, “Compiler-directing statements,” on page 429](#)

Compiler-directing statements (*Enterprise COBOL for z/OS Language Reference*)

Scope terminators

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

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

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

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

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

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

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

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

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

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

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


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

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

Related concepts

[“Conditional statements” on page 18](#)

[“Imperative statements” on page 18](#)

Declaratives

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

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

Related tasks

[“Finding and handling input-output errors” on page 435](#)

Related references

Declaratives (*Enterprise COBOL for z/OS Language Reference*)

Chapter 2. Using data

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

Related concepts

[“Storage and its addressability” on page 39](#)

Related tasks

[“Using variables, structures, literals, and constants” on page 23](#)

[“Assigning values to data items” on page 26](#)

[“Displaying values on a screen or in a file \(DISPLAY\)” on page 35](#)

[“Using intrinsic functions \(built-in functions\)” on page 38](#)

[“Using tables \(arrays\) and pointers” on page 39](#)

[Chapter 7, “Processing data in an international environment,” on page 123](#)

Using variables, structures, literals, and constants

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

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

Related tasks

[“Using variables” on page 23](#)

[“Using data items and group items” on page 24](#)

[“Using literals” on page 25](#)

[“Using constants” on page 26](#)

[“Using figurative constants” on page 26](#)

Related references

Classes and categories of data (*Enterprise COBOL for z/OS Language Reference*)

Using variables

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

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

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

You could instead define the customer names above as national or UTF-8 data items by specifying their `PICTURE` and `USAGE` clauses as `Pic N(20) USAGE NATIONAL` or `Pic U(20) USAGE UTF-8`, respectively. National data items are represented in Unicode UTF-16, in which most characters are

represented in 2 bytes of storage. UTF-8 data items are represented in Unicode UTF-8, which is a variable-width encoding of Unicode, using one to four bytes of storage to represent each character.

Related concepts

[“Unicode and the encoding of language characters” on page 124](#)

Related tasks

[“Using national data \(Unicode\) in COBOL” on page 125](#)

[“Using UTF-8 data \(Unicode\) in COBOL” on page 142](#)

Related references

[“NSYMBOL” on page 377](#)

[“Storage of character data” on page 139](#)

PICTURE clause (*Enterprise COBOL for z/OS Language Reference*)

Using data items and group items

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

A record can be either an elementary item or a group item. A group item can be either an *alphanumeric group item*, a *national group item*, or a *UTF-8 group item*.

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

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

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

```
Data Division.
File Section.
FD Customer-File
   Record Contains 90 Characters.
01 Customer-Record      Group-Usage National.
   05 Customer-Name.
      10 Last-Name       Pic n(17).
      10 Filler          Pic n.
```

```

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

```

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

Related concepts

[“Unicode and the encoding of language characters” on page 124](#)

[“National groups” on page 133](#)

Related tasks

[“Using national data \(Unicode\) in COBOL” on page 125](#)

[“Using national groups” on page 137](#)

Related references

[“FILE SECTION entries” on page 12](#)

[“Storage of character data” on page 139](#)

Classes and categories of group items (*Enterprise COBOL for z/OS Language Reference*)

PICTURE clause (*Enterprise COBOL for z/OS Language Reference*)

MOVE statement (*Enterprise COBOL for z/OS Language Reference*)

USAGE clause (*Enterprise COBOL for z/OS Language Reference*)

Using literals

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

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

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

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

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

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

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

Related concepts

[“Unicode and the encoding of language characters” on page 124](#)

Related tasks

[“Using national literals” on page 126](#)

[“Using DBCS literals” on page 159](#)

Related references

[“NSYMBOL” on page 377](#)

Literals (*Enterprise COBOL for z/OS Language Reference*)

Using constants

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

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

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

Related tasks

[“Using national data \(Unicode\) in COBOL” on page 125](#)

[“Coding for use of DBCS support” on page 158](#)

Using figurative constants

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

For example:

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

Related tasks

[“Using national-character figurative constants” on page 132](#)

[“Coding for use of DBCS support” on page 158](#)

[“Using UTF-8-character figurative constants” on page 147](#)

Related references

Figurative constants (*Enterprise COBOL for z/OS 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.

<i>Table 3. Assignment to data items in a program</i>	
What you want to do	How to do it
Assign values to a data item or large data area.	Use one of these ways: <ul style="list-style-type: none"> • INITIALIZE statement • MOVE statement • STRING or UNSTRING statement • VALUE clause (to set data items to the values you want them to have when the program is in initial state)
Assign the results of arithmetic.	Use COMPUTE, ADD, SUBTRACT, MULTIPLY, or DIVIDE statements.
Examine or replace characters or groups of characters in a data item.	Use the INSPECT statement.
Receive values from a file.	Use the READ (or READ INTO) statement.
Receive values from a system input device or a file.	Use the ACCEPT statement.
Establish a constant.	Use the VALUE clause in the definition of the data item, and do not use the data item as a receiver. Such an item is in effect a constant even though the compiler does not enforce read-only constants.
One of these actions: <ul style="list-style-type: none"> • Place a value associated with a table element in an index. • Set the status of an external switch to ON or OFF. • Move data to a condition-name to make the condition true. • Set a POINTER, PROCEDURE-POINTER, or FUNCTION-POINTER data item to an address. • Associate an OBJECT REFERENCE data item with an object instance. 	Use the SET statement.

[“Examples: initializing data items” on page 27](#)

Related tasks

[“Initializing a structure \(INITIALIZE\)” on page 31](#)

[“Assigning values to elementary data items \(MOVE\)” on page 32](#)

[“Assigning values to group data items \(MOVE\)” on page 33](#)

[“Assigning input from a screen or file \(ACCEPT\)” on page 35](#)

[“Joining data items \(STRING\)” on page 103](#)

[“Splitting data items \(UNSTRING\)” on page 105](#)

[“Assigning arithmetic results \(MOVE or COMPUTE\)” on page 34](#)

[“Tallying and replacing data items \(INSPECT\)” on page 112](#)

[Chapter 7, “Processing data in an international environment,” on page 123](#)

Examples: initializing data items

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

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

Initializing a data item to blanks or zeros:

```
INITIALIZE identifier-1
```

identifier-1 PICTURE	identifier-1 before	identifier-1 after
9(5)	12345	00000
X(5)	AB123	bbbb ¹
N(3)	004100420031 ²	002000200020 ³
99XX9	12AB3	bbbb ¹
XXBX/XX	ABbC/DE	bbbb/bb ¹
*+99.9CR	1234.5CR	*+00.0bb ¹
A(5)	ABCDE	bbbb ¹
+99.99E+99	+12.34E+02	+00.00E+00

1. The symbol *b* represents a blank space.
 2. Hexadecimal representation of the national (UTF-16) characters 'AB1'. The example assumes that *identifier-1* has Usage National.
 3. Hexadecimal representation of the national (UTF-16) characters ' ' (three blank spaces). Note that if *identifier-1* were not defined as Usage National, and if NSYMBOL (DBCS) were in effect, INITIALIZE would instead store DBCS spaces ('4040') into *identifier-1*.

Initializing an alphanumeric data item:

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

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

Initializing an alphanumeric right-justified data item:

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

ALPHABETIC-1	ANJUST before	ANJUST after
ABCD	bbbbbb ¹	bbbbABCD ¹

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

Initializing an alphanumeric-edited data item:

```

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

```

ALPHANUM-EDIT-3	ALPHANUM-EDIT-1 before	ALPHANUM-EDIT-1 after
M/bb ¹	AbbC/DEF ¹	M/bb/bbb ¹
1. The symbol <i>b</i> represents a blank space.		

Initializing a national data item:

```

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

```

NATIONAL-3	NATIONAL-1 before first INITIALIZE	NATIONAL-1 after first INITIALIZE	NATIONAL-1 after second INITIALIZE
00430044 ¹	00410042 ²	00430044 ¹	00410042 ²
1. Hexadecimal representation of the national characters 'CD'			
2. Hexadecimal representation of the national characters 'AB'			

Initializing a national-edited data item:

```

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

```

NATL-3	NATL-EDIT-1 before	NATL-EDIT-1 after
003400350036 ¹	003100320033 ²	003000340035 ³
1. Hexadecimal representation of the national characters '456'		
2. Hexadecimal representation of the national characters '123'		
3. Hexadecimal representation of the national characters '045'		

Initializing a numeric (zoned decimal) data item:

```

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

```

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

Initializing a numeric (national decimal) data item:

```

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

```

NUM-INT-BIN-3	NAT-DEC-1 before	NAT-DEC-1 after
12	003900380037 ¹	003000310032 ²

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

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

```

01 NUM-EDIT-DISP-1  PIC $$$9V  VALUE "$127".
01 NUM-DISP-3       PIC 999V   VALUE 12.
. . .
  INITIALIZE NUM-EDIT-DISP-1
    REPLACING NUMERIC-EDITED DATA BY NUM-DISP-3

```

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

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

```

01 NUM-EDIT-NATL-1  PIC $$$9V  NATIONAL VALUE N"$127".
01 NUM-NATL-3       PIC 999V   NATIONAL VALUE 12.
. . .
  INITIALIZE NUM-EDIT-NATL-1
    REPLACING NUMERIC-EDITED DATA BY NUM-NATL-3

```

NUM-NATL-3	NUM-EDIT-NATL-1 before	NUM-EDIT-NATL-1 after
003000310032 ¹	0024003100320037 ²	0024002000310032 ³

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

Related tasks

- “[Initializing a structure \(INITIALIZE\)](#)” on page 31
- “[Initializing a table \(INITIALIZE\)](#)” on page 73
- “[Defining numeric data](#)” on page 43

Related references

- “[NSYMBOL](#)” on page 377

Initializing a structure (INITIALIZE)

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

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

```

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

Record	TRANSACTION-OUT before	TRANSACTION-OUT after
1	R0013830002400000000000000000	b000000000000000000000000000 ¹
2	R0013900004800000000000000000	b000000000000000000000000000 ¹
3	S0014100001200000000000000000	b000000000000000000000000000 ¹
4	C0013830000000000425000000000	b000000000000000000000000000 ¹
5	C0020100000000000000100000000	b000000000000000000000000000 ¹

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

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

```

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

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

- TRANSACTION-CODE contains NX"0020" (a national space).
- Each of the remaining 27 national character positions of TRANSACTION-OUT contains NX"0030" (a national-decimal zero).

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

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

- ZERO is the implied sending item for numeric and numeric-edited receiving items.

Related concepts

[“National groups” on page 133](#)

Related tasks

[“Initializing a table \(INITIALIZE\)” on page 73](#)

[“Using national groups” on page 137](#)

Related references

INITIALIZE statement (*Enterprise COBOL for z/OS Language Reference*)

Assigning values to elementary data items (MOVE)

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

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

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

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

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

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

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

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

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

```
CBL CODEPAGE(00875)
. . .
01 Data-in-Unicode Pic N(100) usage national.
01 Data-in-Greek   Pic X(100).
. . .
   . . . Read Greek-file into Data-in-Greek
   . . . Move Data-in-Greek to Data-in-Unicode
```

Related concepts

[“Unicode and the encoding of language characters” on page 124](#)

Related tasks

[“Assigning values to group data items \(MOVE\)” on page 33](#)

[“Converting to or from national \(Unicode\) representation” on page 134](#)

Related references

[“CODEPAGE” on page 343](#)

Classes and categories of data (*Enterprise COBOL for z/OS Language Reference*)

MOVE statement (*Enterprise COBOL for z/OS Language Reference*)

Assigning values to group data items (MOVE)

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

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

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

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

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

. . .
MOVE CORR Group-X TO Group-N
```

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

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

Related concepts

[“Unicode and the encoding of language characters” on page 124](#)

[“National groups” on page 133](#)

Related tasks

[“Assigning values to elementary data items \(MOVE\)” on page 32](#)

[“Using national groups” on page 137](#)

[“Converting to or from national \(Unicode\) representation” on page 134](#)

Related references

Classes and categories of group items (*Enterprise COBOL for z/OS Language Reference*)

MOVE statement (*Enterprise COBOL for z/OS Language Reference*)
CORRESPONDING phrase (*Enterprise COBOL for z/OS Language Reference*)

Assigning arithmetic results (MOVE or COMPUTE)

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

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

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

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

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

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

You can assign the results of date, time, mathematical, and other calculations to data items by using Language Environment callable services. Language Environment services are available through a standard COBOL CALL statement, and the values they return are passed in the parameters of the CALL statement. For example, you can call the Language Environment service CEESIABS to find the absolute value of a data item by coding the following statement:

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

As a result of this call, data item Result is assigned the absolute value of the value in data item Arg; data item Feedback-code contains the return code that indicates whether the service completed successfully. You have to define all the data items in the DATA DIVISION using the correct descriptions according to the requirements of the particular callable service. For the example above, the data items could be defined as follows:

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

Related references

[“DIAGTRUNC” on page 352](#)

Intrinsic functions (Enterprise COBOL for z/OS Language Reference)
Language Environment Programming Reference (Callable services)

Assigning input from a screen or file (ACCEPT)

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

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

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

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

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

```
SYSIN is Names-Input
```

device can be a ddname that references a z/OS UNIX file system path. If this ddname is not defined and your program is running in the z/OS UNIX environment, stdin is the input source. If this ddname is not defined and your program is not running in the z/OS UNIX environment, the ACCEPT statement fails.

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

When you assign a value to a USAGE NATIONAL data item, input data from the console is converted from the EBCDIC code page specified in the CODEPAGE compiler option to national (Unicode UTF-16) representation. This is the only case where conversion of national data is done when you use the ACCEPT statement. Conversion is done in this case because the input is known to be coming from a screen.

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

Related concepts

[“Unicode and the encoding of language characters” on page 124](#)

Related tasks

[“Converting alphanumeric or DBCS to national \(NATIONAL-OF\)” on page 135](#)

Related references

[“CODEPAGE” on page 343](#)

ACCEPT statement (*Enterprise COBOL for z/OS Language Reference*)

SPECIAL-NAMES paragraph (*Enterprise COBOL for z/OS Language Reference*)

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

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

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

In the example above, if the content of data item *Customer-Name* is JOHNSON, then the statement displays the following message on the 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 phrase with a destination other than SYSOUT. For example, the following statement writes to the file that is specified in the SYSPUNCH DD statement:

```
Display "Hello" upon syspunch.
```

You can specify a file in the z/OS UNIX file system by using the SYSPUNCH DD statement. For example, the following definition causes DISPLAY output to be written to the 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 and to the TSO screen if you are running under TSO:

```
Display "Hello" upon console.
```

When you display the value of a USAGE NATIONAL data item to the console, the data item is converted from Unicode (UTF-16) representation to EBCDIC based on the value of the CODEPAGE option. This is the only case in which conversion of national data is done when you use the DISPLAY statement. Conversion is done in this case because the output is known to be directed to a screen.

To have a national data item be converted when you direct output to a different device, use the DISPLAY-OF intrinsic function, as in the following example:

```
01 Data-in-Unicode pic N(10) usage national.  
  . . .  
  Display function Display-of(Data-in-Unicode, 00037)
```

Related concepts

[“Unicode and the encoding of language characters” on page 124](#)

Related tasks

[“Displaying data on the system logical output device” on page 36](#)

[“Using WITH NO ADVANCING” on page 37](#)

[“Converting national to alphanumeric \(DISPLAY-OF\)” on page 135](#)

[“Coding COBOL programs to run under CICS” on page 483](#)

Related references

[“CODEPAGE” on page 343](#)

DISPLAY statement (*Enterprise COBOL for z/OS Language Reference*)

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.

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

The output is directed to the ddname that you specify in the OUTDD compiler option. You can specify a file in the z/OS UNIX file system with this ddname.

If the OUTDD ddname is not allocated and you are not running in the z/OS UNIX environment, a default DD of SYSOUT=* is allocated. If the OUTDD ddname is not allocated and you are running in the z/OS 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. The output is written as a single stream of characters without line breaks.

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

Related tasks

[“Setting and accessing environment variables” on page 514](#)

Related references

[“OUTDD” on page 386](#)

DISPLAY statement (*Enterprise COBOL for z/OS Language Reference*)

Using WITH NO ADVANCING

If you specify the WITH NO ADVANCING phrase, and 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 newline 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.

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

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

```
ABC  
CDEF  
+GHIJK  
+LMNOPQ  
RSTUVWX
```

The output that is 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 newline character is appended to the end of the stream.

Related references

DISPLAY statement (*Enterprise COBOL for z/OS Language Reference*)

Using intrinsic functions (built-in functions)

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

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

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

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

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

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

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

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

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

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

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

Related tasks

[“Processing table items using intrinsic functions” on page 87](#)

[“Converting data items \(intrinsic functions\)” on page 113](#)

[“Evaluating data items \(intrinsic functions\)” on page 117](#)

Related references

REPOSITORY paragraph (*Enterprise COBOL for z/OS Language Reference*)

Using tables (arrays) and pointers

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

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

You can perform the following operations with pointer data items:

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

Use pointer data items to:

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

Related tasks

[“Defining a table \(OCCURS\)” on page 67](#)

[“Using procedure and function pointers” on page 545](#)

Storage and its addressability

When you run COBOL programs, the programs and the data that they use reside in virtual storage. Storage that you use with COBOL can be below the 16 MB line, above the 16 MB line but below the 2 GB bar, and above the 2 GB bar. Three modes of addressing are available to address this storage: 24-bit, 31-bit, and 64-bit.

You can address storage below (but not above) the 16 MB line with 24-bit addressing. You can address storage either above or below the 16 MB line with 31-bit addressing. *Unrestricted storage* is addressable by 31-bit addressing and therefore encompasses all the storage available to your program below the 2 GB bar.

Enterprise COBOL can now directly exploit the 64-bit virtual addressing capability of z/OS with the LP(64) compiler option; however, COBOL applications running in 31-bit or 24-bit addressing mode are fully supported on 64-bit z/OS systems.

Addressing mode (AMODE) is the attribute that tells which hardware addressing mode is supported by your program: 24-bit addressing, 31-bit addressing, either 24-bit or 31-bit addressing, and 64-bit addressing. These attributes are AMODE 24, AMODE 31, AMODE ANY, and AMODE 64 respectively. The program object and the executing program each have an AMODE attribute. Enterprise COBOL 6 object programs are either AMODE MIN for cases where AMODE 24 is possible, AMODE 31, or AMODE 64. See [“Restrictions for AMODE” on page 40](#).

Residency mode (RMODE) is the attribute of a program object that identifies where in virtual storage the program will reside: below the 16 MB line, or either below or above. This attribute is RMODE 24 or RMODE ANY.

Note: Enterprise COBOL 6 does not support RMODE 64.

Enterprise COBOL uses Language Environment services to control the storage used at run time. Thus COBOL compiler options and Language Environment runtime options influence the AMODE and RMODE attributes of your program and data, alone and in combination:

DATA

Compiler option that influences the location of storage for WORKING-STORAGE data, I-O buffers, and parameter lists for programs compiled with RENT.

RMODE

Compiler option that influences the residency mode.

RENT

Compiler option to generate a reentrant program.

HEAP

Runtime option that controls storage for the runtime heap. For example, COBOL WORKING-STORAGE is allocated from heap storage when the COBOL program is compiled with the RENT option and is in one of the following cases:

- Compiled with Enterprise COBOL 4.2 or earlier releases
- Compiled with the DATA(24) compiler option
- Running in CICS
- A COBOL 5.1.1 or later program in a program object that contains only COBOL programs (except COBOL 5.1.0) and assembler. There are no Language Environment interlanguage calls within the program object and no COBOL 5.1.0 programs.
- A COBOL 5 program in a program object where the main entry point is COBOL 5. In this case, the program object can contain Language Environment interlanguage calls, with COBOL statically linking with C, C++ or PL/I. All COBOL 5 programs within such program objects (even if they are not the main entry point) have their WORKING-STORAGE allocated from heap storage.
- A COBOL 6.1 or later program

STACK

Runtime option that controls storage for the runtime stack. For example, COBOL LOCAL-STORAGE is allocated from stack storage.

ALL31

Runtime option that specifies whether an application can run entirely in AMODE 31.

Restrictions for AMODE

AMODE 24 execution is not supported in the following cases, and the applications must run in AMODE 31. This is the same set of AMODE 24 restrictions as COBOL 3 and 4.

- Programs containing XML PARSE statements
- Programs containing XML GENERATE statements
- Program objects containing COBOL bound together with C, C++, or PL/I programs, and communicating via static CALL
- Programs containing object-oriented language syntax, such as INVOKE statements, or object-oriented class definitions
- Programs compiled with any of the following compiler options:
 - DLL
 - PGMNAME(LONGUPPER)
 - PGMNAME(LONGMIXED)
- Multithreaded applications

Note: A program compiled with the THREAD option can run in AMODE 24, but only in an application that does not have multiple threads or PL/I tasks.

- Programs run from the z/OS UNIX file system

Note: An AMODE 31 driver program resident in the z/OS UNIX file system can contain a dynamic call to an AMODE 24 program module resident in an MVS PDSE.

- Programs used as COBOL compiler exit modules that are specified on the EXIT compiler option
- Language Environment enclaves that use XPLINK, including either the enclaves that contain non-COBOL programs compiled with the XPLINK compiler option, or run with the XPLINK runtime option

Note:

- To run COBOL programs with addressing mode 24, you must compile all COBOL programs with Enterprise COBOL 5.1.1, or later versions; or Enterprise COBOL 4.2 or earlier versions. If any component of a program object is compiled with Enterprise COBOL 5.1.0, the program object must run in addressing mode 31. COBOL programs that run with addressing mode 24 must be linked with the binder option RMODE (24).
- To run COBOL programs with addressing mode 64, you must compile all COBOL programs with Enterprise COBOL 6.3, or later versions, and use the LP (64) compiler option.

Settings for RMODE

The RMODE and RENT options determine the RMODE attribute of your program.

RMODE compiler option	RENT compiler option	RMODE attribute
RMODE (AUTO)	RENT	RMODE ANY
RMODE (AUTO)	NORENT	RMODE 24
RMODE (24)	RENT or NORENT	RMODE 24
RMODE (ANY)	RENT	RMODE ANY
RMODE (ANY)	NORENT	Compiler option conflict. If the NORENT option is specified, the RMODE (24) or RMODE (AUTO) compiler option must be specified.

Link-edit considerations: When the object code that COBOL generates has an attribute of RMODE 24, you must link-edit it with RMODE 24. When the object code that COBOL generates has an attribute of RMODE ANY, you can link-edit it with RMODE ANY or RMODE 24.

Storage restrictions for passing data

Do not pass parameters that are allocated in storage above the 16 MB line to AMODE 24 subprograms. Force the WORKING-STORAGE data and parameter lists below the line for programs that run in 31-bit addressing mode and pass data to programs that run in AMODE 24:

- Compile with the RENT and DATA (24) compiler options, or if the WORKING-STORAGE is on the HEAP (see previous description of the HEAP option), run them with the HEAP (, , BELOW) runtime option.
- Compile with the NORENT compiler option.

Location of data areas

For reentrant programs, the DATA compiler option, and the HEAP runtime option control whether storage for data areas such as WORKING-STORAGE SECTION and FD record areas is obtained from below the 16 MB line or from unrestricted storage. Compile programs with RENT and RMODE (ANY) or RMODE (AUTO) if they will be run with 31-bit addressing in virtual storage addresses above the 16 MB line. The DATA option does not affect programs that are compiled with NORENT.

Storage for LOCAL-STORAGE data

The location of LOCAL-STORAGE data items is controlled by the STACK runtime option and the AMODE of the program. LOCAL-STORAGE data items are acquired in unrestricted storage when the STACK (, , ANYWHERE) runtime option is in effect and the program is running in AMODE 31. Otherwise

LOCAL-STORAGE is acquired below the 16 MB line. The DATA compiler option does not influence the location of LOCAL-STORAGE data.

Storage for external data

In addition to affecting how storage is obtained for dynamic data areas (WORKING-STORAGE, FD record areas, and parameter lists), the DATA compiler option can also influence where storage for EXTERNAL data is obtained. Storage required for EXTERNAL data is obtained from unrestricted storage if the following conditions are met:

- The program is compiled with the DATA(31) and RENT compiler options.
- The HEAP(, , ANYWHERE) runtime option is in effect.
- The ALL31(ON) runtime option is in effect.

In all other cases, the storage for EXTERNAL data is obtained from below the 16 MB line. If you specify the ALL31(ON) runtime option, all the programs in the run unit must be capable of running in 31-bit addressing mode.

Storage for QSAM input-output buffers

The DATA compiler option can also influence where input-output buffers for QSAM files are obtained. See the related references below for information about allocation of buffers for QSAM files and the DATA compiler option.

Storage for ALLOCATE statement

The DATA compiler option setting influences how ALLOCATE acquires storage:

- If DATA(24) is in effect and the LOC 31 phrase of the ALLOCATE statement is not specified, ALLOCATE acquires storage from below the 16 MB line.
- If DATA(31) is in effect and the LOC 24 phrase of the ALLOCATE statement is not specified, ALLOCATE will attempt to acquire storage from above the 16 MB line.

Related concepts

[“AMODE switching” on page 538](#)
AMODE considerations for heap storage
(*Language Environment Programming Guide*)

Related tasks

[Chapter 28, “Using subprograms,” on page 533](#)
[Chapter 29, “Sharing data,” on page 551](#)

Related references

[“Allocation of buffers for QSAM files” on page 186](#)
[“Allocation of record areas for VSAM files” on page 211](#)
[“DATA” on page 349](#)
[“RENT” on page 391](#)
[“RMODE” on page 392](#)
[“Performance-related compiler options” on page 764](#)
ALLOCATE statement (*Enterprise COBOL for z/OS Language Reference*)
HEAP, STACK, ALL31 (*Language Environment Programming Reference*)
MVS Program Management: User's Guide and Reference

Chapter 3. Working with numbers and arithmetic

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

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

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

Related tasks

[“Defining numeric data” on page 43](#)

[“Displaying numeric data” on page 45](#)

[“Controlling how numeric data is stored” on page 46](#)

[“Checking for incompatible data \(numeric class test\)” on page 53](#)

[“Performing arithmetic” on page 55](#)

[“Using currency signs” on page 64](#)

Defining numeric data

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

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

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

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

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

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

Other characters of special significance that you can code are:

P

Indicates leading or trailing zeros

S

Indicates a sign, positive or negative

V

Implies a decimal point

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

```
05 Price Pic s99v99.
```

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

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

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

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

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

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

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

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

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

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

[“Examples: numeric data and internal representation” on page 49](#)

Related concepts

[“Formats for numeric data” on page 47](#)

[Appendix A, “Intermediate results and arithmetic precision,” on page 779](#)

Related tasks

[“Displaying numeric data” on page 45](#)

[“Controlling how numeric data is stored” on page 46](#)

[“Performing arithmetic” on page 55](#)

[“Defining national numeric data items” on page 132](#)

Related references

[“Sign representation of zoned and packed-decimal data” on page 53](#)

[“Storage of character data” on page 139](#)

[“ARITH” on page 339](#)

[“NUMPROC” on page 382](#)

SIGN clause (*Enterprise COBOL for z/OS Language Reference*)

Displaying numeric data

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

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

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

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

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

To display numeric or numeric-edited data items that have `USAGE NATIONAL` in EBCDIC, direct them to `CONSOLE`. For example, if `Edited-price` in the code above has `USAGE NATIONAL`, `$ 1,500.99` is displayed when you run the program if the last statement above is:

```
Display Edited-price Upon Console
```

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

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

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

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

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

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

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

[“Examples: numeric data and internal representation” on page 49](#)

Related tasks

[“Displaying values on a screen or in a file \(DISPLAY\)” on page 35](#)

[“Controlling how numeric data is stored” on page 46](#)

[“Defining numeric data” on page 43](#)

[“Performing arithmetic” on page 55](#)

[“Defining national numeric data items” on page 132](#)

[“Converting to or from national \(Unicode\) representation” on page 134](#)

Related references

`MOVE` statement (*Enterprise COBOL for z/OS Language Reference*)

`BLANK WHEN ZERO` clause (*Enterprise COBOL for z/OS Language Reference*)

Controlling how numeric data is stored

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

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

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

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

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

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

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

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

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

[“Examples: numeric data and internal representation” on page 49](#)

Related concepts

[“Formats for numeric data” on page 47](#)

[“Data format conversions” on page 51](#)

[Appendix A, “Intermediate results and arithmetic precision,” on page 779](#)

Related tasks

[“Defining numeric data” on page 43](#)

[“Displaying numeric data” on page 45](#)

[“Performing arithmetic” on page 55](#)

Related references

[“Conversions and precision” on page 52](#)

[“Sign representation of zoned and packed-decimal data” on page 53](#)

Formats for numeric data

Several formats are available for numeric data.

External decimal (DISPLAY and NATIONAL) items

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

When USAGE NATIONAL is in effect for a category numeric data item, 2 bytes of storage are required for each decimal digit. The items are stored in UTF-16 format. External decimal items that have USAGE NATIONAL must only contain valid UTF-16 digits. If they do not, the data is illegal and the behavior of the generated code is undefined. External decimal items that have USAGE NATIONAL are referred to as *national decimal* data items.

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

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

External floating-point (DISPLAY and NATIONAL) items

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

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

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

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

or a minus sign for negative numbers. If you instead code a plus sign (+), the sign appears as a plus sign for positive numbers or a minus sign for negative numbers.

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

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

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

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

If Compute-Result-N is displayed, the signs appear as described above for Compute-Result, but in national characters. To instead display Compute-Result-N in EBCDIC characters, direct it to the console:

```
Display Compute-Result-N Upon Console
```

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

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

Binary (COMP) items

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

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

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

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

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

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

PICTURE	Storage representation	Numeric values
S9(1) through S9(4)	Binary halfword (2 bytes)	-32768 through +32767
S9(5) through S9(9)	Binary fullword (4 bytes)	-2,147,483,648 through +2,147,483,647
S9(10) through S9(18)	Binary doubleword (8 bytes)	-9,223,372,036,854,775,808 through +9,223,372,036,854,775,807
9(1) through 9(4)	Binary halfword (2 bytes)	0 through 65535
9(5) through 9(9)	Binary fullword (4 bytes)	0 through 4,294,967,295
9(10) through 9(18)	Binary doubleword (8 bytes)	0 through 18,446,744,073,709,551,615

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

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

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

Packed-decimal (COMP-3) items

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

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

COMP-1 refers to short floating-point format and COMP-2 refers to long floating-point format, which occupy 4 and 8 bytes of storage, respectively. The leftmost bit contains the sign and the next 7 bits contain the exponent; the remaining 3 or 7 bytes contain the mantissa.

COMP-1 and COMP-2 data items are stored in System z® hexadecimal format.

Related concepts

[“Unicode and the encoding of language characters” on page 124](#)

[Appendix A, “Intermediate results and arithmetic precision,” on page 779](#)

Related tasks

[“Defining numeric data” on page 43](#)

[“Defining national numeric data items” on page 132](#)

Related references

[“Storage of character data” on page 139](#)

[“TRUNC” on page 409](#)

Classes and categories of data (*Enterprise COBOL for z/OS Language Reference*)

SIGN clause (*Enterprise COBOL for z/OS Language Reference*)

VALUE clause (*Enterprise COBOL for z/OS Language Reference*)

Examples: numeric data and internal representation

The following table shows the internal representation of numeric items.

Table 6. 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 9999 NATIONAL	1234	00 31 00 32 00 33 00 34
	PIC S9999 DISPLAY SIGN LEADING	+ 1234	C1 F2 F3 F4
		- 1234	D1 F2 F3 F4
	PIC S9999 DISPLAY SIGN LEADING SEPARATE	+ 1234	4E F1 F2 F3 F4
		- 1234	60 F1 F2 F3 F4
	PIC S9999 DISPLAY SIGN TRAILING SEPARATE	+ 1234	F1 F2 F3 F4 4E
		- 1234	F1 F2 F3 F4 60
	PIC S9999 NATIONAL SIGN LEADING SEPARATE	+ 1234	00 2B 00 31 00 32 00 33 00 34
		- 1234	00 2D 00 31 00 32 00 33 00 34
	PIC S9999 NATIONAL SIGN TRAILING SEPARATE	+ 1234	00 31 00 32 00 33 00 34 00 2B
		- 1234	00 31 00 32 00 33 00 34 00 2D
	Binary	PIC S9999 BINARY PIC S9999 COMP PIC S9999 COMP-4	+ 1234
- 1234			FB 2E
PIC S9999 COMP-5		+ 12345 ¹	30 39
		- 12345 ¹	CF C7
PIC 9999 BINARY PIC 9999 COMP PIC 9999 COMP-4		1234	04 D2
PIC 9999 COMP-5		60000 ¹	EA 60

Table 6. *Internal representation of numeric items (continued)*

Numeric type	PICTURE and USAGE and optional SIGN clause	Value	Internal representation
Internal decimal	PIC S9999 PACKED-DECIMAL PIC S9999 COMP-3	+ 1234	01 23 4C
		- 1234	01 23 4D
	PIC 9999 PACKED-DECIMAL PIC 9999 COMP-3	1234	01 23 4F
Internal floating point	COMP-1	+ 1234	43 4D 20 00
		- 1234	C3 4D 20 00
	COMP-2	+ 1234	43 4D 20 00 00 00 00 00
		- 1234	C3 4D 20 00 00 00 00 00
External floating point	PIC +9(2).9(2)E+99 DISPLAY	+ 12.34E+02	4E F1 F2 4B F3 F4 C5 4E F0 F2
		- 12.34E+02	60 F1 F2 4B F3 F4 C5 4E F0 F2
	PIC +9(2).9(2)E+99 NATIONAL	+ 12.34E+02	00 2B 00 31 00 32 00 2E 00 33 00 34 00 45 00 2B 00 30 00 32
		- 12.34E+02	00 2D 00 31 00 32 00 2E 00 33 00 34 00 45 00 2B 00 30 00 32

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

Data format conversions

When the code in your program involves the interaction of items that have different data formats, the compiler converts those items either temporarily, for comparisons and arithmetic operations, or permanently, for assignment to the receiver in a MOVE 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 by using the same rules that are used for MOVE and COMPUTE statements.

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

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

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

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

Related references

[“Conversions and precision” on page 52](#)

[“Sign representation of zoned and packed-decimal data” on page 53](#)

Conversions and precision

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

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

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

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

Conversions that lose precision

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

If a USAGE COMP-1 data item is moved to a fixed-point data item of 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 that has 15 or fewer digits is moved to a USAGE COMP-2 data item and then returned to the fixed-point data item, the original value is recovered.

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

Conversions that result in rounding

If a USAGE COMP-1 data item, a USAGE COMP-2 data item, an external floating-point data item, or a floating-point literal is moved to a fixed-point data item, rounding occurs in the low-order position of the target data item.

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

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

Related concepts

Appendix A, “Intermediate results and arithmetic precision,” on page 779

Sign representation of zoned and packed-decimal data

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

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

Positive:

C, A, E, and F

Negative:

D and B

The COBOL NUMPROC compiler option affects sign processing for zoned decimal and internal decimal data. NUMPROC has no effect on binary data, national decimal 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, zoned-decimal sender is moved to an alphanumeric receiver, the sign is unchanged (even with NUMPROC (NOPFD) in effect).

Related references

[“NUMPROC” on page 382](#)

[“ZWB” on page 421](#)

Checking for incompatible data (numeric class test)

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

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

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

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

The numeric class test checks the contents of a data item against a set of values that are valid for the PICTURE and USAGE of the data item. 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). An external decimal data item that has USAGE DISPLAY is checked for hexadecimal values X'0' through X'9' in the digit positions (the lower 4 bits of each byte), for a valid zone code in the upper 4 bits of each byte and for a valid sign value in the sign position (whether separate or nonseparate). The sign code is in the upper 4 bits of the sign byte or in a separate byte if SIGN IS SEPARATE was specified. If the SIGN IS SEPARATE clause is used, the upper four bits of all bytes must be x'F'.

Note: Although the INVDATA compiler option allows toleration of invalid zone codes in USAGE DISPLAY numeric (zoned decimal) data items in numeric comparisons, invalid zone codes in zoned decimal data items will be treated as nonnumeric by the numeric class test.

For zoned decimal 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)
Signed	C, D, F	C, D, +0 (positive zero)
Unsigned	F	F
Separate sign	+, -	+, -, +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)
Signed	A to F	C, D, +0 (positive zero)
Unsigned	F	F
Separate sign	+, -	+, -, +0 (positive zero)

For more information about numeric class condition, see *Class condition* in the *Enterprise COBOL for z/OS Language Reference*.

You can also use the NUMCHECK (ZON, PAC) option to have the compiler generate implicit numeric class tests for zoned decimal (numeric USAGE DISPLAY) and packed decimal (COMP-3) data items that are used as sending data items. This numeric class test validates data and also validates sign fields against the NUMPROC compiler option to help you decide whether you can use NUMPROC (PFD) or not. For details, see “NUMCHECK” on page 378.

For more information about other invalid data issues, in particular for COBOL migration purposes, see the *FAQ about invalid data* in the *Enterprise COBOL Migration Guide*.

Related references

[“NUMCHECK” on page 378](#)

[“NUMPROC” on page 382](#)

[“INVDATA” on page 366](#)

Performing arithmetic

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

For most common arithmetic evaluations, the COMPUTE statement is appropriate. If you need to use numeric literals, numeric data, or arithmetic operators, you might want to use arithmetic expressions. In places where numeric expressions are allowed, you can save time by using numeric intrinsic functions. Language Environment callable services for mathematical functions and for date and time operations also provide a means of assigning arithmetic results to data items.

Related tasks

[“Using COMPUTE and other arithmetic statements” on page 55](#)

[“Using arithmetic expressions” on page 56](#)

[“Using numeric intrinsic functions” on page 56](#)

[“Using math-oriented callable services” on page 58](#)

[“Using date callable services” on page 59](#)

Using COMPUTE and other arithmetic statements

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

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

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

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

COMPUTE	Equivalent arithmetic statements
Compute Increment = Increment + 1	Add 1 to Increment
Compute Balance = Balance - Overdraft	Subtract Overdraft from Balance
Compute IncrementOne = IncrementOne + 1 Compute IncrementTwo = IncrementTwo + 1 Compute IncrementThree = IncrementThree + 1	Add 1 to IncrementOne, IncrementTwo, IncrementThree

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

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

Related concepts

[“Fixed-point contrasted with floating-point arithmetic” on page 62](#)
[Appendix A, “Intermediate results and arithmetic precision,” on page 779](#)

Related tasks

[“Defining numeric data” on page 43](#)

Using arithmetic expressions

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

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

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

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

Arithmetic operators are evaluated in the following order of precedence:

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

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

Related concepts

[“Fixed-point contrasted with floating-point arithmetic” on page 62](#)
[Appendix A, “Intermediate results and arithmetic precision,” on page 779](#)

Using numeric intrinsic functions

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

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

Numeric functions are classified into the following categories:

Integer

Those that return an integer

Floating point

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

Mixed

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

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

Table 10. Numeric intrinsic functions

Number handling	Date and time	Finance	Mathematics	Statistics
LENGTH MAX MIN NUMVAL NUMVAL-C NUMVAL-F SIGN TEST-NUMVAL TEST- NUMVAL-C TEST- NUMVAL-F ORD-MAX ORD-MIN	CURRENT-DATE DATE-OF-INTEGERS DATE-TO-YYYYMMDD DAY-OF-INTEGERS DAY-TO-YYYYDDD INTEGERS-OF-DATE INTEGERS-OF-DAY WHEN-COMPILED YEAR-TO-YYYY	ANNUITY PRESENT-VALUE	ABS ACOS ASIN ATAN COS E EXP EXP10 FACTORIAL INTEGERS INTEGERS-PART LOG LOG10 MOD PI REM SIN SQRT SUM TAN	MEAN MEDIAN MIDRANGE RANDOM RANGE STANDARD-DEVIATION VARIANCE

[“Examples: numeric intrinsic functions” on page 60](#)

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

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

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

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

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

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

Related concepts

[“Fixed-point contrasted with floating-point arithmetic” on page 62](#)

[Appendix A, “Intermediate results and arithmetic precision,” on page 779](#)

Related references
[“ARITH” on page 339](#)

Using math-oriented callable services

Most COBOL intrinsic functions have corresponding math-oriented callable services that you can use to produce the same results.

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.

For example (considering the first row of the table below), 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 long-precision Language Environment callable service	Corresponding extended-precision Language Environment callable service	Results same for intrinsic function and callable service?
ACOS	CEESDACS	CEESQACS	Yes
ASIN	CEESDASN	CEESQASN	Yes
ATAN	CEESDATN	CEESQATN	Yes
COS	CEESDCOS	CEESQCOS	Yes
E	CEESDEXP with <i>parm1</i> set to 1.0	CEESQEXP with <i>parm1</i> set to 1.0	Yes
EXP	CEESDEXP	CEESQEXP	Yes
EXP10	CEESDXPD with <i>parm1</i> set to 10.0	CEESQXPQ with <i>parm1</i> set to 10.0	Yes
LOG	CEESDLOG	CEESQLOG	Yes
LOG10	CEESDLG1	CEESQLG1	Yes
RANDOM ¹	CEERANO	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 with `ARITH (EXTEND)`.

Both the `RANDOM` intrinsic function and `CEERANO` service generate random numbers between zero and one. However, because each uses its own algorithm, `RANDOM` and `CEERANO` 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 (generally COMP-1 or COMP-2) required by that service.

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.

Related concepts

[“Fixed-point contrasted with floating-point arithmetic” on page 62](#)
[Appendix A, “Intermediate results and arithmetic precision,” on page 779](#)

Related tasks

[“Using Language Environment callable services” on page 773](#)

Related references

[“ARITH” on page 339](#)

Using 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 INTDATE(LILIAN) is in effect, COBOL uses October 15, 1582 as day 1. Language Environment always uses October 15, 1582 as day 1. If you use INTDATE(LILIAN), you get equivalent results from COBOL intrinsic functions and Language Environment date callable services. The following table compares the results when INTDATE(LILIAN) is in effect.

<i>Table 12. INTDATE(LILIAN) and compatibility of date intrinsic functions and callable services</i>		
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
INTEGERS-OF-DATE	CEEDAYS	Compatible
INTEGERS-OF-DATE	CEECBLDY	Incompatible

When the default setting of INTDATE(ANSI) is in effect, COBOL uses January 1, 1601 as day 1. The following table compares the results when INTDATE(ANSI) is in effect.

<i>Table 13. INTDATE(ANSI) and compatibility of date intrinsic functions and callable services</i>		
COBOL intrinsic function	Language Environment callable service	Results
INTEGERS-OF-DATE	CEECBLDY	Compatible
DATE-OF-INTEGERS	CEEDATE with picture string YYYYMMDD	Incompatible
DAY-OF-INTEGERS	CEEDATE with picture string YYYYDDD	Incompatible
INTEGERS-OF-DATE	CEEDAYS	Incompatible

Related tasks

[“Using Language Environment callable services” on page 773](#)

Related references

[“INTDATE” on page 365](#)

Examples: numeric intrinsic functions

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

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

General number handling

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

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

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

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

Date and time

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

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

Finance

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

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

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

You can use the ANNUITY function in business problems that require you to determine the amount of an installment payment (annuity) necessary to repay the principal and interest of a loan. The series of payments is characterized by an equal amount each period, periods of equal length, and an equal interest rate each period. The following example shows how you can calculate the monthly payment required to repay a \$15,000 loan in three years at a 12% annual interest rate (36 monthly payments, interest per month = .12/12):

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

Mathematics

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

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

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

Statistics

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

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

Related tasks

[“Converting to numbers \(NUMVAL, NUMVAL-C, NUMVAL-F\)” on page 115](#)

Fixed-point contrasted with floating-point arithmetic

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

Note: Fixed-point evaluations are sometimes done with decimal floating-point instructions, which are quite different from hex floating-point instructions.

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

- General arithmetic

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

- Expressions and functions

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

- Arithmetic comparisons

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

Floating-point evaluations

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

- An operand or result field is floating point.

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

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

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

- An exponent contains decimal places.

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

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

Fixed-point evaluations

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

Arithmetic comparisons (relation conditions)

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

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

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

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

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

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

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

[“Examples: fixed-point and floating-point evaluations” on page 63](#)

Related references

[“Arithmetic expressions in nonarithmetic statements” on page 787](#)

Examples: fixed-point and floating-point evaluations

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

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

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

These statements are evaluated using floating-point arithmetic:

```
compute report-matrix-col = (emp-count ** .5) + 1
```

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

These statements are evaluated using fixed-point arithmetic:

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

Using currency signs

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

You can use one or more of the following signs:

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

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

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

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

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

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

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

Table 14. Hexadecimal values of the euro sign

Code page CCSID	Applicable countries	Modified from	Euro sign
1140	USA, Canada, Netherlands, Portugal, Australia, New Zealand	037	X'9F'
1141	Austria, Germany	273	X'9F'
1142	Denmark, Norway	277	X'5A'
1143	Finland, Sweden	278	X'5A'
1144	Italy	280	X'9F'
1145	Spain, Latin America - Spanish	284	X'9F'
1146	UK	285	X'9F'
1147	France	297	X'9F'
1148	Belgium, Canada, Switzerland	500	X'9F'
1149	Iceland	871	X'9F'

Related references

“CURRENCY” on page 348

CURRENCY SIGN clause (*Enterprise COBOL for z/OS Language Reference*)

Example: multiple currency signs

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

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EuroSamp.
Environment Division.
Configuration Section.
Special-Names.
    Currency Sign is "CHF " with Picture Symbol "F"
    Currency Sign is "EUR " with Picture Symbol "Y".
Data Division.
WORKING-STORAGE SECTION.
01 Deposit-in-Euro      Pic S9999V99 Value 8000.00.
01 Deposit-in-CHF      Pic S99999V99.
01 Deposit-Report.
    02 Report-in-Franc  Pic -FFFFF9.99.
    02 Report-in-Euro   Pic -YYYYY9.99.
01 EUR-to-CHF-Conv-Rate Pic 9V99999 Value 1.53893.
. . .
PROCEDURE DIVISION.
Report-Deposit-in-CHF-and-EUR.
    Move Deposit-in-Euro to Report-in-Euro
    Compute Deposit-in-CHF Rounded
        = Deposit-in-Euro * EUR-to-CHF-Conv-Rate
    On Size Error
        Perform Conversion-Error
    Not On Size Error
        Move Deposit-in-CHF to Report-in-Franc
        Display "Deposit in euro = " Report-in-Euro
        Display "Deposit in franc = " Report-in-Franc
    End-Compute
    Goback.
Conversion-Error.
    Display "Conversion error from EUR to CHF"
    Display "Euro value: " Report-in-Euro.
```

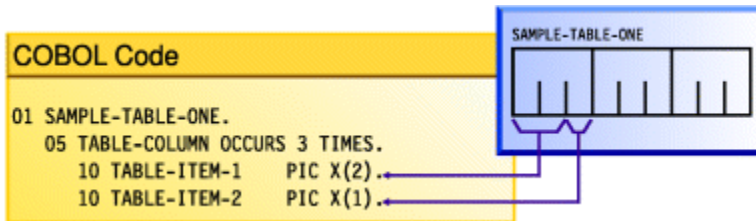
The above example produces the following display output:

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

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

Chapter 4. Handling tables

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



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

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

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

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

Related concepts

[“Complex OCCURS DEPENDING ON” on page 81](#)

Related tasks

[“Defining a table \(OCCURS\)” on page 67](#)

[“Nesting tables” on page 69](#)

[“Referring to an item in a table” on page 70](#)

[“Putting values into a table” on page 73](#)

[“Creating variable-length tables \(DEPENDING ON\)” on page 78](#)

[“Searching a table” on page 84](#)

[“Sorting a table” on page 87](#)

[“Processing table items using intrinsic functions” on page 87](#)

[“Working with unbounded tables and groups” on page 88](#)

[“Handling tables efficiently” on page 760](#)

Defining a table (OCCURS)

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

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

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

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

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

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

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

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

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

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

[“Example: binary search” on page 86](#)

Related concepts

[“National groups” on page 133](#)

Related tasks

[“Nesting tables” on page 69](#)

[“Referring to an item in a table” on page 70](#)

[“Putting values into a table” on page 73](#)

[“Creating variable-length tables \(DEPENDING ON\)” on page 78](#)

[“Using national groups” on page 137](#)

[“Doing a binary search \(SEARCH ALL\)” on page 86](#)

[“Defining numeric data” on page 43](#)

Related references

OCCURS clause (*Enterprise COBOL for z/OS Language Reference*)

SIGN clause (*Enterprise COBOL for z/OS Language Reference*)

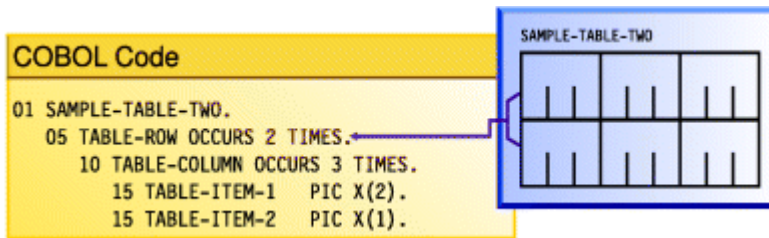
ASCENDING KEY and DESCENDING KEY phrases

(*Enterprise COBOL for z/OS Language Reference*)

SORT statement (*Enterprise COBOL for z/OS Language Reference*)

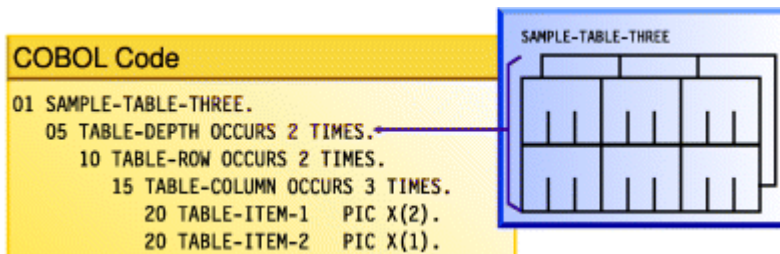
Nesting tables

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



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

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



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

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

[“Example: subscripting” on page 70](#)

[“Example: indexing” on page 70](#)

Related tasks

[“Defining a table \(OCCURS\)” on page 67](#)

[“Referring to an item in a table” on page 70](#)

[“Putting values into a table” on page 73](#)

[“Creating variable-length tables \(DEPENDING ON\)” on page 78](#)

[“Searching a table” on page 84](#)

[“Processing table items using intrinsic functions” on page 87](#)

[“Handling tables efficiently” on page 760](#)

Related references

OCCURS clause (*Enterprise COBOL for z/OS Language Reference*)

Example: subscripting

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

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

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

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

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

Related tasks

[“Subscripting” on page 71](#)

Example: indexing

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

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

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

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

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

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

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

This calculation is based on the following element lengths:

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

Related tasks

[“Indexing” on page 72](#)

Referring to an item in a table

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

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

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

Related tasks

[“Subscripting” on page 71](#)

[“Indexing” on page 72](#)

Subscripting

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

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

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

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

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

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

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

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

<p>ANY-TABLE before the change:</p> <p>ABCDEFGHIJ ABCDEFGHIJ ABCDEFGHIJ</p>	<p>ANY-TABLE after the change:</p> <p>AB??EFGHIJ ABCDEFGHIJ ABCDEFGHIJ</p>
--	---

[“Example: subscripting” on page 70](#)

Related tasks

[“Indexing” on page 72](#)

[“Putting values into a table” on page 73](#)

[“Searching a table” on page 84](#)

[“Handling tables efficiently” on page 760](#)

Indexing

You create an index by using the INDEXED BY phrase of the OCCURS clause to identify an index-name. For example, INX-A in the following code is an index-name:

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

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

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

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

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

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

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

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

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

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

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

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

[“Example: indexing” on page 70](#)

Related tasks

[“Subscripting” on page 71](#)

[“Putting values into a table” on page 73](#)

[“Searching a table” on page 84](#)

[“Processing table items using intrinsic functions” on page 87](#)

[“Handling tables efficiently” on page 760](#)

Related references

INDEXED BY phrase (*Enterprise COBOL for z/OS Language Reference*)

INDEX phrase (*Enterprise COBOL for z/OS Language Reference*)

SET statement (*Enterprise COBOL for z/OS Language Reference*)

Putting values into a table

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

Related tasks

[“Loading a table dynamically” on page 73](#)

[“Loading a variable-length table” on page 80](#)

[“Initializing a table \(INITIALIZE\)” on page 73](#)

[“Assigning values when you define a table \(VALUE\)” on page 74](#)

[“Assigning values to a variable-length table” on page 80](#)

Loading a table dynamically

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

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

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

[“Example: PERFORM and subscripting” on page 76](#)

[“Example: PERFORM and indexing” on page 77](#)

Related references

PERFORM statement (*Enterprise COBOL for z/OS Language Reference*)

Initializing a table (INITIALIZE)

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

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

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

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

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

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

```
01 TABLE-ONE.  
  02 Trans-out Occurs 20.  
    05 Trans-code      Pic X   Value "R".  
    05 Part-number     Pic XX  Value "13".  
    05 Trans-quan      Pic 99  Value 10.  
    05 Price-fields.  
      10 Unit-price     Pic 99V Value 50.  
      10 Discount       Pic 99V Value 25.
```

```

10 Sales-Price Pic 999 Value 375.
Initialize TABLE-ONE Replacing Numeric Data By 3
                        Alphanumeric Data By "X"

```

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

Trans-out (n) before	Trans-out (n) after
R13105025375	XXb030303003 ¹
1. The symbol <i>b</i> represents a blank space.	

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

```

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

```

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

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

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

[“Examples: initializing data items” on page 27](#)

Related tasks

- [“Initializing a structure \(INITIALIZE\)” on page 31](#)
- [“Assigning values when you define a table \(VALUE\)” on page 74](#)
- [“Assigning values to a variable-length table” on page 80](#)
- [“Looping through a table” on page 101](#)
- [“Using data items and group items” on page 24](#)
- [“Using national groups” on page 137](#)

Related references

`INITIALIZE` statement (*Enterprise COBOL for z/OS Language Reference*)

Assigning values when you define a table (VALUE)

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

Set static values in tables in one of these ways:

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

Related tasks

- [“Initializing each table item individually” on page 75](#)
- [“Initializing a table at the group level” on page 75](#)
- [“Initializing all occurrences](#)

of a given table element” on page 76

[“Initializing a structure \(INITIALIZE\)” on page 31](#)

Initializing each table item individually

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

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

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

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

In the example above, the VALUE clause at the 01 level initializes each of the table items to the same value. Each table item could instead be described with its own VALUE clause to initialize that item to a distinct value.

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

Related tasks

[“Initializing a structure \(INITIALIZE\)” on page 31](#)

[“Assigning values to a variable-length table” on page 80](#)

Related references

REDEFINES clause (*Enterprise COBOL for z/OS Language Reference*)

OCCURS clause (*Enterprise COBOL for z/OS Language Reference*)

Initializing a table at the group level

Code an alphanumeric or national group data item and assign to it, through the VALUE clause, the contents of the whole table. Then, in a subordinate data item, use an OCCURS clause to define the individual table items.

In the following example, the alphanumeric group data item TABLE-ONE uses a VALUE clause that initializes each of the four elements of TABLE-TWO:

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

In the following example, the national group data item Table-OneN uses a VALUE clause that initializes each of the three elements of the subordinate data item Table-TwoN (each of which is implicitly USAGE NATIONAL). Note that you can initialize a national group data item with a VALUE clause that uses an alphanumeric literal, as shown below, or a national literal.

```
01 Table-OneN Group-Usage National Value "AB12CD34EF56".
```

```

05 Table-TwoN Occurs 3 Times Indexed By MyI.
10 ElementOneN Pic nn.
10 ElementTwoN Pic 99.

```

After Table-OneN is initialized, ElementOneN(1) contains NX"00410042" (the UTF-16 representation of 'AB'), the national decimal item ElementTwoN(1) contains NX"00310032" (the UTF-16 representation of '12'), and so forth.

Related references

OCCURS clause (*Enterprise COBOL for z/OS Language Reference*)

GROUP-USAGE clause (*Enterprise COBOL for z/OS Language Reference*)

Initializing all occurrences of a given table element

You can use the VALUE clause in the data description of a table element to initialize all instances of that element to the specified value.

```

01 T2.
05 T-OBJ PIC 9 VALUE 3.
05 T OCCURS 5 TIMES
    DEPENDING ON T-OBJ.
10 X PIC XX VALUE "AA".
10 Y PIC 99 VALUE 19.
10 Z PIC XX VALUE "BB".

```

For example, the code above causes all the X elements (1 through 5) to be initialized to AA, all the Y elements (1 through 5) to be initialized to 19, and all the Z elements (1 through 5) to be initialized to BB. T-OBJ is then set to 3.

Related tasks

[“Assigning values to a variable-length table” on page 80](#)

Related references

OCCURS clause (*Enterprise COBOL for z/OS Language Reference*)

Example: PERFORM and subscripting

This example traverses an error-flag table using subscripting until an error code that has been set is found. If an error code is found, the corresponding error message is moved to a print report field.

```

*****
***          E R R O R   F L A G   T A B L E          ***
*****
01 Error-Flag-Table Value Spaces.
88 No-Errors Value Spaces.
05 Type-Error Pic X.
05 Shift-Error Pic X.
05 Home-Code-Error Pic X.
05 Work-Code-Error Pic X.
05 Name-Error Pic X.
05 Initials-Error Pic X.
05 Duplicate-Error Pic X.
05 Not-Found-Error Pic X.
01 Filler Redefines Error-Flag-Table.
05 Error-Flag Occurs 8 Times
    Indexed By Flag-Index Pic X.
77 Error-on Pic X Value "E".
*****
***          E R R O R   M E S S A G E   T A B L E          ***
*****
01 Error-Message-Table.
05 Filler Pic X(25) Value
    "Transaction Type Invalid".
05 Filler Pic X(25) Value
    "Shift Code Invalid".
05 Filler Pic X(25) Value
    "Home Location Code Inval.".
05 Filler Pic X(25) Value
    "Work Location Code Inval.".

```



```

05 Filler                               Pic X(25) Value
   "Last Name - Blanks".
05 Filler                               Pic X(25) Value
   "Initials - Blanks".
05 Filler                               Pic X(25) Value
   "Duplicate Record Found".
05 Filler                               Pic X(25) Value
   "Commuter Record Not Found".
01 Filler Redefines Error-Message-Table.
05 Error-Message Occurs 8 Times
   Indexed By Message-Index           Pic X(25).
.
.
PROCEDURE DIVISION.
.
.
Perform
  Varying Sub From 1 By 1
  Until No-Errors
  If Error-Flag (Sub) = Error-On
    Move Space To Error-Flag (Sub)
    Move Error-Message (Sub) To Print-Message
    Perform 260-Print-Report
  End-If
End-Perform
.
.
.

```

Example: PERFORM and indexing

This example traverses an error-flag table using indexing until an error code that has been set is found. If an error code is found, the corresponding error message is moved to a print report field.

```

*****
***          E R R O R   F L A G   T A B L E          ***
*****
01 Error-Flag-Table                       Value Spaces.
88 No-Errors                             Value Spaces.
05 Type-Error                            Pic X.
05 Shift-Error                           Pic X.
05 Home-Code-Error                       Pic X.
05 Work-Code-Error                       Pic X.
05 Name-Error                            Pic X.
05 Initials-Error                        Pic X.
05 Duplicate-Error                       Pic X.
05 Not-Found-Error                       Pic X.
01 Filler Redefines Error-Flag-Table.
05 Error-Flag Occurs 8 Times
   Indexed By Flag-Index                 Pic X.
77 Error-on                              Pic X Value "E".
*****
***          E R R O R   M E S S A G E   T A B L E      ***
*****
01 Error-Message-Table.
05 Filler                               Pic X(25) Value
   "Transaction Type Invalid".
05 Filler                               Pic X(25) Value
   "Shift Code Invalid".
05 Filler                               Pic X(25) Value
   "Home Location Code Inval.".
05 Filler                               Pic X(25) Value
   "Work Location Code Inval.".
05 Filler                               Pic X(25) Value
   "Last Name - Blanks".
05 Filler                               Pic X(25) Value
   "Initials - Blanks".
05 Filler                               Pic X(25) Value
   "Duplicate Record Found".
05 Filler                               Pic X(25) Value
   "Commuter Record Not Found".
01 Filler Redefines Error-Message-Table.
05 Error-Message Occurs 8 Times
   Indexed By Message-Index           Pic X(25).
.
.
PROCEDURE DIVISION.
.
.
Set Flag-Index To 1
Perform Until No-Errors
  Search Error-Flag
  When Error-Flag (Flag-Index) = Error-On
    Move Space To Error-Flag (Flag-Index)

```

```

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

```

Creating variable-length tables (DEPENDING ON)

If you do not know before run time how many times a table element occurs, define a variable-length table. To do so, use the OCCURS DEPENDING ON (ODO) clause.

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

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

You can also specify unbounded tables and groups, see Variable-length tables in the *Enterprise COBOL for z/OS Language Reference* for details.

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

- Correct calculation of record lengths

The length of the variable portions of a group item is the product of the object of the DEPENDING ON phrase and the length of the subject of the OCCURS clause.

- Conformance of the data in the object of the OCCURS DEPENDING ON clause to its PICTURE clause

If the content of the ODO object does not match its PICTURE clause, the program could terminate abnormally. You must ensure that the ODO object correctly specifies the current number of occurrences of table elements.

The following example shows a group item (REC-1) that contains both the subject and object of the OCCURS DEPENDING ON clause. The way the length of the group item is determined depends on whether it is sending or receiving data.

```

WORKING-STORAGE SECTION.
01 MAIN-AREA.
   03 REC-1.
      05 FIELD-1 PIC 9.
      05 FIELD-2 OCCURS 1 TO 5 TIMES
        DEPENDING ON FIELD-1 PIC X(05).
01 REC-2.
   03 REC-2-DATA PIC X(50).

```

If you want to move REC-1 (the sending item in this case) to REC-2, the length of REC-1 is determined immediately before the move, using the current value in FIELD-1. If the content of FIELD-1 conforms to its PICTURE clause (that is, if FIELD-1 contains a zoned decimal item), the move can proceed based on the actual length of REC-1. Otherwise, the result is unpredictable. You must ensure that the ODO object has the correct value before you initiate the move.

When you do a move to REC-1 (the receiving item in this case), the length of REC-1 is determined using the maximum number of occurrences. In this example, five occurrences of FIELD-2, plus FIELD-1, yields a length of 26 bytes. In this case, you do not need to set the ODO object (FIELD-1) before referencing REC-1 as a receiving item. However, the sending field's ODO object (not shown) must be set to a valid numeric value between 1 and 5 for the ODO object of the receiving field to be validly set by the move.

However, if you do a move to REC-1 (again the receiving item) where REC-1 is followed by a variably located group (a type of *complex ODO*), the actual length of REC-1 is calculated immediately before the

move, using the current value of the ODO object (FIELD-1). In the following example, REC-1 and REC-2 are in the same record, but REC-2 is not subordinate to REC-1 and is therefore variably located:

```
01 MAIN-AREA
  03 REC-1.
    05 FIELD-1 PIC 9.
    05 FIELD-3 PIC 9.
    05 FIELD-2 OCCURS 1 TO 5 TIMES
      DEPENDING ON FIELD-1 PIC X(05).
  03 REC-2.
    05 FIELD-4 OCCURS 1 TO 5 TIMES
      DEPENDING ON FIELD-3 PIC X(05).
```

The compiler issues a message that lets you know that the actual length was used. This case requires that you set the value of the ODO object before using the group item as a receiving field.

The following example shows how to define a variable-length table when the ODO object (LOCATION-TABLE-LENGTH below) is outside the group:

```
DATA DIVISION.
FILE SECTION.
FD LOCATION-FILE
  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.
  08 LOCATION-EOF VALUE "FALSE".
01 MISC-VALUES.
  05 LOCATION-TABLE-LENGTH PIC 9(3) VALUE ZERO.
  05 LOCATION-TABLE-MAX PIC 9(3) VALUE 100.
*****
***          L O C A T I O N   T A B L E          ***
***          FILE CONTAINS LOCATION CODES.      ***
*****
01 LOCATION-TABLE.
  05 LOCATION-CODE OCCURS 1 TO 100 TIMES
    DEPENDING ON LOCATION-TABLE-LENGTH PIC X(80).
```

Related concepts

[“Complex OCCURS DEPENDING ON” on page 81](#)

Related tasks

[“Assigning values to a variable-length table” on page 80](#)

[“Loading a variable-length table” on page 80](#)

[“Preventing overlay when adding elements to a variable table” on page 83](#)

[“Finding the length of data items” on page 120](#)

Related references

OCCURS DEPENDING ON clause

(*Enterprise COBOL for z/OS Language Reference*)

Variable-length tables (*Enterprise COBOL for z/OS Language Reference*)

Loading a variable-length table

You can use a *do-until* structure (a TEST AFTER loop) to control the loading of a variable-length table. For example, after the following code runs, LOCATION-TABLE-LENGTH contains the subscript of the last item in the table.

```
DATA DIVISION.
FILE SECTION.
FD LOCATION-FILE
  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 code a VALUE clause for an alphanumeric or national group item that has a subordinate data item that contains the OCCURS clause with the DEPENDING ON phrase. Each subordinate structure that contains the DEPENDING ON phrase is initialized using the maximum number of occurrences.

If you define the entire table by using the DEPENDING ON phrase, all the elements are initialized using the maximum defined value of the ODO (OCCURS DEPENDING ON) object.

If the ODO object is initialized by a VALUE clause, it is logically initialized after the ODO subject has been initialized.

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

For example, in the code above, the ODO subject Y(1) is initialized to 'A', Y(2) to 'B', . . . , Y(5) to 'E', and finally the ODO object X is initialized to 3. Any subsequent reference to TABLE-THREE (such as in a DISPLAY statement) refers to X and the first three elements, Y(1) through Y(3), of the table.

Related tasks

[“Assigning values when you define a table \(VALUE\)” on page 74](#)

Related references

OCCURS DEPENDING ON clause
(*Enterprise COBOL for z/OS Language Reference*)

Complex OCCURS DEPENDING ON

Several types of complex OCCURS DEPENDING ON (*complex ODO*) are possible. Complex ODO is supported as an extension to the 85 COBOL Standard.

The basic forms of complex ODO permitted by the compiler are as follows:

- Variably located item or group: A data item described by an OCCURS clause with the DEPENDING ON phrase is followed by a nonsubordinate elementary or group data item.
- Variably located table: A data item described by an OCCURS clause with the DEPENDING ON phrase is followed by a nonsubordinate data item described by an OCCURS clause.
- Table that has variable-length elements: A data item described by an OCCURS clause contains a subordinate data item described by an OCCURS clause with the DEPENDING ON phrase.
- Index name for a table that has variable-length elements.
- Element of a table that has variable-length elements.

[“Example: complex ODO” on page 81](#)

Related tasks

[“Preventing index errors when changing ODO object value” on page 82](#)

[“Preventing overlay when adding elements to a variable table” on page 83](#)

Related references

[“Effects of change in ODO object value” on page 82](#)
OCCURS DEPENDING ON clause
(*Enterprise COBOL for z/OS Language Reference*)

Example: complex ODO

The following example illustrates the possible types of occurrence of complex ODO.

```
01 FIELD-A.
   02 COUNTER-1                PIC S99.
   02 COUNTER-2                PIC S99.
   02 TABLE-1.
      03 RECORD-1 OCCURS 1 TO 5 TIMES
         DEPENDING ON COUNTER-1  PIC X(3).
   02 EMPLOYEE-NUMBER          PIC X(5). (1)
   02 TABLE-2 OCCURS 5 TIMES   (2) (3)
      INDEXED BY INDX.         (4)
   03 TABLE-ITEM              PIC 99.  (5)
   03 RECORD-2 OCCURS 1 TO 3 TIMES
      DEPENDING ON COUNTER-2.
   04 DATA-NUM                PIC S99.
```

Definition: In the example, COUNTER-1 is an *ODO object*, that is, it is the object of the DEPENDING ON clause of RECORD-1. RECORD-1 is said to be an *ODO subject*. Similarly, COUNTER-2 is the ODO object of the corresponding ODO subject, RECORD-2.

The types of complex ODO occurrences shown in the example above are as follows:

(1)

A variably located item: EMPLOYEE-NUMBER is a data item that follows, but is not subordinate to, a variable-length table in the same level-01 record.

(2)

A variably located table: TABLE - 2 is a table that follows, but is not subordinate to, a variable-length table in the same level-01 record.

(3)

A table with variable-length elements: TABLE - 2 is a table that contains a subordinate data item, RECORD - 2, whose number of occurrences varies depending on the content of its ODO object.

(4)

An index-name, INDX, for a table that has variable-length elements.

(5)

An element, TABLE - ITEM, of a table that has variable-length elements.

How length is calculated

The length of the variable portion of each record is the product of its ODO object and the length of its ODO subject. For example, whenever a reference is made to one of the complex ODO items shown above, the actual length, if used, is computed as follows:

- The length of TABLE - 1 is calculated by multiplying the contents of COUNTER - 1 (the number of occurrences of RECORD - 1) by 3 (the length of RECORD - 1).
- The length of TABLE - 2 is calculated by multiplying the contents of COUNTER - 2 (the number of occurrences of RECORD - 2) by 2 (the length of RECORD - 2), and adding the length of TABLE - ITEM.
- The length of FIELD - A is calculated by adding the lengths of COUNTER - 1, COUNTER - 2, TABLE - 1, EMPLOYEE - NUMBER, and TABLE - 2 times 5.

Setting values of ODO objects

You must set *every* ODO object in a group item before you reference any complex ODO item in the group. For example, before you refer to EMPLOYEE - NUMBER in the code above, you must set COUNTER - 1 and COUNTER - 2 even though EMPLOYEE - NUMBER does not directly depend on either ODO object for its value.

Restriction: An ODO object cannot be variably located.

Effects of change in ODO object value

If a data item that is described by an OCCURS clause with the DEPENDING ON phrase is followed in the same group by one or more nonsubordinate data items (a form of complex ODO), any change in value of the ODO object affects subsequent references to complex ODO items in the record.

For example:

- The size of any group that contains the relevant ODO clause reflects the new value of the ODO object.
- A MOVE to a group that contains the ODO subject is made based on the new value of the ODO object.
- The location of any nonsubordinate items that follow the item described with the ODO clause is affected by the new value of the ODO object. (To preserve the contents of the nonsubordinate items, move them to a work area before the value of the ODO object changes, then move them back.)

The value of an ODO object can change when you move data to the ODO object or to the group in which it is contained. The value can also change if the ODO object is contained in a record that is the target of a READ statement.

Related tasks

[“Preventing index errors when changing ODO object value” on page 82](#)

[“Preventing overlay when adding elements to a variable table” on page 83](#)

Preventing index errors when changing ODO object value

Be careful if you reference a complex-ODO index-name, that is, an index-name for a table that has variable-length elements, after having changed the value of the ODO object for a subordinate data item in the table.

When you change the value of an ODO object, the byte offset in an associated complex-ODO index is no longer valid because the table length has changed. Unless you take precautions, you will have unexpected results if you then code a reference to the index-name such as:

- A reference to an element of the table
- A SET statement of the form SET *integer-data-item* TO *index-name* (format 1)
- A SET statement of the form SET *index-name* UP | DOWN BY *integer* (format 2)

To avoid this type of error, do these steps:

1. Save the index in an integer data item. (Doing so causes an implicit conversion: the integer item receives the table element occurrence number that corresponds to the offset in the index.)
2. Change the value of the ODO object.
3. Immediately restore the index from the integer data item. (Doing so causes an implicit conversion: the index-name receives the offset that corresponds to the table element occurrence number in the integer item. The offset is computed according to the table length then in effect.)

The following code shows how to save and restore the index-name (shown in [“Example: complex ODO” on page 81](#)) when the ODO object COUNTER-2 changes.

```
77 INTEGER-DATA-ITEM-1      PIC 99.
. . .
  SET INDX TO 5.
*   INDX is valid at this point.
  SET INTEGER-DATA-ITEM-1 TO INDX.
*   INTEGER-DATA-ITEM-1 now has the
*   occurrence number that corresponds to INDX.
  MOVE NEW-VALUE TO COUNTER-2.
*   INDX is not valid at this point.
  SET INDX TO INTEGER-DATA-ITEM-1.
*   INDX is now valid, containing the offset
*   that corresponds to INTEGER-DATA-ITEM-1, and
*   can be used with the expected results.
```

Related references

SET statement (*Enterprise COBOL for z/OS Language Reference*)

Preventing overlay when adding elements to a variable table

Be careful if you increase the number of elements in a variable-occurrence table that is followed by one or more nonsubordinate data items in the same group. When you increment the value of the ODO object and add an element to a table, you can inadvertently overlay the variably located data items that follow the table.

To avoid this type of error, do these steps:

1. Save the variably located data items that follow the table in another data area.
2. Increment the value of the ODO object.
3. Move data into the new table element (if needed).
4. Restore the variably located data items from the data area where you saved them.

In the following example, suppose you want to add an element to the table VARY-FIELD-1, whose number of elements depends on the ODO object CONTROL-1. VARY-FIELD-1 is followed by the nonsubordinate variably located data item GROUP-ITEM-1, whose elements could potentially be overlaid.

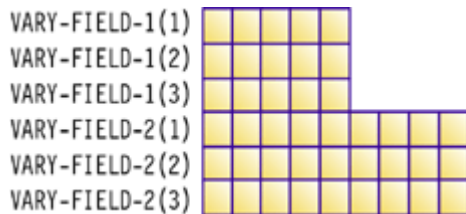
```
WORKING-STORAGE SECTION.
01 VARIABLE-REC.
   05 FIELD-1                PIC X(10).
   05 CONTROL-1              PIC S99.
   05 CONTROL-2              PIC S99.
   05 VARY-FIELD-1 OCCURS 1 TO 10 TIMES
       DEPENDING ON CONTROL-1 PIC X(5).
   05 GROUP-ITEM-1.
       10 VARY-FIELD-2
           OCCURS 1 TO 10 TIMES
```

```

        DEPENDING ON CONTROL-2          PIC X(9) .
01  STORE-VARY-FIELD-2.
    05  GROUP-ITEM-2.
        10  VARY-FLD-2
            OCCURS 1 TO 10 TIMES
            DEPENDING ON CONTROL-2      PIC X(9) .

```

Each element of VARY-FIELD-1 has 5 bytes, and each element of VARY-FIELD-2 has 9 bytes. If CONTROL-1 and CONTROL-2 both contain the value 3, you can picture storage for VARY-FIELD-1 and VARY-FIELD-2 as follows:



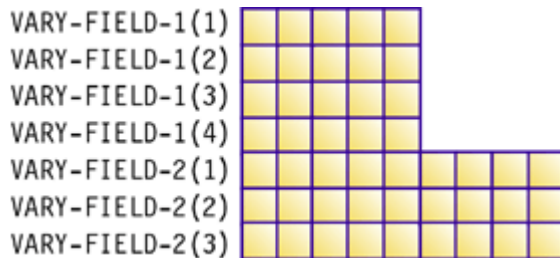
To add a fourth element to VARY-FIELD-1, code as follows to prevent overlaying the first 5 bytes of VARY-FIELD-2. (GROUP-ITEM-2 serves as temporary storage for the variably located GROUP-ITEM-1.)

```

MOVE GROUP-ITEM-1 TO GROUP-ITEM-2.
ADD 1 TO CONTROL-1.
MOVE five-byte-field TO
    VARY-FIELD-1 (CONTROL-1).
MOVE GROUP-ITEM-2 TO GROUP-ITEM-1.

```

You can picture the updated storage for VARY-FIELD-1 and VARY-FIELD-2 as follows:



Note that the fourth element of VARY-FIELD-1 did not overlay the first element of VARY-FIELD-2.

Searching a table

COBOL provides two search techniques for tables: *serial* and *binary*.

To do serial searches, use SEARCH and indexing. For variable-length tables, you can use PERFORM with subscripting or indexing.

To do binary searches, use SEARCH ALL and indexing.

A binary search can be considerably more efficient than a serial search. For a serial search, the number of comparisons is of the order of n , the number of entries in the table. For a binary search, the number of comparisons is of the order of only the logarithm (base 2) of n . A binary search, however, requires that the table items already be sorted.

Related tasks

[“Doing a serial search \(SEARCH\)” on page 85](#)

[“Doing a binary search \(SEARCH ALL\)” on page 86](#)

Doing a serial search (SEARCH)

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

The conditions in the WHEN phrase are evaluated in the order in which they appear:

- If none of the conditions is satisfied, the index is increased to correspond to the next table element, and the WHEN conditions are evaluated again.
- If one of the WHEN conditions is satisfied, the search ends. The index remains pointing to the table element that satisfied the condition.
- If the entire table has been searched and no conditions were met, the AT END imperative statement is executed if there is one. If you did not code AT END, control passes to the next statement in the program.

You can reference only one level of a table (a table element) with each SEARCH statement. To search multiple levels of a table, use nested SEARCH statements. Delimit each nested SEARCH statement with END-SEARCH.

Performance: If the found condition comes after some intermediate point in the table, you can speed up the search by using the SET statement to set the index to begin the search after that point. Arranging the table so that the data used most often is at the beginning of the table also enables more efficient serial searching. If the table is large and is presorted, a binary search is more efficient.

[“Example: serial search” on page 85](#)

Related references

SEARCH statement (*Enterprise COBOL for z/OS Language Reference*)

Example: serial search

The following example shows how you might find a particular string in the innermost table of a three-dimensional table.

Each dimension of the table has its own index (set to 1, 4, and 1, respectively). The innermost table (TABLE-ENTRY3) has an ascending key.

```
01 TABLE-ONE.
   05 TABLE-ENTRY1 OCCURS 10 TIMES
      INDEXED BY TE1-INDEX.
   10 TABLE-ENTRY2 OCCURS 10 TIMES
      INDEXED BY TE2-INDEX.
   15 TABLE-ENTRY3 OCCURS 5 TIMES
      ASCENDING KEY IS KEY1
      INDEXED BY TE3-INDEX.
      20 KEY1 PIC X(5).
      20 KEY2 PIC X(10).

PROCEDURE DIVISION.
   SET TE1-INDEX TO 1
   SET TE2-INDEX TO 4
   SET TE3-INDEX TO 1
   MOVE "A1234" TO KEY1 (TE1-INDEX, TE2-INDEX, TE3-INDEX + 2)
   MOVE "AAAAAAAAA00" 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)
      = "A1234AAAAAAAAA00"
      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 "A1234AAAAAAAAA00"
RETURN-CODE = 0
```

Doing a binary search (SEARCH ALL)

If you use `SEARCH ALL` to do a binary search, you do not need to set the index before you begin. The index is always the one that is associated with the first index-name in the `OCCURS` clause. The index varies during execution to maximize the search efficiency.

To use the `SEARCH ALL` statement to search a table, the table must specify the `ASCENDING` or `DESCENDING KEY` phrases of the `OCCURS` clause, or both, and must already be ordered on the key or keys that are specified in the `ASCENDING` and `DESCENDING KEY` phrases. You can use a format 2 `SORT` statement to order the table according to its defined keys, thereby making the table searchable by the `SEARCH ALL` statement. Note that `SEARCH ALL` will return unpredictable results if the table has not been ordered according to the keys.

In the `WHEN` phrase of the `SEARCH ALL` statement, you can test any key that is named in the `ASCENDING` or `DESCENDING KEY` phrases for the table, but you must test all preceding keys, if any. The test must be an equal-to condition, and the `WHEN` phrase must specify either a key (subscripted by the first index-name associated with the table) or a condition-name that is associated with the key. The `WHEN` condition can be a compound condition that is formed from simple conditions that use `AND` as the only logical connective.

Each key and its object of comparison must be compatible according to the rules for comparison of data items. Note though that if a key is compared to a national literal or identifier, the key must be a national data item.

[“Example: binary search” on page 86](#)

Related tasks

[“Defining a table \(OCCURS\)” on page 67](#)

Related references

`SEARCH` statement (*Enterprise COBOL for z/OS Language Reference*)

General relation conditions (*Enterprise COBOL for z/OS Language Reference*)

Example: binary search

The following example shows how you can code a binary search of a table.

Suppose you define a table that contains 90 elements of 40 bytes each, and three keys. The primary and secondary keys (`KEY-1` and `KEY-2`) are in ascending order, but the least significant key (`KEY-3`) is in descending order:

```
01 TABLE-A.
   05 TABLE-ENTRY OCCURS 90 TIMES
       ASCENDING KEY-1, KEY-2
       DESCENDING KEY-3
       INDEXED BY INDX-1.
   10 PART-1          PIC 99.
   10 KEY-1           PIC 9(5).
   10 PART-2          PIC 9(6).
   10 KEY-2           PIC 9(4).
   10 PART-3          PIC 9(18).
   10 KEY-3           PIC 9(5).
```

You can search this table by using the following statements:

```
SEARCH ALL TABLE-ENTRY
```

```

AT END
  PERFORM NOENTRY
WHEN KEY-1 (INDX-1) = VALUE-1 AND
      KEY-2 (INDX-1) = VALUE-2 AND
      KEY-3 (INDX-1) = VALUE-3
  MOVE PART-1 (INDX-1) TO OUTPUT-AREA
END-SEARCH

```

If an entry is found in which each of the three keys is equal to the value to which it is compared (VALUE-1, VALUE-2, and VALUE-3, respectively), PART-1 of that entry is moved to OUTPUT-AREA. If no matching key is found in the entries in TABLE-A, the NOENTRY routine is performed.

Sorting a table

You can sort a table by using the format 2 SORT statement. It is part of the 2002 COBOL Standard.

The format 2 SORT statement sorts table elements according to the specified table keys, and it is especially useful for tables used with SEARCH ALL. You can specify the keys for sorting as part of the table definition, which can also be used in the SEARCH ALL statement. Alternatively, you can also specify the keys for sorting as part of the SORT statement, either if you want to sort the table using different keys than those specified in the table definition, or if the table has no keys specified.

With the format 2 SORT statement, you don't need to use the input and output procedures as you do with the format 1 SORT statement.

See the following example in which the table is sorted based on specified keys:

```

WORKING-STORAGE SECTION.
01 GROUP-ITEM.
   05 TABL OCCURS 10 TIMES
      10 ELEM-ITEM1 PIC X.
      10 ELEM-ITEM2 PIC X.
      10 ELEM-ITEM3 PIC X.
...
PROCEDURE DIVISION.
...
  SORT TABL DESCENDING ELEM-ITEM2 ELEM-ITEM3.
  IF TABL (1)...

```

Related references

SORT statement (*Enterprise COBOL for z/OS Language Reference*)
[“Using the format 2 SORT statement to sort a table” on page 777](#)

Processing table items using intrinsic functions

You can use intrinsic functions to process alphabetic, alphanumeric, national, or numeric table items. (You can process DBCS data items only with the NATIONAL-OF intrinsic function.) The data descriptions of the table items must be compatible with the requirements for the function arguments.

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

```

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

```

You might often need to iteratively process the data in tables. For intrinsic functions that accept multiple arguments, you can use the subscript ALL to reference all the items in the table or in a single dimension of the table. The iteration is handled automatically, which can make your code shorter and simpler.

You can mix scalars and array arguments for functions that accept multiple arguments:

```

Compute Table-Median = Function Median(Arg1 Table-One(ALL))

```

[“Example: processing tables using intrinsic functions” on page 88](#)

Related tasks

[“Using intrinsic functions \(built-in functions\)” on page 38](#)

[“Converting data items \(intrinsic functions\)” on page 113](#)

[“Evaluating data items \(intrinsic functions\)” on page 117](#)

Related references

Intrinsic functions (*Enterprise COBOL for z/OS Language Reference*)

Example: processing tables using intrinsic functions

These examples show how you can apply an intrinsic function to some or all of the elements in a table by using the ALL subscript.

Assuming that Table-Two is a 2 x 3 x 2 array, the following statement adds the values in elements Table-Two(1, 3, 1), Table-Two(1, 3, 2), Table-Two(2, 3, 1), and Table-Two(2, 3, 2):

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

The following example computes various salary values for all the employees whose salaries are encoded in Employee-Table:

```
01 Employee-Table.
   05 Emp-Count      Pic s9(4) usage binary.
   05 Emp-Record     Occurs 1 to 500 times
                     depending on Emp-Count.
       10 Emp-Name   Pic x(20).
       10 Emp-Idme  Pic 9(9).
       10 Emp-Salary Pic 9(7)v99.
. . .
Procedure Division.
   Compute Max-Salary = Function Max(Emp-Salary(ALL))
   Compute I          = Function Ord-Max(Emp-Salary(ALL))
   Compute Avg-Salary = Function Mean(Emp-Salary(ALL))
   Compute Salary-Range = Function Range(Emp-Salary(ALL))
   Compute Total-Payroll = Function Sum(Emp-Salary(ALL))
```

Working with unbounded tables and groups

You can process an unbounded group as the input parameter to a called program. The memory for the unbounded group is provided by the calling program. Alternatively, you can define, initialize, and process unbounded groups in a single program.

To work with unbounded tables and groups in a single program, do these steps:

1. In the LINKAGE SECTION, define an unbounded table (with the syntax of OCCURS *n* TO UNBOUNDED), which will be part of an unbounded group.
2. In the WORKING-STORAGE SECTION or LOCAL-STORAGE SECTION, define the OCCURS DEPENDING ON objects.
3. In the PROCEDURE DIVISION, do these steps to process unbounded groups:
 - a. Set the OCCURS DEPENDING ON objects.
 - b. Use the LENGTH special register or the LENGTH intrinsic function to compute the total size of the group.
 - c. Use the CALL statement to call a storage allocation service, such as the Language Environment service CEEGTST. Allocate enough memory for the total length of the group. You will need a pointer to this memory (the CEEGTST service returns a pointer).
 - d. Use the SET statement to establish addressability. For example, SET ADDRESS OF *group* TO *pointer*.

4. Use the unbounded table and its containing unbounded group according to the following rules:

- You can reference unbounded tables in COBOL syntax anywhere a table can be referenced.
- You can reference unbounded groups in COBOL syntax anywhere an alphanumeric or national group can be referenced, with the following exceptions:
 - You cannot specify unbounded groups as a BY CONTENT argument in a CALL statement.
 - You cannot specify unbounded groups as *data-name-2* on the PROCEDURE DIVISION RETURNING phrase.
 - You cannot specify unbounded groups as arguments to intrinsic functions, except as an argument to the LENGTH intrinsic function.

Related references

“Example:

Using unbounded tables for parsing XML documents” on page 89

Example: ALLOCATE and FREE storage for unbounded tables

(*Enterprise COBOL for z/OS Language Reference*)

Variable-length tables (*Enterprise COBOL for z/OS Language Reference*)

OCCURS DEPENDING ON clause

(*Enterprise COBOL for z/OS Language Reference*)

Example: Using unbounded tables for parsing XML documents

Consider using unbounded tables when parsing an XML document with an unknown number of repetitive elements.

You can use any of the following methods:

- Predetermine the number of elements to expect. One method to determine the number of elements is to parse the XML document twice. During the first parse, count the number of occurrences of each unbounded element in the corresponding OCCURS UNBOUNDED DEPENDING ON object. Then, allocate storage for the data items using these computed values, and parse the XML document a second time to process its payload.
- Pick initial sizes and allow for expansion of the tables. It might be more efficient to set arbitrary limits in the OCCURS UNBOUNDED DEPENDING ON objects based on previous experience, and parse the document directly to process its content. For each unbounded element, check if the current limit is about to be exceeded. If so, allocate more storage for the corresponding array, copy the data from the old array to the expanded array, then free the storage for the old array.

The following examples illustrate the first method. See the *XML schema* example, and note that elements B and C have a maxOccurs value of unbounded, and thus can occur an unlimited number of times in the sequence within element G. In the *XML document* example, element B in fact occurs three times, and element C occurs five times.

In the *XML processing program* example, the processing procedure for the first XML PARSE statement simply computes the number of occurrences of elements B and C. After allocating the required storage, the program executes a second XML PARSE statement to process the XML payload.

XML schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="http://example.org"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="G">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="A" type="xsd:string" maxOccurs="1" />
<xsd:element name="B" type="xsd:int" maxOccurs="unbounded" />
<xsd:element name="C" type="xsd:int" maxOccurs="unbounded" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>
```

XML document

```
<?xml version="1.0" encoding="UTF-8"?>
<p:G xmlns:p="http://example.org" >
<A>Hello</A>
<B>1</B>
<B>2</B>
<B>3</B>
<C>1</C>
<C>2</C>
<C>3</C>
<C>4</C>
<C>5</C>
</p:G>
```

XML processing program

```
Identification division.
Program-id. XMLProc.
Data division.
Working-storage section.
01 NB pic S9(9) binary value zero.
01 NC pic S9(9) binary value zero.
01 Gptr pointer.
01 Gsize pic 9(9) binary.
01 Heap0 pic 9(9) binary value zero.
Linkage section.
01 XML-Doc pic X(500000).
01 G.
  02 A pic x(5).
  02 B pic s9(9) occurs 1 to unbounded depending on NB.
  02 C pic s9(9) occurs 1 to unbounded depending on NC.
Procedure division using XML-Doc.
XML parse XML-Doc processing procedure CountElements
Move length of G to Gsize
Call "CEEGTST" using Heap0 Gsize Gptr omitted
Set address of G to Gptr
XML parse XML-doc processing procedure acquireContent
...
Goback.
CountElements.
If xml-event = 'START-OF-ELEMENT'
  Evaluate xml-text
  When 'B'
    Add 1 to NB
  When 'C'
    Add 1 to NC
  When other
    Continue
  End-evaluate
End-if.
End program XMLProc.
```

Related tasks

[“Working with unbounded tables and groups” on page 88](#)

Chapter 5. Selecting and repeating program actions

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

These controls include the IF, EVALUATE, and PERFORM statements, and the use of switches and flags.

Related tasks

[“Selecting program actions” on page 91](#)

[“Repeating program actions” on page 99](#)

Selecting program actions

You can provide for different program actions depending on the tested value of one or more data items.

The IF and EVALUATE statements in COBOL test one or more data items by means of a conditional expression.

Related tasks

[“Coding a choice of actions” on page 91](#)

[“Coding conditional expressions” on page 95](#)

Related references

IF statement (*Enterprise COBOL for z/OS Language Reference*)

EVALUATE statement (*Enterprise COBOL for z/OS Language Reference*)

Coding a choice of actions

Use IF . . . ELSE to code a choice between two processing actions. (The word THEN is optional.) Use the EVALUATE statement to code a choice among three or more possible actions.

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

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

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

Such coding is suitable for simple cases. For complex logic, you probably need to use the ELSE clause. For example, suppose you have nested IF statements in which there is an action for only one of the processing choices. You could use the ELSE clause and code the null branch of the IF statement with the CONTINUE statement:

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

Note: NEXT SENTENCE can be very different from CONTINUE, based on location of the following period, as shown in this example:

```
IF condition-r
  statement-1
ELSE
  CONTINUE or NEXT SENTENCE
END-IF
*> CONTINUE goes to statement-2
  statement-2
  statement-3.
*> NEXT SENTENCE goes to statement-4
  statement-4
```

For details about NEXT SENTENCE, see IF statement in the *Enterprise COBOL for z/OS Language Reference*.

The EVALUATE statement is an expanded form of the IF statement that allows you to avoid nesting IF statements, a common source of logic errors and debugging problems.

Related tasks

[“Using nested IF statements” on page 92](#)

[“Using the EVALUATE statement” on page 93](#)

[“Coding conditional expressions” on page 95](#)

Using nested IF statements

If an IF statement contains an IF statement as one of its possible branches, the IF statements are said to be *nested*. Theoretically, there is no limit to the depth of nested IF statements.

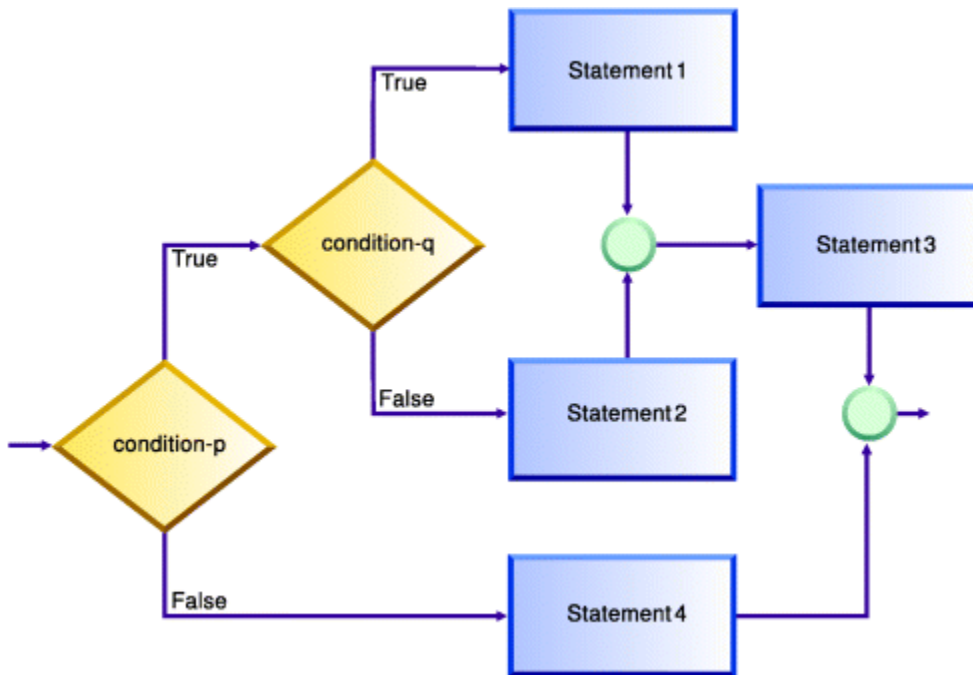
However, use nested IF statements sparingly. The logic can be difficult to follow, although explicit scope terminators and indentation can help. If a program has to test a variable for more than two values, EVALUATE is probably a better choice.

The following pseudocode depicts a nested IF statement:

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

In the pseudocode above, an IF statement and a sequential structure are nested in one branch of the outer IF. In this structure, the END-IF that closes the nested IF is very important. Use END-IF instead of a period, because a period would end the outer IF structure also.

The following figure shows the logic structure of the pseudocode above.



Related tasks

[“Coding a choice of actions” on page 91](#)

Related references

Explicit scope terminators (*Enterprise COBOL for z/OS Language Reference*)

Using the EVALUATE statement

You can use the EVALUATE statement instead of a series of nested IF statements to test several conditions and specify a different action for each. Thus you can use the EVALUATE statement to implement a *case structure* or decision table.

You can also use the EVALUATE statement to cause multiple conditions to lead to the same processing, as shown in these examples:

[“Example: EVALUATE using THRU phrase” on page 94](#)

[“Example: EVALUATE using multiple WHEN phrases” on page 94](#)

In an EVALUATE statement, the operands before the WHEN phrase are referred to as *selection subjects*, and the operands in the WHEN phrase are called the *selection objects*. Selection subjects can be identifiers, literals, conditional expressions, or the word TRUE or FALSE. Selection objects can be identifiers, literals, conditional or arithmetic expressions, or the word TRUE, FALSE, or ANY.

You can separate multiple selection subjects with the ALSO phrase. You can separate multiple selection objects with the ALSO phrase. The number of selection objects within each set of selection objects must be equal to the number of selection subjects, as shown in this example:

[“Example: EVALUATE testing several conditions” on page 95](#)

Identifiers, literals, or arithmetic expressions that appear within a selection object must be valid operands for comparison to the corresponding operand in the set of selection subjects. Conditions or the word TRUE or FALSE that appear in a selection object must correspond to a conditional expression or the word TRUE or FALSE in the set of selection subjects. (You can use the word ANY as a selection object to correspond to any type of selection subject.)

The execution of the EVALUATE statement ends when one of the following conditions occurs:

- The statements associated with the selected WHEN phrase are performed.

- The statements associated with the WHEN OTHER phrase are performed.
- No WHEN conditions are satisfied.

WHEN phrases are tested in the order that they appear in the source program. Therefore, you should order these phrases for the best performance. First code the WHEN phrase that contains selection objects that are most likely to be satisfied, then the next most likely, and so on. An exception is the WHEN OTHER phrase, which must come last.

Related tasks

[“Coding a choice of actions” on page 91](#)

Related references

EVALUATE statement (*Enterprise COBOL for z/OS Language Reference*)

General relation conditions (*Enterprise COBOL for z/OS Language Reference*)

Example: EVALUATE using THRU phrase

This example shows how you can code several conditions in a range of values to lead to the same processing action by coding the THRU phrase. Operands in a THRU phrase must be of the same class.

In this example, CARPOOL-SIZE is the *selection subject*; 1, 2, and 3 THRU 6 are the *selection objects*:

```
EVALUATE CARPOOL-SIZE
  WHEN 1
    MOVE "SINGLE" TO PRINT-CARPOOL-STATUS
  WHEN 2
    MOVE "COUPLE" TO PRINT-CARPOOL-STATUS
  WHEN 3 THRU 6
    MOVE "SMALL GROUP" TO PRINT-CARPOOL STATUS
  WHEN OTHER
    MOVE "BIG GROUP" TO PRINT-CARPOOL STATUS
END-EVALUATE
```

The following nested IF statements represent the same logic:

```
IF CARPOOL-SIZE = 1 THEN
  MOVE "SINGLE" TO PRINT-CARPOOL-STATUS
ELSE
  IF CARPOOL-SIZE = 2 THEN
    MOVE "COUPLE" TO PRINT-CARPOOL-STATUS
  ELSE
    IF CARPOOL-SIZE >= 3 and CARPOOL-SIZE <= 6 THEN
      MOVE "SMALL GROUP" TO PRINT-CARPOOL-STATUS
    ELSE
      MOVE "BIG GROUP" TO PRINT-CARPOOL-STATUS
    END-IF
  END-IF
END-IF
```

Example: EVALUATE using multiple WHEN phrases

The following example shows that you can code multiple WHEN phrases if several conditions should lead to the same action. Doing so gives you more flexibility than using only the THRU phrase, because the conditions do not have to evaluate to values in a range nor have the same class.

```
EVALUATE MARITAL-CODE
  WHEN "M"
    ADD 2 TO PEOPLE-COUNT
  WHEN "S"
  WHEN "D"
  WHEN "W"
    ADD 1 TO PEOPLE-COUNT
END-EVALUATE
```

The following nested IF statements represent the same logic:

```
IF MARITAL-CODE = "M" THEN
```

```

ADD 2 TO PEOPLE-COUNT
ELSE
  IF MARITAL-CODE = "S" OR
     MARITAL-CODE = "D" OR
     MARITAL-CODE = "W" THEN
    ADD 1 TO PEOPLE-COUNT
  END-IF
END-IF

```

Example: EVALUATE testing several conditions

This example shows the use of the ALSO phrase to separate two selection subjects (True ALSO True) and to separate the two corresponding selection objects within each set of selection objects (for example, When A + B < 10 Also C = 10).

Both selection objects in a WHEN phrase must satisfy the TRUE, TRUE condition before the associated action is performed. If both objects do not evaluate to TRUE, the next WHEN phrase is processed.

```

Identification Division.
  Program-ID. MiniEval.
Environment Division.
  Configuration Section.
  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.
  PNO1.
  Evaluate True Also True
    When Age < 13 Also Sex = "M"
      Move "Young Boy" To Description
    When Age < 13 Also Sex = "F"
      Move "Young Girl" To Description
    When Age > 12 And Age < 20 Also Sex = "M"
      Move "Teenage Boy" To Description
    When Age > 12 And Age < 20 Also Sex = "F"
      Move "Teenage Girl" To Description
    When Age > 19 Also Sex = "M"
      Move "Adult Man" To Description
    When Age > 19 Also Sex = "F"
      Move "Adult Woman" To Description
    When Other
      Move "Invalid Data" To Description
  End-Evaluate
  Evaluate True Also True
    When A + B < 10 Also C = 10
      Move "Case 1" To Description
    When A + B > 50 Also C = ( D + E ) / F
      Move "Case 2" To Description
    When Other
      Move "Case Other" To Description
  End-Evaluate
  Stop Run.

```

Coding conditional expressions

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

You can specify the following conditions:

- Relation conditions, such as:
 - Numeric comparisons
 - Alphanumeric comparisons
 - DBCS comparisons

- National comparisons
- Class conditions; for example, to test whether a data item:
 - IS NUMERIC
 - IS ALPHABETIC
 - IS ALPHABETIC-LOWER
 - IS ALPHABETIC-UPPER
 - IS DBCS
 - IS KANJI
- Condition-name conditions, to test the value of a conditional variable that you define
- Sign conditions, to test whether a numeric operand IS POSITIVE, NEGATIVE, or ZERO
- Switch-status conditions, to test the status of UPSI switches that you name in the SPECIAL -NAMES paragraph
- Complex conditions, such as:
 - Negated conditions; for example, NOT (A IS EQUAL TO B)
 - Combined conditions (conditions combined with logical operators AND or OR)

Related concepts

[“Switches and flags” on page 96](#)

Related tasks

[“Defining switches and flags” on page 97](#)

[“Resetting switches and flags” on page 98](#)

[“Checking for incompatible data \(numeric class test\)” on page 53](#)

[“Comparing national \(UTF-16\) data” on page 140](#)

[“Testing for valid DBCS characters” on page 159](#)

Related references

General relation conditions (*Enterprise COBOL for z/OS Language Reference*)

Class condition (*Enterprise COBOL for z/OS Language Reference*)

Rules for condition-name entries (*Enterprise COBOL for z/OS Language Reference*)

Sign condition (*Enterprise COBOL for z/OS Language Reference*)

Combined conditions (*Enterprise COBOL for z/OS Language Reference*)

Switches and flags

Some program decisions are based on whether the value of a data item is true or false, on or off, yes or no. Control these two-way decisions by using level-88 items with meaningful names (*condition-names*) to act as switches.

Other program decisions depend on the particular value or range of values of a data item. When you use condition-names to give more than just on or off values to a field, the field is generally referred to as a *flag*.

Flags and switches make your code easier to change. If you need to change the values for a condition, you have to change only the value of that level-88 condition-name.

For example, suppose a program uses a condition-name to test a field for a given salary range. If the program must be changed to check for a different salary range, you need to change only the value of the condition-name in the DATA DIVISION. You do not need to make changes in the PROCEDURE DIVISION.

Related tasks

[“Defining switches and flags” on page 97](#)

[“Resetting switches and flags” on page 98](#)

Defining switches and flags

In the DATA DIVISION, define level-88 items that will act as switches or flags, and give them meaningful names.

To test for more than two values with flags, assign more than one condition-name to a field by using multiple level-88 items.

The reader can easily follow your code if you choose meaningful condition-names and if the values assigned to them have some association with logical values.

[“Example: switches” on page 97](#)

[“Example: flags” on page 97](#)

Example: switches

The following examples show how you can use level-88 items to test for various binary-valued (on-off) conditions in your program.

For example, to test for the end-of-file condition for an input file named Transaction-File, you can use the following data definitions:

```
WORKING-STORAGE SECTION.  
01 Switches.  
   05 Transaction-EOF-Switch Pic X value space.  
   88 Transaction-EOF      value "y".
```

The level-88 description says that a condition named Transaction-EOF is in effect when Transaction-EOF-Switch has value 'y'. Referencing Transaction-EOF in the PROCEDURE DIVISION expresses the same condition as testing Transaction-EOF-Switch = "y". For example, the following statement causes a report to be printed only if Transaction-EOF-Switch has been set to 'y':

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

Example: flags

The following examples show how you can use several level-88 items together with an EVALUATE statement to determine which of several conditions in a program is true.

Consider for example a program that updates a main file. The updates are read from a transaction file. The records in the file contain a field that indicates which of the three functions is to be performed: add, change, or delete. In the record description of the input file, code a field for the function code using level-88 items:

```
01 Transaction-Input Record  
   05 Transaction-Type      Pic X.  
   88 Add-Transaction       Value "A".  
   88 Change-Transaction    Value "C".  
   88 Delete-Transaction    Value "D".
```

The code in the PROCEDURE DIVISION for testing these condition-names to determine which function is to be performed might look like this:

```
Evaluate True  
  When Add-Transaction  
    Perform Add-Main-Record-Paragraph  
  When Change-Transaction  
    Perform Update-Existing-Record-Paragraph  
  When Delete-Transaction
```

```
Perform Delete-Main-Record-Paragraph
End-Evaluate
```

Resetting switches and flags

Throughout your program, you might need to reset switches or flags to the original values they had in their data descriptions. To do so, either use a SET statement or define a data item to move to the switch or flag.

When you use the SET *condition-name* TO TRUE statement, the switch or flag is set to the original value that it was assigned in its data description. For a level-88 item that has multiple values, SET *condition-name* TO TRUE assigns the first value (A in the example below):

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

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

[“Example: set switch on” on page 98](#)

[“Example: set switch off” on page 99](#)

Example: set switch on

The following examples show how you can set a switch on by coding a SET statement that moves the condition name value to the conditional variable.

For example, the SET statement in the following example has the same effect as coding the statement Move "y" to Transaction-EOF-Switch:

```
01 Switches
  05 Transaction-EOF-Switch Pic X Value space.
  88 Transaction-EOF      Value "y".
. . .
Procédure Division.
000-Do-Main-Logic.
  Perform 100-Initialize-Paragraph
  Read Update-Transaction-File
  At End Set Transaction-EOF to True
  End-Read
```

The following example shows how to assign a value to a field in an output record based on the transaction code of an input record:

```
01 Input-Record.
  05 Transaction-Type Pic X(9).
01 Data-Record-Out.
  05 Data-Record-Type Pic X.
    88 Record-Is-Active Value "A".
    88 Record-Is-Suspended Value "S".
    88 Record-Is-Deleted Value "D".
  05 Key-Field Pic X(5).
. . .
Procédure Division.
  Evaluate Transaction-Type of Input-Record
  When "ACTIVE"
    Set Record-Is-Active to TRUE
  When "SUSPENDED"
    Set Record-Is-Suspended to TRUE
  When "DELETED"
    Set Record-Is-Deleted to TRUE
  End-Evaluate
```

Example: set switch off

The following example shows how you can set a switch off by coding a MOVE statement that moves the condition name value to the conditional variable.

For example, you can use a data item called SWITCH-OFF to set an on-off switch to off, as in the following code, which resets a switch to indicate that end-of-file has not been reached:

```
01 Switches
   05 Transaction-EOF-Switch      Pic X Value space.
       88 Transaction-EOF        Value "y".
01 SWITCH-OFF                    Pic X Value "n".
.
.
.
Procedure Division.
.
.
.
Move SWITCH-OFF to Transaction-EOF-Switch
```

Repeating program actions

Use a PERFORM statement to repeat the same code (that is, loop) either a specified number of times or based on the outcome of a decision.

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

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

Related tasks

[“Choosing inline or out-of-line PERFORM” on page 99](#)

[“Coding a loop” on page 100](#)

[“Looping through a table” on page 101](#)

[“Executing multiple paragraphs or sections” on page 101](#)

Related references

PERFORM statement (*Enterprise COBOL for z/OS Language Reference*)

Choosing inline or out-of-line PERFORM

An inline PERFORM is an imperative statement that is executed in the normal flow of a program; an out-of-line PERFORM entails a branch to a named paragraph and an implicit return from that paragraph.

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

- Is the PERFORM statement used in several places?

Use an out-of-line PERFORM when you want to use the same portion of code in several places in your program.

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

If the code to be performed is short, an inline PERFORM can be easier to read. But if the code extends over several screens, the logical flow of the program might be clearer if you use an out-of-line PERFORM. (Each paragraph in structured programming should perform one logical function, however.)

- What are the efficiency tradeoffs?

An inline PERFORM avoids the overhead of branching that occurs with an out-of-line PERFORM. But even out-of-line PERFORM coding can improve code optimization, so efficiency gains should not be overemphasized.

In the 1974 COBOL standard, the PERFORM statement is out-of-line and thus requires a branch to a separate procedure and an implicit return. If the performed procedure is in the subsequent sequential

flow of your program, it is also executed in that logic flow. To avoid this additional execution, place the procedure outside the normal sequential flow (for example, after the GOBACK) or code a branch around it.

The subject of an inline PERFORM is an imperative statement. Therefore, you must code statements (other than imperative statements) within an inline PERFORM with explicit scope terminators.

[“Example: inline PERFORM statement” on page 100](#)

Example: inline PERFORM statement

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

```

Perform 100-Initialize-Paragraph
* The following statement is an inline PERFORM:
Perform Until Transaction-EOF
  Read Update-Transaction-File Into WS-Transaction-Record
  At End
  Set Transaction-EOF To True
  Not At End
  Perform 200-Edit-Update-Transaction
  If No-Errors
    Perform 300-Update-Commuter-Record
  Else
    Perform 400-Print-Transaction-Errors
* End-If is a required scope terminator
  End-If
  Perform 410-Re-Initialize-Fields
* End-Read is a required scope terminator
  End-Read
End-Perform

```

Coding a loop

Use the PERFORM . . . TIMES statement to execute a procedure a specified number of times.

```

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

```

In the example above, when control reaches the PERFORM statement, the code for the procedure 010-PROCESS-ONE-MONTH is executed 12 times before control is transferred to the INSPECT statement.

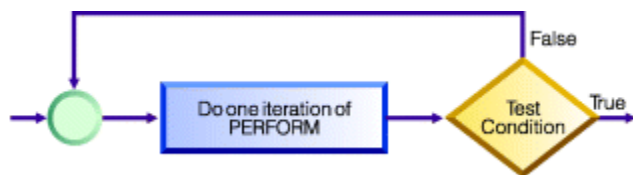
Use the PERFORM . . . UNTIL statement to execute a procedure until a condition you choose is satisfied. You can use either of the following forms:

```

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

```

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



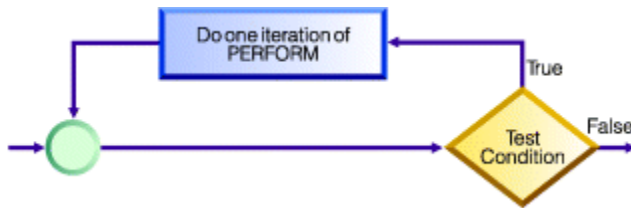
In the following example, the implicit WITH TEST BEFORE phrase provides a do-while structure:

```

PERFORM 010-PROCESS-ONE-MONTH
  UNTIL MONTH GREATER THAN 12
INSPECT . . .

```


When control reaches the PERFORM statement, the condition MONTH GREATER THAN 12 is tested. If the condition is satisfied, control is transferred to the INSPECT statement. If the condition is not satisfied, 010-PROCESS-ONE-MONTH is executed, and the condition is tested again. This cycle continues until the condition tests as true. (To make your program easier to read, you might want to code the WITH TEST BEFORE clause.)



Looping through a table

You can use the PERFORM . . . VARYING statement to initialize a table. In this form of the PERFORM statement, a variable is increased or decreased and tested until a condition is satisfied.

Thus you use the PERFORM statement to control looping through a table. You can use either of these forms:

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

The following section of code shows an example of looping through a table to check for invalid data:

```
PERFORM TEST AFTER VARYING WS-DATA-IX
  FROM 1 BY 1 UNTIL WS-DATA-IX = 12
  IF WS-DATA (WS-DATA-IX) EQUALS SPACES
    SET SERIOUS-ERROR TO TRUE
    DISPLAY ELEMENT-NUM-MSG5
  END-IF
END-PERFORM
INSPECT . . .
```

When control reaches the PERFORM statement above, WS-DATA-IX is set equal to 1 and the PERFORM statement is executed. Then the condition WS-DATA-IX = 12 is tested. If the condition is true, control drops through to the INSPECT statement. If the condition is false, WS-DATA-IX is increased by 1, the PERFORM statement is executed, and the condition is tested again. This cycle of execution and testing continues until WS-DATA-IX is equal to 12.

The loop above controls input-checking for the 12 fields of item WS-DATA. Empty fields are not allowed in the application, so the section of code loops and issues error messages as appropriate.

Executing multiple paragraphs or sections

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

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

Related tasks

[“Processing table items using intrinsic functions” on page 87](#)

Related references

EXIT PERFORM or EXIT PERFORM CYCLE statement

(Enterprise COBOL for z/OS Language Reference)

EXIT PARAGRAPH or EXIT SECTION statement

(Enterprise COBOL for z/OS Language Reference)

Chapter 6. Handling strings

COBOL provides language constructs for performing many different operations on string data items.

For example, you can:

- Join or split data items.
- Manipulate null-terminated strings, such as count or move characters.
- Refer to substrings by their ordinal position and, if needed, length.
- Tally and replace data items, such as count the number of times a specific character occurs in a data item.
- Convert data items, such as change to uppercase or lowercase.
- Evaluate data items, such as determine the length of a data item.

Related tasks

[“Joining data items \(STRING\)” on page 103](#)

[“Splitting data items \(UNSTRING\)” on page 105](#)

[“Manipulating null-terminated strings” on page 108](#)

[“Referring to substrings of data items” on page 109](#)

[“Tallying and replacing data items \(INSPECT\)” on page 112](#)

[“Converting data items \(intrinsic functions\)” on page 113](#)

[“Evaluating data items \(intrinsic functions\)” on page 117](#)

[Chapter 7, “Processing data in an international environment,” on page 123](#)

Joining data items (STRING)

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

The STRING statement transfers data into a receiving data item in the order that you indicate. In the STRING statement you also specify:

- A delimiter for each set of sending fields that, if encountered, causes those sending fields to stop being transferred (DELIMITED BY phrase)
- (Optional) Action to be taken if the receiving field is filled before all of the sending data has been processed (ON OVERFLOW phrase)
- (Optional) An integer data item that indicates the leftmost character position within the receiving field into which data should be transferred (WITH POINTER phrase)

The receiving data item must not be an edited item, or a display or national floating-point item. If the receiving data item has:

- USAGE DISPLAY, each identifier in the statement except the POINTER identifier must have USAGE DISPLAY, and each literal in the statement must be alphanumeric
- USAGE NATIONAL, each identifier in the statement except the POINTER identifier must have USAGE NATIONAL, and each literal in the statement must be national
- USAGE DISPLAY-1, each identifier in the statement except the POINTER identifier must have USAGE DISPLAY-1, and each literal in the statement must be DBCS
- USAGE UTF-8, each identifier in the statement except the POINTER identifier must have USAGE UTF-8, and each literal in the statement must be DBCS

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

[“Example: STRING statement” on page 104](#)

Related tasks

[“Handling errors in joining and splitting strings” on page 242](#)

Related references

STRING statement (*Enterprise COBOL for z/OS Language Reference*)

Example: STRING statement

The following example shows the STRING statement selecting and formatting information from a record into an output line.

The FILE SECTION defines the following record:

```
01 RCD-01.
   05 CUST-INFO.
      10 CUST-NAME    PIC X(15).
      10 CUST-ADDR    PIC X(35).
   05 BILL-INFO.
      10 INV-NO       PIC X(6).
      10 INV-AMT      PIC $$,$$$ .99.
      10 AMT-PAID     PIC $$,$$$ .99.
      10 DATE-PAID    PIC X(8).
      10 BAL-DUE      PIC $$,$$$ .99.
      10 DATE-DUE     PIC X(8).
```

The WORKING-STORAGE SECTION defines the following fields:

```
77 RPT-LINE          PIC X(120).
77 LINE-POS          PIC S9(3).
77 LINE-NO           PIC 9(5) VALUE 1.
77 DEC-POINT         PIC X VALUE ".".
```

The record RCD-01 contains the following information (the symbol *b* indicates a blank space):

```
J . B . bSMITHbbbb
444bSPRINGbST . , bCHICAGO , bBILL . bbbbb
A14275
$4,736.85
$2,400.00
09/22/76
$2,336.85
10/22/76
```

In the PROCEDURE DIVISION, these settings occur before the STRING statement:

- RPT-LINE is set to SPACES.
- LINE-POS, the data item to be used as the POINTER field, is set to 4.

Here is the STRING statement:

```
STRING
  LINE-NO SPACE CUST-INFO INV-NO SPACE DATE-DUE SPACE
  DELIMITED BY SIZE
  BAL-DUE
  DELIMITED BY DEC-POINT
  INTO RPT-LINE
  WITH POINTER LINE-POS.
```

Because the POINTER field LINE-POS has value 4 before the STRING statement is performed, data is moved into the receiving field RPT-LINE beginning at character position 4. Characters in positions 1 through 3 are unchanged.

The sending items that specify DELIMITED BY SIZE are moved in their entirety to the receiving field. Because BAL-DUE is delimited by DEC-POINT, the moving of BAL-DUE to the receiving field stops when a decimal point (the value of DEC-POINT) is encountered.

STRING results

When the STRING statement is performed, items are moved into RPT - LINE as shown in the table below.

Item	Positions
LINE - NO	4 - 8
Space	9
CUST - INFO	10 - 59
INV - NO	60 - 65
Space	66
DATE - DUE	67 - 74
Space	75
Portion of BAL - DUE that precedes the decimal point	76 - 81

After the STRING statement is performed, the value of LINE - POS is 82, and RPT - LINE has the values shown below.

Column

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

Splitting data items (UNSTRING)

Use the UNSTRING statement to split a sending field into several receiving fields. One UNSTRING statement can take the place of several MOVE statements.

In the UNSTRING statement you can specify:

- Delimiters that, when one of them is encountered in the sending field, cause the current receiving field to stop receiving and the next, if any, to begin receiving (DELIMITED BY phrase)
- A field for the delimiter that, when encountered in the sending field, causes the current receiving field to stop receiving (DELIMITER IN phrase)
- An integer data item that stores the number of characters placed in the current receiving field (COUNT IN phrase)
- An integer data item that indicates the leftmost character position within the sending field at which UNSTRING processing should begin (WITH POINTER phrase)
- An integer data item that stores a tally of the number of receiving fields that are acted on (TALLYING IN phrase)
- Action to be taken if all of the receiving fields are filled before the end of the sending data item is reached (ON OVERFLOW phrase)

The sending data item and the delimiters in the DELIMITED BY phrase must be of category alphabetic, alphanumeric, alphanumeric-edited, DBCS, national, national-edited, or UTF-8.

Receiving data items can be of category alphabetic, alphanumeric, numeric, DBCS, national, or UTF-8. If numeric, a receiving data item must be zoned decimal or national decimal. If a receiving data item has:

- USAGE DISPLAY, the sending item and each delimiter item in the statement must have USAGE DISPLAY, and each literal in the statement must be alphanumeric
- USAGE NATIONAL, the sending item and each delimiter item in the statement must have USAGE NATIONAL, and each literal in the statement must be national

- USAGE DISPLAY-1, the sending item and each delimiter item in the statement must have USAGE DISPLAY-1, and each literal in the statement must be DBCS
- USAGE UTF-8, the sending item and each delimiter item in the statement must have USAGE UTF-8, and each literal in the statement must be DBCS

[“Example: UNSTRING statement” on page 106](#)

Related concepts

[“Unicode and the encoding of language characters” on page 124](#)

Related tasks

[“Handling errors in joining and splitting strings” on page 242](#)

Related references

UNSTRING statement (*Enterprise COBOL for z/OS Language Reference*)

Classes and categories of data (*Enterprise COBOL for z/OS Language Reference*)

Example: UNSTRING statement

The following example shows the UNSTRING statement transferring selected information from an input record. Some information is organized for printing and some for further processing.

The FILE SECTION defines the following records:

```
* Record to be acted on by the UNSTRING statement:
01 INV-RCD.
   05 CONTROL-CHARS          PIC XX.
   05 ITEM-INDENT            PIC X(20).
   05 FILLER                 PIC X.
   05 INV-CODE              PIC X(10).
   05 FILLER                 PIC X.
   05 NO-UNITS              PIC 9(6).
   05 FILLER                 PIC X.
   05 PRICE-PER-M           PIC 99999.
   05 FILLER                 PIC X.
   05 RTL-AMT               PIC 9(6).99.

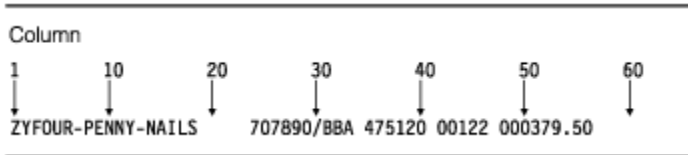
*
* UNSTRING receiving field for printed output:
01 DISPLAY-REC.
   05 INV-NO                 PIC X(6).
   05 FILLER                 PIC X VALUE SPACE.
   05 ITEM-NAME              PIC X(20).
   05 FILLER                 PIC X VALUE SPACE.
   05 DISPLAY-DOLS          PIC 9(6).

*
* UNSTRING receiving field for further processing:
01 WORK-REC.
   05 M-UNITS                PIC 9(6).
   05 FIELD-A                PIC 9(6).
   05 WK-PRICE REDEFINES FIELD-A PIC 9999V99.
   05 INV-CLASS              PIC X(3).

*
* UNSTRING statement control fields:
77 DBY-1                    PIC X.
77 CTR-1                    PIC S9(3).
77 CTR-2                    PIC S9(3).
77 CTR-3                    PIC S9(3).
77 CTR-4                    PIC S9(3).
77 DLTR-1                   PIC X.
77 DLTR-2                   PIC X.
77 CHAR-CT                  PIC S9(3).
77 FLDS-FILLED              PIC S9(3).
```

In the PROCEDURE DIVISION, these settings occur before the UNSTRING statement:

- A period (.) is placed in DBY-1 for use as a delimiter.
- CHAR-CT (the POINTER field) is set to 3.
- The value zero (0) is placed in FLDS-FILLED (the TALLYING field).
- Data is read into record INV-RCD, whose format is as shown below.



Here is the UNSTRING statement:

```

* Move subfields of INV-RCD to the subfields of DISPLAY-REC
* and WORK-REC:
UNSTRING INV-RCD
  DELIMITED BY ALL SPACES OR "/" OR DBY-1
  INTO ITEM-NAME      COUNT IN CTR-1
     INV-NO          DELIMITER IN DLTR-1  COUNT IN CTR-2
     INV-CLASS
     M-UNITS         COUNT IN CTR-3
     FIELD-A
     DISPLAY-DOLS   DELIMITER IN DLTR-2  COUNT IN CTR-4
  WITH POINTER CHAR-CT
  TALLYING IN  FLDS-FILLED
  ON OVERFLOW GO TO UNSTRING-COMPLETE.

```

Because the POINTER field CHAR-CT has value 3 before the UNSTRING statement is performed, the two character positions of the CONTROL-CHARS field in INV-RCD are ignored.

UNSTRING results

When the UNSTRING statement is performed, the following steps take place:

1. Positions 3 through 18 (FOUR-PENNY-NAILS) of INV-RCD are placed in ITEM-NAME, left justified in the area, and the four unused character positions are padded with spaces. The value 16 is placed in CTR-1.
2. Because ALL SPACES is coded as a delimiter, the five contiguous space characters in positions 19 through 23 are considered to be one occurrence of the delimiter.
3. Positions 24 through 29 (707890) are placed in INV-NO. The delimiter character slash (/) is placed in DLTR-1, and the value 6 is placed in CTR-2.
4. Positions 31 through 33 (BBA) are placed in INV-CLASS. The delimiter is SPACE, but because no field has been defined as a receiving area for delimiters, the space in position 34 is bypassed.
5. Positions 35 through 40 (475120) are placed in M-UNITS. The value 6 is placed in CTR-3. The delimiter is SPACE, but because no field has been defined as a receiving area for delimiters, the space in position 41 is bypassed.
6. Positions 42 through 46 (00122) are placed in FIELD-A and right justified in the area. The high-order digit position is filled with a zero (0). The delimiter is SPACE, but because no field was defined as a receiving area for delimiters, the space in position 47 is bypassed.
7. Positions 48 through 53 (000379) are placed in DISPLAY-DOLS. The period (.) delimiter in DBY-1 is placed in DLTR-2, and the value 6 is placed in CTR-4.
8. Because all receiving fields have been acted on and two characters in INV-RCD have not been examined, the ON OVERFLOW statement is executed. Execution of the UNSTRING statement is completed.

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

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

Manipulating null-terminated strings

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

For example, you can:

- Use null-terminated literal constants (Z" . . . ").
- Use an INSPECT statement to count the number of characters in a null-terminated string:

```
MOVE 0 TO char-count
INSPECT source-field TALLYING char-count
                     FOR CHARACTERS
                     BEFORE X"00"
```

- Use an UNSTRING statement to move characters in a null-terminated string to a target field, and get the character count:

```
WORKING-STORAGE SECTION.
01 source-field          PIC X(1001).
01 char-count           COMP-5 PIC 9(4).
01 target-area.
   02 individual-char OCCURS 1 TO 1000 TIMES DEPENDING ON char-count
      PIC X.

PROCEDURE DIVISION.
  UNSTRING source-field DELIMITED BY X"00"
                    INTO target-area
                    COUNT IN char-count

  ON OVERFLOW
    DISPLAY "source not null terminated or target too short"
  END-UNSTRING
```

- Use a SEARCH statement to locate trailing null or space characters. Define the string being examined as a table of single characters.
- Check each character in a field in a loop (PERFORM). You can examine each character in a field by using a reference modifier such as source-field (I:1).

[“Example: null-terminated strings” on page 108](#)

Related tasks

[“Handling null-terminated strings” on page 556](#)

Related references

Alphanumeric literals (*Enterprise COBOL for z/OS Language Reference*)

Example: null-terminated strings

The following example shows several ways in which you can process null-terminated strings.

```
01 L pic X(20) value z'ab'.
01 M pic X(20) value z'cd'.
01 N pic X(20).
01 N-Length pic 99 value zero.
01 Y pic X(13) value 'Hello, World!'.

* Display null-terminated string:
  Inspect N tallying N-length
           for characters before initial x'00'
  Display 'N: ' N(1:N-Length) ' Length: ' N-Length

* Move null-terminated string to alphanumeric, strip null:
  Unstring N delimited by X'00' into X

* Create null-terminated string:
  String Y           delimited by size
           X'00' delimited by size
```



```

        into N.
    . . .
    * Concatenate two null-terminated strings to produce another:
      String L      delimited by x'00'
      M            delimited by x'00'
      X'00'        delimited by size
      into N.

```

Referring to substrings of data items

Refer to a substring of a data item that has USAGE DISPLAY, DISPLAY-1, or NATIONAL by using a reference modifier. You can also refer to a substring of an alphanumeric or national character string that is returned by an intrinsic function by using a reference modifier.

Note: To get a substring of a character string argument that is encoded in UTF-8, use the USUBSTR function as described in [“Using intrinsic functions to process UTF-8 encoded data”](#) on page 153.

The following example shows how to use a reference modifier to refer to a twenty-character substring of a data item called Customer-Record:

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

You code a reference modifier in parentheses immediately after the data item. As the example shows, a reference modifier can contain two values that are separated by a colon, in this order:

1. Ordinal position (from the left) of the character that you want the substring to start with
2. (Optional) Length of the required substring in *character positions*

The reference-modifier position and length for an item that has USAGE DISPLAY are expressed in terms of single-byte characters. The reference-modifier position and length for items that have USAGE DISPLAY-1 or NATIONAL are expressed in terms of DBCS character positions and national character positions, respectively.

If you omit the length in a reference modifier (coding only the ordinal position of the first character, followed by a colon), the substring extends to the end of the item. Omit the length where possible as a simpler and less error-prone coding technique.

You can refer to substrings of USAGE DISPLAY data items, including alphanumeric groups, alphanumeric-edited data items, numeric-edited data items, display floating-point data items, and zoned decimal data items, by using reference modifiers. When you reference-modify any of these data items, the result is of category alphanumeric. When you reference-modify an alphabetic data item, the result is of category alphabetic.

You can refer to substrings of USAGE NATIONAL data items, including national groups, national-edited data items, numeric-edited data items, national floating-point data items, and national decimal data items, by using reference modifiers. When you reference-modify any of these data items, the result is of category national. For example, suppose that you define a national decimal data item as follows:

```
01 NATL-DEC-ITEM Usage National Pic 999 Value 123.
```

You can use NATL-DEC-ITEM in an arithmetic expression because NATL-DEC-ITEM is of category numeric. But you cannot use NATL-DEC-ITEM(2:1) (the national character 2, which in hexadecimal notation is NX"0032") in an arithmetic expression, because it is of category national.

You can refer to substrings of table entries, including variable-length entries, by using reference modifiers. To refer to a substring of a table entry, code the subscript expression before the reference modifier. For example, assume that PRODUCT-TABLE is a properly coded table of character strings. To move D to the fourth character in the second string in the table, you can code this statement:

```
MOVE 'D' to PRODUCT-TABLE (2), (4:1)
```

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

[“Example: arithmetic expressions as reference modifiers” on page 111](#)

Because numeric function identifiers can be used anywhere that arithmetic expressions can be used, you can code a numeric function identifier in a reference modifier as the leftmost character position or as the length, or both.

[“Example: intrinsic functions as reference modifiers” on page 111](#)

Each number in the reference modifier must have a value of at least 1. The sum of the two numbers must not exceed the total length of the data item by more than 1 character position so that you do not reference beyond the end of the substring.

If the leftmost character position or the length value is a fixed-point noninteger, truncation occurs to create an integer. If either is a floating-point noninteger, rounding occurs to create an integer.

The SSRANGE compiler option detects out-of-range reference modifiers, and flags violations with a runtime message.

Related concepts

[“Reference modifiers” on page 110](#)

[“Unicode and the encoding of language characters” on page 124](#)

Related tasks

[“Referring to an item in a table” on page 70](#)

Related references

[“SSRANGE” on page 401](#)

Reference modification (*Enterprise COBOL for z/OS Language Reference*)

Function definitions (*Enterprise COBOL for z/OS Language Reference*)

Reference modifiers

Reference modifiers let you easily refer to a substring of a data item.

For example, assume that you want to retrieve the current time from the system and display its value in an expanded format. You can retrieve the current time with the ACCEPT statement, which returns the hours, minutes, seconds, and hundredths of seconds in this format:

```
HHMMSSss
```

However, you might prefer to view the current time in this format:

```
HH:MM:SS
```

Without reference modifiers, you would have to define data items for both formats. You would also have to write code to convert from one format to the other.

With reference modifiers, you do not need to provide names for the subfields that describe the TIME elements. The only data definition you need is for the time as returned by the system. For example:

```
01 REFMOD-TIME-ITEM    PIC X(8).
```

The following code retrieves and expands the time value:

```
ACCEPT REFMOD-TIME-ITEM FROM TIME.  
DISPLAY "CURRENT TIME IS: "  
* Retrieve the portion of the time value that corresponds to
```

```

*   the number of hours:
      REFMOD-TIME-ITEM (1:2)
      ":"
* Retrieve the portion of the time value that corresponds to
*   the number of minutes:
      REFMOD-TIME-ITEM (3:2)
      ":"
* Retrieve the portion of the time value that corresponds to
*   the number of seconds:
      REFMOD-TIME-ITEM (5:2)

```

[“Example: arithmetic expressions as reference modifiers” on page 111](#)

[“Example: intrinsic](#)

[functions as reference modifiers” on page 111](#)

Related tasks

[“Assigning input from a screen or file \(ACCEPT\)” on page 35](#)

[“Referring to substrings of data items” on page 109](#)

[“Using national data \(Unicode\) in COBOL” on page 125](#)

Related references

Reference modification (*Enterprise COBOL for z/OS Language Reference*)

Example: arithmetic expressions as reference modifiers

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

Suppose a program has the following data:

```

01 LEFTY      PIC X(30).
01 RIGHTY    PIC X(30) JUSTIFIED RIGHT.
01 I         PIC 9(9)  USAGE BINARY.

```

The program counts the number of leading spaces and, using arithmetic expressions in a reference modifier, moves the right-justified characters into another field, justified to the left:

```

MOVE SPACES TO LEFTY
MOVE ZERO TO I
INSPECT RIGHTY
      TALLYING I FOR LEADING SPACE.
IF I IS LESS THAN LENGTH OF RIGHTY THEN
      MOVE RIGHTY ( I + 1 : LENGTH OF RIGHTY - I ) TO LEFTY
END-IF

```

The MOVE statement transfers characters from RIGHTY, beginning at the position computed as I + 1 for a length that is computed as LENGTH OF RIGHTY - I, into the field LEFTY.

Example: intrinsic functions as reference modifiers

You can use intrinsic functions in reference modifiers if you do not know the leftmost position or length of a substring at compile time.

For example, the following code fragment causes a substring of Customer-Record to be moved into the data item WS-name. The substring is determined at run time.

```

05 WS-name      Pic x(20).
05 Left-posn    Pic 99.
05 I           Pic 99.
. . .
Move Customer-Record(Function Min(Left-posn I):Function Length(WS-name)) to WS-name

```

If you want to use a noninteger function in a position that requires an integer function, you can use the INTEGER or INTEGER-PART function to convert the result to an integer. For example:

```
Move Customer-Record(Function Integer(Function Sqrt(I)): ) to WS-name
```

Related references

INTEGER (*Enterprise COBOL for z/OS Language Reference*)

INTEGER-PART (*Enterprise COBOL for z/OS Language Reference*)

Tallying and replacing data items (INSPECT)

Use the INSPECT statement to inspect characters or groups of characters in a data item and to optionally replace them.

Use the INSPECT statement to do the following tasks:

- Count the number of times a specific character occurs in a data item (TALLYING phrase).
- Fill a data item or selected portions of a data item with specified characters such as spaces, asterisks, or zeros (REPLACING phrase).
- Convert all occurrences of a specific character or string of characters in a data item to replacement characters that you specify (CONVERTING phrase).

You can specify one of the following data items as the item to be inspected:

- An elementary item described explicitly or implicitly as USAGE DISPLAY, USAGE DISPLAY-1, or USAGE NATIONAL
- An alphanumeric group item or national group item

If the inspected item has:

- USAGE DISPLAY, each identifier in the statement (except the TALLYING count field) must have USAGE DISPLAY, and each literal in the statement must be alphanumeric
- USAGE NATIONAL, each identifier in the statement (except the TALLYING count field) must have USAGE NATIONAL, and each literal in the statement must be national
- USAGE DISPLAY-1, each identifier in the statement (except the TALLYING count field) must have USAGE DISPLAY-1, and each literal in the statement must be a DBCS literal

[“Examples: INSPECT statement” on page 112](#)

Related concepts

[“Unicode and the encoding of language characters” on page 124](#)

Related references

INSPECT statement (*Enterprise COBOL for z/OS Language Reference*)

Examples: INSPECT statement

The following examples show some uses of the INSPECT statement to examine and replace characters.

In the following example, the INSPECT statement examines and replaces characters in data item DATA-2. The number of times a leading zero (0) occurs in the data item is accumulated in COUNTR. The first instance of the character A that follows the first instance of the character C is replaced by the character 2.

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

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

In the following example, the INSPECT statement examines and replaces characters in data item DATA-3. Each character that precedes the first instance of a quotation mark (") is replaced by the character 0.

```

77 COUNTR          PIC 9   VALUE ZERO.
01 DATA-3        PIC X(8).
. . .
    INSPECT DATA-3
      REPLACING CHARACTERS BY ZEROS BEFORE INITIAL QUOTE

```

DATA-3 before	COUNTR after	DATA-3 after
456"ABEL	0	000"ABEL
ANDES"12	0	00000"12
"Twas BR	0	"Twas BR

The following example shows the use of INSPECT CONVERTING with AFTER and BEFORE phrases to examine and replace characters in data item DATA-4. All characters that follow the first instance of the character / but that precede the first instance of the character ? (if any) are translated from lowercase to uppercase.

```

01 DATA-4        PIC X(11).
. . .
    INSPECT DATA-4
      CONVERTING
        "abcdefghijklmnopqrstuvwxy" TO
        "ABCDEFGHIJKLMNopRSTUVWXYZ"
      AFTER INITIAL "/"
      BEFORE INITIAL "?"

```

DATA-4 before	DATA-4 after
a/five/?six	a/FIVE/?six
r/Rexx/RRRr	r/REXX/RRRR
zfour?inspe	zfour?inspe

Converting data items (intrinsic functions)

You can use intrinsic functions to convert character-string data items to several other formats, for example, to uppercase or lowercase, to reverse order, to numbers, to one code page from another, or to hexadecimal or binary digits. You can also convert hexadecimal character strings or bit character strings to alphanumeric data items.

You can use the NATIONAL -OF and DISPLAY -OF intrinsic functions to convert to and from national (Unicode) strings.

You can also use the INSPECT statement to convert characters.

[“Examples: INSPECT statement” on page 112](#)

Related tasks

[“Changing case \(UPPER-CASE, LOWER-CASE\)” on page 114](#)

[“Transforming to reverse order \(REVERSE\)” on page 114](#)

[“Converting to numbers \(NUMVAL, NUMVAL-C, NUMVAL-F\)” on page 115](#)

[“Converting from one code page to another” on page 116](#)

[“Converting](#)

[to hexadecimal or bit data \(HEX-OF, BIT-OF\)” on page 116](#)

[“Converting from hexadecimal or bit data \(HEX-TO-CHAR, BIT-TO-CHAR\)” on page 117](#)

Changing case (UPPER-CASE, LOWER-CASE)

You can use the UPPER-CASE and LOWER-CASE intrinsic functions to easily change the case of alphanumeric, alphabetic, or national strings.

```
01 Item-1 Pic x(30) Value "Hello World!".
01 Item-2 Pic x(30).
. . .
  Display Item-1
  Display Function Upper-case(Item-1)
  Display Function Lower-case(Item-1)
  Move Function Upper-case(Item-1) to Item-2
  Display Item-2
```

The code above displays the following messages on the system logical output device:

```
Hello World!
HELLO WORLD!
hello world!
HELLO WORLD!
```

The DISPLAY statements do not change the actual contents of Item-1, but affect only how the letters are displayed. However, the MOVE statement causes uppercase letters to replace the contents of Item-2.

Note: The UPPER-CASE and LOWER-CASE intrinsic functions do not support alphanumeric arguments that contain UTF-8 encoded data.

Related tasks

[“Assigning input from a screen or file \(ACCEPT\)” on page 35](#)

[“Displaying values on a screen or in a file \(DISPLAY\)” on page 35](#)

Transforming to reverse order (REVERSE)

You can reverse the order of the characters in a string by using the REVERSE intrinsic function.

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

For example, the statement above reverses the order of the characters in Orig-cust-name. If the starting value is JOHNSONbbb, the value after the statement is performed is bbbNOSNH0J, where *b* represents a blank space.

Related concepts

[“Unicode and the encoding of language characters” on page 124](#)

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

The NUMVAL, NUMVAL - C and NUMVAL - F functions convert character strings (alphanumeric or national literals, or class alphanumeric or class national data items) to numbers. Use these functions to convert free-format character-representation numbers to numeric form so that you can process them numerically.

```
01 R      Pic x(20) Value "- 1234.5678".
01 S      Pic x(20) Value "$12,345.67CR".
01 T      Pic x(20) Value "+ 12.345678E+2".
01 Total  Usage is Comp-1.
. . .
Compute Total = Function Numval(R) + Function Numval-C(S) + Function Numval-F(T)
```

Use NUMVAL - C when the argument includes a currency symbol or comma or both, as shown in the example above. You can also place an algebraic sign before or after the character string, and the sign will be processed. The arguments must not exceed 18 digits when you compile with the default option ARITH (COMPAT) (*compatibility mode*) nor 31 digits when you compile with ARITH (EXTEND) (*extended mode*), not including the editing symbols.

Use NUMVAL - F when the argument includes an exponent value, as shown in the example above. You can also place an algebraic sign before the character string, and the sign will be processed. The arguments must not exceed 18 digits when you compile with the default option ARITH (COMPAT) (*compatibility mode*) nor 31 digits when you compile with ARITH (EXTEND) (*extended mode*), not including the editing symbols.

NUMVAL, NUMVAL - C and NUMVAL - F return long (64-bit) floating-point values in compatibility mode, and return extended-precision (128-bit) floating-point values in extended mode. A reference to either of these functions represents a reference to a numeric data item.

At most 15 decimal digits can be converted accurately to long-precision floating point (as described in the related reference below about conversions and precision). Internally NUMVAL uses long-precision floating calculations to convert the given number to the output, so if the argument to NUMVAL, NUMVAL - C, or NUMVAL - F has more than 15 digits, it is recommended that you specify the ARITH (EXTEND) compiler option so that an extended-precision function result that can accurately represent the value of the argument is returned. Otherwise, the result may lose precision in an unexpected manner.

When you use NUMVAL, NUMVAL - C, or NUMVAL - F, you do not need to statically define numeric data in a fixed format nor input data in a precise manner. For example, suppose you define numbers to be entered as follows:

```
01 X      Pic S999V99 leading sign is separate.
. . .
Accept X from Console
```

The user of the application must enter the numbers exactly as defined by the PICTURE clause. For example:

```
+001.23
-300.00
```

However, using the NUMVAL function, you could code:

```
01 A      Pic x(10).
01 B      Pic S999V99.
. . .
Accept A from Console
Compute B = Function Numval(A)
```

The input could then be:

```
1.23  
-300
```

Related concepts

[“Formats for numeric data” on page 47](#)

[“Data format conversions” on page 51](#)

[“Unicode and the encoding of language characters” on page 124](#)

Related tasks

[“Converting to or from national \(Unicode\) representation” on page 134](#)

Related references

[“Conversions and precision” on page 52](#)

[“ARITH” on page 339](#)

Converting from one code page to another

You can nest the DISPLAY-OF and NATIONAL-OF intrinsic functions to easily convert from any code page to any other code page.

For example, the following code converts an EBCDIC string to an ASCII string:

```
77 EBCDIC-CCSID PIC 9(4) BINARY VALUE 1140.  
77 ASCII-CCSID PIC 9(4) BINARY VALUE 819.  
77 Input-EBCDIC PIC X(80).  
77 ASCII-Output PIC X(80).  
.  
* Convert EBCDIC to ASCII  
  Move Function Display-of  
    (Function National-of (Input-EBCDIC EBCDIC-CCSID),  
     ASCII-CCSID)  
  to ASCII-output
```

Related concepts

[“Unicode and the encoding of language characters” on page 124](#)

Related tasks

[“Converting to or from national \(Unicode\) representation” on page 134](#)

Converting to hexadecimal or bit data (HEX-OF, BIT-OF)

You can use the HEX-OF or BIT-OF intrinsic functions to convert data of any type to hexadecimal or binary digits.

The HEX-OF intrinsic function can be used to convert data of any type to a human readable string of hexadecimal digits ("0" through "9", "A" through "F", and "a" through "f") that represent, in hexadecimal form, the underlying byte values of the data to be converted. The length of the output hex string in bytes is two times the length of the input argument string in bytes.

For example, FUNCTION HEX-OF('Hello, world!') returns 'C8859393966B40A6969993845A'.

Note: The first two hexadecimal digits 'C8' correspond to the EBCDIC encoding of the letter 'H'.

The argument to the HEX-OF intrinsic function can be a literal, a data item, or the result of an intrinsic function.

The BIT-OF intrinsic function can be used to convert data of any type to a human readable string of binary digits ("0" or "1") that represent, in bit string form, the underlying byte values of the data to be converted. The length of the output bit string in bytes is eight times the length of the input argument string in bytes.

For example, `FUNCTION BIT-OF('Hello, world!')` returns
'110010001000010110010011100100111001011001101011010000001010011010010110
10011001100100111000010001011010'.

Note: The first eight characters '11001000' of the output string correspond to the hexadecimal value x'C8', which matches the output of the `HEX-OF` intrinsic function shown above and corresponds to the EBCDIC encoding of the letter 'H'.

The argument to the `BIT-OF` intrinsic function can be a literal, a data item, or the result of an intrinsic function.

Related references

`BIT-OF` (*Enterprise COBOL for z/OS Language Reference*)

`HEX-OF` (*Enterprise COBOL for z/OS Language Reference*)

Converting from hexadecimal or bit data (HEX-TO-CHAR, BIT-TO-CHAR)

You can use the `HEX-TO-CHAR` or `BIT-TO-CHAR` intrinsic functions to convert hexadecimal character strings (consisting of characters "0" through "9", "A" through "F", and "a" through "f") or bit character strings (consisting of characters "0" and "1") to alphanumeric data items.

HEX-TO-CHAR

The `HEX-TO-CHAR` intrinsic function can be used to convert a character string consisting of hexadecimal digits ("0" through "9", "A" through "F", and "a" through "f") to an alphanumeric character string consisting of bytes that correspond to the hexadecimal digits in the input character string.

For example,

```
MOVE 'FFAABB' TO MY-HEX-DATA
```

`FUNCTION HEX-TO-CHAR(MY-HEX-DATA)` returns a character string with value x'FFAABB'.

The argument to the `HEX-TO-CHAR` intrinsic function can be an alphanumeric literal, alphanumeric data item, or alphanumeric group item. The length of the argument must be a multiple of 2 bytes.

BIT-TO-CHAR

The `BIT-TO-CHAR` intrinsic function can be used to convert a character string consisting of characters "0" and "1" to an alphanumeric character string consisting of bytes that correspond to the bit pattern indicated by the sequence of "0" and "1" characters in the input character string.

For example,

```
MOVE '1111001000000110' TO MY-BIT-DATA
```

`FUNCTION BIT-TO-CHAR(MY-BIT-DATA)` returns a character string with value x'F206'.

The argument to the `BIT-TO-CHAR` intrinsic function can be an alphanumeric literal, alphanumeric data item or alphanumeric group item. The length of the argument must be a multiple of 8 bytes.

Related references

`BIT-TO-CHAR` (*Enterprise COBOL for z/OS Language Reference*)

`HEX-TO-CHAR` (*Enterprise COBOL for z/OS Language Reference*)

Evaluating data items (intrinsic functions)

You can use intrinsic functions to determine the ordinal position of a character in the collating sequence, to find the largest or smallest item in a series, to find the length of data item, or to determine when a program was compiled.

Use these intrinsic functions:

- `CHAR` and `ORD` to evaluate integers and single alphabetic or alphanumeric characters with respect to the collating sequence used in a program

- MAX, MIN, ORD-MAX, and ORD-MIN to find the largest and smallest items in a series of data items, including USAGE NATIONAL data items
- LENGTH to find the length of data items, including USAGE NATIONAL data items, and BYTE-LENGTH to find the length of data items in bytes, including DBCS data items
- WHEN-COMPILED to find the date and time when a program was compiled

Related concepts

[“Unicode and the encoding of language characters” on page 124](#)

Related tasks

[“Evaluating single characters for collating sequence” on page 118](#)

[“Finding the largest or smallest data item” on page 118](#)

[“Finding the length of data items” on page 120](#)

[“Finding the date of compilation” on page 121](#)

Evaluating single characters for collating sequence

To find out the ordinal position of a given alphabetic or alphanumeric character in the collating sequence, use the ORD function with the character as the argument. ORD returns an integer that represents that ordinal position.

You can use a one-character substring of a data item as the argument to ORD:

```
IF Function Ord(Customer-record(1:1)) IS > 194 THEN . . .
```

If you know the ordinal position in the collating sequence of a character, and want to find the character that it corresponds to, use the CHAR function with the integer ordinal position as the argument. CHAR returns the required character. For example:

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

The ordinal number associated with a character is not the same as the numeric value (in decimal) of the hex value of the character. For example, if you are using the EBCDIC collating sequence, the ordinal number of X'00' is one instead of zero. Similarly, the ordinal number of X'FF' is 256 instead of 255. Therefore, the ordinal values returned from the ORD intrinsic function when using the EBCDIC collating sequence range from 1 - 256, not 0 - 255 that are the decimal values of the hex values of the valid EBCDIC characters.

Related references

CHAR (*Enterprise COBOL for z/OS Language Reference*)

ORD (*Enterprise COBOL for z/OS Language Reference*)

Finding the largest or smallest data item

To determine which of two or more alphanumeric, alphabetic, or national data items has the largest value, use the MAX or ORD-MAX intrinsic function. To determine which item has the smallest value, use MIN or ORD-MIN. These functions evaluate according to the collating sequence.

To compare numeric items, including those that have USAGE NATIONAL, you can use MAX, ORD-MAX, MIN, or ORD-MIN. With these intrinsic functions, the algebraic values of the arguments are compared.

The MAX and MIN functions return the content of one of the arguments that you supply. For example, suppose that your program has the following data definitions:

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

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

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

If you used MIN instead, then THOMASbbbb would be assigned.

The functions ORD-MAX and ORD-MIN return an integer that represents the ordinal position (counting from the left) of the argument that has the largest or smallest value in the list of arguments that you supply. If you used the ORD-MAX function in the previous example, the compiler would issue an error message because numeric functions are not valid as senders in MOVE statements. Using the same arguments as in the previous example, ORD-MAX can be used as follows:

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

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

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

Related tasks

[“Performing arithmetic” on page 55](#)

[“Processing table items using intrinsic functions” on page 87](#)

[“Returning variable results with alphanumeric or national functions” on page 119](#)

Related references

MAX (*Enterprise COBOL for z/OS Language Reference*)

MIN (*Enterprise COBOL for z/OS Language Reference*)

ORD-MAX (*Enterprise COBOL for z/OS Language Reference*)

ORD-MIN (*Enterprise COBOL for z/OS Language Reference*)

Returning variable results with alphanumeric or national functions

The results of alphanumeric or national functions could be of varying lengths and values depending on the function arguments.

In the following example, the amount of data moved to R3 and the results of the COMPUTE statement depend on the values and sizes of R1 and R2:

```
01 R1    Pic x(10) value "e".
01 R2    Pic x(05) value "f".
01 R3    Pic x(20) value spaces.
01 L     Pic 99.
. . .
      Move Function Max(R1 R2) to R3
      Compute L = Function Length(Function Max(R1 R2))
```

This code has the following results:

- R2 is evaluated to be larger than R1.
- The string 'fbbbb' is moved to R3, where *b* represents a blank space. (The unfilled character positions in R3 are padded with spaces.)
- L evaluates to the value 5.

If R1 contained 'g' instead of 'e', the code would have the following results:

- R1 would evaluate as larger than R2.
- The string 'gbbbbbbbbb' would be moved to R3. (The unfilled character positions in R3 would be padded with spaces.)
- The value 10 would be assigned to L.

If a program uses national data for function arguments, the lengths and values of the function results could likewise vary. For example, the following code is identical to the fragment above, but uses national data instead of alphanumeric data.

```
01 R1 Pic n(10) national value "e".
01 R2 Pic n(05) national value "f".
01 R3 Pic n(20) national value spaces.
01 L Pic 99 national.
. . .
Move Function Max(R1 R2) to R3
Compute L = Function Length(Function Max(R1 R2))
```

This code has the following results, which are similar to the first set of results except that these are for national characters:

- R2 is evaluated to be larger than R1.
- The string NX"0066 0020 0020 0020 0020" (the equivalent in national characters of 'fbbb', where *b* represents a blank space), shown here in hexadecimal notation with added spaces for readability, is moved to R3. The unfilled character positions in R3 are padded with national spaces.
- L evaluates to the value 5, the length in national character positions of R2.

You might be dealing with variable-length output from alphanumeric or national functions. Plan your program accordingly. For example, you might need to think about using variable-length files when the records that you are writing could be of different lengths:

```
File Section.
FD Output-File Recording Mode V.
01 Short-Customer-Record Pic X(50).
01 Long-Customer-Record Pic X(70).
Working-Storage Section.
01 R1 Pic x(50).
01 R2 Pic x(70).
. . .
If R1 > R2
Write Short-Customer-Record from R1
Else
Write Long-Customer-Record from R2
End-if
```

Related tasks

[“Finding the largest or smallest data item” on page 118](#)

[“Performing arithmetic” on page 55](#)

Related references

MAX (*Enterprise COBOL for z/OS Language Reference*)

Finding the length of data items

You can use the LENGTH function in many contexts (including tables and numeric data) to determine the length of an item. For example, you can use the LENGTH function to determine the length of an alphanumeric or national literal, or a data item of any type except DBCS. You can also use the BYTE-LENGTH function to determine the length of an item in bytes.

LENGTH intrinsic function

The LENGTH function returns the length of a national item (a literal, or any item that has USAGE NATIONAL, including national group items) as an integer equal to the length of the argument in national character positions. It returns the length of any other data item as an integer equal to the length of the argument in alphanumeric character positions.

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

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

BYTE-LENGTH intrinsic function

The BYTE - LENGTH function returns the length of a national item, a UTF-8 item, an alphanumeric item, or a DBCS literal as an integer that is equal to the length of the argument in bytes.

LENGTH OF special register

You can also use the LENGTH OF special register, which returns the length in bytes even for national data. Coding either `Function Length(Customer-name)` or `LENGTH OF Customer-name` returns the same result for alphanumeric items: the length of `Customer-name` in bytes. The BYTE - LENGTH function returns the same result as the LENGTH OF special register for all arguments types.

You can use the LENGTH and BYTE - LENGTH functions only where arithmetic expressions are allowed. However, you can use the LENGTH OF special register in a greater variety of contexts. For example, you can use the LENGTH OF special register as an argument to an intrinsic function that accepts integer arguments. (You cannot use an intrinsic function as an operand to the LENGTH OF special register.) You can also use the LENGTH OF special register as a parameter in a CALL statement.

Related tasks

[“Performing arithmetic” on page 55](#)

[“Creating variable-length tables \(DEPENDING ON\)” on page 78](#)

[“Processing table items using intrinsic functions” on page 87](#)

Related references

BYTE-LENGTH (*Enterprise COBOL for z/OS Language Reference*)

LENGTH (*Enterprise COBOL for z/OS Language Reference*)

LENGTH OF (*Enterprise COBOL for z/OS Language Reference*)

Finding the date of compilation

You can use the WHEN - COMPILED intrinsic function to determine when a program was compiled. The 21-character result indicates the four-digit year, month, day, and time (in hours, minutes, seconds, and hundredths of seconds) of compilation, and the difference in hours and minutes from Greenwich mean time.

The first 16 positions are in the following format:

```
YYYYMMDDhhmmsshh
```

You can instead use the WHEN - COMPILED special register to determine the date and time of compilation in the following format:

```
MM/DD/YYhh.mm.ss
```

The WHEN - COMPILED special register supports only a two-digit year, and does not carry fractions of a second. You can use this special register only as the sending field in a MOVE statement.

Related references

WHEN-COMPILED (*Enterprise COBOL for z/OS Language Reference*)

Chapter 7. Processing data in an international environment

Enterprise COBOL supports the UTF-16 and UTF-8 Unicode encodings for character data at run time. UTF-16 is a fixed-width Unicode encoding that provides a consistent and efficient way to encode plain text. Using UTF-16, you can develop software that will work with various national languages.

Use these COBOL facilities to code and compile programs that process national data:

- Data types and literals:
 - Character data types, defined with the `USAGE NATIONAL` clause and a `PICTURE` clause that defines data of category national, national-edited, or numeric-edited
 - Numeric data types, defined with the `USAGE NATIONAL` clause and a `PICTURE` clause that defines a numeric data item (a *national decimal item*) or an external floating-point data item (a *national floating-point item*)
 - National literals, specified with literal prefix N or NX
 - Figurative constant ALL *national-literal*
 - Figurative constants QUOTE, SPACE, HIGH-VALUE, LOW-VALUE, or ZERO, which have national character (UTF-16) values when used in national-character contexts
- The COBOL statements shown in the related reference below about COBOL statements and national data
- Intrinsic functions:
 - NATIONAL-OF to convert an alphanumeric or double-byte character set (DBCS) character string to `USAGE NATIONAL` (UTF-16)
 - DISPLAY-OF to convert a national character string to `USAGE DISPLAY` in a selected code page (EBCDIC, ASCII, EUC, or UTF-8)
 - The other intrinsic functions shown in the related reference below about intrinsic functions and national data
- The `GROUP-USAGE NATIONAL` clause to define groups that contain only `USAGE NATIONAL` data items and that behave like elementary category national items in most operations
- Compiler options:
 - `CODEPAGE` to specify the code page to use for alphanumeric and DBCS data in your program
 - `NSYMBOL` to control whether national or DBCS processing is used for the N symbol in literals and `PICTURE` clauses

You can also take advantage of implicit conversions of alphanumeric or DBCS data items to national representation. The compiler performs such conversions (in most cases) when you move these items to national data items, or compare these items with national data items.

UTF-8 is a variable-width Unicode encoding that is popular for data transmission and World Wide Web-related data formats such as HTML and JSON. Using UTF-8, you can develop software that will work with various national languages.

Use these COBOL facilities to code and compile programs that process UTF-8 data:

- Data types and literals:
 - Character data types, defined with the `USAGE UTF-8` clause and a `PICTURE` clause that defines data of category UTF-8
 - UTF-8 literals, specified with literal prefix U or UX

- U literals might contain a special Unicode escape sequence `\uhhhh` or `\U00hhhhh` to specify an individual Unicode code point, where `hhhh` and `hhhhhhh` are the Unicode code point value specified as a sequence of hexadecimal digits.
- Figurative constant ALL *utf-8-literal*
- Figurative constants QUOTE, SPACE, HIGH-VALUE, LOW-VALUE, or ZERO, which have UTF-8 character (UTF-8) values when used in UTF-8 character contexts
- The COBOL statements shown in the related reference about COBOL statements and UTF-8 data
 - Intrinsic functions:
 - NATIONAL-OF to convert a UTF-8 character string to USAGE NATIONAL (UTF-16)
 - DISPLAY-OF to convert a UTF-8 character string to USAGE DISPLAY in a selected code page (EBCDIC, ASCII, EUC) and to convert national data to USAGE UTF-8 (UTF-8)
 - The other intrinsic functions shown in the related reference about intrinsic functions and UTF-8 data
 - The GROUP-USAGE UTF-8 clause to define groups that contain only USAGE UTF-8 data items and that behave like elementary category UTF-8 items in most operations

You can also take advantage of implicit conversions of alphanumeric or national data items to UTF-8 representation. The compiler performs such conversions (in most cases) when you move these items to UTF-8 data items, or compare these items with UTF-8 data items.

Related concepts

[“Unicode and the encoding of language characters” on page 124](#)

[“National groups” on page 133](#)

Related tasks

[“Using national data \(Unicode\) in COBOL” on page 125](#)

[“Converting to or from national \(Unicode\) representation” on page 134](#)

[“Processing UTF-8 data using UTF-16 \(national\) data types” on page 153](#)

[“Processing Chinese GB 18030 data” on page 157](#)

[“Comparing national \(UTF-16\) data” on page 140](#)

[“Coding for use of DBCS](#)

[support” on page 158](#)

[Appendix B, “Converting double-byte character set \(DBCS\) data,” on page 789](#)

Related references

[“COBOL statements and national data” on page 127](#)

[“Intrinsic functions and national data” on page 130](#)

[“CODEPAGE” on page 343](#)

[“NSYMBOL” on page 377](#)

Classes and categories of data (*Enterprise COBOL for z/OS Language Reference*)

Data categories and PICTURE rules

(*Enterprise COBOL for z/OS Language Reference*)

MOVE statement (*Enterprise COBOL for z/OS Language Reference*)

General relation conditions (*Enterprise COBOL for z/OS Language Reference*)

Unicode and the encoding of language characters

Enterprise COBOL provides basic runtime support for Unicode, which can handle tens of thousands of characters that cover all commonly used characters and symbols in the world.

A *character set* is a defined set of characters, but is not associated with a coded representation. A *coded character set* (also referred to in this documentation as a *code page*) is a set of unambiguous rules that relate the characters of the set to their coded representation. Each code page has a name and is like a table that sets up the symbols for representing a character set; each symbol is associated with a unique

bit pattern, or *code point*. Each code page also has a *coded character set identifier (CCSID)*, which is a value from 1 to 65,536.

Unicode has several encoding schemes, called *Unicode Transformation Format (UTF)*, such as UTF-8, UTF-16, and UTF-32. Enterprise COBOL uses UTF-16 (CCSID 1200) in big-endian format as the representation for national literals and data items that have USAGE NATIONAL. Enterprise COBOL uses UTF-8 (CCSID 1208) as the representation for UTF-8 literals and data items that have USAGE UTF-8.

UTF-8 represents ASCII invariant characters a-z, A-Z, 0-9, and certain special characters such as ' @ , . + - = / * () as UX ' nn ' where X ' nn ' is the representation of the character in ASCII. UTF-16 represents these characters as NX ' 00nn '.

For example, the string 'ABC' is represented in UTF-16 as NX ' 004100420043 '. In UTF-8, 'ABC' is represented as X ' 414243 '.

One or more *encoding units* are used to represent a character from a coded character set. For UTF-16, an encoding unit takes 2 bytes of storage. Any character defined in any EBCDIC, ASCII, or EUC code page is represented in one UTF-16 encoding unit when the character is converted to the national data representation.

Cross-platform considerations: Enterprise COBOL and COBOL for AIX® support UTF-16 in big-endian format in national data. If you are porting Unicode data that is encoded in UTF-16LE representation to Enterprise COBOL from another platform, you must convert that data to UTF-16 in big-endian format to process the data as national data.

Related tasks

[“Converting to or from national \(Unicode\) representation” on page 134](#)

Related references

[“Storage of character data” on page 139](#)

Character sets and code pages (*Enterprise COBOL for z/OS Language Reference*)

Using national data (Unicode) in COBOL

In Enterprise COBOL, you can specify national (UTF-16) data in any of several ways.

These types of national data are available:

- National data items (categories national, national-edited, and numeric-edited)
- National literals
- Figurative constants as national characters
- Numeric data items (national decimal and national floating-point)

In addition, you can define national groups that contain only data items that explicitly or implicitly have USAGE NATIONAL, and that behave in the same way as elementary category national data items in most operations.

These declarations affect the amount of storage that is needed.

Related concepts

[“Unicode and the encoding of language characters” on page 124](#)

[“National groups” on page 133](#)

Related tasks

[“Defining national data items” on page 126](#)

[“Using national literals” on page 126](#)

[“Using national-character figurative constants” on page 132](#)

[“Defining national numeric data items” on page 132](#)

[“Using national groups” on page 137](#)

[“Converting to or from national \(Unicode\) representation” on page 134](#)

[“Comparing national \(UTF-16\) data” on page 140](#)

Related references

[“Storage of character data” on page 139](#)

Classes and categories of data (*Enterprise COBOL for z/OS Language Reference*)

Defining national data items

Define national data items with the USAGE NATIONAL clause to hold national (UTF-16) character strings.

You can define national data items of the following categories:

- National
- National-edited
- Numeric-edited

To define a category national data item, code a PICTURE clause that contains only one or more PICTURE symbols N.

To define a national-edited data item, code a PICTURE clause that contains at least one of each of the following symbols:

- Symbol N
- Simple insertion editing symbol B, 0, or /

To define a numeric-edited data item of class national, code a PICTURE clause that defines a numeric-edited item (for example, -\$999.99) and code a USAGE NATIONAL clause. You can use a numeric-edited data item that has USAGE NATIONAL in the same way that you use a numeric-edited item that has USAGE DISPLAY.

You can also define a data item as numeric-edited by coding the BLANK WHEN ZERO clause for an elementary item that is defined as numeric by its PICTURE clause.

If you code a PICTURE clause but do not code a USAGE clause for data items that contain only one or more PICTURE symbols N, you can use the compiler option NSYMBOL (NATIONAL) to ensure that such items are treated as national data items instead of as DBCS items.

Related tasks

[“Displaying numeric data” on page 45](#)

Related references

[“NSYMBOL” on page 377](#)

BLANK WHEN ZERO clause (*Enterprise COBOL for z/OS Language Reference*)

Using national literals

To specify national literals, use the prefix character N and compile with the option NSYMBOL (NATIONAL).

You can use either of these notations:

- N"character-data"
- N'character-data'

If you compile with the option NSYMBOL (DBCS), the literal prefix character N specifies a DBCS literal, not a national literal.

To specify a national literal as a hexadecimal value, use the prefix NX. You can use either of these notations:

- NX"hexadecimal-digits"
- NX'hexadecimal-digits'

Each of the following MOVE statements sets the national data item Y to the UTF-16 value of the characters 'AB':

```
01 Y pic NN usage national.
. . .
Move NX"00410042" to Y
Move N"AB"         to Y
Move "AB"          to Y
```

Do not use alphanumeric hexadecimal literals in contexts that call for national literals, because such usage is easily misunderstood. For example, the following statement also results in moving the UTF-16 characters 'AB' (not the hexadecimal bit pattern C1C2) to Y, where Y is defined as USAGE NATIONAL:

```
Move X"C1C2" to Y
```

You cannot use national literals in the SPECIAL-NAMES paragraph or as program-names. You can use a national literal to name an object-oriented method in the METHOD-ID paragraph or to specify a method-name in an INVOKE statement.

Related tasks

[“Using literals” on page 25](#)

Related references

[“NSYMBOL” on page 377](#)

National literals (*Enterprise COBOL for z/OS Language Reference*)

COBOL statements and national data

You can use national data with the PROCEDURE DIVISION and compiler-directing statements shown in the table below.

COBOL statement	Can be national	Comment	For more information
ACCEPT	<i>identifier-1, identifier-2</i>	<i>identifier-1</i> is converted from the native code page specified in the CODEPAGE compiler option only if input is from CONSOLE.	“Assigning input from a screen or file (ACCEPT)” on page 35
ADD	All identifiers can be numeric items that have USAGE NATIONAL. <i>identifier-3</i> (GIVING) can be numeric-edited with USAGE NATIONAL.		“Using COMPUTE and other arithmetic statements” on page 55
CALL	<i>identifier-2, identifier-3, identifier-4, identifier-5; literal-2, literal-3</i>		“Passing data” on page 551
COMPUTE	<i>identifier-1</i> can be numeric or numeric-edited with USAGE NATIONAL. <i>arithmetic-expression</i> can contain numeric items that have USAGE NATIONAL.		“Using COMPUTE and other arithmetic statements” on page 55

Table 15. **COBOL statements and national data** (continued)

COBOL statement	Can be national	Comment	For more information
COPY . . . REPLACING	<i>operand-1</i> , <i>operand-2</i> of the REPLACING phrase		Chapter 20, “Compiler-directing statements,” on page 429
DISPLAY	<i>identifier-1</i>	<i>identifier-1</i> is converted to EBCDIC only if the CONSOLE mnemonic-name is specified directly or indirectly.	“Displaying values on a screen or in a file (DISPLAY)” on page 35
DIVIDE	All identifiers can be numeric items that have USAGE NATIONAL. <i>identifier-3</i> (GIVING) and <i>identifier-4</i> (REMAINDER) can be numeric-edited with USAGE NATIONAL.		“Using COMPUTE and other arithmetic statements” on page 55
INITIALIZE	<i>identifier-1</i> ; <i>identifier-2</i> or <i>literal-1</i> of the REPLACING phrase	If you specify REPLACING NATIONAL or REPLACING NATIONAL - EDITED, <i>identifier-2</i> or <i>literal-1</i> must be valid as a sending operand in a move to <i>identifier-1</i> .	“Examples: initializing data items” on page 27
INSPECT	All identifiers and literals. (<i>identifier-2</i> , the TALLYING integer data item, can have USAGE NATIONAL.)	If any of these (other than <i>identifier-2</i> , the TALLYING identifier) have USAGE NATIONAL, all must be national.	“Tallying and replacing data items (INSPECT)” on page 112
INVOKE	Method-name as <i>identifier-2</i> or <i>literal-1</i> ; <i>identifier-3</i> or <i>literal-2</i> in the BY VALUE phrase		“Invoking methods (INVOKE)” on page 685
JSON PARSE	<i>identifier-2</i> (the target data item); <i>identifier-3</i> (the NAME identifier); <i>literal-1</i> (the NAME substitution); <i>identifier-4</i> (the SUPPRESS identifier)	<i>identifier-1</i> is not supported as a national data item.	Chapter 35, “Processing JSON input,” on page 597
MERGE	Merge keys	The COLLATING SEQUENCE phrase does not apply.	“Setting sort or merge criteria” on page 229
MOVE	Both the sender and receiver, or only the receiver	Implicit conversions are performed for valid MOVE operands.	“Assigning values to elementary data items (MOVE)” on page 32 “Assigning values to group data items (MOVE)” on page 33

Table 15. **COBOL statements and national data** (continued)

COBOL statement	Can be national	Comment	For more information
MULTIPLY	All identifiers can be numeric items that have USAGE NATIONAL. <i>identifier-3</i> (GIVING) can be numeric-edited with USAGE NATIONAL.		“Using COMPUTE and other arithmetic statements” on page 55
SEARCH ALL (binary search)	Both the key data item and its object of comparison	The key data item and its object of comparison must be compatible according to the rules of comparison. If the object of comparison is of class national, the key must be also.	“Doing a binary search (SEARCH ALL)” on page 86
SORT	Sort keys	The COLLATING SEQUENCE phrase does not apply.	“Setting sort or merge criteria” on page 229
STRING	All identifiers and literals. (<i>identifier-4</i> , the POINTER integer data item, can have USAGE NATIONAL.)	If <i>identifier-3</i> , the receiving data item, is national, all identifiers and literals (other than <i>identifier-4</i> , the POINTER identifier) must be national.	“Joining data items (STRING)” on page 103
SUBTRACT	All identifiers can be numeric items that have USAGE NATIONAL. <i>identifier-3</i> (GIVING) can be numeric-edited with USAGE NATIONAL.		“Using COMPUTE and other arithmetic statements” on page 55
UNSTRING	All identifiers and literals. (<i>identifier-6</i> and <i>identifier-7</i> , the COUNT and TALLYING integer data items, respectively, can have USAGE NATIONAL.)	If <i>identifier-4</i> , a receiving data item, has USAGE NATIONAL, the sending data item and each delimiter must have USAGE NATIONAL, and each literal must be national.	“Splitting data items (UNSTRING)” on page 105
XML GENERATE	<i>identifier-1</i> (the generated XML document); <i>identifier-2</i> (the source field or fields); <i>identifier-4</i> or <i>literal-4</i> (the namespace identifier); <i>identifier-5</i> or <i>literal-5</i> (the namespace prefix)		Chapter 38, “Producing XML output,” on page 651

Table 15. **COBOL statements and national data** (continued)

COBOL statement	Can be national	Comment	For more information
XML PARSE	<i>identifier-1</i> (the XML document)	The XML -NTEXT special register contains national character document fragments during parsing. XML -NNAMESPACE and XML -NNAMESPACE -PREFIX special registers contain the associated namespace identifier and namespace prefix, if any, in national characters.	Chapter 37, “Processing XML input,” on page 611

Related tasks

- “Defining numeric data” on page 43
- “Displaying numeric data” on page 45
- “Using national data (Unicode) in COBOL” on page 125
- “Comparing national (UTF-16) data” on page 140

Related references

- “CODEPAGE” on page 343
- Classes and categories of data (*Enterprise COBOL for z/OS Language Reference*)

Intrinsic functions and national data

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

Table 16. **Intrinsic functions and national character data**

Intrinsic function	Function type	For more information
BIT-OF	Alphanumeric	BIT-OF (<i>Enterprise COBOL for z/OS Language Reference</i>)
BYTE-LENGTH	Integer	“Finding the length of data items” on page 120
CONTENT-OF	Alphanumeric, national, UTF-8, integer, or numeric	CONTENT-OF (<i>Enterprise COBOL for z/OS Language Reference</i>)
DISPLAY-OF	Alphanumeric	“Converting national to alphanumeric (DISPLAY-OF)” on page 135
FORMATTED-CURRENT-DATE, FORMATTED-DATE, FORMATTED-TIME, FORMATTED-DATETIME	Alphanumeric, national, or UTF-8	<ul style="list-style-type: none"> • FORMATTED-CURRENT-DATE (<i>Enterprise COBOL for z/OS Language Reference</i>) • FORMATTED-DATE (<i>Enterprise COBOL for z/OS Language Reference</i>) • FORMATTED-TIME (<i>Enterprise COBOL for z/OS Language Reference</i>) • FORMATTED-DATETIME (<i>Enterprise COBOL for z/OS Language Reference</i>)
HEX-OF	Alphanumeric	HEX-OF (<i>Enterprise COBOL for z/OS Language Reference</i>)
INTEGER-OF-FORMATTED-DATE	Integer	INTEGRE-OF-FORMATTED-DATE (<i>Enterprise COBOL for z/OS Language Reference</i>)

<i>Table 16. Intrinsic functions and national character data (continued)</i>		
Intrinsic function	Function type	For more information
LENGTH	Integer	“Finding the length of data items” on page 120
LOWER-CASE, UPPER-CASE	National	“Changing case (UPPER-CASE, LOWER-CASE)” on page 114
NUMVAL, NUMVAL-C, NUMVAL-F	Numeric	“Converting to numbers (NUMVAL, NUMVAL-C, NUMVAL-F)” on page 115
MAX, MIN	National	“Finding the largest or smallest data item” on page 118
ORD-MAX, ORD-MIN	Integer	“Finding the largest or smallest data item” on page 118
REVERSE	Alphanumeric or national	REVERSE (<i>Enterprise COBOL for z/OS Language Reference</i>)
SECONDS-FORM-FORMATTED-TIME	Numeric	SECONDS-FROM-FORMATTED-TIME (<i>Enterprise COBOL for z/OS Language Reference</i>)
TEST-FORMATTED-DATETIME	Integer	TEST-FORMATTED-DATETIME (<i>Enterprise COBOL for z/OS Language Reference</i>)
TEST-NUMVAL, TEST-NUMVAL-C, TEST-NUMVAL-F	Integer	<ul style="list-style-type: none"> • TEST-NUMVAL (<i>Enterprise COBOL for z/OS Language Reference</i>) • TEST-NUMVAL-C (<i>Enterprise COBOL for z/OS Language Reference</i>) • TEST-NUMVAL-F (<i>Enterprise COBOL for z/OS Language Reference</i>)
TRIM	Alphanumeric or national	TRIM (<i>Enterprise COBOL for z/OS Language Reference</i>)
ULENGTH	Integer	ULENGTH (<i>Enterprise COBOL for z/OS Language Reference</i>)
UPOS	Integer	UPOS (<i>Enterprise COBOL for z/OS Language Reference</i>)
USUBSTR	Alphanumeric or national	USUBSTR (<i>Enterprise COBOL for z/OS Language Reference</i>)
USUPPLEMENTARY	Integer	USUPPLEMENTARY (<i>Enterprise COBOL for z/OS Language Reference</i>)
UVALID	Integer	UVALID (<i>Enterprise COBOL for z/OS Language Reference</i>)
UWIDTH	Integer	UWIDTH (<i>Enterprise COBOL for z/OS Language Reference</i>)

You can use national decimal arguments wherever zoned decimal arguments are allowed. You can use national floating-point arguments wherever display floating-point arguments are allowed. (See the related reference below about arguments for a complete list of intrinsic functions that can take integer or numeric arguments.)

Related tasks

[“Defining numeric data” on page 43](#)

[“Using national data \(Unicode\) in COBOL” on page 125](#)

Related references

Arguments (*Enterprise COBOL for z/OS Language Reference*)

Classes and categories of data (*Enterprise COBOL for z/OS Language Reference*)

Intrinsic functions (*Enterprise COBOL for z/OS Language Reference*)

Using national-character figurative constants

You can use the figurative constant ALL *national-literal* in a context that requires national characters. ALL *national-literal* represents all or part of the string that is generated by successive concatenations of the encoding units that make up the national literal.

You can use the figurative constants QUOTE, SPACE, HIGH-VALUE, LOW-VALUE, or ZERO in a context that requires national characters, such as a MOVE statement, an implicit move, or a relation condition that has national operands. In these contexts, the figurative constant represents a national-character (UTF-16) value.

When you use the figurative constant QUOTE in a context that requires national characters, and the QUOTE compiler option is in effect, its value is NX'0022'. If the APOST compiler option is in effect, its value is NX'0027'.

When you use the figurative constant HIGH-VALUE in a context that requires national characters, its value is NX'FFFF'. When you use LOW-VALUE in a context that requires national characters, its value is NX'0000'.

Restrictions: You must not use HIGH-VALUE or the value assigned from HIGH-VALUE in a way that results in conversion of the value from one data representation to another (for example, between USAGE DISPLAY and USAGE NATIONAL). X'FF' (the value of HIGH-VALUE in an alphanumeric context when the EBCDIC collating sequence is being used) does not represent a valid EBCDIC character, and NX'FFFF' does not represent a valid national character. Conversion of such a value to another representation results in a *substitution character* being used (not X'FF' or NX'FFFF'). Consider the following example:

```
01 natl-data PIC NN Usage National.
01 alph-data PIC XX.
. . .
MOVE HIGH-VALUE TO natl-data, alph-data
IF natl-data = alph-data. . .
```

The IF statement above evaluates as false even though each of its operands was set to HIGH-VALUE. Before an elementary alphanumeric operand is compared to a national operand, the alphanumeric operand is treated as though it were moved to a temporary national data item, and the alphanumeric characters are converted to the corresponding national characters. When X'FF' is converted to UTF-16, however, the UTF-16 item gets a substitution character value and so does not compare equally to NX'FFFF'.

Related tasks

[“Converting to or from national \(Unicode\) representation” on page 134](#)

[“Comparing national \(UTF-16\) data” on page 140](#)

Related references

Figurative constants (*Enterprise COBOL for z/OS Language Reference*)

DISPLAY-OF (*Enterprise COBOL for z/OS Language Reference*)

Support for Unicode: Using Unicode Services

Defining national numeric data items

Define data items with the USAGE NATIONAL clause to hold numeric data that is represented in national characters (UTF-16). You can define national decimal items and national floating-point items.

To define a national decimal item, code a PICTURE clause that contains only the symbols 9, P, S, and V. If the PICTURE clause contains S, the SIGN IS SEPARATE clause must be in effect for that item.

To define a national floating-point item, code a PICTURE clause that defines a floating-point item (for example, +99999.9E-99).

You can use national decimal items in the same way that you use zoned decimal items. You can use national floating-point items in the same way that you use display floating-point items.

Related tasks

[“Defining numeric data” on page 43](#)

[“Displaying numeric data” on page 45](#)

Related references

SIGN clause (*Enterprise COBOL for z/OS Language Reference*)

National groups

National groups, which are specified either explicitly or implicitly with the GROUP-USAGE NATIONAL clause, contain only data items that have USAGE NATIONAL. In most cases, a national group item is processed as though it were redefined as an elementary category national item described as PIC N(*m*), where *m* is the number of national (UTF-16) characters in the group.

For some operations on national groups, however (just as for some operations on alphanumeric groups), group semantics apply. Such operations (for example, MOVE CORRESPONDING and INITIALIZE) recognize or process the elementary items within the national group.

Where possible, use national groups instead of alphanumeric groups that contain USAGE NATIONAL items. National groups provide several advantages for the processing of national data compared to the processing of national data within alphanumeric groups:

- When you move a national group to a longer data item that has USAGE NATIONAL, the receiving item is padded with national characters. By contrast, if you move an alphanumeric group that contains national characters to a longer alphanumeric group that contains national characters, alphanumeric spaces are used for padding. As a result, mishandling of data items could occur.
- When you move a national group to a shorter data item that has USAGE NATIONAL, the national group is truncated at national-character boundaries. By contrast, if you move an alphanumeric group that contains national characters to a shorter alphanumeric group that contains national characters, truncation might occur between the 2 bytes of a national character.
- When you move a national group to a national-edited or numeric-edited item, the content of the group is edited. By contrast, if you move an alphanumeric group to an edited item, no editing takes place.
- When you use a national group as an operand in a STRING, UNSTRING, or INSPECT statement:
 - The group content is processed as national characters rather than as single-byte characters.
 - TALLYING and POINTER operands operate at the logical level of national characters.
 - The national group operand is supported with a mixture of other national operand types.

By contrast, if you use an alphanumeric group that contains national characters in these contexts, the characters are processed byte by byte. As a result, invalid handling or corruption of data could occur.

USAGE NATIONAL groups: A group item can specify the USAGE NATIONAL clause at the group level as a convenient shorthand for the USAGE of each of the elementary data items within the group. Such a group is *not* a national group, however, but an alphanumeric group, and behaves in many operations, such as moves and compares, like an elementary data item of USAGE DISPLAY (except that no editing or conversion of data occurs).

Related tasks

[“Assigning values to group data items \(MOVE\)” on page 33](#)

[“Joining data items \(STRING\)” on page 103](#)

[“Splitting data items \(UNSTRING\)” on page 105](#)

[“Tallying and replacing data items \(INSPECT\)” on page 112](#)

[“Using national groups” on page 137](#)

Related references

GROUP-USAGE clause (*Enterprise COBOL for z/OS Language Reference*)

Converting to or from national (Unicode) representation

You can implicitly or explicitly convert data items to national (UTF-16) representation.

You can implicitly convert alphabetic, alphanumeric, UTF-8, DBCS, or integer data to national data by using the MOVE statement. Implicit conversions also take place in other COBOL statements, such as IF statements that compare an alphanumeric data item with a data item that has USAGE NATIONAL.

You can explicitly convert to and from national data items by using the intrinsic functions NATIONAL-OF and DISPLAY-OF, respectively. By using these intrinsic functions, you can specify a code page for the conversion that is different from the code page that is in effect with the CODEPAGE compiler option.

Related tasks

[“Converting alphanumeric, DBCS, and integer to national \(MOVE\)” on page 134](#)

[“Converting alphanumeric or DBCS to national \(NATIONAL-OF\)” on page 135](#)

[“Converting national to alphanumeric \(DISPLAY-OF\)” on page 135](#)

[“Overriding the default code page” on page 135](#)

[“Comparing national \(UTF-16\) data” on page 140](#)

Related references

[“CODEPAGE” on page 343](#)

[“Conversion exceptions” on page 136](#)

Converting alphanumeric, DBCS, and integer to national (MOVE)

You can use a MOVE statement to implicitly convert data to national representation.

You can move the following kinds of data to category national or national-edited data items, and thus convert the data to national representation:

- Alphabetic
- Alphanumeric
- Alphanumeric-edited
- UTF-8
- DBCS
- Integer of USAGE DISPLAY
- Numeric-edited of USAGE DISPLAY

You can likewise move the following kinds of data to numeric-edited data items that have USAGE NATIONAL:

- Alphanumeric
- Display floating-point (floating-point of USAGE DISPLAY)
- Numeric-edited of USAGE DISPLAY
- Integer of USAGE DISPLAY

For complete rules about moves to national data, see the related reference about the MOVE statement.

For example, the MOVE statement below moves the alphanumeric literal "AB" to the national data item UTF16-Data:

```
01 UTF16-Data Pic N(2) Usage National.  
   Move "AB" to UTF16-Data
```

After the MOVE statement above, UTF16-Data contains NX '00410042', the national representation of the alphanumeric characters 'AB'.

If padding is required in a receiving data item that has USAGE NATIONAL, the default UTF-16 space character (NX '0020') is used. If truncation is required, it occurs at the boundary of a national-character position.

Related tasks

[“Assigning values to elementary data items \(MOVE\)” on page 32](#)

[“Assigning values to group data items \(MOVE\)” on page 33](#)

[“Displaying numeric data” on page 45](#)

[“Coding for use of DBCS support” on page 158](#)

Related references

MOVE statement (*Enterprise COBOL for z/OS Language Reference*)

Converting alphanumeric or DBCS to national (NATIONAL-OF)

Use the NATIONAL-OF intrinsic function to convert alphabetic, alphanumeric, UTF-8, or DBCS data to a national data item. Specify the source code page as the second argument if the source is encoded in a different code page than is in effect with the CODEPAGE compiler option.

[“Example: converting to and from national data” on page 136](#)

Related tasks

[“Processing UTF-8 data using UTF-16 \(national\) data types” on page 153](#)

[“Processing Chinese GB 18030 data” on page 157](#)

[“Processing alphanumeric data items that contain DBCS data” on page 160](#)

Related references

[“CODEPAGE” on page 343](#)

NATIONAL-OF (*Enterprise COBOL for z/OS Language Reference*)

Converting national to alphanumeric (DISPLAY-OF)

Use the DISPLAY-OF intrinsic function to convert national data to an alphanumeric (USAGE DISPLAY) character string that is represented in a code page that you specify as the second argument.

If you omit the second argument, the output code page is the one that was in effect with the CODEPAGE compiler option when the source was compiled.

If you specify an EBCDIC or ASCII code page that combines single-byte character set (SBCS) and DBCS characters, the returned string might contain a mixture of SBCS and DBCS characters. The DBCS substrings are delimited by shift-in and shift-out characters if the code page in effect for the function is an EBCDIC code page.

[“Example: converting to and from national data” on page 136](#)

Related tasks

[“Processing UTF-8 data using UTF-16 \(national\) data types” on page 153](#)

[“Processing Chinese GB 18030 data” on page 157](#)

Related references

DISPLAY-OF (*Enterprise COBOL for z/OS Language Reference*)

Overriding the default code page

In some cases, you might need to convert data to or from a code page that differs from the CCSID that is specified as the CODEPAGE option value. To do so, convert the item by using a conversion function in which you explicitly specify the code page.

If you specify a code page as an argument to the DISPLAY-OF intrinsic function, and the code page differs from the code page that is in effect with the CODEPAGE compiler option, do not use the function result in any operations that involve implicit conversion (such as an assignment to, or comparison with, a national data item). Such operations assume the EBCDIC code page that is specified with the CODEPAGE compiler option.

Related references

[“CODEPAGE” on page 343](#)

Conversion exceptions

Implicit or explicit conversion between national data and alphanumeric data can fail and generate a severity-3 Language Environment condition.

Failure can occur if the code page that you specified implicitly or explicitly is not a valid code page.

A character that does not have a counterpart in the target CCSID does not result in a conversion exception. Such a character is converted to a *substitution character* in the target code page.

Related references

[“CODEPAGE” on page 343](#)

Example: converting to and from national data

The following example shows the NATIONAL-OF and DISPLAY-OF intrinsic functions and the MOVE statement for converting to and from national (UTF-16) data items. It also demonstrates the need for explicit conversions when you operate on strings that are encoded in multiple code pages.

```
CBL CODEPAGE(00037)
* . . .
01 Data-in-Unicode          pic N(100) usage national.
01 Data-in-Greek           pic X(100).
01 other-data-in-US-English pic X(12) value "PRICE in $ =".
* . . .
  Read Greek-file into Data-in-Greek
  Move function National-of(Data-in-Greek, 00875)
    to Data-in-Unicode
* . . . process Data-in-Unicode here . . .
  Move function Display-of(Data-in-Unicode, 00875)
    to Data-in-Greek
  Write Greek-record from Data-in-Greek
```

The example above works correctly because the input code page is specified. Data-in-Greek is converted as data represented in CCSID 00875 (Greek). However, the following statement results in an incorrect conversion unless all the characters in the item happen to be among those that have a common representation in both the Greek and the English code pages:

```
Move Data-in-Greek to Data-in-Unicode
```

The MOVE statement above converts Data-in-Greek to Unicode representation based on the CCSID 00037 (U.S. English) to UTF-16 conversion. This conversion does not produce the expected results because Data-in-Greek is encoded in CCSID 00875.

If you can correctly set the CODEPAGE compiler option to CCSID 00875 (that is, the rest of your program also handles EBCDIC data in Greek), you can code the same example correctly as follows:

```
CBL CODEPAGE(00875)
* . . .
01 Data-in-Unicode pic N(100) usage national.
01 Data-in-Greek   pic X(100).
* . . .
  Read Greek-file into Data-in-Greek
* . . . process Data-in-Greek here ...
* . . . or do the following (if need to process data in Unicode):
```

```

Move Data-in-Greek to Data-in-Unicode
* . . . process Data-in-Unicode
Move function Display-of(Data-in-Unicode) to Data-in-Greek
Write Greek-record from Data-in-Greek

```

Using national groups

To define a group data item as a national group, code a `GROUP-USAGE NATIONAL` clause at the group level for the item. The group can contain only data items that explicitly or implicitly have `USAGE NATIONAL`.

The following data description entry specifies that a level-01 group and its subordinate groups are national group items:

```

01 Nat-Group-1    GROUP-USAGE NATIONAL.
  02 Group-1.
    04 Month      PIC 99.
    04 DayOf      PIC 99.
    04 Year        PIC 9999.
  02 Group-2     GROUP-USAGE NATIONAL.
    04 Amount     PIC 9(4).99 USAGE NATIONAL.

```

In the example above, `Nat-Group-1` is a national group, and its subordinate groups `Group-1` and `Group-2` are also national groups. A `GROUP-USAGE NATIONAL` clause is implied for `Group-1`, and `USAGE NATIONAL` is implied for the subordinate items in `Group-1`. `Month`, `DayOf`, and `Year` are national decimal items, and `Amount` is a numeric-edited item that has `USAGE NATIONAL`.

You can subordinate national groups within alphanumeric groups as in the following example:

```

01 Alpha-Group-1.
  02 Group-1.
    04 Month      PIC 99.
    04 DayOf      PIC 99.
    04 Year        PIC 9999.
  02 Group-2     GROUP-USAGE NATIONAL.
    04 Amount     PIC 9(4).99.

```

In the example above, `Alpha-Group-1` and `Group-1` are alphanumeric groups; `USAGE DISPLAY` is implied for the subordinate items in `Group-1`. (If `Alpha-Group-1` specified `USAGE NATIONAL` at the group level, `USAGE NATIONAL` would be implied for each of the subordinate items in `Group-1`. However, `Alpha-Group-1` and `Group-1` would be alphanumeric groups, not national groups, and would behave like alphanumeric groups during operations such as moves and compares.) `Group-2` is a national group, and `USAGE NATIONAL` is implied for the numeric-edited item `Amount`.

You cannot subordinate alphanumeric groups within national groups. All elementary items within a national group must be explicitly or implicitly described as `USAGE NATIONAL`, and all group items within a national group must be explicitly or implicitly described as `GROUP-USAGE NATIONAL`.

Related concepts

[“National groups” on page 133](#)

Related tasks

[“Using national groups as elementary items” on page 138](#)

[“Using national groups as group items” on page 138](#)

Related references

`GROUP-USAGE` clause (*Enterprise COBOL for z/OS Language Reference*)

Using national groups as elementary items

In most cases, you can use a national group as though it were an elementary data item.

In the following example, a national group item, Group-1, is moved to a national-edited item, Edited-date. Because Group-1 is treated as an elementary data item during the move, editing takes place in the receiving data item. The value in Edited-date after the move is 06/23/2010 in national characters.

```
01 Edited-date PIC NN/NN/NNNN USAGE NATIONAL.
01 Group-1     GROUP-USAGE NATIONAL.
   02 Month    PIC 99 VALUE 06.
   02 DayOf    PIC 99 VALUE 23.
   02 Year     PIC 9999 VALUE 2010.
.
.
MOVE Group-1 to Edited-date.
```

If Group-1 were instead an alphanumeric group in which each of its subordinate items had USAGE NATIONAL (specified either explicitly with a USAGE NATIONAL clause on each elementary item, or implicitly with a USAGE NATIONAL clause at the group level), a group move, rather than an elementary move, would occur. Neither editing nor conversion would take place during the move. The value in the first eight character positions of Edited-date after the move would be 06232010 in national characters, and the value in the remaining two character positions would be 4 bytes of alphanumeric spaces.

Related tasks

[“Assigning values to group data items \(MOVE\)” on page 33](#)

[“Comparing national data and alphanumeric-group operands” on page 142](#)

[“Using national groups as group items” on page 138](#)

Related references

MOVE statement (*Enterprise COBOL for z/OS Language Reference*)

Using national groups as group items

In some cases when you use a national group, it is handled with group semantics; that is, the elementary items in the group are recognized or processed.

In the following example, an INITIALIZE statement that acts upon national group item Group-OneN causes the value 15 in national characters to be moved to only the numeric items in the group:

```
01 Group-OneN Group-Usage National.
   05 Trans-codeN Pic N Value "A".
   05 Part-numberN Pic NN Value "XX".
   05 Trans-quantN Pic 99 Value 10.
.
.
Initialize Group-OneN Replacing Numeric Data By 15
```

Because only Trans-quantN in Group-OneN above is numeric, only Trans-quantN receives the value 15. The other subordinate items are unchanged.

The table below summarizes the cases where national groups are processed with group semantics.

Language feature	Uses of national group items	Comment
CORRESPONDING phrase of the ADD, SUBTRACT, or MOVE statement	Specify a national group item for processing as a group in accordance with the rules of the CORRESPONDING phrase.	Elementary items within the national group are processed like elementary items that have USAGE NATIONAL within an alphanumeric group.

Table 17. **National group items that are processed with group semantics** (continued)

Language feature	Uses of national group items	Comment
Host variable in EXEC SQL statement	Specify a national group item as a host variable.	The national group item is in effect shorthand for the set of host variables that are subordinate to the group item.
INITIALIZE statement	Specify a national group for processing as a group in accordance with the rules of the INITIALIZE statement.	Elementary items within the national group are initialized like elementary items that have USAGE NATIONAL within an alphanumeric group.
Name qualification	Use the name of a national group item to qualify the names of elementary data items and of subordinate group items in the national group.	Follow the same rules for qualification as for an alphanumeric group.
THROUGH phrase of the RENAMES clause	To specify a national group item in the THROUGH phrase, use the same rules as for an alphanumeric group item.	The result is an alphanumeric group item.
FROM phrase of the XML GENERATE statement	Specify a national group item in the FROM phrase for processing as a group in accordance with the rules of the XML GENERATE statement.	Elementary items within the national group are processed like elementary items that have USAGE NATIONAL within an alphanumeric group.

Related tasks

- [“Initializing a structure \(INITIALIZE\)” on page 31](#)
- [“Initializing a table \(INITIALIZE\)” on page 73](#)
- [“Assigning values to elementary data items \(MOVE\)” on page 32](#)
- [“Assigning values to group data items \(MOVE\)” on page 33](#)
- [“Finding the length of data items” on page 120](#)
- [“Generating XML output” on page 651](#)
- [“Using national group items in SQL statements” on page 496](#)

Related references

- Qualification (*Enterprise COBOL for z/OS Language Reference*)
- RENAMES clause (*Enterprise COBOL for z/OS Language Reference*)

Storage of character data

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

Table 18. **Encoding and size of alphanumeric, DBCS, and national data**

Characteristic	DISPLAY	DISPLAY-1	NATIONAL
Character encoding unit	1 byte	2 bytes	2 bytes
Code page	EBCDIC	EBCDIC DBCS	UTF-16BE ¹
Encoding units per graphic character	1	1	1 or 2 ²
Bytes per graphic character	1 byte	2 bytes	2 or 4 bytes

Table 18. **Encoding and size of alphanumeric, DBCS, and national data** (continued)

Characteristic	DISPLAY	DISPLAY-1	NATIONAL
1. Use the CODEPAGE compiler option to specify the EBCDIC code page that is applicable to alphanumeric or DBCS data.			
2. Most characters are represented in UTF-16 using one encoding unit. In particular, the following characters are represented using a single UTF-16 encoding unit per character: <ul style="list-style-type: none"> • COBOL characters A-Z, a-z, 0-9, space, + - * / = \$, ; . " () > < :' • All characters that are converted from an EBCDIC or ASCII code page 			

Related concepts

[“Unicode and the encoding of language characters” on page 124](#)

Comparing national (UTF-16) data

You can compare national (UTF-16) data, that is, national literals and data items that have USAGE NATIONAL (whether of class national or class numeric), explicitly or implicitly with other kinds of data in relation conditions.

You can code conditional expressions that use national data in the following statements:

- EVALUATE
- IF
- INSPECT
- PERFORM
- SEARCH
- STRING
- UNSTRING

For full details about comparing national data items to other data items, see the Related references.

Related tasks

[“Comparing two class national operands” on page 140](#)

[“Comparing class national and class numeric operands” on page 141](#)

[“Comparing national numeric and other numeric operands” on page 141](#)

[“Comparing national and other character-string operands” on page 141](#)

[“Comparing national data and alphanumeric-group operands” on page 142](#)

Related references

Relation conditions (*Enterprise COBOL for z/OS Language Reference*)

General relation conditions (*Enterprise COBOL for z/OS Language Reference*)

National comparisons (*Enterprise COBOL for z/OS Language Reference*)

Group comparisons (*Enterprise COBOL for z/OS Language Reference*)

Comparing two class national operands

You can compare the character values of two operands of class national.

Either operand (or both) can be any of the following types of items:

- A national group
- An elementary category national or national-edited data item
- A numeric-edited data item that has USAGE NATIONAL

One of the operands can instead be a national literal or a national intrinsic function.

When you compare two class national operands that have the same length, they are determined to be equal if all pairs of the corresponding characters are equal. Otherwise, comparison of the binary values of the first pair of unequal characters determines the operand with the larger binary value.

When you compare operands that have unequal lengths, the shorter operand is treated as if it were padded on the right with default UTF-16 space characters (NX '0020 ') to the length of the longer operand.

The PROGRAM COLLATING SEQUENCE clause does not affect the comparison of two class national operands.

Related concepts

[“National groups” on page 133](#)

Related tasks

[“Using national groups” on page 137](#)

Related references

National comparisons (*Enterprise COBOL for z/OS Language Reference*)

Comparing class national and class numeric operands

You can compare national literals or class national data items to integer literals or numeric data items that are defined as integer (that is, national decimal items or zoned decimal items). At most one of the operands can be a literal.

You can also compare national literals or class national data items to floating-point data items (that is, display floating-point or national floating-point items).

Numeric operands are converted to national (UTF-16) representation if they are not already in national representation. A comparison is made of the national character values of the operands.

Related references

General relation conditions (*Enterprise COBOL for z/OS Language Reference*)

Comparing national numeric and other numeric operands

National numeric operands (national decimal and national floating-point operands) are data items of class numeric that have USAGE NATIONAL.

You can compare the algebraic values of numeric operands regardless of their USAGE. Thus you can compare a national decimal item or a national floating-point item with a binary item, an internal-decimal item, a zoned decimal item, a display floating-point item, or any other numeric item.

Related tasks

[“Defining national numeric data items” on page 132](#)

Related references

General relation conditions (*Enterprise COBOL for z/OS Language Reference*)

Comparing national and other character-string operands

You can compare the character value of a national literal or class national data item with the character value of any of the following other character-string operands: alphabetic, alphanumeric, alphanumeric-edited, UTF-8, DBCS, or numeric-edited of USAGE DISPLAY.

These operands are treated as if they were moved to an elementary national data item. The characters are converted to national (UTF-16) representation, and the comparison proceeds with two national character operands.

Related tasks

[“Using national-character figurative constants” on page 132](#)

Related references

National comparisons (*Enterprise COBOL for z/OS Language Reference*)

Comparing national data and alphanumeric-group operands

You can compare a national literal, a national group item, or any elementary data item that has USAGE NATIONAL to an alphanumeric group.

Neither operand is converted. The national operand is treated as if it were moved to an alphanumeric group item of the same size in bytes as the national operand, and the two groups are compared. An alphanumeric comparison is done regardless of the representation of the subordinate items in the alphanumeric group operand.

For example, Group-XN is an alphanumeric group that consists of two subordinate items that have USAGE NATIONAL:

```
01 Group-XN.  
   02 TransCode PIC NN   Value "AB"   Usage National.  
   02 Quantity  PIC 999  Value 123    Usage National.  
  
   If N"AB123" = Group-XN Then Display "EQUAL"  
   Else Display "NOT EQUAL".
```

When the IF statement above is executed, the 10 bytes of the national literal N"AB123" are compared byte by byte to the content of Group-XN. The items compare equally, and "EQUAL" is displayed.

Related references

Group comparisons (*Enterprise COBOL for z/OS Language Reference*)

Using UTF-8 data (Unicode) in COBOL

UTF-8 is a variable-width Unicode encoding that encodes each valid Unicode code point using one to four 8-bit bytes. UTF-8 has many desirable properties, including that it is backwards compatible with ASCII, often provides a more compact representation of Unicode data than UTF-16, and is endianness independent. UTF-8 is the preferred encoding of HTML and related languages and it is by far the most common encoding of data on the World Wide Web.

For details about the UTF-8 support in Enterprise COBOL, see the related references.

Related references

[“Defining UTF-8 data items” on page 142](#)

[“Using UTF-8 literals” on page 144](#)

[“COBOL statements and UTF-8 data items” on page 145](#)

[“Intrinsic functions support for UTF-8 data items” on page 146](#)

[“Db2 considerations” on page 147](#)

Defining UTF-8 data items

Enterprise COBOL provides native support for defining, moving, and comparing UTF-8 data items.

Three different types of UTF-8 data items are supported in Enterprise COBOL for z/OS:

- **Fixed character-length UTF-8 data items**

This type of UTF-8 data item is defined when the PICTURE clause contains one or more 'U' characters, or a single 'U' character followed by a repetition factor, and neither the BYTE - LENGTH phrase of the PICTURE clause nor the DYNAMIC LENGTH clause is specified.

Each 'U' character in the PICTURE clause corresponds to one UTF-8 character, which in Enterprise COBOL is treated as the equivalent of a single Unicode code point. The UTF-8 encoding of a character varies in length and it is always between one and four bytes.

The following code fragment illustrates two different fixed character-length UTF-8 data item definitions:

```
01 u1 pic u(10).  *> fixed character-length UTF-8 data item holding 10 UTF-8 characters (40
bytes reserved)
01 u2 pic uuuu.  *> fixed character-length UTF-8 data item holding 4 UTF-8 characters (16
bytes reserved)
```

For fixed character-length UTF-8 data items, the number of bytes reserved for the data item in memory is $4 \times n$, where n is the number of characters specified in the definition of the item. Note that, due to the varying length nature of the UTF-8 encoding, even after moving n characters to a UTF-8 data item of length n , it is not necessarily the case that all $4 \times n$ reserved bytes are needed to hold the data. It depends on the size of each character in the data.

During moves, the fixed character-length UTF-8 data items are always padded with UTF-8 blanks (x'20') to the maximum byte-length of the data item. When truncation is performed on the fixed character-length UTF-8 data item, it is done on a character boundary.

Whenever a fixed character-length UTF-8 data item is used as a sender, the byte-length of the item is computed at run time based on its known fixed-character length so that the number of characters used in the operation is the same as the number of characters indicated in the item definition.

- **Fixed byte-length UTF-8 data items**

This type of UTF-8 data item is defined when the PICTURE clause contains a single 'U' character and the BYTE - LENGTH phrase of the PICTURE clause is specified. This phrase indicates that the data item is a fixed byte-length UTF-8 item consisting of exactly n valid UTF-8 bytes. Note that, due to the varying length nature of the UTF-8 encoding, the number of characters in the data item at any time is variable and depends on the size of each character, but should always be in the range $[\text{ceil}(4/n), n]$, where ceil is the mathematical function that returns the least integer greater than its argument.

The following code fragment illustrates two different fixed byte-length UTF-8 data item definitions:

```
01 u1 pic u byte-length 10.  *> fixed byte-length UTF-8 data item
```

When truncation is needed for a fixed byte-length UTF-8 data item, it is done at a character boundary and the data item is always padded out with UTF-8 spaces (x'20') to the specified byte length of n .

Whenever a fixed byte-length UTF-8 data item is used as a sender, the byte-length of the item is always taken to be n .

Fixed byte-length UTF-8 data items are provided for compatibility with the Db2[®], DFSORT, and MQ products, all of which provide support for fixed byte-length UTF-8 data strings. For example, to use a COBOL UTF-8 data item as a Db2 host variable that corresponds to a CHAR(n) column in a Unicode table, the data item must contain the BYTE - LENGTH phrase of the PICTURE clause. Similarly, DFSORT only supports UTF-8 sort key parts that have a fixed byte-length and therefore fixed byte-length UTF-8 data items in COBOL are strongly recommended for sort and merge keys.

- **Dynamic-length UTF-8 data items**

UTF-8 data items can be declared with the DYNAMIC LENGTH clause, which is a natural fit for UTF-8 data items since they vary in byte length. In this case, there is no restriction on the number of characters in the data item and the number of bytes is only limited when there is a LIMIT phrase of the DYNAMIC LENGTH clause.

When a UTF-8 data item defined with the DYNAMIC LENGTH clause requires truncation due to the LIMIT phrase, truncation is performed at the UTF-8 character level.

Truncation for these data items, when it is needed due to the LIMIT phrase, is done at a character boundary.

Whenever a dynamic-length UTF-8 data item is used as a sender, the byte-length of the item is always the current, runtime byte-length of the item.

For example:

```
01 u1 pic u dynamic length limit 10.  *> dynamic-length UTF-8 data item
```

The following rules apply to data items declared with the 'U' pic symbol:

- UTF-8 data items can be elementary data items.
- UTF-8 data items can appear in groups, including file records, except where the data item is dynamic length, which is not supported in file records.
- Group-items can be UTF-8 items via the UTF-8 phrase of the GROUP-USAGE clause.

Note: The groups defined with the GROUP-USAGE UTF-8 clause can only contain UTF-8 items defined with the BYTE-LENGTH phrase of the PICTURE clause. No other classes of data items are permitted.

- Condition variables associated with condition names (level 88 items) can be UTF-8.
- The VALUE clause for UTF-8 data items accepts alphanumeric, national and UTF-8 literals. In the alphanumeric case, the literal is automatically converted from EBCDIC to UTF-8.
- UTF-8 support does not include support for the following types of data items:
 - UTF-8 edited, UTF-8 numeric-edited
 - UTF-8 decimal
 - UTF-8 external float

Note: The USAGE UTF-8 clause can only appear in data definitions for data items declared with the 'U' pic symbol, i.e., numeric items cannot be defined with USAGE UTF-8.

The following code fragment illustrates valid and invalid UTF-8 data item definitions:

Examples of valid UTF-8 data item definitions:

```
01 u1 pic u(10).           *> fixed character-length item, usage utf-8
implied
01 u2 pic u(10) usage utf-8. *> fixed character-length item, usage utf-8
specified explicitly
77 u3 pic uuu usage utf-8.  *> fixed character-length item
77 u4 pic u dynamic length usage utf-8. *> dynamic length item
01 u5 pic u(5) value u'abcde'. *> fixed character-length item, VALUE clause
applied
01 u6 pic u byte-length 5.  *> fixed byte-length
```

Examples of invalid UTF-8 data item definitions:

```
01 u1 pic uuu,uuu,uuu usage utf-8. *> utf-8 edited not supported
01 u2 pic 9(9) usage utf-8.         *> utf-8 decimal not supported
77 u3 pic 999,999 usage utf-8.     *> utf-8 numeric edited not supported
77 u4 pic 999e+99 usage utf-8.     *> utf-8 external float not supported
01 u5 pic u(10) byte-length 20.    *> cannot include repetition factor and byte-
length phrase
01 u6 pic uuu byte-length 5.       *> cannot include repetition factor and byte-
length phrase
01 u7 pic u byte-length 5 dynamic length. *> only one of byte-length and dynamic length
allowed
```

Using UTF-8 literals

Two types of UTF-8 literals are supported:

1. Basic UTF-8 literal:
 - *u'character-data'*

- *character-data* is converted from EBCDIC to UTF-8.
- *character-data* may contain double-byte EBCDIC characters, but those characters must be delimited by shift-out and shift-in characters.
- The maximum number of Unicode code points that can be represented in a basic UTF-8 literal can vary depending on the size of each UTF-8 character. However, before truncation occurs, a maximum of 160 bytes after UTF-8 conversion is allowed.
- *character-data* can contain the following Unicode escape sequences:
 - \uhhhh, where each *h* represents a hexadecimal digit in the range ‘0’ to ‘9’, ‘a’ to ‘f’, and ‘A’ to ‘F’. This Unicode escape sequence represents a Unicode code point from the *Basic Multilingual Plane* (BMP) (that is, Unicode code points in the range U+0000 through U+FFFF).
 - \U00hhhhhh, where each *h* represents a hexadecimal digit in the range ‘0’ to ‘9’, ‘a’ to ‘f’, and ‘A’ to ‘F’. This Unicode escape sequence can represent any legal Unicode code point, including code points from the Supplementary Planes, i.e., Unicode code points in the range U+10000 through U+10FFFF (for example, an emoji symbol).

Notes:

- Code points U+D800 through U+DFFF are reserved for the high and low halves of surrogate pairs used by UTF-16. There is no legal encoding of these Unicode code points in UTF-8 and hence \uD800 through \uDFFF and \U0000D800 through \U0000DFFF cannot be specified as Unicode escape sequences in UTF-8 literals.
- To avoid having a string of characters of the form \uhhhh or \U00hhhhhh in a UTF-8 literal to be interpreted as a Unicode escape sequence, the escape character ‘\’ can itself be escaped with ‘\’ in order to be interpreted literally. Thus, the sequence \\u00E9 will not be treated as a Unicode escape sequence.
- Wherever a Unicode escape sequence appears in a basic UTF-8 literal, it is replaced by the compiler with the UTF-8 encoding for the Unicode code point corresponding to the escape sequence. This makes it convenient to represent general Unicode code points in the literal using only EBCDIC characters. For example, u’caf\u00E9’ represents the string ‘café’.

2. Hexadecimal UTF-8 literal

• ux'*hexadecimal-digits*'

- *hexadecimal-digits* are converted to a sequence of bytes in order to be used verbatim as the UTF-8 literal value.
- A minimum of 2 hexadecimal digits and up to a maximum of 320 hexadecimal digits are allowed.

Note: The sequence of bytes represented by *hexadecimal-digits* is validated to ensure that it contains a legal sequence of UTF-8 bytes.

COBOL statements and UTF-8 data items

The following COBOL statements directly support UTF-8 items:

ALLOCATE and FREE

UTF-8 data items declared in the LINKAGE SECTION can be dynamically allocated and freed.

EVALUATE and IF

The comparison of UTF-8 data items in conditions is supported. Comparisons of UTF-8 data items are done by using binary byte-by-byte comparisons.

UTF-8 data items can only be compared with items of class alphabetic, class alphanumeric, class national, or class UTF-8.

Conversions from EBCDIC to UTF-8 and UTF-16 to UTF-8 are done automatically where necessary during comparisons.

INITIALIZE

The category default for UTF-8 data items is x'20', a UTF-8 space.

JSON GENERATE and JSON PARSE

Data items of category UTF-8 can be used as key parts for JSON GENERATE and JSON PARSE operations.

MERGE and SORT

Data items of category UTF-8 can be used as key parts for sort and merge operations.

All comparisons of UTF-8 data in COBOL are done by using a binary, byte-by-byte comparison. This should produce the same ordering for a set of UTF-8 strings as the corresponding set of NATIONAL strings representing the same Unicode code points, assuming all code points in the strings are from the Basic Multilingual Plane (BMP).

MOVE

Basic move rules:

- A category UTF-8 sender can be moved only to an item of class and category national or an item of class and category UTF-8.
- A category UTF-8 item can only receive an item of class alphabetic, class alphanumeric, class national, or class UTF-8.

Note: This includes items such as numeric-edited, alphanumeric-edited, national-edited, and national-numeric-edited.

Padding and truncation, where needed, are always done at the UTF-8 character (that is, Unicode code point) level.

STRING and UNSTRING

Data is transferred from the sending fields to the receiving field using the rules of the MOVE statement for elementary UTF-8-to-UTF-8 moves.

The following COBOL statements do not support UTF-8 arguments:

- ACCEPT
- INSPECT
- XML GENERATE
- XML PARSE

Tip: If you need to use a UTF-8 data item with a statement such as INSPECT that does not yet support UTF-8 data items natively, you can always move the UTF-8 data item to a national data item first and use the national item in the statement instead.

Related tasks

[“Processing UTF-8 data using UTF-16 \(national\) data types” on page 153](#)

Intrinsic functions support for UTF-8 data items

UTF-8 data is a class and category of arguments and a return type for intrinsic functions.

The following intrinsic functions support UTF-8 arguments:

- BIT-OF
- BYTE-LENGTH
- CONTENT-OF¹
- DISPLAY-OF
- FORMATTED-CURRENT-DATE¹
- FORMATTED-DATE¹
- FORMATTED-DATETIME¹
- FORMATTED-TIME¹
- HEX-OF
- INTEGER-OF-FORMATTED-DATE

- LENGTH
- LOWER-CASE¹
- NATIONAL-OF
- SECONDS-FROM-FORMATTED-TIME
- TEST-FORMATTED-DATETIME
- TRIM¹
- ULENGTH
- UPOS
- UPPER-CASE¹
- USUBSTR¹
- USUPPLEMENTARY
- UVALID
- UWIDTH

1. This intrinsic function returns a result of category UTF-8 when the input argument contains UTF-8 data items.

Related references

Intrinsic functions (*Enterprise COBOL for z/OS Language Reference*)

Db2 considerations

The codepage of Db2 host variables is governed by various factors such as CODEPAGE, SQLCCSID, and DSNHDECP file. However, for host variables declared as UTF-8, these are overridden and the Db2 coprocessor assumes that the item has a CCSID of 1208.

Only UTF-8 data items declared with the BYTE-LENGTH phrase of the PICTURE clause can be used as Db2 host variables. In this case, when the host variable corresponds to a CHAR column in a Db2 table, the byte length of the CHAR column and the number of bytes specified in the BYTE-LENGTH phrase of the PICTURE clause should match to avoid unexpected truncation or padding of data.

Note: The Db2 precompiler cannot be used with a COBOL program containing UTF-8 data items.

Using UTF-8-character figurative constants

You can use the figurative constant ALL *utf-8-literal* in a context that requires UTF-8 characters. ALL *utf-8-literal* represents all or part of the string that is generated by successive concatenations of the encoding units that make up the UTF-8 literal.

You can use the figurative constants QUOTE, SPACE, HIGH-VALUE, LOW-VALUE, or ZERO in a context that requires UTF-8 characters, such as a MOVE statement, an implicit move, or a relation condition that has UTF-8 operands. In these contexts, the figurative constant represents a UTF-8-character (UTF-8) value.

When you use the figurative constant QUOTE in a context that requires UTF-8 characters, and the QUOTE compiler option is in effect, its value is UX'22'. If the APOST compiler option is in effect, its value is UX'27'.

When you use the figurative constant HIGH-VALUE in a context that requires UTF-8 characters, its value is UTF-8 character UX'F48FBFBF' corresponding to Unicode code point U+10FFFF, except when HIGH-VALUE is used in a move or compare operation with a fixed byte-length UTF-8 data item that has a length that is not a multiple of 4 bytes. In that case, when HIGH-VALUE is moved into or compared against the final 3, 2, or 1 byte(s) of the UTF-8 data item, the value of HIGH-VALUE is UX'EFBFBF' (U+FFFF), UX'DFBF' (U+07FF), and UX'7F' (U+007F), respectively.

When you use LOW-VALUE in a context that requires UTF-8 characters, its value is UX'00'.

Restriction: When a UTF-8 data item set to HIGH-VALUES is compared to an alphanumeric or national data item set to HIGH-VALUES, they will not compare equal. This is because the HIGH-VALUE value for one data type does not get translated to the corresponding HIGH-VALUE value for a different data type.

Related tasks

[“Converting to or from UTF-8 \(Unicode\) representation” on page 150](#)

[“Comparing UTF-8 data” on page 151](#)

Related references

Figurative constants (*Enterprise COBOL for z/OS Language Reference*)

DISPLAY-OF (*Enterprise COBOL for z/OS Language Reference*)

Support for Unicode: Using Unicode Services

Reference modifiers and UTF-8 data items

Similar to national data items, reference modifiers can be applied to UTF-8 data items. The start position and length of UTF-8 reference modifiers are specified in terms of characters (Unicode code points), not bytes, so that you do not need to worry about the varying byte-width of UTF-8 characters when generating UTF-8 substrings. However, due to this varying nature of UTF-8 characters, there are some special considerations:

- The starting byte position of a reference modified UTF-8 data item cannot be determined at compile time and must be determined from its starting character position at run time, even when the starting character position is specified as a literal.

Example 1

```
01 u1 pic u(10) value u'\u00e9cran'. *> écran
:
display u1(3:2)
```

In this example, `u1(3:2)` starts at character position 3, which is the 4th byte of the data item. This is because 'é' has a 2-byte encoding in UTF-8 and 'c' and 'r' both have 1-byte encodings in UTF-8.

Example 2

```
01 u1 pic u(10) value u'ecran'.
:
display u1(3:2)
```

In this example, `u1(3:2)` starts at character position 3, which is the 3rd byte of the data item. This is because 'e', 'c', and 'r' all have 1-byte encodings in UTF-8.

- The actual byte length of a UTF-8 reference modification cannot be determined at compile time and must be determined from its character length at run time, even when the character length of the reference modifier is specified as a literal.

Example 3

```
01 u1 pic u(10) value u'caf\u00e9'. *> café
:
display u1(3:2)
```

In this example, `u1(3:2)` is 2 UTF-8 characters in length but 3 bytes in length because 'f' has a 1-byte encoding in UTF-8 and 'é' has a 2-byte encoding in UTF-8.

Example 4

```
01 u1 pic u(10) value u'cafe'.
:
display u1(3:2)
```

In this example, `u1(3:2)` is 2 UTF-8 characters in length but 2 bytes in length because both 'f' and 'e' have a 1-byte encoding in UTF-8.

- When a reference modified UTF-8 data item is a receiver in a MOVE statement, the number of bytes that make up the substring indicated by the reference modification may not be the same after the move as before the move. For example, if the substring has a length of 4 characters, those 4 characters might be represented by 12 bytes before the move, but after the move, those 4 characters could be represented by only 4 bytes, or as many as 16 bytes. It depends on the UTF-8 sender in the move. When this situation occurs the remaining data after the substring will automatically be shifted left if the new substring has fewer bytes than it originally had, and the remaining bytes will be shifted right if the new substring has more bytes than it originally had. In the latter case, it is possible that some characters in the remaining portion of the underlying data item may be truncated. However, truncation can only happen if the underlying UTF-8 data item is defined with the BYTE-LENGTH phrase of the PICTURE clause. In the case of fixed character-length UTF-8 items, there will always be enough space to accommodate the right-shift without truncation.

Example 5

```
01 u1 pic u(13) value '01\u00e9, 01\u00e9, 01\u00e9'. *> 01é, 01é, 01é
:
move 'abcdef' to u1(3:6)
```

In this example, u1(3:6) represents 'é, Olé', which is 8 bytes long. After the move, the 8-byte substring represented by the reference modification is replaced with 'abcdef' which is only 6 bytes long in UTF-8. This means that the remaining characters ', Olé' in the underlying data item located immediately following the reference modified portion will be shifted left in memory 2 bytes as a result of the move.

Example 6

```
01 u1 pic u(13) value '01\u00e9, 01\u00e9, 01\u00e9'. *> 01é, 01é, 01é
:
move '\u00e9\u00e9\u00e9\u00e9\u00e9\u00e9' to u1(3:6)
```

In this example, u1(3:6) represents 'é, Olé' which is 8 bytes long. After the move, the 8-byte substring represented by the reference modification is replaced with 'éééééé' which is 12 bytes long in UTF-8. This means that the remaining characters ', Olé' in the underlying data item located immediately following the reference modified portion will be shifted right in memory 4 bytes as a result of the move.

Related references

[“Reference modifiers” on page 110](#)

UTF-8 groups

UTF-8 groups are specified with the GROUP-USAGE UTF-8 clause. These groups can contain fixed byte-length UTF-8 data items (items defined as PIC U BYTE-LENGTH *n*) or dynamic-length data items (items defined as PIC U DYNAMIC). Fixed character-length UTF-8 data items are not permitted. If the group contains only fixed byte-length UTF-8 data items then, in most cases, the group is processed as though it were redefined as an elementary category UTF-8 item described as PIC U BYTE-LENGTH *m*, where *m* is the total byte length of all items in the group.

For some operations on UTF-8 groups, however (just as for some operations on alphanumeric groups), group semantics apply. Such operations (for example, MOVE CORRESPONDING and INITIALIZE) recognize or process the elementary items within the UTF-8 group.

In use cases where the current limitation of UTF-8 groups is acceptable, that is, they can only contain fixed byte-length UTF-8 data items, UTF-8 groups provide several advantages for the processing of UTF-8 data compared to the processing of UTF-8 data within alphanumeric groups:

- When you move a UTF-8 group to a longer data item that has USAGE UTF-8, the receiving item is padded with UTF-8 characters. By contrast, if you move an alphanumeric group that contains UTF-8 characters to a longer alphanumeric group that contains UTF-8 characters, alphanumeric spaces are used for padding. As a result, mishandling of data items could occur.
- When you move a UTF-8 group to a shorter data item that has USAGE UTF-8, the UTF-8 group is truncated at UTF-8-character boundaries. By contrast, if you move an alphanumeric group that contains

UTF-8 characters to a shorter alphanumeric group that contains UTF-8 characters, truncation might occur between the 2 bytes of a UTF-8 character.

USAGE UTF-8 groups: A group item can specify the `USAGE UTF-8` clause at the group level as a convenient shorthand for the `USAGE` of each of the elementary data items within the group. Such a group is *not* a national group, but an alphanumeric group, and behaves in many operations, such as moves and compares, like an elementary data item of `USAGE DISPLAY` (except that no editing or conversion of data occurs).

Related tasks

[“Assigning values to group data items \(MOVE\)” on page 33](#)

Related references

`GROUP-USAGE` clause (*Enterprise COBOL for z/OS Language Reference*)

Converting to or from UTF-8 (Unicode) representation

You can implicitly or explicitly convert data items to UTF-8 representation.

You can implicitly convert alphabetic or national data to UTF-8 data by using the `MOVE` statement. Implicit conversions also take place in other COBOL statements, such as `IF` statements that compare an alphanumeric data item with a data item that has `USAGE UTF-8`.

You can explicitly convert UTF-16 data to UTF-8 data and UTF-8 data to alphanumeric data by using the `DISPLAY-OF` intrinsic function. You can also explicitly convert UTF-8 data to UTF-16 data by using the `NATIONAL-OF` intrinsic function. When using the `DISPLAY-OF` intrinsic function to convert UTF-16 to UTF-8, you must specify an output code page of 1208.

Related tasks

[“Converting alphanumeric or national to UTF-8 \(MOVE\)” on page 150](#)

[“Converting UTF-8 to national \(NATIONAL-OF\)” on page 151](#)

[“Converting national to UTF-8 and UTF-8 to alphanumeric \(DISPLAY-OF\)” on page 151](#)

[“Overriding the default code page” on page 135](#)

[“Comparing UTF-8 data” on page 151](#)

Related references

`CODEPAGE` on page 343

[“Conversion exceptions” on page 136](#)

`DISPLAY-OF` (*Enterprise COBOL for z/OS Language Reference*)

`NATIONAL-OF` (*Enterprise COBOL for z/OS Language Reference*)

Converting alphanumeric or national to UTF-8 (MOVE)

You can use a `MOVE` statement to implicitly convert data to UTF-8 representation.

You can move the following kinds of data to category UTF-8, and thus convert the data to UTF-8 representation:

- Alphabetic
- Alphanumeric
- Alphanumeric-edited
- Numeric-edited
- National
- National-edited
- Numeric-edited of `USAGE NATIONAL`

For complete rules about moves to UTF-8 data, see `MOVE` statement (*Enterprise COBOL for z/OS Language Reference*).

For example, the following MOVE statement moves the alphanumeric literal "AB" to the UTF-8 data item UTF8-Data:

```
01 UTF8-Data Pic U(2) Usage UTF-8.  
   Move "AB" to UTF8-Data
```

After the previous MOVE statement, UTF8-Data contains UX '4142', the UTF-8 representation of the alphanumeric characters 'AB'.

If padding is required in a receiving data item that has USAGE UTF-8, the default UTF-8 space character (UX '20') is used. If truncation is required, it occurs at the boundary of a UTF-8 character position.

Related tasks

[“Assigning values to elementary data items \(MOVE\)” on page 32](#)

[“Assigning values to group data items \(MOVE\)” on page 33](#)

Related references

MOVE statement (*Enterprise COBOL for z/OS Language Reference*)

Converting UTF-8 to national (NATIONAL-OF)

Use the NATIONAL-OF intrinsic function to explicitly convert a UTF-8 data item to a national data item.

Related references

NATIONAL-OF (*Enterprise COBOL for z/OS Language Reference*)

Converting national to UTF-8 and UTF-8 to alphanumeric (DISPLAY-OF)

Use the DISPLAY-OF intrinsic function to convert national data to UTF-8, or to convert UTF-8 data to an alphanumeric (USAGE DISPLAY) character string that is represented in a code page that you specify as the second argument.

When you convert national data to UTF-8, it is necessary to specify the second argument of DISPLAY-OF, the output code page, to be code page 1208.

When you convert UTF-8 to alphanumeric, if you omit the second argument, the output code page is the one that was in effect with the CODEPAGE compiler option when the source was compiled. If you specify an EBCDIC or ASCII code page that combines single-byte character set (SBCS) and DBCS characters, the returned string might contain a mixture of SBCS and DBCS characters. The DBCS substrings are delimited by shift-in and shift-out characters if the code page in effect for the function is an EBCDIC code page.

Related references

DISPLAY-OF (*Enterprise COBOL for z/OS Language Reference*)

Comparing UTF-8 data

You can compare UTF-8 data, that is, UTF-8 literals and data items that have USAGE UTF-8, explicitly or implicitly with other kinds of data in relation conditions.

You can code conditional expressions that use UTF-8 data in the following statements:

- EVALUATE
- IF
- PERFORM
- SEARCH

For full details about comparing UTF-8 data items to other data items, see the Related references.

Related tasks

[“Comparing two class UTF-8 operands” on page 152](#)

[“Comparing UTF-8 and other character-string operands” on page 152](#)
[“Comparing UTF-8 data and alphanumeric-group operands” on page 152](#)

Related references

Relation conditions (*Enterprise COBOL for z/OS Language Reference*)
General relation conditions (*Enterprise COBOL for z/OS Language Reference*)
National comparisons (*Enterprise COBOL for z/OS Language Reference*)
Group comparisons (*Enterprise COBOL for z/OS Language Reference*)

Comparing two class UTF-8 operands

You can compare the character values of two operands of class UTF-8.

Either operand (or both) can be any of the following types of items:

- A UTF-8 group
- An elementary category UTF-8 data item

One of the operands can instead be a UTF-8 literal or a UTF-8 intrinsic function.

When you compare two class UTF-8 operands that have the same length, they are determined to be equal if the corresponding characters in each operand are equal. Otherwise, comparison of the binary values of the first pair of unequal characters determines the operand with the larger binary value.

When you compare operands that have unequal lengths, the shorter operand is treated as if it were padded on the right with default UTF-8 space characters (UX '20 ') to the length of the longer operand.

The PROGRAM COLLATING SEQUENCE clause does not affect the comparison of two class UTF-8 operands.

Related references

UTF-8 comparisons (*Enterprise COBOL for z/OS Language Reference*)

Comparing UTF-8 and other character-string operands

You can compare the character value of a UTF-8 literal or class UTF-8 data item with the character value of any of the following character-string operands: alphabetic, alphanumeric, alphanumeric-edited, or numeric-edited of USAGE DISPLAY, national, national-edited, numeric-edited of USAGE NATIONAL.

These operands are treated as if they were moved to an elementary UTF-8 data item. The characters are converted to national (UTF-8) representation, and the comparison proceeds with two UTF-8 character operands.

Related tasks

[“Using national-character figurative constants” on page 132](#)

Related references

UTF-8 comparisons (*Enterprise COBOL for z/OS Language Reference*)

Comparing UTF-8 data and alphanumeric-group operands

You can compare a UTF-8 literal, a UTF-8 group item, or any elementary data item that has USAGE UTF-8 with an alphanumeric group or a national group.

Neither operand is converted. The UTF-8 operand is treated as if it were moved to an alphanumeric group item of the same size in bytes as the UTF-8 operand, and the two groups are compared. An alphanumeric comparison is done regardless of the representation of the subordinate items in the alphanumeric or national group operand.

For example, Group-XN is an alphanumeric group that consists of two subordinate items that have USAGE NATIONAL:

```
01 Group-XN.  
02 TransCode PIC NN Value "AB" Usage National.
```

```
02 Quantity PIC 999 Value 123 Usage National.
If U"AB123" = Group-XN Then Display "EQUAL"
Else Display "NOT EQUAL".
```

When the IF statement above is executed, the 5 bytes of the UTF-8 literal U"AB123" are compared byte by byte with the content of Group-XN. The items compare unequally, and "NOT EQUAL" is displayed.

Related references

Group comparisons (*Enterprise COBOL for z/OS Language Reference*)

Processing UTF-8 data using UTF-16 (national) data types

As an alternative to the recommended method of processing UTF-8 data using USAGE UTF-8 data items, you can also process UTF-8 data by storing it in alphanumeric data items and then converting it to UTF-16 in a national data item. After processing the national data, convert it back to UTF-8 for output. For the conversions, use the intrinsic functions NATIONAL-OF and DISPLAY-OF, respectively. Use code page 1208 for UTF-8 data.

As an alternative to the recommended method of processing UTF-8 data using USAGE UTF-8 data items, you can also process UTF-8 data by storing it in alphanumeric data items and then converting it to UTF-16 in a national data item.

National data is encoded in UTF-16, which uses one encoding unit for almost all commonly encountered characters. With this property, you can use string operations such as reference modification on the national data. If it is more convenient to retain the UTF-8 encoding, use the Unicode intrinsic functions to assist with processing the data. For details, see [“Using intrinsic functions to process UTF-8 encoded data” on page 153](#).

Take the following steps to convert ASCII or EBCDIC data to UTF-8:

1. Use the function NATIONAL-OF to convert the ASCII or EBCDIC string to a national (UTF-16) string.
2. Use the function DISPLAY-OF to convert the national string to UTF-8.

The following example converts Greek EBCDIC data to UTF-8:

```
01 Greek-EBCDIC pic X(10) value "αβγδεζηθ".
01 UnicodeString pic N(10).
01 UTF-8-String pic X(20).
   Move function National-of(Greek-EBCDIC, 00875) to UnicodeString
   Move function Display-of(UnicodeString, 01208) to UTF-8-String
```

Usage note: Use care if you use reference modification to refer to data encoded in UTF-8. UTF-8 characters are encoded with a varying number of bytes per character. Avoid operations that might split a multibyte character.

Related tasks

[“Referring to substrings of data items” on page 109](#)

[“Converting to or from national \(Unicode\) representation” on page 134](#)

[“Parsing XML documents encoded in UTF-8” on page 632](#)

[“Using](#)

[intrinsic functions to process UTF-8 encoded data” on page 153](#)

Using intrinsic functions to process UTF-8 encoded data

If it is more convenient to keep your data encoded in UTF-8, use the Unicode intrinsic functions to facilitate testing and processing the UTF-8 data.

You can use the following intrinsic functions:

UVALID

To verify that the UTF-8 character data is well-formed

USUPPLEMENTARY

If the data is to be converted to national, and it is important that every character can be represented by a single 16-bit encoding unit, use the USUPPLEMENTARY function to determine whether a valid UTF-8 character string contains a Unicode supplementary code point; that is, a code point with a Unicode scalar value above U+FFFF, requiring a 4-byte representation in UTF-8.

USUBSTR

It provides a convenient alternative to reference modification for referring to substrings of the UTF-8 character string. USUBSTR expects character position and length arguments versus the computed byte locations and counts required by reference modification.

Auxiliary functions can provide additional information about a valid UTF-8 character string:

ULENGTH

To determine the total number of Unicode code points in the string

UPOS

To determine the byte position in the string of the *n*th Unicode code point

UWIDTH

To determine the width in bytes of the *n*th Unicode code point in the string

The following code fragment illustrates UTF-8 validity checking, and the use of the auxiliary functions:

```
checkUTF-8-validity.  
  Compute u = function UVALID(UTF-8-testStr)  
  If u not = 0  
  Display 'checkUTF-8-validity failure:'  
  Display ' The UTF-8 representation is not valid,'  
    'starting at byte ' u '.'  
  Compute v = function ULENGTH(UTF-8-testStr(1:u - 1))  
  Compute u = function UPOS(UTF-8-testStr v)  
  Compute w = function UWIDTH(UTF-8-testStr v)  
  Display ' The ' v 'th and last valid code point starts '  
    'at byte ' u ' for ' w ' bytes.'  
End-if.
```

In the following string, the sequence that starts with x'F5' is not valid UTF-8 because no byte can have a value in the range x'F5' to x'FF':

```
x'6162D0B0E4BA8CF5646364'
```

The output from checkUTF-8-validity for this string is as follows:

```
checkUTF-8-validity failure:  
The UTF-8 representation is not valid, starting at byte 08.  
The 04th and last valid code point starts at byte 05 for 03 bytes.
```

The following code fragment illustrates checking for the presence of a Unicode supplementary code point, requiring a 4-byte representation in UTF-8:

```
checkUTF-8-sup.  
  Compute u = function USUPPLEMENTARY(UTF-8-testStr)  
  If u not = 0  
  Display ' checkUTF-8-sup hit:'  
  Compute v = function ULENGTH(UTF-8-testStr(1:u - 1))  
  Compute w = function UWIDTH(UTF-8-testStr v + 1)  
  Display ' The ' v 'th code point of the string'  
    ', starting at byte ' u ','  
  Display ' is a Unicode supplementary code point, '  
    'width ' w ' bytes.'  
End-if.
```

In the following string, the sequence x'F0908C82' is a supplementary character (as is any valid UTF-8 sequence beginning with a byte in the range x'F0' to x'F4'):

```
x'6162D0B0E4BA8CF0908C826364'
```

The output from checkUTF-8-supp for this string is as follows:

```
checkUTF-8-supp hit:
  The 04th code point of the string, starting at byte 08,
  is a Unicode supplementary code point, width 04 bytes.
```

Related references

[“CODEPAGE” on page 343](#)

Example: deriving initials from UTF-8 names

The following program uses the Unicode functions to derive composers’ initials from a table of names in Czech. It is intended to illustrate these functions, and is not necessarily the most efficient way of doing the task. Although the program processes the composer names in UTF-8, the data begins and ends in EBCDIC in order to permit a meaningful display of the program source and output. The compiler option CODEPAGE (1153) ensures that the names are interpreted correctly when translated to and from Unicode.

Program initials

```
Process codepage(1153)
-----*
* For a table of Czech composer names represented in UTF-8, *
* determine and print out the initials of each name. *
-----*
Identification division.
  Program-id. initials.
Data division.
Working-storage section.
  1 utilityVariables.
    2 UTF-8-space pic x value x'20'.
    2 UTF-8-hyphen pic x value x'2D'.
    2 UTF-8-ch pic xxx.
    2 i comp pic 9.
    2 j comp pic 99.
    2 hex pic x(160).
  1 EBCDICnameData.
    2 pic x(40) value 'Antonín Leopold Dvořák'.
    2 pic x(40) value 'Leoš Janáček'.
    2 pic x(40) value 'Rafael Jeroným Kubelík'.
    2 pic x(40) value 'Pavel Křížkovský'.
    2 pic x(40) value 'Jan Václav Hugo Voříšek'.
  1 redefines EBCDICnameData.
    2 EBCDICname pic x(40) occurs 5 times.
  1 UTF-8-nameData.
    2 composer pic x(40) occurs 5 times.
  1 composerInitials.
    2 occurs 5.
    3 cInitSize comp pic 99.
    3 cInit pic x(8).
  1 state pic 9.
    88 seekingInitial value 0.  *> Skip space and hyphen
    88 seekingSeparator value 1. *> Skip all but space and hyphen
```

Program initials, continued

```
Procedure division.
  main.
    Display 'Compute composer initials...'
    Initialize composerInitials
    Perform test before varying i from 1 by 1 until i > 5
  * Start by translating each composer name from EBCDIC to UTF-8.
    Move function display-of
      (function national-of(EBCDICname(i)) 1208)
    to composer(i)
  * Test each character of the name; skip leading spaces, etc.
    Set seekingInitial to true
    Move 1 to cInitSize(i)
    Perform varying j from 1 by 1
      until j > function ULENGTH(composer(i))
    Move function USUBSTR(composer(i) j 1) to UTF-8-ch
  * Initial found. Save in buffer, then skip to next space/hyphen.
    If seekingInitial and
      UTF-8-ch not = UTF-8-Space and UTF-8-Hyphen
    String function USUBSTR(composer(i) j 1)
      delimited by size
      into cInit(i) with pointer cInitSize(i)
    Set seekingSeparator to true
    End-if
  * Space/hyphen found; skip spaces or hyphens to next initial.
    If seekingSeparator and
      (UTF-8-ch = UTF-8-Space or UTF-8-Hyphen)
    Set seekingInitial to true
    End-if
  End-perform
  * Adjust string pointer to number of initials found.
    Subtract 1 from cInitSize(i)
  End-perform
  * Print out the UTF-8 initials, translated to EBCDIC, and
  * also in hexadecimal, using program ToHex (listed later).
    Perform test before varying i from 1 by 1 until i > 5
    Call 'toHex' using hex cInit(i) value cInitSize(i)
    Display ' #' i ': ' function display-of(
      function national-of(cInit(i) (1:cInitSize(i)) 1208))
      ' (x''' hex(1:2 * cInitSize(i)) ''')'
  End-perform.
  Goback.
End program initials.
```

Output from program initials

```
Compute composer initials...
#1: ALD (x'414C44')
#2: LJ (x'4C4A')
#3: RJK (x'524A4B')
#4: PK (x'504B')
#5: JVHV (x'4A564856')
```

Program toHex

```
Identification division.
  Program-id. toHex.
Data division.
  Working-storage section.
  1 hexv.
  2 pic x(32) value '000102030405060708090A0B0C0D0E0F'.
  2 pic x(32) value '101112131415161718191A1B1C1D1E1F'.
  2 pic x(32) value '202122232425262728292A2B2C2D2E2F'.
  2 pic x(32) value '303132333435363738393A3B3C3D3E3F'.
  2 pic x(32) value '404142434445464748494A4B4C4D4E4F'.
  2 pic x(32) value '505152535455565758595A5B5C5D5E5F'.
  2 pic x(32) value '606162636465666768696A6B6C6D6E6F'.
```



```

2 pic x(32) value '707172737475767778797A7B7C7D7E7F'.
2 pic x(32) value '808182838485868788898A8B8C8D8E8F'.
2 pic x(32) value '909192939495969798999A9B9C9D9E9F'.
2 pic x(32) value 'A0A1A2A3A4A5A6A7A8A9AAABACADAEAF'.
2 pic x(32) value 'B0B1B2B3B4B5B6B7B8B9BABBBBCBDBEBF'.
2 pic x(32) value 'C0C1C2C3C4C5C6C7C8C9CACBCCCDCECF'.
2 pic x(32) value 'D0D1D2D3D4D5D6D7D8D9DADBDDCDDDEDF'.
2 pic x(32) value 'E0E1E2E3E4E5E6E7E8E9EAEBECEDEEEF'.
2 pic x(32) value 'F0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF'.
1 redefines hexv.
2 hex pic xx occurs 256 times.
Local-storage section.
1 i pic 9(4) binary.
1 j pic 9(4) binary value 0.
1 jx redefines j.
2 pic x.
2 jxd pic x.
Linkage section.
1 ostr.
2 ostrv pic xx occurs 1024 times.
1 istr.
2 istrv pic x occurs 1024 times.
1 len pic 9(9) binary.
Procedure division using ostr istr value len.
  If len > 1024
    Display '>> Error: length ' len ' greater than toHex '
      'supported maximum of 1024.'
    Stop run
  End-if
  Perform with test before varying i from 1 by 1 until i > len
    Move 0 to j
    Move istrv(i) to jxd
    Add 1 to j
    Move hex(j) to ostrv(i)
  End-perform
  Goback
.
End program toHex.

```

Processing Chinese GB 18030 data

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

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

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

- Use DBCS data items to process GB 18030 characters that are represented in CCSID 1388.
- Use national data items to define and process GB 18030 characters that are represented in UTF-16, CCSID 01200.
- Process data in any code page (including CCSID 1388 or 1392) by converting the data to UTF-16, processing the UTF-16 data, and then converting the data back to the original code-page representation.

When you need to process Chinese GB 18030 data that requires conversion, first convert the input data to UTF-16 in a national data item. After you process the national data item, convert it back to Chinese GB 18030 for output. For the conversions, use the intrinsic functions NATIONAL-OF and DISPLAY-OF, respectively, and specify code page 1388 or 1392 as the second argument of each function.

The following example illustrates these conversions:

```

01 Chinese-EBCDIC pic X(16) value "奥林匹克运动会".
01 Chinese-GB18030-String pic X(16).
01 UnicodeString pic N(14).
. . .
      Move function National-of(Chinese-EBCDIC, 1388) to UnicodeString
* Process data in Unicode
      Move function Display-of(UnicodeString, 1388) to Chinese-GB18030-String

```

Related tasks

[“Converting to or from national \(Unicode\) representation” on page 134](#)

[“Coding for use of DBCS support” on page 158](#)

Related references

[“Storage of character data” on page 139](#)

Coding for use of DBCS support

IBM Enterprise COBOL for z/OS supports using applications in any of many national languages, including languages that use double-byte character sets (DBCS).

The following list summarizes the support for DBCS:

- DBCS characters in user-defined words (DBCS names)
- DBCS characters in comments
- DBCS data items (defined with PICTURE N, G, or G and B)
- DBCS literals
- DBCS compiler option

Related tasks

[“Defining DBCS data” on page 158](#)

[“Using DBCS literals” on page 159](#)

[“Testing for valid DBCS characters” on page 159](#)

[“Processing alphanumeric data items that contain DBCS data” on page 160](#)

Appendix B, “Converting double-byte character set (DBCS) data,” on page 789

Related references

[“DBCS” on page 350](#)

Defining DBCS data

Use the PICTURE and USAGE clauses to define DBCS data items. DBCS data items can use PICTURE symbols G, G and B, or N. Each DBCS character position is 2 bytes in length.

You can specify a DBCS data item by using the USAGE DISPLAY-1 clause. When you use PICTURE symbol G, you must specify USAGE DISPLAY-1. When you use PICTURE symbol N but omit the USAGE clause, USAGE DISPLAY-1 or USAGE NATIONAL is implied depending on the setting of the NSYMBOL compiler option.

If you use a VALUE clause with the USAGE clause in the definition of a DBCS item, you must specify a DBCS literal or the figurative constant SPACE or SPACES.

For the purpose of handling reference modifications, each character in a DBCS data item is considered to occupy the number of bytes that corresponds to the code-page width (that is, 2).

Related references

[“NSYMBOL” on page 377](#)

Using DBCS literals

You can use the prefix N or G to represent a DBCS literal.

That is, you can specify a DBCS literal in either of these ways:

- N ' *dbcs characters* ' (provided that the compiler option NSYMBOL (DBCS) is in effect)
- G ' *dbcs characters* '

You can use quotation marks (") or apostrophes (') as the delimiters of a DBCS literal irrespective of the setting of the APOST or QUOTE compiler option. You must code the same opening and closing delimiter for a DBCS literal.

The shift-out (SO) control character X'0E' must immediately follow the opening delimiter, and the shift-in (SI) control character X'0F' must immediately precede the closing delimiter.

In addition to DBCS literals, you can use alphanumeric literals to specify any character in one of the supported code pages. However, any string of DBCS characters that is within an alphanumeric literal must be delimited by the SO and SI characters, and the DBCS compiler option must be in effect for the SO and SI characters to be recognized as shift codes.

You cannot continue an alphanumeric literal that contains DBCS characters. The length of a DBCS literal is likewise limited by the available space in Area B on a single source line. The maximum length of a DBCS literal is thus 28 double-byte characters.

An alphanumeric literal that contains DBCS characters is processed byte by byte, that is, with semantics appropriate for single-byte characters, except when it is converted explicitly or implicitly to national data representation, as for example in an assignment to or comparison with a national data item.

Related tasks

[“Using figurative constants” on page 26](#)

Related references

[“APOST/QUOTE” on page 337](#)

[“DBCS” on page 350](#)

[“NSYMBOL” on page 377](#)

DBCS literals (*Enterprise COBOL for z/OS Language Reference*)

Testing for valid DBCS characters

The Kanji class test tests for valid Japanese graphic characters. This testing includes Katakana, Hiragana, Roman, and Kanji character sets.

The Kanji class test is done by checking characters for the range X'41' through X'7E' in the first byte and X'41' through X'FE' in the second byte, plus the space character X'4040'.

The DBCS class test tests for valid graphic characters for the code page.

The DBCS class test is done by checking characters for the range X'41' through X'FE' in both the first and second byte of each character, plus the space character X'4040'.

Related tasks

[“Coding conditional expressions” on page 95](#)

Related references

Class condition (*Enterprise COBOL for z/OS Language Reference*)

Processing alphanumeric data items that contain DBCS data

If you use byte-oriented operations (for example, STRING, UNSTRING, or reference modification) on an alphanumeric data item that contains DBCS characters, results are unpredictable. You should instead convert the item to a national data item before you process it.

That is, do these steps:

1. Convert the item to UTF-16 in a national data item by using a MOVE statement or the NATIONAL -OF intrinsic function.
2. Process the national data item as needed.
3. Convert the result back to an alphanumeric data item by using the DISPLAY-OF intrinsic function.

Related tasks

[“Joining data items \(STRING\)” on page 103](#)

[“Splitting data items \(UNSTRING\)” on page 105](#)

[“Referring to substrings of data items” on page 109](#)

[“Converting to or from national \(Unicode\) representation” on page 134](#)

Chapter 8. Processing files

Processing 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 items:

- Another program
- Hierarchical or relational database
- Messages from subsystem software
- Direct-access storage device
- Magnetic tape
- Printer
- Terminal
- Card reader or punch

The information as it exists on an external device might be 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” on page 161](#)

Related tasks

[“Choosing file organization and access mode” on page 162](#)

[“Allocating files” on page 164](#)

[“Checking for input or output errors” on page 165](#)

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 in the z/OS UNIX file system and that contain only characters as data. Each record ends with a newline 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. When you read or write indexed files randomly, the sequence is in a programmer-specified manner. When you read or write indexed files dynamically, the sequence is sequential or random or both, determined by the input-output statements used.

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 IBM Enterprise COBOL for z/OS, 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 z/OS 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.

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 (z/OS UNIX)
- Indexed (VSAM)
- Relative (VSAM)

Related tasks

[“Allocating files” on page 164](#)

[Chapter 9, “Processing QSAM files,” on page 167](#)

[Chapter 10, “Processing VSAM files,” on page 189](#)

[Chapter 11, “Processing line-sequential files,” on page 215](#)

[“Choosing file organization and access mode” on page 162](#)

Choosing file organization and access mode

There are several guidelines you can use to determine which file organization and access mode to use in an application.

Consider the following guidelines when choosing file organization:

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

Consider the following guidelines when choosing access mode:

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

File organization	Sequential access	Random access	Dynamic access	Fixed length	Variable length
QSAM (physical sequential)	X			X	X
Line sequential	X			X ¹	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” on page 163

“Control characters in line-sequential files” on page 216

Format for coding input and output

The following example shows the general format of input-output coding. Explanations of the user-supplied information are shown after the code.

```
IDENTIFICATION DIVISION.
.
.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT filename ASSIGN TO assignment-name (1) (2)
    ORGANIZATION IS org ACCESS MODE IS access (3) (4)
    FILE STATUS IS file-status (5)
.
.
DATA DIVISION.
FILE SECTION.
FD filename
01 recordname (6)
   nn . . . fieldlength & type (7) (8)
   nn . . . fieldlength & type
.
.
WORKING-STORAGE SECTION
01 file-status 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 described below:

(1) filename

Any legal COBOL name. You must use the same file-name in the SELECT clause and in the FD entry, and in 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 clause, FD entry, 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* in a DD statement, in an ALLOCATE command (TSO), or as an environment variable (for example, in an export command) (z/OS 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 COBOL file status key. You can specify the file status key as a two-character category alphanumeric or category national item, or as a two-digit zoned decimal (USAGE DISPLAY) or national decimal (USAGE NATIONAL) item.

(6) recordname

The name of the record used in the WRITE or 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, map each element 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 a file, code OUTPUT or EXTEND. If you are both reading and writing, code I-0, except for organization LINE SEQUENTIAL.

Related tasks

[Chapter 9, "Processing QSAM files," on page 167](#)

[Chapter 10, "Processing VSAM files," on page 189](#)

[Chapter 11, "Processing line-sequential files," on page 215](#)

Allocating files

For any type of file (sequential, line sequential, indexed, or relative) in your z/OS or z/OS 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 z/OS UNIX file system, 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 cases:

- 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 steps occur:
 - a) If the allocation is for a QSAM file and the CBLQDA runtime 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.
 - b) Otherwise, the allocation fails.

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

Related tasks

[“Setting and accessing environment variables” on page 514](#)

[“Defining and allocating QSAM files” on page 180](#)

[“Dynamically creating QSAM files” on page 177](#)

[“Allocating VSAM files” on page 208](#)

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 item in the FILE STATUS clause to get additional VSAM status code information.

Another way of handling errors in input and output operations is to code ERROR (synonymous with EXCEPTION) declaratives.

Related tasks

[“Handling errors in input and output operations” on page 243](#)

[“Coding ERROR declaratives” on page 246](#)

[“Using file status keys” on page 247](#)

Chapter 9. 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 z/OS UNIX file system.

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 z/OS UNIX file system 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

z/OS DFSMS: Using Data Sets (Access methods)

Related tasks

[“Defining QSAM files and records in COBOL” on page 167](#)

[“Coding input and output statements for QSAM files” on page 176](#)

[“Handling errors in QSAM files” on page 179](#)

[“Working with QSAM files” on page 180](#)

[“Accessing z/OS UNIX files](#)

[using QSAM” on page 186](#)

[“Processing QSAM ASCII files on tape” on page 187](#)

Defining QSAM files and records in COBOL

Use the FILE-CONTROL entry to define the files in a COBOL program as QSAM files, and to associate the files with their external file-names.

An *external file-name* (a ddname or environment variable 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.
```

The ASSIGN clause *name* can include an S- before the external name to document that the file is a QSAM file. Both the ORGANIZATION and ACCESS MODE clauses are optional.

Related tasks

[“Establishing record formats” on page 168](#)

[“Setting block sizes” on page 174](#)

Establishing record formats

In the FD entry in the DATA DIVISION, code the record format and indication of whether the records are blocked. In the associated record description entry or entries, specify 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 the FD entry for the file. If you want the records to be blocked, code the BLOCK CONTAINS clause in the FD entry.

The following example shows how the FD entry might look for a file that has 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 z/OS UNIX file system. The above example is appropriate for such a file.

Related concepts

[“Logical records” on page 168](#)

Related tasks

[“Requesting fixed-length format” on page 168](#)

[“Requesting variable-length format” on page 170](#)

[“Requesting spanned format” on page 171](#)

[“Requesting undefined format” on page 173](#)

[“Defining QSAM files and records in COBOL” on page 167](#)

Related references

[“FILE SECTION entries” on page 12](#)

Logical records

COBOL uses the term *logical record* in a slightly different way than z/OS QSAM.

For format-V and format-S files, a 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 byte-stream files in the z/OS UNIX file system, 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” on page 169](#)

[“Layout of format-V records” on page 171](#)

[“Layout of format-S records” on page 173](#)

[“Layout of format-U records” on page 174](#)

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 (format-1 RECORD clause) to indicate the length of the record in bytes.

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 z/OS UNIX file system are never blocked.

Related concepts

[“Logical records” on page 168](#)

Related tasks

[“Requesting variable-length format” on page 170](#)

[“Requesting spanned format” on page 171](#)

[“Requesting undefined format” on page 173](#)

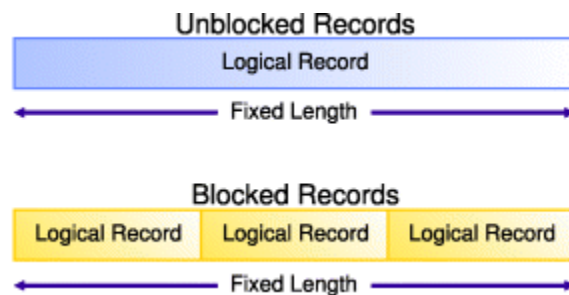
[“Establishing record formats” on page 168](#)

Related references

[“Layout of format-F records” on page 169](#)

Layout of format-F records

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



Related concepts

[“Logical records” on page 168](#)

Related tasks

[“Requesting fixed-length format” on page 168](#)

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

Related references

[“Layout of format-V records” on page 171](#)

[“Layout of format-S records” on page 173](#)

[“Layout of format-U records” on page 174](#)

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 (format-3 RECORD clause).

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. The integer sizes indicate the minimum and maximum record lengths in numbers of bytes regardless of the USAGE of the data items in the record.

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 (format-2 RECORD clause). Make *integer-1* and *integer-2* match the minimum length and the maximum length in bytes 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 level-01 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).

Related tasks

[“Requesting fixed-length format” on page 168](#)

[“Requesting spanned format” on page 171](#)

[“Requesting undefined format” on page 173](#)

[“Establishing record formats” on page 168](#)

Related references

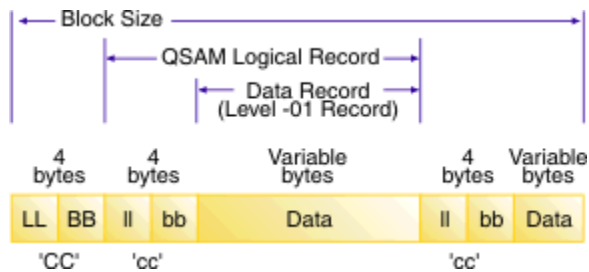
[“FILE SECTION entries” on page 12](#)

[“Layout of format-V records” on page 171](#)

[Enterprise COBOL for z/OS Migration Guide \(Moving from the VS COBOL II run time\)](#)

Layout of format-V records

Format-V QSAM records have control fields that precede 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. However, 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.

II 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 the description of the logical record in the `DATA DIVISION` of your program. COBOL allocates input and output buffers that are 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, the records will be moved without the control bytes.

Files in the z/OS UNIX file system are never blocked.

Related concepts

[“Logical records” on page 168](#)

Related tasks

[“Requesting variable-length format” on page 170](#)

Related references

[“Layout of format-F records” on page 169](#)

[“Layout of format-S records” on page 173](#)

[“Layout of format-U records” on page 174](#)

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 that are assigned to magnetic tape or to direct access devices. Do not request spanned records for files in the z/OS UNIX file system. You can omit the `RECORDING MODE` clause. The compiler determines the recording mode to be S if the maximum record length (in bytes) 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 as are used for format V. The length is based on your usage of the RECORD clause.

When creating files that contain 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 32,760 bytes in length.

When retrieving files that have format-S records, a 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: When you specify a READ INTO statement for a format-S file, the compiler generates a MOVE statement that uses the size of the record that it just read for that file. If the record just read does not correspond to the level-01 record description, you might not get the result that you expect. All other rules of the MOVE statement apply.

Related concepts

[“Logical records” on page 168](#)

[“Spanned blocked and unblocked files” on page 172](#)

Related tasks

[“Requesting fixed-length format” on page 168](#)

[“Requesting variable-length format” on page 170](#)

[“Requesting undefined format” on page 173](#)

[“Establishing record formats” on page 168](#)

Related references

[“FILE SECTION entries” on page 12](#)

[“Layout of format-S records” on page 173](#)

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. A spanned unblocked file is made up of physical blocks, each containing one logical record or one segment of a logical record.

In a spanned blocked file, a logical record can be either fixed or variable in length, and its 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.

In a spanned unblocked file, 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 168](#)

Related tasks

[“Requesting spanned format” on page 171](#)

Layout of format-S records

Spanned records are preceded by control fields, as explained below.



Each block is preceded by a 4-byte block descriptor field ('BDF' in the image above). There is only one block descriptor field at the beginning of each physical block.

Each segment of a record in a block is preceded by a 4-byte segment descriptor field ('SDF' in the image) even if the segment is the entire record. 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, and the fields are not available for you to use in your COBOL program.

Related tasks

[“Requesting spanned format” on page 171](#)

Related references

[“Layout of format-F records” on page 169](#)

[“Layout of format-V records” on page 171](#)

[“Layout of format-U records” on page 174](#)

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.

When you use format-U files, each block of storage is one logical record. A read of a format-U file returns the entire block as a record. A write to a format-U file writes a record out as a block. The compiler determines the recording mode to be U only if you code `RECORDING MODE U`.

It is recommended that you not use format U to update or extend a file that was written with a different record format. If you use format U to update a file that was written with a different format, the RECFM value in the data-set label could be changed or the data set could contain records written in different formats.

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

- If you use the `RECORD CONTAINS integer` clause (format-1 RECORD clause), the record length is the *integer* value regardless of the lengths of the level-01 record description entries associated with the file. The integer size indicates the number of bytes in a record regardless of the USAGE of its data items.
- If you use the `RECORD IS VARYING` clause (format-3 RECORD clause), 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. The integer sizes indicate the minimum and maximum record lengths in numbers of bytes regardless of the USAGE of the data items in the record.

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 (format-2 RECORD clause), with *integer-1* and *integer-2* matching the minimum length and the maximum length in bytes 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 compiler generates a `MOVE` statement that uses the size of the record that it just read for that file. If the record just read does not correspond to the level-01 record description, you might not get the result that you expect. All other rules of the `MOVE` statement apply.

Related tasks

[“Requesting fixed-length format” on page 168](#)

[“Requesting variable-length format” on page 170](#)

[“Requesting spanned format” on page 171](#)

[“Establishing record formats” on page 168](#)

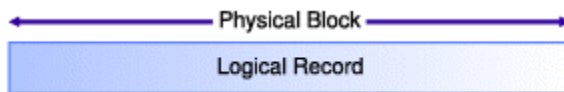
Related references

[“FILE SECTION entries” on page 12](#)

[“Layout of format-U records” on page 174](#)

Layout of format-U records

With format-U, 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 168](#)

Related tasks

[“Requesting undefined format” on page 173](#)

Related references

[“Layout of format-F records” on page 169](#)

[“Layout of format-V records” on page 171](#)

[“Layout of format-S records” on page 173](#)

Setting block sizes

In COBOL, you establish the size of a physical record by using 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 files in the z/OS UNIX file system, 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, the size must not be greater than the maximum block size for the device. If you specify the `CHARACTERS` phrase of the `BLOCK CONTAINS` clause, size must indicate the number of bytes in a record regardless of the `USAGE` of the data items in the record. The block size that is 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 during one of the following actions:

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

Larger blocks generally give you better performance. Blocks of only a few kilobytes are particularly inefficient; you should choose a block size of at least tens of kilobytes. If you specify record blocking and omit the block size, the system will pick a block size that is optimal for device utilization and for data transfer speed.

Letting z/OS determine block size: To maximize performance, do not explicitly set the block size for a blocked file in your COBOL source program. For new blocked data sets, it is simpler to allow z/OS to supply a system-determined block size. To use this feature, follow these guidelines:

- Code `BLOCK CONTAINS 0` in your source program or compile with the `BLOCK0` option. For details about `BLOCK0`, see “[BLOCK0](#)” on page 340.
- 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, your program will be most flexible if you follow these guidelines:

- Code `BLOCK CONTAINS 0` in your source program or compile with the `BLOCK0` option.
- Code a `BLKSIZE` value in the ddname definition (the JCL DD statement).

For extended-format data sets on z/OS, z/OS 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. z/OS DFSMS allocates the space used to read in the block suffix. However, when you calculate how many blocks of an extended-format 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 that is larger than 32760 directly in the `BLOCK CONTAINS` clause or indirectly with the use of `BLOCK CONTAINS n RECORDS`, the `OPEN` of the data set fails with file status code 90 unless you define the data set to be on tape.

For existing blocked data sets, it is simplest to:

- Code `BLOCK CONTAINS 0` in your source program or compile with the `BLOCK0` option.
- Not code a `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 the large block interface (LBI) for large block sizes. When the LBI 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 a system that 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 runtime message is issued.

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

- The `BLKSZLIM` keyword on the DD statement for the data set
- `BLKSZLIM` for the data class by using the `BLKSZLIM` keyword (must be set by your systems programmer)
- A block-size limit for the system in the `DEVSUPxx` member of `SYS1.PARMLIB` by using the keyword `TAPEBLKSZLIM` (must be set by your systems programmer)

The block-size limit is the first nonzero value that the compiler finds by checking these items.

If no BLKSIZE or BLKSZLIM value is available from any source, the system limits BLKSIZE to 32760. You can then enable block sizes larger than 32760 in one of two ways:

- Specify a BLKSZLIM value greater than 32760 in the DD statement for the file and use BLOCK CONTAINS 0 in your COBOL source.
- 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 z/OS, 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 167](#)

[z/OS DFSMS: Using Data Sets](#)

Related references

[“FILE SECTION entries” on page 12](#)

[“BLOCK0” on page 340](#)

BLOCK CONTAINS clause (*Enterprise COBOL for z/OS Language Reference*)

Coding input and output statements for QSAM files

You can code the following input and output statements to process a QSAM file or a byte-stream file in the z/OS UNIX file system using QSAM: OPEN, READ, WRITE, REWRITE, and CLOSE.

OPEN

Initiates the processing of files. 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-O. You cannot open z/OS UNIX files as I-O; a file status of 37 results 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 z/OS UNIX file system using REWRITE.

CLOSE

Releases the connection between the file and your program.

Related tasks

[“Opening QSAM files” on page 177](#)

[“Dynamically creating QSAM files” on page 177](#)

[“Adding records to QSAM files” on page 178](#)

[“Updating QSAM files” on page 178](#)

[“Writing QSAM files to a printer or spooled data set” on page 178](#)

[“Closing QSAM files” on page 179](#)

Related references

OPEN statement (*Enterprise COBOL for z/OS Language Reference*)
READ statement (*Enterprise COBOL for z/OS Language Reference*)
WRITE statement (*Enterprise COBOL for z/OS Language Reference*)
REWRITE statement (*Enterprise COBOL for z/OS Language Reference*)
CLOSE statement (*Enterprise COBOL for z/OS Language Reference*)
File status key (*Enterprise COBOL for z/OS Language Reference*)

Opening QSAM files

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

An OPEN statement works if both of the following conditions 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 the file match the attributes coded for that file in the SELECT clause and FD entry.

Mismatches in the file-organization attributes, code set, maximum record size, or record format (fixed or variable) result in file status code 39, and the failure of the OPEN statement. Mismatches in maximum record size and record format are not errors when opening files in the z/OS UNIX file system.

For fixed-length QSAM files, if you code RECORD CONTAINS 0 in the FD entry, 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. The file is positioned at the end, and READ statements read the data records in reverse order, starting with the last record. The REVERSED option is supported only for files that have fixed-length records.

Related tasks

[“Dynamically creating QSAM files” on page 177](#)

[“Ensuring that file attributes match your program” on page 183](#)

Related references

OPEN statement (*Enterprise COBOL for z/OS Language Reference*)

Dynamically creating QSAM files

Sometimes a QSAM file is unavailable on the operating system, but a COBOL program specifies that the file be created. Under certain circumstances, the file is created for you dynamically.

A QSAM file is considered to be *available* on z/OS when it has been identified to the operating system using a valid DD statement, an export command for an environment variable, or a TSO ALLOCATE command. Otherwise the file is *unavailable*.

Note that a DD statement with a misspelled ddname is equivalent to a missing DD statement, and an environment variable with a value that is not valid is equivalent to an unset variable.

The QSAM file is implicitly created if you use the runtime option CBLQDA and one of the following circumstances exists:

- An optional file is being opened as EXTEND or I-0.

Optional files are files that are not necessarily available each time the program is run. You define a file that is being opened in INPUT, I-0, or EXTEND mode as optional by coding the SELECT OPTIONAL clause in the FILE-CONTROL paragraph.

- The file is being opened for OUTPUT, regardless of the OPTIONAL phrase.

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

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

Under z/OS, files created using the CBLQDA option are temporary data sets and do not exist after the program has run.

Related tasks

[“Opening QSAM files” on page 177](#)

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 (*Enterprise COBOL for z/OS Language Reference*)

WRITE statement (*Enterprise COBOL for z/OS Language Reference*)

Updating QSAM files

You can update QSAM files only if they reside on direct access storage devices. You cannot update files in the z/OS UNIX file system.

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

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

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

Related references

REWRITE statement (*Enterprise COBOL for z/OS 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 phrase, COBOL first calculates the sum of LINAGE-COUNTER plus *nn*. Subsequent actions depend on the size of *nn*. The END-OF-PAGE imperative phrase 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 phrase is performed.

AT END-OF-PAGE or NOT AT END-OF-PAGE imperative phrases 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, and all conditional phrases are omitted.

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 phrase from a WRITE statement, the effect is as if you had coded:

```
AFTER ADVANCING 1 LINE
```

Related references

WRITE statement (*Enterprise COBOL for z/OS Language Reference*)

Closing QSAM files

Use the CLOSE statement to disconnect your program from a 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 COBOL programs in the run unit.
- If the run unit ends abnormally and the TRAP(ON) runtime option is in effect, the run time closes all open files that are defined in any COBOL programs in the run unit.
- When Language Environment condition handling has 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 COBOL programs in the run unit that might be called again and reentered.

You can change the location 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 language constructs such as a C longjmp.

- When you use CANCEL for a COBOL subprogram, the run time closes any open nonexternal files that are defined in that program.
- When a COBOL subprogram with the INITIAL attribute returns control, the run time closes any open nonexternal files that are defined in that program.
- When a thread of a multithreaded application ends, both external and nonexternal files that you opened from within that same thread are closed.

File status key data items in the DATA DIVISION are set when these implicit CLOSE operations are performed, but your EXCEPTION/ERROR declarative is not invoked.

Errors: If you open a QSAM file in a multithreaded application, you must close it from the same thread of execution from which the file was opened. Attempting to close the file from a different thread results in a close failure with file-status condition 90.

Related references

CLOSE statement (*Enterprise COBOL for z/OS Language Reference*)

Handling errors in QSAM files

When an input statement or output statement fails, COBOL does not take corrective action for you. You choose whether your program should 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 runtime 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 243](#)

Working with QSAM files

To work with QSAM files in a COBOL program, you define and allocate the files, retrieve them, and ensure that their file attributes match those in the program. You can also use striped extended-format QSAM data sets to help improve performance.

Related tasks

[“Defining and allocating QSAM files” on page 180](#)

[“Retrieving QSAM files” on page 182](#)

[“Ensuring that file attributes match your program” on page 183](#)

[“Using striped extended-format QSAM data sets” on page 185](#)

Related references

[“Allocation of buffers for QSAM files” on page 186](#)

Defining and allocating QSAM files

You can define a QSAM file or a byte-stream file in the z/OS UNIX file system by using either a DD statement or an environment variable. Allocation of these files follows the general rules for the allocation of COBOL files.

When you use an environment variable, the name must be in uppercase. Specify the MVS data set in one of these ways:

- DSN (*data-set-name*)
- DSN (*data-set-name (member-name)*)

data-set-name must be fully qualified and cannot be a temporary data set (that is, it must not start with &).

Restriction: You cannot create a PDS or PDSE by using an environment variable.

You can optionally specify the following attributes in any order after 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 z/OS UNIX file system. To do so, define one of the following items with a name that matches the external name in the 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 releases of COBOL before COBOL for OS/390 & VM 2.2, you can also specify `FILEDATA=TEXT` when using a DD allocation for z/OS UNIX files, but this use is not recommended. To process text files in the z/OS UNIX file system, use `LINE SEQUENTIAL` organization. If you do use QSAM to process text files in the z/OS UNIX file system, you cannot use environment variables to define the files.

When you define a QSAM file, use the parameters as shown below.

<i>Table 20. QSAM file allocation</i>		
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. (Only for direct-access storage devices.)	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 that you want to add.	DCB subparameters	n/a

Some of the information about the QSAM file must always be coded in the FILE-CONTROL paragraph, 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 by 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 items:

- Block size (`BLKSIZE=`), if `BLOCK CONTAINS 0 RECORDS` or `BLOCK0` option was specified at compile time (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.

[“Example: setting and accessing environment variables” on page 516](#)

Related tasks

[“Setting block sizes” on page 174](#)

[“Defining QSAM files and records in COBOL” on page 167](#)

[“Allocating files” on page 164](#)

Related references

[“BLOCK0” on page 340](#)

[“Parameters for creating](#)

[QSAM files” on page 182](#)

MVS Program Management: User's Guide and Reference

Parameters for creating QSAM files

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

```
DSNAME= [ dataset-name  
         dataset-name(member-name)  
         &&name  
         &&name(member-name) ]  
DSN=    [ &&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-sequence-number,] [ NL  
                                         SL  
                                         SUL ] [ ,EXPDT= [ yyddd  
                                                         yyyy/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 180](#)

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, and 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 DSNAME and DISP.

Related references

[“Parameters for retrieving QSAM files” on page 183](#)

Parameters for retrieving QSAM files

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

```
DSNAME= [ dataset-name  
DSN=    dataset-name(member-name)  
        *.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 ]  
            [ ,CATLG ] [ ,UNCATLG ]  
            [ ,UNCATLG ]  
  
DCB= ( subparameter-list )
```

Related tasks

[“Retrieving QSAM files” on page 182](#)

Ensuring that file attributes match your program

When the fixed file attributes in the DD statement or the data-set label and the attributes that are coded for that file in the SELECT clause and FD entry 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 file status code 39 and the failure of the OPEN statement. An exception exists for files in the z/OS UNIX file system: mismatches in record format and record length do not cause an error.

To prevent common file status 39 problems, follow the guidelines for processing existing or new files.

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

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

Related tasks

[“Processing existing files” on page 184](#)

[“Processing new files” on page 185](#)

[“Opening QSAM files” on page 177](#)

Related references

[“FILE SECTION entries” on page 12](#)

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 the guidelines below to define the maximum record length.

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

The easiest way to define variable-length (format-V) records in a program is to use the `RECORD IS VARYING FROM integer-1 TO integer-2` clause in the FD entry and set an appropriate value for *integer-2*. Express® the integer sizes in bytes regardless of the underlying USAGE of the data items in the record. For example, assume that you determine that the length attribute of the data set is 104 bytes (LRECL=104). Remembering that the maximum record length is determined from the `RECORD IS VARYING` clause 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).
```

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).
```

To define fixed-length records in your program, either code 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. If you intend to use the same program to process different files at run time, and those files have differing fixed lengths, avoid record-length conflicts by coding `RECORD CONTAINS 0`.

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

Related tasks

[“Processing new files” on page 185](#)

[“Requesting fixed-length](#)

[format” on page 168](#)

[“Requesting variable-length format” on page 170](#)

[“Requesting undefined format” on page 173](#)

[“Opening QSAM files” on page 177](#)

Related references

[“FILE SECTION entries” on page 12](#)

Processing new files

If your COBOL program writes records to a new file that will be made available before the program runs, ensure that the file attributes in the DD statement, the environment variable, or the allocation do not conflict with the attributes in the program.

Usually you need to code only a minimum of parameters when predefining files. But if you need to explicitly set a length attribute for the data set (for example, you are using an ISPF allocation panel, or your DD statement is for a batch job in which the program uses `RECORD CONTAINS 0`), follow these guidelines:

- 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 the 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 language used in your program. In such a case, take the added byte into account when coding the LRECL value.

For example, if your program contains the following code for a file that has variable-length records, the LRECL value in the DD statement or allocation should be 54.

```
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) .
```

Related tasks

[“Processing existing files” on page 184](#)

[“Requesting fixed-length format” on page 168](#)

[“Requesting variable-length format” on page 170](#)

[“Requesting undefined format” on page 173](#)

[“Opening QSAM files” on page 177](#)

[“Dynamically creating QSAM files” on page 177](#)

Related references

[“FILE SECTION entries” on page 12](#)

Using striped extended-format QSAM data sets

Striped extended-format QSAM data sets can benefit applications that process files that have large amounts of data or in which the time needed for I/O operations significantly affects overall performance.

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

For you to gain the maximum benefit from using QSAM striped data sets, z/OS 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 entry for the file. Doing so lets z/OS DFSMS determine the optimum number of buffers for the data set.

- Compile your program with the DATA(31) and RENT compiler options, and make the program object AMODE 31.
- Specify the ALL31(ON) runtime option if the file is an EXTERNAL file with format-F, format-V, or format-U records.

Note that all striped data sets are extended-format data sets, but not all extended-format data sets are striped.

Related tasks

z/OS DFSMS: Using Data Sets

Allocation of buffers for QSAM files

z/OS DFSMS automatically allocates buffers for storing input and output for a QSAM file above or below the 16 MB line as appropriate for the file.

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 option.
- EXTERNAL files when the ALL31(OFF) runtime option is specified. To specify the ALL31(ON) runtime 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 ones:
 - An EXTERNAL file (even if ALL31(ON) 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 concepts

[“Storage and its addressability” on page 39](#)

Related tasks

[“Using striped extended-format QSAM data sets” on page 185](#)

Accessing z/OS UNIX files using QSAM

You can process byte-stream files in the z/OS UNIX file system as ORGANIZATION SEQUENTIAL files using QSAM. To do this, specify as the *assignment-name* in the ASSIGN clause either a ddname or an environment-variable name.

ddname

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

Environment-variable name

An environment variable that holds the runtime value of the z/OS UNIX file system path for the file

Observe the following restrictions:

- Spanned record format is not supported.
- OPEN I-O and REWRITE are not supported. If you attempt one of these operations, one of the following file-status conditions results:
 - 37 from OPEN I-O
 - 47 from REWRITE (because you could not have successfully opened the file as I-O)

Usage notes

- File status 39 (fixed file attribute conflict) is not enforced for either of the following types of conflicts:

- Record-length conflict
- Record-type conflict (fixed as opposed to variable)
- A READ returns the number of bytes of the maximum logical record size for the file except for the last record, which might be shorter.

For example, suppose that a file definition has level-01 record descriptions of 3, 5, and 10 bytes long, and you write the following three records: 'abc', 'defgh', and 'ijklmnopqr', in that order. The first READ of this file returns 'abcdefghij', the second READ returns 'klmnopqr ', and the third READ results in the AT END condition.

For compatibility with releases of IBM COBOL before COBOL for OS/390 & VM 2.2, you can also specify FILEDATA=TEXT when using a DD allocation for z/OS UNIX files, but this use is not recommended. To process text files in the z/OS UNIX file system, use LINE SEQUENTIAL organization. If you use QSAM to process text files in the z/OS UNIX file system, you cannot use environment variables to define the files.

Related tasks

[“Allocating files” on page 164](#)

[“Defining and allocating QSAM files” on page 180](#)

[z/OS DFSMS: Using Data Sets \(Using HFS data sets\)](#)

Processing QSAM ASCII files on tape

If your program processes a QSAM ASCII file, you must request the ASCII alphabet, define the record formats, and define the ddname (with JCL).

In addition, if your program processes signed numeric data items from ASCII files, define the numeric data as zoned decimal items with separate signs, that is, as USAGE DISPLAY and with the SEPARATE phrase of the SIGN clause.

The CODEPAGE compiler option has no effect on the code page used for conversions between ASCII and EBCDIC for ASCII tape support. For information about how CCSIDs used for the ASCII tape support are selected and what the default CCSIDs are, see the z/OS DFSMS documentation.

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

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

In the FD entry 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 use 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 z/OS, 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 is created using Enterprise COBOL.

Related references

z/OS DFSMS: Using Data Sets (Character data conversion)

Chapter 10. Processing VSAM files

Virtual storage access method (VSAM) is an access method for files on direct-access storage devices. With VSAM you can load files, retrieve records from files, update files, and add, replace, and delete records in files.

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 following table shows how VSAM terms differ from COBOL terms and other terms that you might be familiar with.

VSAM term	COBOL term	Similar non-VSAM term
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		Block
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 documentation refers to either a COBOL file or a VSAM data set.

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

Related concepts

[“VSAM files” on page 190](#)

Related tasks

[“Defining VSAM file organization and records” on page 191](#)

[“Coding input and output statements for VSAM files” on page 196](#)

[“Handling errors in VSAM files” on page 204](#)

[“Protecting VSAM files with a password” on page 204](#)

[“Working with VSAM data sets under z/OS and z/OS UNIX” on page 205](#)

[“Improving VSAM performance” on page 211](#)

Related references

z/OS DFSMS: Using Data Sets

z/OS DFSMS Macro Instructions for Data Sets

z/OS DFSMS: Access Method Services for Catalogs

[“Allocation of record areas for VSAM files” on page 211](#)

[“Extended addressability support” on page 212](#)

VSAM files

The physical organization of VSAM data sets differs considerably from the organizations used by other access methods.

VSAM data sets are held in control intervals (CI) and control areas (CA). The size of the CI and CA 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, which 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 that contains 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 information, 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 characteristics of the different types of VSAM data sets.

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

Table 23. Comparison of VSAM data-set types (continued)

Characteristic	Entry-sequenced data set (ESDS)	Key-sequenced data set (KSDS)	Relative-record data set (RRDS)
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 192](#)

[“Specifying indexed organization for VSAM files” on page 192](#)

[“Specifying relative organization for VSAM files” on page 193](#)

[“Defining VSAM files” on page 205](#)

Defining VSAM file organization and records

Use an entry in the FILE-CONTROL paragraph in the ENVIRONMENT DIVISION to define the file organization and access modes for the VSAM 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 by using the RECORD clause.

Important: You can process VSAM data sets in Enterprise COBOL programs only after you define them by using access method services.

Table 24. VSAM file organization, access mode, and record format					
File organization	Sequential access	Random access	Dynamic access	Fixed length	Variable length
VSAM sequential (ESDS)	Yes	No	No	Yes	Yes

Table 24. VSAM file organization, access mode, and record format (continued)

File organization	Sequential access	Random access	Dynamic access	Fixed length	Variable length
VSAM indexed (KSDS)	Yes	Yes	Yes	Yes	Yes
VSAM relative (RRDS)	Yes	Yes	Yes	Yes	Yes

Related tasks

[“Specifying sequential organization for VSAM files” on page 192](#)

[“Specifying indexed organization for VSAM files” on page 192](#)

[“Specifying relative organization for VSAM files” on page 193](#)

[“Specifying access modes for VSAM files” on page 194](#)

[“Defining record lengths for VSAM files” on page 195](#)

[“Using file status keys” on page 247](#)

[“Using VSAM status codes \(VSAM files only\)” on page 248](#)

[“Defining VSAM files” on page 205](#)

Specifying sequential organization for VSAM files

Identify VSAM ESDS files in a 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 190](#)

Specifying indexed organization for VSAM files

Identify a VSAM KSDS file in a COBOL program by using the `ORGANIZATION IS INDEXED` clause. Code a prime key for the record by using the `RECORD KEY` clause. You can also use alternate keys and an alternate index.

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

In the example above, *data-name* is the name of the prime key field as you define it in the record description entry in the `DATA DIVISION`. The prime key data item can be class alphabetic, alphanumeric, DBCS, numeric, national, or UTF-8. If it has `USAGE NATIONAL`, the prime key can be category national, or can be a national-edited, numeric-edited, national decimal, or national floating-point data item. The collation of record keys is based on the binary value of the keys regardless of the class or category of the keys.

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

```
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.
```

Related concepts

[“VSAM files” on page 190](#)

Related tasks

[“Using alternate keys” on page 193](#)

[“Using an alternate index” on page 193](#)

Related references

RECORD KEY clause (*Enterprise COBOL for z/OS Language Reference*)

Classes and categories of data (*Enterprise COBOL for z/OS Language Reference*)

Using alternate keys

In addition to the primary key, you can code one or more alternate keys for a VSAM KSDS file. By using alternate keys, you can access an indexed file to read records in some sequence other than the prime-key sequence.

Alternate keys do not need to be unique. More than one record could be accessed if alternate keys are coded to allow duplicates. For example, you could access the file through employee department rather than through employee number.

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

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

In the example above, *data-name* is the name of the alternate key field as you define it in the record description entry in the `DATA DIVISION`. Alternate key data items, like prime key data items, can be class alphabetic, alphanumeric, DBCS, numeric, national, or UTF-8. The collation of alternate keys is based on the binary value of the keys regardless of the class or category of the keys.

Using an alternate index

To use an alternate index for a VSAM KSDS file, you need to define a data set called the *alternate index (AIX)* by using access method services.

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 tasks

[“Creating alternate indexes” on page 206](#)

Specifying relative organization for VSAM files

Identify VSAM RRDS files in a COBOL program by using 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:

```
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 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 is a valid relative record number.

Alternate indexes are not supported for VSAM RRDS.

Related concepts

[“VSAM files” on page 190](#)

[“Fixed-length and variable-length RRDS” on page 194](#)

Related tasks

[“Using variable-length RRDS” on page 194](#)

[“Defining VSAM files” on page 205](#)

Fixed-length and variable-length RRDS

In an RRDS that has fixed-length records, each record occupies one slot. You store and retrieve records according to the relative record number of the slot. A variable-length RRDS does not have slots; instead, the free space that you define allows for more efficient record insertions.

When you load an RRDS that has fixed-length records, you have the option of skipping over slots and leaving them empty. When you load an RRDS that has variable-length records, you can skip over relative record numbers.

Using variable-length RRDS

To use relative-record data sets (RRDS) that have variable-length records, you must use VSAM variable-length RRDS support.

Do these steps:

1. Define the file with the `ORGANIZATION IS RELATIVE` clause.
2. Use FD entries to describe the records with variable-length sizes.
3. Define the VSAM file through access-method services as an RRDS.

Related tasks

[“Defining VSAM files” on page 205](#)

Related references

z/OS DFSMS: Access Method Services for Catalogs

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. To determine if you want to alter the COBOL runtime `ACCESS IS DYNAMIC` logic path, see "VSAM dynamic access optional logic path" in the *Performance Tuning Guide*.

Related tasks

[“Reading records from a VSAM file” on page 201](#)

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.

If your program processes those employees who earn between \$15.00 and \$20.00 per hour and those who earn \$25.00 per hour and above, using dynamic access of VSAM files, the program would:

1. Retrieve the first record randomly (with a random-retrieval READ) based on the key of 1500.
2. Read sequentially (using READ NEXT) until the salary field exceeds 2000.
3. Retrieve the next record randomly, based on a key of 2500.
4. Read sequentially until the end of the file.

Related tasks

[“Reading records from a VSAM file” on page 201](#)

Defining record lengths for VSAM files

You can define VSAM records to be fixed or variable in length. COBOL determines the record format from the RECORD clause and the record descriptions that are associated with the FD entry for a 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.

Related tasks

[“Defining fixed-length records” on page 195](#)

[“Defining variable-length records” on page 196](#)

Related references

[“FILE SECTION entries” on page 12](#)
Enterprise COBOL for z/OS Migration Guide

Defining fixed-length records

To define VSAM records as fixed length, use one of these 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-3</i> bytes	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 that are associated with the file as the same size; and code none with an OCCURS DEPENDING ON clause.		The fixed size that you coded	

Related references

RECORD clause (*Enterprise COBOL for z/OS Language Reference*)

Defining variable-length records

To define VSAM records as variable length, use one of these coding options.

RECORD clause	Clause format	Maximum record length	Comments
Code RECORD IS VARYING FROM <i>integer-6</i> TO <i>integer-7</i> .	3	<i>integer-7</i> bytes	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-4</i> TO <i>integer-5</i> .	2	<i>integer-5</i> bytes	The minimum record length is <i>integer-4</i> bytes.
Omit the RECORD clause, but code multiple level-01 records that are associated with the file and are of different sizes or 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.

When you specify a READ INTO statement for a format-V file, the record size that is 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 references

RECORD clause (*Enterprise COBOL for z/OS Language Reference*)

Coding input and output statements for VSAM files

Use the COBOL statements shown below to process 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-O, or EXTEND)

The following table shows the possible combinations of statements and open modes for sequential files (ESDS). The X indicates that you can use a statement with the open mode shown at the top of the column.

Table 27. I/O statements for VSAM sequential files

Access mode	COBOL statement	OPEN INPUT	OPEN OUTPUT	OPEN I-O	OPEN EXTEND
Sequential	OPEN	X	X	X	X
	WRITE		X		X
	START				
	READ	X		X	
	REWRITE			X	
	DELETE				
	CLOSE	X	X	X	X

The following table shows the possible combinations of statements and open modes that you can use with indexed (KSDS) files and relative (RRDS) files. The X indicates that you can use the statement with the open mode shown at the top of the column.

Table 28. I/O statements for VSAM relative and indexed files

Access mode	COBOL statement	OPEN INPUT	OPEN OUTPUT	OPEN I-O	OPEN EXTEND
Sequential	OPEN	X	X	X	X
	WRITE		X		X
	START	X		X	
	READ	X		X	
	REWRITE			X	
	DELETE			X	
	CLOSE	X	X	X	X
Random	OPEN	X	X	X	
	WRITE		X	X	
	START				
	READ	X		X	
	REWRITE			X	
	DELETE			X	
	CLOSE	X	X	X	

Table 28. I/O statements for VSAM relative and indexed files (continued)

Access mode	COBOL statement	OPEN INPUT	OPEN OUTPUT	OPEN I-O	OPEN EXTEND
Dynamic	OPEN	X	X	X	
	WRITE		X	X	
	START	X		X	
	READ	X		X	
	REWRITE			X	
	DELETE			X	
	CLOSE	X	X	X	

The fields that 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” on page 198](#)

Related tasks

[“Opening a file \(ESDS, KSDS, or RRDS\)” on page 198](#)

[“Reading records from a VSAM file” on page 201](#)

[“Updating records in a VSAM file” on page 201](#)

[“Adding records to a VSAM file” on page 202](#)

[“Replacing records in a VSAM file” on page 203](#)

[“Deleting records from a VSAM file” on page 203](#)

[“Closing VSAM files” on page 203](#)

Related references

File status key (*Enterprise COBOL for z/OS Language Reference*)

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 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 201](#)

Opening a file (ESDS, KSDS, or RRDS)

Before you can use WRITE, START, READ, REWRITE, or DELETE statements to process records in a file, you must first open the file with an OPEN statement.

Whether a file is available or optional affects OPEN processing, file creation, and the resulting file status key. For example, if you open in EXTEND, I-O, or INPUT mode a nonexistent non-OPTIONAL file, the result is an OPEN error, and file status 35 is returned. If the file is OPTIONAL, however, the same OPEN statement returns file status 05, and, for open modes EXTEND and I-O, creates the file.

An OPEN operation works successfully only if you set fixed file attributes in the DD statement or data-set label for a file, and specify consistent attributes for the file in the SELECT clause and FD entries of your

COBOL program. Mismatches in the following items result in a file status key 39 and the failure of the OPEN statement:

- 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 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.

Note: The VSAMOPENFS option affects the file status key reported from successful OPEN statements on VSAM files. If the VSAMOPENFS (COMPAT) option is in effect, you will get the status value of 97 when a VSAM OPEN statement is successfully verified. For details about the VSAMOPENFS option, see [“VSAMOPENFS” on page 414](#).

Related tasks

[“Opening an empty file” on page 199](#)

[“Opening a loaded file \(a file with records\)” on page 200](#)

Related references

[“Statements to load records into a VSAM file” on page 200](#)

[“VSAMOPENFS” on page 414](#)

Opening an empty file

To open a file that has never contained records (an empty file), use a form of the OPEN statement.

Depending on the type of file that you are opening, use one of the following statements:

- OPEN OUTPUT for ESDS files.
- OPEN OUTPUT or OPEN EXTEND for KSDS and RRDS files. (Either coding has the same effect.) If you 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 available each time a program is run. You can define files opened in INPUT, I-0, or OUTPUT mode as optional by defining them with the SELECT OPTIONAL clause in the FILE-CONTROL paragraph.

Initially loading a file sequentially: Initially loading a file means writing records into the file for the first time. Doing so 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). (Sequential processing is faster than random or dynamic processing.)
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-0 or OPEN EXTEND has a negative effect 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 because it has 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.

When you are loading an extended-format VSAM data set, file status 30 will occur for the `OPEN` if z/OS DFSMS system-managed buffering sets the buffering to local shared resources (LSR). To successfully load the VSAM data set in this case, specify `ACCBIAS=USER` in the `DD AMP` parameter for the VSAM data set to bypass system-managed buffering.

Loading a VSAM data set with access method services: You can load or update a VSAM data set by using the access method services `REPRO` command. Use `REPRO` whenever possible.

Related tasks

[“Opening a loaded file \(a file with records\)” on page 200](#)

Related references

[“Statements to load records into a VSAM file” on page 200](#)
[z/OS DFSMS: Access Method Services for Catalogs \(REPRO\)](#)

Statements to load records into a VSAM file

Use the statements shown below 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 199](#)
[“Updating records in a VSAM file” on page 201](#)

Opening a loaded file (a file with records)

To open a file that already contains records, use `OPEN INPUT`, `OPEN I-O`, 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 199](#)

[“Working with VSAM data sets under z/OS and z/OS UNIX” on page 205](#)

Related references

[“Statements to load records into a VSAM file” on page 200](#)

z/OS DFSMS: Access Method Services for Catalogs

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 198](#)

Related tasks

[“Specifying access modes for VSAM files” on page 194](#)

Updating records in a VSAM file

To update a VSAM file, use these PROCEDURE DIVISION statements.

Table 30. Statements to update records in a VSAM file

Access method	ESDS	KSDS	RRDS
ACCESS IS SEQUENTIAL	OPEN EXTEND WRITE CLOSE or OPEN I-O READ REWRITE CLOSE	OPEN EXTEND WRITE CLOSE or OPEN I-O READ REWRITE DELETE CLOSE	OPEN EXTEND WRITE CLOSE or OPEN I-O READ REWRITE DELETE CLOSE
ACCESS IS RANDOM	Not applicable	OPEN I-O READ WRITE REWRITE DELETE CLOSE	OPEN I-O READ WRITE REWRITE DELETE CLOSE
ACCESS IS DYNAMIC (sequential processing)	Not applicable	OPEN I-O READ NEXT WRITE REWRITE START DELETE CLOSE	OPEN I-O READ NEXT WRITE REWRITE START DELETE CLOSE
ACCESS IS DYNAMIC (random processing)	Not applicable	OPEN I-O READ WRITE REWRITE DELETE CLOSE	OPEN I-O READ WRITE REWRITE DELETE CLOSE

Related references

“Statements to load records into a VSAM file” on page 200

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 a record in a VSAM file, use REWRITE on a file that you opened as I-0. If the file was not opened as I-0, the record is not rewritten and the status key is set to 49. Check the file status key after each REWRITE statement.

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

To replace a record randomly or dynamically, you do not have to first READ the record. Instead, locate the record you want to replace as follows:

- For indexed files, move the record key to the RECORD KEY data item, and then issue the REWRITE.
- For relative files, move the relative record number to the RELATIVE KEY data item, and then issue the REWRITE.

Deleting records from a VSAM file

To remove an existing record from an indexed or relative file, open the file I-0 and use the DELETE statement. You cannot use DELETE 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 does not have to 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 a 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 programs in the run unit are closed.
- When the run unit ends abnormally, if the TRAP(ON) runtime option has been set, all open files defined in any COBOL programs in the run unit are closed.
- When Language Environment condition handling has 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 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 language constructs such as a C longjmp statement.

- When you issue CANCEL for a COBOL subprogram, any open nonexternal files defined in that program are closed.

- When a COBOL subprogram with the INITIAL attribute returns control, any open nonexternal files defined in that program are closed.
- When a thread of a multithreaded application ends, both external and nonexternal files that were opened from within that same thread are closed.

File status key data items in the DATA DIVISION are set when these implicit CLOSE operations are performed, but your EXCEPTION/ERROR declarative is not invoked.

Errors: If you open a VSAM file in a multithreaded application, you must close it from the same thread of execution. Attempting to close the file from a different thread results in a close failure with file-status condition 90.

Handling errors in VSAM files

When an input 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 if you 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 error.

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 status 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 each 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 runtime option ABTERMENC (ABEND).

Related tasks

[“Handling errors in input and output operations” on page 243](#)

[“Using VSAM status codes \(VSAM files only\)” on page 248](#)

Related references

z/OS DFSMS Macro Instructions for Data Sets (VSAM macro return and reason codes)

Protecting VSAM files with a password

Although the preferred security mechanism on a z/OS system is RACF, Enterprise COBOL also supports using explicit passwords on VSAM files to prevent unauthorized access and update.

To use explicit passwords, code the PASSWORD clause in the FILE-CONTROL paragraph. 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 each alternate index in order to be fully password protected. Where you place the PASSWORD clause is 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

The following example shows the COBOL code used for a VSAM indexed file that has 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 z/OS and z/OS UNIX

Be aware of special coding considerations for VSAM files under z/OS and z/OS UNIX for access method services (IDCAMS) commands, environment variables, and JCL.

A VSAM file is *available* if all of the following conditions 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.
- It has previously contained a record.

A VSAM file is *unavailable* if it has never contained a record, even if you have defined the file.

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 8](#)

[“Defining VSAM files” on page 205](#)

[“Creating alternate indexes” on page 206](#)

[“Allocating VSAM files” on page 208](#)

[“Sharing VSAM files through](#)

[RLS” on page 209](#)

Defining VSAM files

You can process VSAM entry-sequenced, key-sequenced, and relative-record data sets in Enterprise COBOL 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 `DEFINE CLUSTER access-method services` command to define 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)
```

If you define a data set in this way, all records are 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` uses the length specified in `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.

Related tasks

[“Creating alternate indexes” on page 206](#)

[“Allocating VSAM files” on page 208](#)

[“Specifying relative organization for VSAM files” on page 193](#)

Related references

z/OS DFSMS: Access Method Services for Catalogs

Creating alternate indexes

An alternate index provides access to the records in a data set that uses 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, do these steps:

1. Define the alternate index by using the `DEFINE ALTERNATEINDEX` command. In it, specify these items:

- 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
- Whether 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 in the `FILE-CONTROL` paragraph. The `ALTERNATE RECORD KEY` definitions must match the definitions in the catalog entry. Any password entries that you 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, specify these items:

- 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. `BLDINDEX` reads all the records in the VSAM indexed data set (or base cluster) and extracts the data needed to build the alternate index.

Alternatively, you can use the runtime option `AIXBLD` to build the alternate index at run time. However, this option might adversely affect performance.

[“Example: entries for alternate indexes” on page 207](#)

Related tasks

[“Using an alternate index” on page 193](#)

Related references

Language Environment Programming Reference (AIXBLD (COBOL only))

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 that has two alternate indexes.

Using JCL:

```
//MAINA    DD    DSN=clustername,DISP=OLD    (1)
//MAINA1   DD    DSN=path1,DISP=OLD        (2)
//MAINA2   DD    DSN=path2,DISP=OLD        (3)
```

Using environment variables:

```
export MAINA=DSN(clustername),OLD        (1)
```

```

export MAINA=DSN(path1),OLD           (2)
export MAINA=DSN(path2),OLD           (3)
FILE-CONTROL.
  SELECT MAIN-FILE ASSIGN TO MAINA      (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 *clustername*.
- (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 206](#)

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, so you need to specify only minimal DD or environment variable information.

Allocation of VSAM files (indexed, relative, and sequential) follows the general rules for the allocation of COBOL files.

When you use an environment variable to allocate a VSAM file, the variable name must be in uppercase. Usually the input and data buffers are the only variables that you are concerned about. You must specify these options in the order shown, but no others:

1. DSN(*dsname*), where *dsname* is the name of the base cluster
2. OLD or SHR

The basic DD statement that you need for VSAM files and the corresponding export command are these:

```

//ddname DD DSN=dsname,DISP=SHR,AMP=AMORG
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 that supplements 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 required only 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 a VSAM data set with an environment variable.

For a VSAM base cluster, specify the same system-name (ddname or environment variable name) that you specify in the ASSIGN clause after the SELECT clause.

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 a system-name 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 FILE-CONTROL paragraph).
- 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 together with the 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 164](#)

Related references

[MVS Program Management: User's Guide and Reference](#)

Sharing VSAM files through RLS

By using the VSAM JCL parameter RLS, you can specify record-level sharing with VSAM. Specifying RLS is the only way to request the RLS mode when running COBOL programs.

Use RLS=CR when consistent read protocols are required, and RLS=NRI when no read integrity protocols are required. You cannot specify RLS if you allocate your VSAM data set with an environment variable

Related tasks

[“Preventing update problems with VSAM files in RLS mode” on page 210](#)

[“Handling errors in VSAM files in RLS mode” on page 210](#)

Related references

[“Restrictions when using RLS” on page 210](#)

Preventing update problems with VSAM files in RLS mode

When you open a VSAM data set in RLS mode for I-O (updates), the first READ causes an exclusive lock of the record regardless of the value of RLS (RLS=CR or RLS=NRI) that you specify.

If the COBOL file is defined as ACCESS RANDOM, VSAM releases the exclusive lock on the record after a WRITE or REWRITE statement is executed or a READ statement is executed for another record. When a WRITE or REWRITE is done, VSAM writes the record immediately.

However, if the COBOL file is defined as ACCESS DYNAMIC, VSAM does not release the exclusive lock on the record after a WRITE or REWRITE statement, nor after a READ statement, unless the I-O statement causes VSAM to move to another control interval (CI). As a result, if a WRITE or REWRITE was done, VSAM does not write the record until processing is moved to another CI and the lock is released. When you use ACCESS DYNAMIC, one way to cause the record to be written immediately, to release the exclusive lock immediately, or both, is to define the VSAM data set to allow only one record per CI.

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 executed. Another user might update the record.

The locking rules for RLS=CR can cause the application to wait for availability of a record lock. This wait might slow down the READ for input. You might need to modify your application logic to use RLS=CR. Do not use the RLS 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 good to issue an OPEN or START *immediately* before a READ. If there is a delay between the OPEN or START and the 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

When you use RLS mode, several restrictions apply to VSAM cluster attributes and to runtime options.

Be aware of these restrictions:

- The VSAM cluster attributes KEYRANGE and IMBED are not supported when you open a VSAM file.
- The VSAM cluster attribute REPLICATE is not recommended because the benefits are negated by the system-wide buffer pool and potentially large CF cache structure in the storage hierarchy.
- The AIXBLD runtime option is not supported when you open a VSAM file because VSAM does not allow an empty path to be opened. If you need the AIXBLD runtime option to build the alternate index data set, open the VSAM data set in non-RLS mode.
- Temporary data sets are not allowed.

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 such failures, and there is no feedback code. You can have COBOL programs check the first 2 bytes of the second file status area for 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.

Related tasks

[“Using VSAM status codes \(VSAM files only\)” on page 248](#)

Allocation of record areas for VSAM files

For reentrant COBOL programs, the record areas for VSAM files are allocated above the 16 MB line by default.

If you specify the DATA(24) compiler option, the VSAM record areas and other dynamic storage areas are allocated in storage below 16 MB.

Programs that pass data in VSAM file records as CALL . . . USING parameters to AMODE 24 subprograms are impacted. You can recompile such programs with the DATA(24) compiler option, or use the Language Environment HEAP runtime option, to ensure that the records are addressable by the AMODE 24 programs.

Improving VSAM performance

Your system programmer is most likely responsible for tuning the performance of COBOL and VSAM. As an application programmer, you can control the aspects of VSAM that are listed in the following table.

Aspect of VSAM	What you can do	Rationale and comments
Invoking access methods service	Build your alternate indexes in advance, using IDCAMS.	
Buffering	For sequential access, request more data buffers; for random access, request more index buffers. Specify both BUFND and BUFNI if ACCESS IS DYNAMIC. Avoid coding additional buffers unless your application will run interactively; then code buffers only when response-time problems arise that might be caused by delays in input and output.	The default is one index (BUFNI) and two data buffers (BUFND).
Loading records, using access methods services	Use the access methods service REPRO command when: <ul style="list-style-type: none"> • The target indexed data set already contains records. • The input sequential data set contains records to be updated or inserted into the indexed data set. If you use a COBOL program to load the file, use OPEN OUTPUT and ACCESS SEQUENTIAL.	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.

Table 31. **Methods for improving VSAM performance** (continued)

Aspect of VSAM	What you can do	Rationale and comments
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.
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, 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 194

z/OS DFSMS: Using Data Sets (Building a resource pool, Selecting the optimal percentage of free space)

Related references

z/OS DFSMS: Access Method Services for Catalogs

Extended addressability support

You can access VSAM data sets that are defined with the extended addressability attribute, use those VSAM data sets in COBOL programs without COBOL source changes, and maintain compatibility with previous versions of COBOL.

With extended addressability support, you can define larger VSAM data sets outside of COBOL. The 4 GB VSAM architectural limit for data set size imposed by using the 4-byte field for the relative byte address (RBA) is eliminated.

To use the extended addressability, the VSAM data set must be Storage Management Subsystem (SMS)-managed and be defined as extended format. The size limit for a VSAM data set is determined in either of the following ways:

- Control Interval (CI) size multiplied by 4 GB

- Volume size multiplied by 59

For example, a 4 KB CI size yields a maximum data set size of 16 TB, and a 32 KB CI size yields a maximum data set size of 128 TB. A 4 KB CI size is preferred by many applications for performance reasons. For extended-format data sets that grow beyond 4 GB, the processing time does not increase.

Extended addressability is also supported for programs compiled with earlier versions: VS COBOL II programs compiled with RES and any later compilers.

Extended addressability and extended format are not the same concept. Extended format is a prerequisite for extended addressability. Extended format is a technique that affects the way of storing count key data (CKD) in a 3390/3380 logical track. Extended format implements data striping and increases the performance and the reliability of an I/O operation. If a data set is allocated as an extended-format data set, 32 bytes are added to each physical block.

Restriction: Extended addressability was introduced for KSDS data sets in DFSMS/MVS 1.3. Since DFSMS/MVS 1.4, extended addressability is supported in record level sharing (RLS). With DFSMS/MVS 1.5, support for extended addressability is extended to all other VSAM record organizations.

Related tasks

z/OS DFSMS: Using Data Sets

Chapter 11. Processing line-sequential files

Line-sequential files reside in the z/OS UNIX file system and can contain both printable characters and control characters as data. Each record ends with an EBCDIC newline character (X'15'), which is not included in the record length.

Because line-sequential files are sequential, 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 a program, code 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 tasks

[“Defining line-sequential files and records in COBOL” on page 215](#)

[“Allocating line-sequential files” on page 216](#)

[“Coding input-output statements for line-sequential files” on page 217](#)

[“Handling errors in line-sequential files” on page 219](#)

UNIX System Services User's Guide

Defining line-sequential files and records in COBOL

Use the FILE-CONTROL paragraph in the ENVIRONMENT DIVISION to define the files in a COBOL program as line-sequential files, and to associate the files with the corresponding 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.
```

The ASSIGN *assignment-name* clause must not include an organization field (S- or AS-) before the external name. The ACCESS phrase and the FILE STATUS clause are optional.

Related tasks

[“Describing the structure of a line-sequential file” on page 215](#)

[“Allocating line-sequential files” on page 216](#)

[“Coding input-output statements for line-sequential files” on page 217](#)

Related references

[“Control characters in line-sequential files” on page 216](#)

Describing the structure of a line-sequential file

In the FILE SECTION, 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 in bytes of the records by using 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

[“Allocating line-sequential files” on page 216](#)

[“Coding input-output statements for line-sequential files” on page 217](#)

Related references

Data division--file description entries
(*Enterprise COBOL for z/OS Language Reference*)

Control characters in line-sequential files

A line-sequential file can contain control characters. Be aware though that if a line-sequential file contains a newline character (X '15 '), the newline character will function as a record delimiter.

Control characters other than newline are treated as data and are part of the record.

Allocating line-sequential files

You can allocate a line-sequential file in the z/OS UNIX file system by using either a DD statement or an environment variable. Allocation of line-sequential files follows the general rules for allocating COBOL files.

To allocate a line-sequential file, code a DD allocation or an environment variable that has a name that matches the external name in the 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 that has a value of PATH(*absolute-path-name*). No other values can be specified.

For example, to have your program use z/OS UNIX file /u/myfiles/commuterfile for a COBOL file that has an *assignment-name* of COMMUTR, you can use the following command:

```
export COMMUTR="PATH(/u/myfiles/commuterfile)"
```

Related tasks

[“Allocating files” on page 164](#)

[“Defining line-sequential files and records in COBOL” on page 215](#)

Related references

[MVS Program Management: User's Guide and Reference](#)

Coding input-output statements for line-sequential files

Code the input and output statements shown below to process a line-sequential file.

OPEN

To initiate the processing of a file.

You can open a line-sequential file as INPUT, OUTPUT, or EXTEND. You cannot open a line-sequential file as I-O.

READ

To read a record from a file.

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

WRITE

To create a record in a file.

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

CLOSE

To release the connection between a file and the program.

Related tasks

[“Defining line-sequential files and records in COBOL” on page 215](#)

[“Describing the structure of a line-sequential file” on page 215](#)

[“Opening line-sequential files” on page 217](#)

[“Reading records from line-sequential files” on page 218](#)

[“Adding records to line-sequential files” on page 218](#)

[“Closing line-sequential files” on page 219](#)

[“Handling errors in line-sequential files” on page 219](#)

Related references

OPEN statement (*Enterprise COBOL for z/OS Language Reference*)

READ statement (*Enterprise COBOL for z/OS Language Reference*)

WRITE statement (*Enterprise COBOL for z/OS Language Reference*)

CLOSE statement (*Enterprise COBOL for z/OS 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” on page 218](#)

[“Adding records to line-sequential files” on page 218](#)

[“Closing line-sequential files” on page 219](#)

[“Allocating line-sequential files” on page 216](#)

Related references

OPEN statement (*Enterprise COBOL for z/OS Language Reference*)

CLOSE statement (*Enterprise COBOL for z/OS Language Reference*)

Reading records from line-sequential files

To read from a line-sequential file, open the file and use the READ statement. 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 newline 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.)

- End-of-file is encountered.

The remainder of the record area is filled with spaces. (Record area is longer than the file record.)

Related tasks

[“Opening line-sequential files” on page 217](#)

[“Adding records to line-sequential files” on page 218](#)

[“Closing line-sequential files” on page 219](#)

[“Allocating line-sequential files” on page 216](#)

Related references

OPEN statement (*Enterprise COBOL for z/OS Language Reference*)

WRITE statement (*Enterprise COBOL for z/OS 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. Zoned decimal data items must be unsigned or declared with the SEPARATE phrase of the SIGN clause if signed.

Related tasks

[“Opening line-sequential files” on page 217](#)

[“Reading records from line-sequential files” on page 218](#)
[“Closing line-sequential files” on page 219](#)
[“Allocating line-sequential files” on page 216](#)

Related references

OPEN statement (*Enterprise COBOL for z/OS Language Reference*)

WRITE statement (*Enterprise COBOL for z/OS 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) runtime 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 217](#)

[“Reading records from line-sequential files” on page 218](#)

[“Adding records to line-sequential files” on page 218](#)

[“Allocating line-sequential files” on page 216](#)

Related references

CLOSE statement (*Enterprise COBOL for z/OS Language Reference*)

Handling errors in line-sequential files

When an input or output statement fails, COBOL does not take corrective action for you. You choose whether your program should continue running after an input or output statement fails.

COBOL provides these language elements 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, your program might continue, but the results will probably not be what you expected.

Related tasks

[“Coding input-output statements for line-sequential files” on page 217](#)

[“Handling errors in input and output operations” on page 243](#)

Chapter 12. Sorting and merging files

You can arrange records in a particular sequence by using a SORT or MERGE statement. You can mix SORT and MERGE statements in the same COBOL program.

Note: The SORT statement, sort processes, and sort restrictions that are described in this topic relate to the format 1 SORT statement only. For more information about sorting a table by using the format 2 SORT statement, see [“Sorting a table” on page 87](#).

SORT statement

Accepts input (from a file or an internal procedure) that is not in sequence, and produces output (to a file or an internal procedure) in a requested sequence. You can add, delete, or change records before or after they are sorted.

MERGE statement

Compares records from two or more sequenced files and combines them in order. You can add, delete, or change records after they are merged.

A program can contain any number of sort and merge operations. They can be the same operation performed many times or different operations. However, one operation must finish before another begins.

With Enterprise COBOL, 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 that contain SORT or MERGE statements can reside above or below the 16 MB line.

The steps you take to sort or merge are generally as follows:

1. Describe the sort or merge file to be used for sorting or merging.
2. Describe the input to be sorted or merged. If you want to process the records before you sort them, code an input procedure.
3. Describe the output from sorting or merging. If you want to process the records after you sort or merge them, code an output procedure.
4. Request the sort or merge.
5. Determine whether the sort or merge operation was successful.

Restrictions:

- You cannot run a COBOL program that contains SORT or MERGE statements under z/OS UNIX. This restriction includes BPXBATCH.
- You cannot use SORT or MERGE statements in programs compiled with the THREAD option. This includes programs that use object-oriented syntax and multithreaded applications, both of which require the THREAD option. In addition, the COBOL program that uses SORT or MERGE statements cannot call directly or indirectly other applications that require z/OS Unix System Services or applications that use multithreading. For example, the JVM uses both of them.
- You cannot use the DFSORT conventional technique. The conventional technique is used in the following cases:
 - The NOBLKSET option is specified. BLKSET is the default when invoking DFSORT.
 - Tape device is used for intermediate work storage.
 - L5 is used in the RECORD statement of DFSORT OPTION control. L5 specifies the average record length. Instead of using L5, the same can be specified by using the AVGLEN=n statement.

To assist migration of legacy COBOL programs, toleration runtime APARs PH20569(z/OS 2.2) and PH21261(z/OS 2.3/2.4) are provided for programs running in AMODE 31. There is no toleration for AMODE 64. For details, see Using DFSORT option NOBLKSET (*Enterprise COBOL Migration Guide*).

Related concepts

[“Sort and merge process” on page 222](#)

Related tasks

[“Sorting](#)

[a table” on page 87](#)

[“Describing the sort or merge file” on page 222](#)

[“Describing the input to sorting or merging” on page 223](#)

[“Describing the output from sorting or merging” on page 225](#)

[“Requesting the sort or merge” on page 228](#)

[“Determining whether the sort or merge was successful” on page 231](#)

[“Stopping a sort or merge operation prematurely” on page 232](#)

[“Improving sort performance with FASTSORT” on page 232](#)

[“Controlling sort behavior” on page 235](#)

[DFSORT Application Programming Guide](#)

Related references

[“CICS SORT application restrictions” on page 238](#)

[SORT statement \(*Enterprise COBOL for z/OS Language Reference*\)](#)

[MERGE statement \(*Enterprise COBOL for z/OS Language Reference*\)](#)

Sort and merge process

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

If there are multiple keys, the records are first sorted according to the content of the first (or primary) key, then according to the content of the second key, and so on.

To sort a file, use the format 1 SORT statement.

During the merging of two or more files (which must already be sorted), the records are combined and ordered according to the contents of one or more keys in each record. You can order the records in either ascending or descending order of each key. As with sorting, the records are first ordered according to the content of the primary key, then according to the content of the second key, and so on.

Use MERGE . . . USING to name the files that you want to combine into one sequenced file. The merge operation compares keys in the records of the input files, and passes the sequenced records one by one to the RETURN statement of an output procedure or to the file that you name in the GIVING phrase.

Related tasks

[“Setting sort or merge criteria” on page 229](#)

Related references

[SORT statement \(*Enterprise COBOL for z/OS Language Reference*\)](#)

[MERGE statement \(*Enterprise COBOL for z/OS Language Reference*\)](#)

Describing the sort or merge file

Describe the sort file to be used for sorting or merging. You need SELECT clauses and SD entries even if you are sorting or merging data items only from WORKING-STORAGE or LOCAL-STORAGE.

Code as follows:

1. Write one or more SELECT clauses in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION to name a sort file. For example:

```
ENVIRONMENT DIVISION.  
INPUT-OUTPUT SECTION.
```

```
FILE-CONTROL.  
  SELECT Sort-Work-1 ASSIGN TO SortFile.
```

Sort-Work-1 is the name of the file in your program. Use this name to refer to the file.

2. Describe the sort file in an SD entry in the FILE SECTION of the DATA DIVISION. Every SD entry must contain a record description. For example:

```
DATA DIVISION.  
FILE SECTION.  
SD Sort-Work-1  
  RECORD CONTAINS 100 CHARACTERS.  
01 SORT-WORK-1-AREA.  
  05 SORT-KEY-1 PIC X(10).  
  05 SORT-KEY-2 PIC X(10).  
  05 FILLER PIC X(80).
```

The file described in an SD entry is the working file used for a sort or merge operation. You cannot perform any input or output operations on this file and you do not need to provide a ddname definition for it.

Related references

[“FILE SECTION entries” on page 12](#)

Describing the input to sorting or merging

Describe the input file or files for sorting or merging by following the procedure below.

1. Write one or more SELECT clauses in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION to name the input files. For example:

```
ENVIRONMENT DIVISION.  
INPUT-OUTPUT SECTION.  
FILE-CONTROL.  
  SELECT Input-File ASSIGN TO InFile.
```

Input-File is the name of the file in your program. Use this name to refer to the file.

2. Describe the input file (or files when merging) in an FD entry in the FILE SECTION of the DATA DIVISION. For example:

```
DATA DIVISION.  
FILE SECTION.  
FD Input-File  
  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 224](#)

[“Requesting the sort or merge” on page 228](#)

Related references

[“FILE SECTION entries” on page 12](#)

Example: describing sort and input files for SORT

The following example shows the ENVIRONMENT DIVISION and DATA DIVISION entries needed to describe sort work files and an input file.

```
ID Division.  
Program-ID. 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 228](#)

Coding the input procedure

To process the records in an input file before they are released to the sort program, use the INPUT PROCEDURE phrase of the format 1 SORT statement.

You can use an input procedure to:

- Release data items to the sort file from WORKING-STORAGE or LOCAL-STORAGE.
- Release records that have already been read elsewhere in the program.
- Read records from an input file, select or process them, and release them to the sort file.

Each input procedure must be contained in either paragraphs or sections. For example, to release records from a table in WORKING-STORAGE or LOCAL-STORAGE to the sort file SORT-WORK-2, you could code as follows:

```

SORT SORT-WORK-2
  ON ASCENDING KEY SORT-KEY
  INPUT PROCEDURE 600-SORT3-INPUT-PROC
. . .
600-SORT3-INPUT-PROC SECTION.
  PERFORM WITH TEST AFTER
    VARYING X1 FROM 1 BY 1 UNTIL X1 = 100
  RELEASE SORT-WORK-2-AREA FROM TABLE-ENTRY (X1)
  END-PERFORM.

```

To transfer records to the sort program, all input procedures must contain at least one RELEASE or RELEASE FROM statement. To release A from X, for example, you can code:

```

MOVE X TO A.
RELEASE A.

```

Alternatively, you can code:

```
RELEASE A FROM X.
```

The following table compares the RELEASE and RELEASE FROM statements.

RELEASE	RELEASE FROM
<pre>MOVE SORT-EXT-RECORD TO SORT-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 227](#)

RELEASE statement (*Enterprise COBOL for z/OS Language Reference*)

Describing the output from sorting or merging

If the output from sorting or merging is a file, describe the file by following the procedure below.

1. Write a SELECT clause in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION to name the output file. For example:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT Output-File ASSIGN TO OutFile.
```

Output-File is the name of the file in your program. Use this name to refer to the file.

2. Describe the output file (or files when merging) in an FD entry in the FILE SECTION of the DATA DIVISION. For example:

```
DATA DIVISION.
FILE SECTION.
FD Output-File
  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” on page 225](#)

[“Requesting the sort or merge” on page 228](#)

Related references

[“FILE SECTION entries” on page 12](#)

Coding the output procedure

To select, edit, or otherwise change sorted records before writing them from the sort work file into another file, use the OUTPUT PROCEDURE phrase of the format 1 SORT statement.

Each output procedure must be contained in either a section or a paragraph. An output procedure must include both of the following elements:

- At least one RETURN statement or one RETURN statement with the INTO phrase

- Any statements necessary to process the records that are made available, one at a time, by the RETURN statement

The RETURN statement makes each sorted record available to the output procedure. (The RETURN statement for a sort file is similar to a READ statement for an input file.)

You can use the AT END and END-RETURN phrases with the RETURN statement. The imperative statements in the AT END phrase are performed after all the records have been returned from the sort file. The END-RETURN explicit scope terminator delimits the scope of the RETURN statement.

If you use RETURN INTO instead of RETURN, the records will be returned to WORKING-STORAGE, LOCAL-STORAGE, or to an output area.

DFSORT coding: If you use DFSORT and a RETURN statement does not encounter an AT END condition before a COBOL program finishes running, the format 1 SORT statement could end abnormally with DFSORT message IEC025A. To avoid this situation, be sure to code the RETURN statement with the AT END phrase. In addition, 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” on page 226](#)

Related references

[“Restrictions on input and output procedures” on page 227](#)

RETURN statement (*Enterprise COBOL for z/OS 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

Several restrictions apply to each input or output procedure called by SORT and to each output procedure called by MERGE.

Observe these restrictions:

- The procedure must not contain any SORT or MERGE statements.

Restriction: In an input or output procedure, you can call a program that follows standard linkage conventions. However, the called program cannot issue a SORT, MERGE, RELEASE, or RETURN statement.

- 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.
- 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 let user-written condition handlers be established in an input or output procedure.

Related tasks

[“Coding the input procedure” on page 224](#)

[“Coding the output procedure” on page 225](#)

Language Environment Programming Guide (Preparing to link-edit and run)

Defining sort and merge data sets

To use DFSORT under z/OS, code DD statements in the runtime JCL to describe the necessary data sets that are listed below.

Sort or merge work

Define a minimum of three data sets: SORTWK01, SORTWK02, SORTWK03, . . . , SORTWK nn (where nn is 99 or less). These data sets cannot be in the z/OS UNIX file system.

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 that contains the sort modules, for example, SYS1.SORTLIB.

Related tasks

[“Controlling sort behavior” on page 235](#)

[“Using checkpoint/restart with DFSORT” on page 237](#)

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 determines that the sort work file is variable length if you code one of the following elements in the SD entry:

- A RECORD IS VARYING clause
- Two or more record descriptions that define records that have different sizes, or records that contain an OCCURS DEPENDING ON clause

You cannot use RECORDING MODE V for the sort work file because the SD entry does not allow the RECORDING MODE clause.

Performance consideration: To improve sort performance of 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 236](#)

[“Controlling sort behavior” on page 235](#)

Requesting the sort or merge

To read records from an input file (files for MERGE) without preliminary processing, use SORT . . . USING or MERGE . . . USING and the name of the input file (files) that you declared in a SELECT clause.

To transfer sorted or merged records from the sort or merge program to another file without any further processing, use SORT . . . GIVING or MERGE . . . GIVING and the name of the output file that you declared in a SELECT clause. For example:

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

For SORT . . . USING or MERGE . . . USING, the compiler generates an input procedure to open the file (files), read the records, release the records to the sort or merge program, and close the file (files). The file (files) must not be open when the SORT or MERGE statement begins execution. For SORT . . . GIVING or MERGE . . . GIVING, the compiler generates an output procedure to open the file, return the records, write the records, and close the file. The file must not be open when the SORT or MERGE statement begins execution.

The USING or GIVING files in a SORT or MERGE statement can be sequential files residing in the z/OS UNIX file system.

[“Example: describing sort and input files for SORT” on page 223](#)

You can also use the FASTSORT compiler option to let IBM DFSORT, or an equivalent product, perform sort input and output instead of the Enterprise COBOL compiler (along with Language Environment run time). Using FASTSORT improves the performance of most sort operations. For details, see [“FASTSORT” on page 359](#).

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 230](#)

Restriction: You cannot use an input procedure with the MERGE statement. The source of input to the merge operation must be a collection of already sorted files. However, if you want an output procedure to be performed on the merged records, use MERGE . . . OUTPUT PROCEDURE. For example:

```
MERGE Merge-Work
  ON ASCENDING KEY Merge-Key
  USING Input-File-1 Input-File-2 Input-File-3
  OUTPUT PROCEDURE ProcessOutput.
```

In the FILE SECTION, you must define *Merge-Work* in an SD entry, and the input files in FD entries.

Related tasks

[“Defining sort and merge data sets” on page 227](#)

Related references

[SORT statement \(Enterprise COBOL for z/OS Language Reference\)](#)

[MERGE statement \(Enterprise COBOL for z/OS Language Reference\)](#)

Setting sort or merge criteria

To set sort or merge criteria, define the keys on which the operation is to be performed.

Note: The process of setting sort criteria that is described in this topic relates to the format 1 SORT statement only. For more information about sorting a table by using the format 2 SORT statement, see [“Sorting a table” on page 87](#).

Do these steps:

1. In the record description of the files to be sorted or merged, define the key or keys.

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 by coding the ASCENDING or DESCENDING KEY phrase, or both. When you code more than one key, some can be ascending, and some descending.

Specify the names of the keys in decreasing order of significance. The leftmost key is the primary key. The next key is the secondary key, and so on.

SORT and MERGE keys can be of class alphabetic, alphanumeric, national, or numeric (but not numeric of USAGE NATIONAL). If it has USAGE NATIONAL, a key can be of category national or can be a national-edited or numeric-edited data item. A key cannot be a national decimal data item or a national floating-point data item.

The collation order for national keys is determined by the binary order of the keys. If you specify a national data item as a key, any COLLATING SEQUENCE phrase in the SORT or MERGE statement does not apply to that key.

You can mix SORT and MERGE statements in the same COBOL program. A program can perform any number of sort or merge operations. However, one operation must end before another can begin.

Related tasks

[“Sorting a table” on page 87](#)

Related references

[DFSORT Application Programming Guide \(SORT control statement\)](#)

SORT statement (*Enterprise COBOL for z/OS Language Reference*)
MERGE statement (*Enterprise COBOL for z/OS Language Reference*)

Example: sorting with input and output procedures

The following example shows the use of an input and an output procedure in a format 1 SORT statement. The example also shows how you can define a primary key (SORT-GRID-LOCATION) and a secondary key (SORT-SHIFT) before using them in the format 1 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.
*****
* This SORT statement will do a 'table sort' using Format 1 SORT, and will *
* sort records in SORT-FILE in an ascending sequence based on the data *
* in sort keys SORT-GRID-LOCATION (primary) and SORT-SHIFT (secondary). *
* The source of the file records is data in table TAB-ENTRY *
* which is acquired in the input procedure 600-SORT3-INPUT and then *
* output back to the table in the output procedure 700-SORT3-OUTPUT. *
*****
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 228](#)

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 default sequence, use the `COLLATING SEQUENCE` phrase of the `SORT` or `MERGE` statement. You can use different collating sequences for each `SORT` or `MERGE` statement in your program.

The `PROGRAM COLLATING SEQUENCE` clause and the `COLLATING SEQUENCE` phrase apply only to keys of class alphabetic or alphanumeric.

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, and define the *alphabet-name* as `STANDARD-1` in the `SPECIAL-NAMES` paragraph.

Related tasks

[“Specifying the collating sequence” on page 7](#)

[“Setting sort or merge criteria” on page 229](#)

Related references

`OBJECT-COMPUTER` paragraph (*Enterprise COBOL for z/OS Language Reference*)

`SORT` statement (*Enterprise COBOL for z/OS Language Reference*)

Classes and categories of data (*Enterprise COBOL for z/OS Language Reference*)

Preserving the original sequence of records with equal keys

You can preserve the order of identical collating records from input to output.

Use one of these techniques:

- Install DFSORT with the `EQUALS` option as the default.
- Provide, at run time, an `OPTION` card that has 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 both the `NOEQUALS` keyword on the `OPTION` card *and* the `DUPLICATES` phrase, or the run unit will end.

Related references

DFSORT Application Programming Guide (`OPTION` control statement)

Determining whether the sort or merge was successful

The DFSORT program returns a completion code of either 0 (successful completion) or 16 (unsuccessful completion) after each sort or merge has finished. The completion code is stored in the `SORT-RETURN` special register.

You should test for successful completion after each `SORT` or `MERGE` statement. For example:

```
SORT SORT-WORK-2
  ON ASCENDING KEY SORT-KEY
  INPUT PROCEDURE IS 600-SORT3-INPUT-PROC
  OUTPUT PROCEDURE IS 700-SORT3-OUTPUT-PROC.
IF SORT-RETURN NOT=0
  DISPLAY "SORT ENDED ABNORMALLY. SORT-RETURN = " SORT-RETURN.
. . .
600-SORT3-INPUT-PROC SECTION.
. . .
700-SORT3-OUTPUT-PROC SECTION.
. . .
```

If you do not reference `SORT-RETURN` anywhere in your program, the COBOL run time tests the completion code. If it is 16, COBOL issues a runtime diagnostic message.

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.

If you test SORT-RETURN for one or more (but not necessarily all) SORT or MERGE statements, the COBOL run time does not check the completion code.

Related tasks

[“Checking for sort errors with NOFASTSRT” on page 234](#)

[“Controlling sort behavior” on page 235](#)

Related references

DFSORT Application Programming Guide (DFSORT messages and return codes)

Stopping a sort or merge operation prematurely

To stop a sort or merge operation, move the integer 16 into the SORT-RETURN special register.

Move 16 into the register in either of the following ways:

- Use MOVE in an input or output procedure.

Sort or merge processing will be stopped immediately after the next RELEASE or RETURN statement is performed.

- Reset the register in a declarative section entered during processing of a USING or GIVING file.

Sort or merge processing will be stopped 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 Enterprise COBOL) performs the I/O on the input and output files you name in the SORT . . . USING and SORT . . . GIVING statements.

The compiler issues informational messages to point out statements in which FASTSRT can improve performance.

Usage notes

- 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 359](#)

[“FASTSRT requirements for JCL” on page 232](#)

[“FASTSRT requirements for sort input and output files” on page 233](#)

FASTSRT requirements for JCL

In the runtime 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 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 that are otherwise possible.

Note: The "sort input and output files" that is described in this topic relates to the format 1 SORT statement only.

To use FASTSRT, you must describe and process the input files to the sort and the output files from the sort in these ways:

- You can name only one input file in the USING phrase. You can name 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 these DFSORT control statements:

- INREC
- OUTFILE
- 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 or 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.

If you code a RELATIVE KEY clause for an output file, it will not be set by the sort.

Performance 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 this:

```
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, assuming that data set INOUT has been cataloged, you would code:

```
//INPUTF DD DSN=INOUT,DISP=SHR
//OUTPUTF DD DSN=INOUT,DISP=SHR
```

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 FASTSRT either for input or output, but not both. If no other conditions disqualify the file from being eligible for FASTSRT on input, then the file will be accepted for FASTSRT on input, but not on output. If the file was found to be ineligible for FASTSRT on input, it might be eligible for FASTSRT on output.

A QSAM file that qualifies for FASTSRT can be accessed by the COBOL program while the format 1 SORT statement is being performed. For example, if the file is used for FASTSRT on input, you can access it in an output procedure; if it is used for FASTSRT 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 FASTSRT cannot be accessed by the COBOL program until the format 1 SORT statement processing is completed. For example, if the file qualifies for FASTSRT on input, you cannot access it in an output procedure and vice versa. (If you do so, OPEN fails.)

Related tasks

DFSORT Application Programming Guide

Checking for sort errors with NOFASTSRT

When you compile with the NOFASTSRT option, the sort process does not check for errors in open, close, or input or output operations for files that you reference in the USING or GIVING phrase of the format 1 SORT statement. Therefore, you might need to check whether SORT completed successfully.

Note: This topic relates to the format 1 SORT statement only.

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 format 1 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.)

Table 32. *Methods for checking for sort errors with NOFASTSORT (continued)*

FILE STATUS clause?	ERROR declarative?	Then do:
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 format 1 SORT statement.

Related tasks

[“Determining whether the sort or merge was successful” on page 231](#)

[“Using file status keys” on page 247](#)

[“Coding ERROR declaratives” on page 246](#)

[“Stopping a sort or merge operation prematurely” on page 232](#)

Controlling sort behavior

You can control several aspects of sort behavior by inserting values in special registers before the sort or by using compiler options. You might also have a choice of control statements and keywords.

You can 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 by using special registers or compiler options, and the equivalent sort control statement keywords if any are available.

Table 33. *Methods for controlling sort behavior*

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=nnnnn
Name of sort control statement data set (default IGZSRTCD)	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

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 IGZSRTCD, 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 in 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.

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 231](#)

[“Stopping a sort or merge operation prematurely” on page 232](#)

[“Changing DFSORT defaults with control statements” on page 236](#)

[“Allocating space for sort files” on page 237](#)

DFSORT Application Programming Guide (Using DFSORT program control statements)

Related references

[“Default characteristics of the IGZSRTCD data set” on page 236](#)

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 runtime data set IGZSRTCD.

The control statements that you can include in IGZSRTCD (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. You cannot use labels or comments on a record, and a record itself cannot be a DFSORT comment statement.

Related tasks

[“Controlling sort behavior” on page 235](#)

DFSORT Application Programming Guide (Using DFSORT program control statements)

Related references

[“Default characteristics of the IGZSRTCD data set” on page 236](#)

Default characteristics of the IGZSRTCD data set

The IGZSRTCD data set is optional. Its defaults are LRECL=80, BLKSIZE=400, and ddname IGZSRTCD.

You can use a different ddname by coding it in the SORT-CONTROL special register. 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 235](#)

Allocating storage for sort or merge operations

Certain parameters set during the installation of DFSORT determine the amount of storage that DFSORT 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:

- COBOL programs that are dynamically called from an input or output procedure
- Language Environment runtime 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 for your COBOL program.

Related tasks

[“Controlling sort behavior” on page 235](#)
DFSORT Installation and Customization

Related references

DFSORT Application Programming Guide (OPTION control statement)

Allocating space for sort files

If you use NOFASTSORT or an input procedure, DFSORT does not know the size of the file that you are sorting. This can lead to an out-of-space condition when you sort large files or to overallocation of resources when you sort small files.

If this occurs, you can use the SORT-FILE-SIZE special register to help DFSORT determine the amount of resource (for example, workspace or *hiperspace*) needed for the sort. Set SORT-FILE-SIZE to a reasonable estimate of the number of input records. This value is passed to DFSORT as its FILSZ=*En* value.

Related tasks

[“Controlling sort behavior” on page 235](#)
[“Coding the input procedure” on page 224](#)
DFSORT Application Programming Guide

Using checkpoint/restart with DFSORT

You cannot use checkpoints taken while DFSORT is running under z/OS to restart, unless the checkpoints are taken by DFSORT.

Checkpoints taken by a COBOL program while SORT or MERGE statements execute are invalid; such restarts are detected and canceled.

To take a checkpoint during a sort or merge operation, do these steps:

1. Add a DD statement for SORTCKPT in the JCL.
2. Code the RERUN clause in the I-O-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 IGZSRTCD).

Related concepts

[Chapter 47, “Interrupts and checkpoint/restart,” on page 743](#)

Related tasks

[“Changing DFSORT defaults with control statements” on page 236](#)
[“Setting checkpoints” on page 743](#)

Using COBOL user exit routines with DFSORT

You can write DFSORT E15 and E35 user exit routines in COBOL. For details about the COBOL user exit routine interface, refer to [Using your own user exit routines](#) in the *z/OS DFSORT Application Programming Guide*.

Note that the following rules apply:

- The COBOL routine should not contain SORT or MERGE statements.
- The COBOL routine should not return by using STOP RUN.
- The return code to be passed back to DFSORT must be set using the RETURN-CODE special register.
- DFSORT might use high-half of the general registers. If the COBOL exit routine is compiled with Enterprise COBOL 5 or later versions, the HGPR (PRESERVE) compiler option must be used. This is the default setting of the HGPR option.

Related references

[“HGPR” on page 362](#)

RETURN-CODE (*Enterprise COBOL for z/OS Language Reference*)

Sorting under CICS

There is no IBM sort product that is supported under CICS. However, you can use the format 1 SORT statement with a sort program you write that runs under CICS to sort small amounts of data. You can also use the format 2 SORT statement under CICS to sort tables.

You must have both an input and an output procedure for the format 1 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.

The format 2 SORT statement is supported under CICS and you don't need to write a special SORT program.

Related tasks

[“Coding the input procedure” on page 224](#)

[“Coding the output procedure” on page 225](#)

[“Coding COBOL programs to run under CICS” on page 483](#)

Related references

[“CICS SORT application restrictions” on page 238](#)

[“CICS reserved-word table” on page 490](#)

CICS SORT application restrictions

Several restrictions apply to COBOL applications that run under CICS and use the format 1 SORT statement.

The restrictions are:

- Format 1 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.
- These 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 these if you use the NOHANDLE or RESP option. Unpredictable results can occur if you do not use NOHANDLE or RESP.

Related references

[“CICS reserved-word table” on page 490](#)

Chapter 13. Handling errors

Put code in your programs that anticipates possible system or runtime problems. If you do not include such code, output data or files could be corrupted, and the user might not even be aware that there is a problem.

The error-handling code can take actions such as handling the situation, issuing a message, or halting the program. You might for example create error-detection routines for data-entry errors or for errors as your installation defines them. In any event, coding a warning message is a good idea.

Enterprise 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
- Elements for handling 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 242](#)

[“Handling errors in arithmetic operations” on page 242](#)

[“Handling errors in input and output operations” on page 243](#)

[“Handling errors when calling programs” on page 251](#)

[“Writing routines for handling errors” on page 252](#)

Requesting dumps

You can cause a formatted dump of the Language Environment runtime environment and the member language libraries at any prespecified point in your program by coding a call to the Language Environment callable service CEE3DMP.

```
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, compile with the TEST compiler option and use the VARIABLES subparameter of CEE3DMP. You can also request, through runtime options, that a dump be produced for error conditions of your choosing.

You can cause a system dump at any prespecified point in your program. 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

[“TEST” on page 404](#)

Language Environment Debugging Guide

Language Environment Programming Reference (CEE3DMP--generate dump)

Handling errors in joining and splitting strings

During the joining or splitting of strings, the pointer used by STRING or UNSTRING might fall outside the range of the receiving field. A potential overflow condition exists, but COBOL does not let the overflow happen.

Instead, the STRING or UNSTRING operation is not completed, the receiving field remains unchanged, and control passes to the next sequential statement. If you do not code the ON OVERFLOW phrase of the STRING or UNSTRING statement, you are not notified of the incomplete operation.

Consider the following statement:

```
String Item-1 space Item-2 delimited by Item-3
  into Item-4
  with pointer String-ptr
  on overflow
    Display "A string overflow occurred"
End-String
```

These are the data values before and after the statement is performed:

Data item	PICTURE	Value before	Value after
Item-1	X(5)	AAAAA	AAAAA
Item-2	X(5)	EEEEA	EEEEA
Item-3	X(2)	EA	EA
Item-4	X(8)	<i>bbbbbbb</i> ¹	<i>bbbbbbb</i> ¹
String-ptr	9(2)	0	0

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

Because String-ptr has a value (0) that falls short of the receiving field, an overflow condition occurs and the STRING operation is not completed. (Overflow would also occur if String-ptr were greater than 9.) If ON OVERFLOW had not been specified, you would not be notified that the contents of Item-4 remained unchanged.

Handling errors in arithmetic operations

The results of arithmetic operations might be larger than the fixed-point field that is to hold them, or you might have tried dividing by zero. In either case, the ON SIZE ERROR clause after the ADD, SUBTRACT, MULTIPLY, DIVIDE, or COMPUTE statement can handle the situation.

For ON SIZE ERROR to work correctly for fixed-point overflow and decimal overflow, you must specify the TRAP(ON) runtime option.

The imperative statement of the ON SIZE ERROR clause will be performed and the result field will not change in these cases:

- Fixed-point overflow
- Division by zero
- Zero raised to the zero power
- Zero raised to a negative number
- Negative number raised to a fractional power

Floating-point exponent overflow occurs when the value of a floating-point computation cannot be represented in the System z 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

The following example shows how you can code an `ON SIZE ERROR` imperative statement so that the program issues an informative message if division by zero occurs.

```
DIVIDE-TOTAL-COST.  
  DIVIDE TOTAL-COST BY NUMBER-PURCHASED  
  GIVING ANSWER  
  ON SIZE ERROR  
    DISPLAY "ERROR IN DIVIDE-TOTAL-COST PARAGRAPH"  
    DISPLAY "SPENT " TOTAL-COST, " FOR " NUMBER-PURCHASED  
  PERFORM FINISH  
END-DIVIDE  
  
FINISH.  
STOP RUN.
```

If division by zero occurs, the program writes a message and halts program execution.

Handling errors in input and output operations

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

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

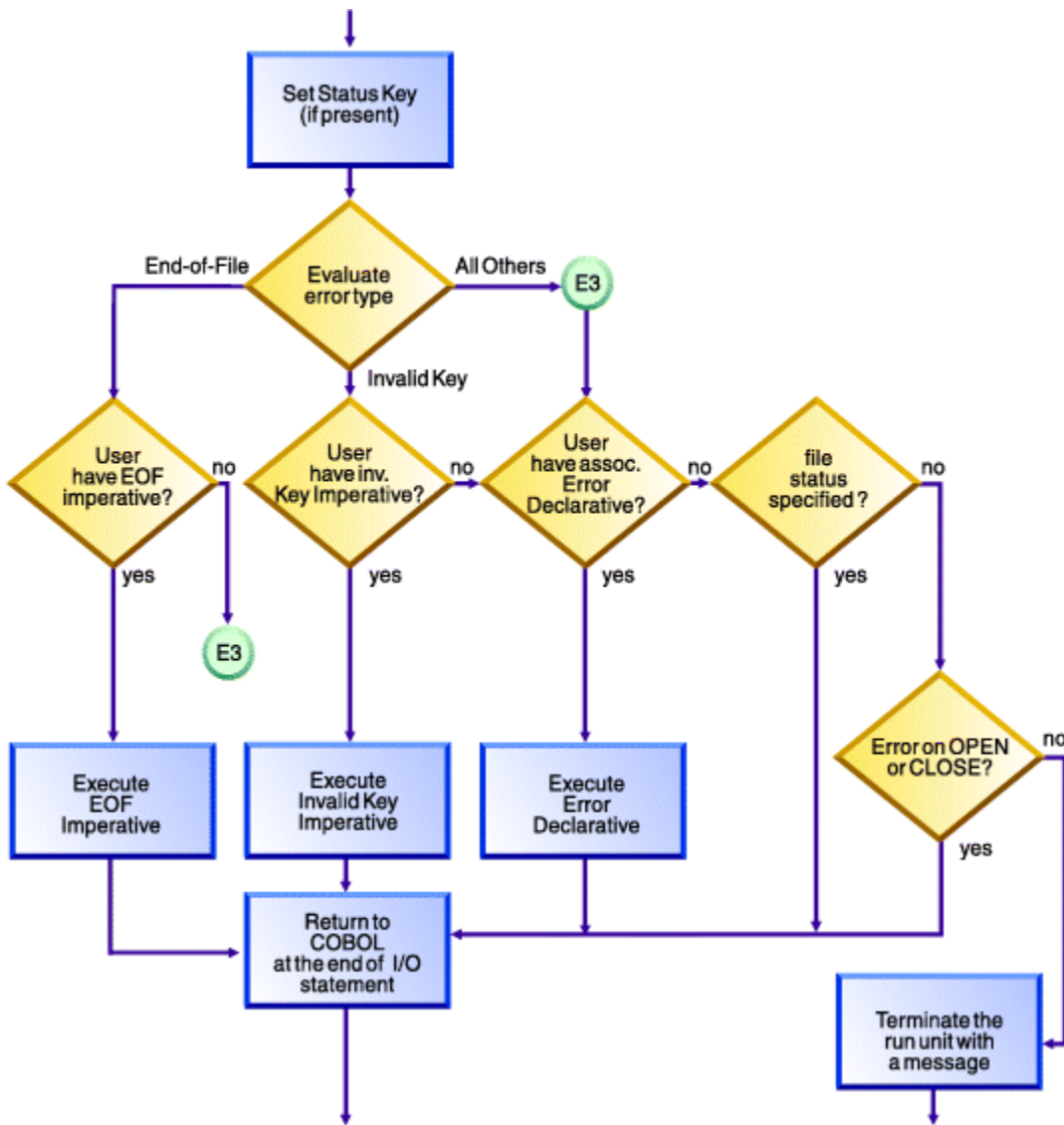
- End-of-file condition (`AT END`)
- `ERROR` declaratives
- `FILE STATUS` clause and file status key
- File system status code
- Imperative-statement phrases in `READ` or `WRITE` statements

For VSAM files, if you specify a `FILE STATUS` clause, you can also test the VSAM status code to direct your program to error-handling logic.

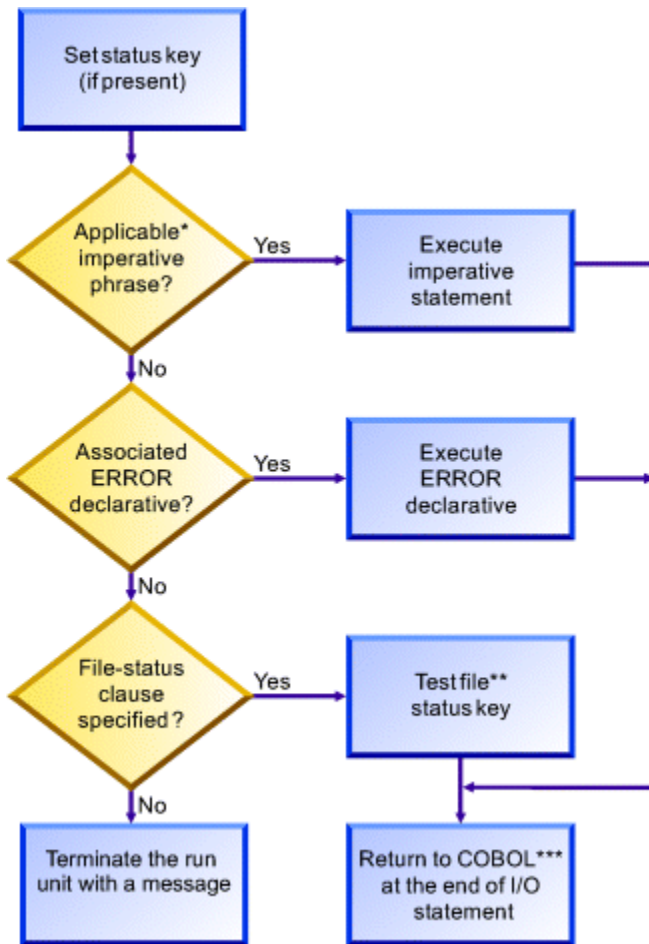
- `INVALID KEY` phrase

To have your program continue, you must code the appropriate error-recovery procedure. You might code, for example, a procedure to check the value of the file status key. If you do not handle an input or output error in any of these ways, a severity-3 Language Environment condition is signaled, which causes the run unit to end if the condition is not handled.

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 \(AT END\)” on page 246](#)

[“Coding ERROR declaratives” on page 246](#)

[“Using file status keys” on page 247](#)

[“Handling errors in QSAM files” on page 179](#)

[“Using VSAM status codes \(VSAM files only\)” on page 248](#)

[“Handling errors in line-sequential files” on page 219](#)

[“Coding INVALID KEY phrases” on page 250](#)

Related references

File status key (*Enterprise COBOL for z/OS Language Reference*)

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

You code the AT END phrase of the READ statement to handle errors or normal conditions, according to your program design. At end-of-file, the AT END phrase is performed. If you do not code an AT END phrase, the associated ERROR declarative is performed.

In many designs, reading sequentially to the end of a file is done intentionally, and the AT END condition is expected. For example, suppose you are processing a file that contains transactions in order to update a main file:

```
PERFORM UNTIL TRANSACTION-EOF = "TRUE"  
  READ UPDATE-TRANSACTION-FILE INTO WS-TRANSACTION-RECORD  
  AT END  
    DISPLAY "END OF TRANSACTION UPDATE FILE REACHED"  
    MOVE "TRUE" TO TRANSACTION-EOF  
  END READ  
END PERFORM
```

Any NOT AT END phrase is performed only if the READ statement completes successfully. If the READ operation fails because of a condition other than end-of-file, neither the AT END nor the NOT AT END phrase is performed. Instead, control passes to the end of the READ statement after any associated declarative procedure is performed.

You might choose not to code either an AT END phrase or an EXCEPTION declarative procedure, but to code a status key clause for the file instead. In that case, control passes to the next sequential instruction after the input or output statement that detected the end-of-file condition. At that place, have some code that takes appropriate action.

Related references

AT END phrases (*Enterprise COBOL for z/OS Language Reference*)

Coding ERROR declaratives

You can code one or more ERROR declarative procedures that will be given control if an input or output error occurs during the execution of your program. If you do not code such procedures, your job could be canceled or abnormally terminated after an input or output error occurs.

Place each such procedure in the declaratives section of the PROCEDURE DIVISION. You can code:

- A single, common procedure for the entire program
- Procedures for each file open mode (whether INPUT, OUTPUT, I-O, or EXTEND)
- Individual procedures for each file

In an ERROR declarative procedure, you can code corrective action, retry the operation, continue, or end execution. (If you continue processing a blocked file, though, you might lose the remaining records in a block after the record that caused the error.) You can use the ERROR declaratives procedure in combination with the file status key if you want a further analysis of the error.

Multithreading: Avoid deadlocks when coding I/O declaratives in multithreaded applications. When an I/O operation results in a transfer of control to an I/O declarative, the automatic serialization lock associated with the file is held during the execution of the statements within the declarative. If you code I/O operations within your declaratives, your logic might result in a deadlock as illustrated by the following sample:

```
Declaratives.  
D1 section.  
Use after standard error procedure on F1  
  Read F2.  
  .  
D2 section.  
Use after standard error procedure on F2  
  Read F1.
```

```
End declaratives.  
Rewrite R1.  
Rewrite R2.
```

When this program is running on two threads, the following sequence of events could occur:

1. Thread 1: Rewrite R1 acquires lock on F1 and encounters I/O error.
2. Thread 1: Enter declarative D1, holding lock on F1.
3. Thread 2: Rewrite R2 acquires lock on F2 and encounters I/O error.
4. Thread 2: Enter declarative D2.
5. Thread 1: Read F2 from declarative D1; wait on F2 lock held by thread 2.
6. Thread 2: Read F1 from declarative D2; wait on F1 lock held by thread 1.
7. Deadlock.

Related references

EXCEPTION/ERROR declarative (*Enterprise COBOL for z/OS Language Reference*)

Using file status keys

After each input or output statement is performed on a file, the system updates values in the two digit positions of the file status key. In general, a zero in the first position indicates a successful operation, and a zero in both positions means that nothing abnormal occurred.

Establish a file status key by coding:

- The FILE STATUS clause in the FILE-CONTROL paragraph:

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

- Data definitions in the DATA DIVISION (WORKING-STORAGE, LOCAL-STORAGE, or LINKAGE SECTION), for example:

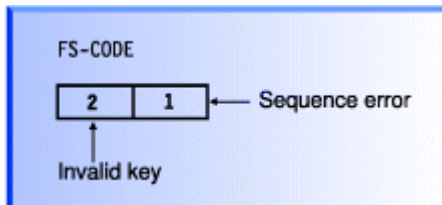
```
WORKING-STORAGE SECTION.  
01 data-name-1 PIC 9(2) USAGE NATIONAL.
```

Specify the file status key *data-name-1* as a two-character category alphanumeric or category national item, or as a two-digit zoned decimal or national decimal item. This *data-name-1* cannot be variably located.

Your program can check the file status key to discover whether an error occurred, and, if so, what type of error occurred. For example, suppose that a FILE STATUS clause is coded like this:

```
FILE STATUS IS FS-CODE
```

FS-CODE is used by COBOL to hold status information like this:



Follow these rules for each file:

- Define a different file status key for each file.

Doing so means that you can determine the cause of a file input or output exception, such as an application logic error or a disk error.

- Check the file status key after each input or output request.

If the file status key contains a value other than 0, your program can issue an error message or can take action based on that value.

You do not have to reset the file status key code, because it is set after each input or output attempt.

For VSAM files, you can additionally code a second identifier in the FILE STATUS clause to get more detailed information about VSAM input or output requests.

You can use the file status key alone or in conjunction with the INVALID KEY phrase, or to supplement the EXCEPTION or ERROR declarative. Using the file status key in this way gives you precise information about the results of each input or output operation.

[“Example: file status key” on page 248](#)

Related tasks

[“Using VSAM status codes \(VSAM files only\)” on page 248](#)

[“Coding INVALID KEY phrases” on page 250](#)

[“Finding and handling input-output errors” on page 435](#)

Related references

FILE STATUS clause (*Enterprise COBOL for z/OS Language Reference*)

File status key (*Enterprise COBOL for z/OS Language Reference*)

Example: file status key

The following example shows how you can perform a simple check of the file status key after opening a file.

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. SIMCHK.  
ENVIRONMENT DIVISION.  
INPUT-OUTPUT SECTION.  
FILE-CONTROL.  
    SELECT MAINFILE ASSIGN TO AS-MAINA  
    FILE STATUS IS MAIN-CHECK-KEY  
  
DATA DIVISION.  
  
WORKING-STORAGE SECTION.  
01 MAIN-CHECK-KEY          PIC X(2).  
  
PROCEDURE DIVISION.  
    OPEN INPUT MAINFILE  
    IF MAIN-CHECK-KEY NOT = "00"  
        DISPLAY "Nonzero file status returned from OPEN " MAIN-CHECK-KEY  
    . . .
```

Using VSAM status codes (VSAM files only)

Often the COBOL 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 data item in the FILE STATUS clause.

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

The data item *data-name-1* shown above specifies the COBOL file status key, which you define as a two-character alphanumeric or national data item, or as a two-digit zoned decimal or national decimal item.

The data item *data-name-8* specifies the VSAM status code, which you define as a 6-byte alphanumeric group data item that has three subordinate 2-byte binary fields. The VSAM status code contains meaningful values when the COBOL file status key is not 0.

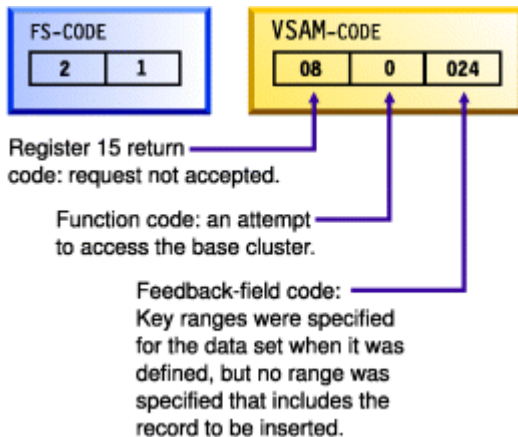
You can define *data-name-8* in the WORKING-STORAGE SECTION, as in VSAM-CODE below.

```

01 RETURN-STATUS.
  05 FS-CODE          PIC X(2).
  05 VSAM-CODE.
    10 VSAM-R15-RETURN PIC S9(4) Usage Comp-5.
    10 VSAM-FUNCTION   PIC S9(4) Usage Comp-5.
    10 VSAM-FEEDBACK   PIC S9(4) Usage Comp-5.

```

Enterprise COBOL 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” on page 249](#)

Related references

FILE STATUS clause (*Enterprise COBOL for z/OS Language Reference*)
 File status key (*Enterprise COBOL for z/OS Language Reference*)
 z/OS DFSMS Macro Instructions for Data Sets (VSAM macro return and reason codes)

Example: checking VSAM status codes

The following example reads an indexed file (starting at the fifth record), checks the file status key after each input or output request, and displays the VSAM status 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 contained 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
ELSE
   DISPLAY VSAMFILE-REC
END-IF.

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 status-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 an INVALID KEY phrase in READ, START, WRITE, REWRITE, and DELETE statements for VSAM indexed and relative files. The INVALID KEY phrase is given control if an input or output error occurs due to a faulty index key.

You can also include the INVALID KEY phrase in WRITE requests for QSAM files, but the phrase has limited meaning for QSAM files. It is used only if you try to write to a disk that is full.

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

INVALID KEY phrases differ from ERROR declaratives in several ways. INVALID KEY phrases:

- Operate for only limited types of errors. ERROR declaratives encompass all forms.
- Are coded directly with the input or output statement. ERROR declaratives are coded separately.
- Are specific for a single input or output operation. ERROR declaratives are more general.

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

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

[“Example: FILE STATUS and INVALID KEY” on page 251](#)

Example: FILE STATUS and INVALID KEY

The following example shows how you can use the file status code and the `INVALID KEY` phrase to determine more specifically why an input or output statement failed.

Assume that you have a file that contains main customer records and you need to update some of these records with information from a transaction update file. The program reads each transaction record, finds the corresponding record in the main file, and makes the necessary updates. The records in both files contain a field for a customer number, and each record in the main file has a unique customer number.

The `FILE-CONTROL` entry for the main file of customer records includes statements that define indexed organization, random access, `MAIN-CUSTOMER-NUMBER` as the prime record key, and `CUSTOMER-FILE-STATUS` as the file status key.

```
. (read the update transaction record)
.
MOVE "TRUE" TO TRANSACTION-MATCH
MOVE UPDATE-CUSTOMER-NUMBER TO MAIN-CUSTOMER-NUMBER
READ MAIN-CUSTOMER-FILE INTO WS-CUSTOMER-RECORD
  INVALID KEY
    DISPLAY "MAIN CUSTOMER RECORD NOT FOUND"
    DISPLAY "FILE STATUS CODE IS: " CUSTOMER-FILE-STATUS
    MOVE "FALSE" TO TRANSACTION-MATCH
  END-READ
```

Handling errors when calling programs

When a program dynamically calls a separately compiled program, the called program might be unavailable. For example, the system might be out of storage or unable to locate the program object. If the `CALL` statement does not have an `ON EXCEPTION` or `ON OVERFLOW` phrase, your application might abend.

Use the `ON EXCEPTION` phrase to perform a series of statements and to perform your own error handling. For example, in the code fragment below, if program `REPORTA` is unavailable, control passes to the `ON EXCEPTION` phrase.

```
MOVE "REPORTA" TO REPORT-PROG
CALL REPORT-PROG
  ON EXCEPTION
    DISPLAY "Program REPORTA not available, using REPORTB."
    MOVE "REPORTB" TO REPORT-PROG
    CALL REPORT-PROG
  END-CALL
END-CALL
```

The `ON EXCEPTION` phrase applies only to the availability of the called program on its initial load. If the called program is loaded but fails for any other reason (such as initialization), the `ON EXCEPTION` phrase is not performed.

Related references

Enterprise COBOL for z/OS Migration Guide

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, `ON SIZE ERROR` phrase, or other language constructs. But if an extraordinary condition such as a machine check occurs, usually your application is abnormally terminated.

Enterprise COBOL 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 routines in COBOL. They can report, analyze, or even fix up a program and enable it to resume running.

Most messages from the COBOL runtime are produced by signaling the corresponding condition. This means that your error handler will get control. For I (Information) and W (Warning) level conditions, you can simply percolate the condition, and the runtime will take care of displaying the message and resuming the execution. In particular, this is required for various compiler generated runtime diagnostics with a `MSG` option (for example, `SSRANGE (, MSG)`) to work as intended.

When you write your own error-handling routines for an application, the COBOL programs must be compiled with appropriate compiler options. For more information, see [“OPTIMIZE” on page 385](#).

To have Language Environment pass control to a user-written error program, you must first identify and register its entry point to Language Environment. `PROCEDURE-POINTER` data items enable you to pass the entry address of procedure entry points to Language Environment services.

Related tasks

[“Using procedure and function pointers” on page 545](#)

Related references

[“OPTIMIZE” on page 385](#)

Part 2. Compiling and debugging your program

Chapter 14. Compiling under z/OS

You can compile Enterprise COBOL programs under z/OS 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 can also compile in a z/OS UNIX shell by using the cob2 command.

You might instead want to start the Enterprise COBOL compiler from an assembler program, for example, if your shop has developed a tool or interface that calls the Enterprise COBOL compiler.

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 required 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” on page 255](#)

[“Compiling under TSO” on page 262](#)

[Chapter 15, “Compiling under z/OS UNIX,” on page 283](#)

[“Starting the compiler from an assembler program” on page 264](#)

[“Defining compiler input and output” on page 266](#)

[“Specifying compiler options under z/OS” on page 273](#)

[“Compiling multiple programs \(batch compilation\)” on page 276](#)

[“Correcting errors in your source program” on page 280](#)

Related references

[Chapter 20, “Compiler-directing statements,” on page 429](#)

[“Data sets used by the compiler under z/OS” on page 267](#)

[“Compiler options and compiler output under z/OS” on page 275](#)

Compiling with JCL

Include the following information in the JCL for compilation: job description, statement to invoke the compiler, and definitions of the needed data sets (including the directory paths of z/OS UNIX files, if any).

The simplest way to compile your program under z/OS is to code JCL that uses a cataloged procedure. A *cataloged procedure* is a set of job control statements in a partitioned data set called the *procedure library* (SYS1.PROCLIB).

The following JCL shows the general format for using 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 a procedural DLL application” on page 569](#)

Related tasks

[“Using a cataloged procedure” on page 256](#)

[“Writing JCL to compile programs” on page 260](#)

[“Specifying compiler options under z/OS” on page 273](#)

[“Specifying compiler options in a batch compilation” on page 278](#)

[“Compiling programs to create DLLs” on page 568](#)

Related references

[“Data sets used by the compiler under z/OS” on page 267](#)

Using a cataloged procedure

Specify a cataloged procedure in an EXEC statement in your JCL.

For example, the following JCL calls the IBM-supplied cataloged procedure IGYWC for compiling an Enterprise COBOL program and defining the required data sets:

```
//JOB1      JOB1
//STEPA    EXEC PROC=IGYWC
//COBOL.SYSIN DD *
000100 IDENTIFICATION DIVISION
          * (the source code)
          .
          .
/*
```

You can omit `/*` after the source code. If your source code is stored in a data set, replace `SYSIN DD *` with appropriate parameters that describe the data set.

You can use these procedures with any of the job schedulers that are part of z/OS. 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 required options.

You can specify data sets to be in the z/OS UNIX 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 about 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 \(IGYWC\)” on page 257](#)

[“Compile and link-edit procedure \(IGYWCL\)” on page 258](#)

[“Compile, link-edit, and run procedure \(IGYWCLG\)” on page 259](#)

[MVS Program Management: User's Guide and Reference](#)

Compile procedure (IGYWC)

IGYWC is a single-step cataloged 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)
```

If you use copybooks in the program that you are compiling, you must also supply a DD statement for SYSLIB or other libraries that you specify in COPY statements. For example:

```
//COBOL.SYSLIB DD DISP=SHR,DSN=DEPT88.BOBS.COBLIB
```

```
//IGYWC PROC LNGPRFX='IGY.V6R4M0',
//          LIBPREFIX='CEE'
//*
//* COMPILE A COBOL PROGRAM
//*
//* PARAMETER  DEFAULT VALUE  USAGE
//* LNGPRFX   IGY.V6R4M0     PREFIX FOR LANGUAGE DATA SET NAMES
//* LIBPREFIX CEE           PREFIX FOR LIBRARY DATA SET NAMES
//*
//* CALLER MUST SUPPLY //COBOL.SYSIN DD . . .
//*
//* CALLER MUST ALSO SUPPLY //COBOL.SYSLIB DD . . . for COPY statements
//*
//COBOL EXEC PGM=IGYCRCTL,REGION=0M
//STEPLIB DD DSNAME=&LNGPRFX..SIGYCOMP,DISP=SHR          (1)
//          DD DSNAME=&LIBPREFIX..SCEERUN,DISP=SHR
//          DD DSNAME=&LIBPREFIX..SCEERUN2,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN DD DSNAME=&&LOADSET,UNIT=SYSALLDA,
//          DISP=(MOD,PASS),SPACE=(CYL,(1,1)),
//SYSUT1 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT2 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT3 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT4 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT5 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT6 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT7 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT8 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT9 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT10 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT11 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT12 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT13 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT14 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT15 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSDMDECK DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
```

(1)

STEPLIB can be installation-dependent.

[“Example: JCL for compiling in the z/OS UNIX file system” on page 257](#)

Example: JCL for compiling in the z/OS UNIX file system

The following job uses procedure IGYWC to compile a COBOL program, demo.cb1, that is located in the z/OS UNIX file system. The job writes the generated compiler listing demo.lst, object file demo.o, and SYSDATA file demo.adt in the z/OS UNIX file system.

```
//UNIXDEMO JOB ,
// TIME=(1),MSGLEVEL=(1,1),MSGCLASS=H,CLASS=A,REGION=50M,
// NOTIFY=&SYSUID,USER=&SYSUID
//COMPILE EXEC IGYWC,
// PARM.COBOBOL='LIST,MAP,RENT,FLAG(I,I),XREF,ADATA'
//SYSPRINT DD PATH='/u/userid/cobol/demo.lst',          (1)
```

```

// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),           (2)
// PATHMODE=(SIRUSR,SIWUSR),                   (3)
// FILEDATA=TEXT                               (4)
//SYSLIN DD PATH='/u/userid/cobol/demo.o',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=(SIRUSR,SIWUSR)
//SYSADATA DD PATH='/u/userid/cobol/demo.adt',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=(SIRUSR,SIWUSR)
//SYSIN DD PATH='/u/userid/cobol/demo.cbl',
// PATHOPTS=ORDONLY,
// FILEDATA=TEXT,
// RECFM=F

```

- (1) PATH specifies the path name of a file in the z/OS UNIX file system.
- (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 as binary.

You can use a mixture of files in the z/OS UNIX file system (PATH= ' *unix-directory-path* ') and traditional MVS data sets (DSN=*mvs-data-set-name*) in 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

[“Data sets used by the compiler under z/OS” on page 267](#)

[UNIX System Services Command Reference](#)

[MVS JCL Reference](#)

Compile and link-edit procedure (IGYWCL)

IGYWCL is a two-step cataloged procedure to compile and link-edit a program.

The COBOL job step produces an object module that is input to the binder (linkage-editor). You can add other object modules. You must supply the following DD statement, indicating the location of the source program, in the input stream:

```
//COBOL.SYSIN DD *           (or appropriate parameters)
```

If the program uses copybooks, you must also supply a DD statement for SYSLIB or other libraries that you specify in COPY statements. For example:

```
//COBOL.SYSLIB DD DISP=SHR,DSN=DEPT88.BOBS.COBLIB
```

```

//IGYWCL PROC  LNGPRFX='IGY.V6R4M0',
//             LIBPRFX='CEE',
//             PGMLIB='&&GOSET',GOPGM=GO
//*
//*  COMPILE AND LINK EDIT A COBOL PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE  USAGE
//*  LNGPRFX   IGY.V6R4M0      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 . . .
//*

```

```

//* CALLER MUST ALSO SUPPLY //COBOL.SYSLIB DD . . . for COPY statements
//*
//COBOL EXEC PGM=IGYCRCTL,REGION=0M
//STEPLIB DD DSNAME=&LNGPRFX..SIGYCOMP,DISP=SHR (1)
// DD DSNAME=&LIBPRFX..SCEERUN,DISP=SHR
// DD DSNAME=&LIBPRFX..SCEERUN2,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN DD DSNAME=&&LOADSET,UNIT=SYSALLDA,
// DISP=(MOD,PASS),SPACE=(CYL,(1,1)),
//SYSUT1 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT2 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT3 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT4 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT5 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT6 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT7 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT8 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT9 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT10 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT11 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT12 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT13 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT14 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT15 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSMDECK DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//LKED EXEC PGM=IEWBLINK,COND=(8,LT,COBOL),REGION=0M
//SYSLIB DD DSNAME=&LIBPRFX..SCEELKEX,DISP=SHR (2)
// DD DSNAME=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN DD DSNAME=&&LOADSET,DISP=(OLD,DELETE)
// DD DDNAME=SYSIN
//SYSLMOD DD DSNAME=&PGMLIB(&GOPGM),
// SPACE=(CYL,(3,1,1)),
// UNIT=SYSALLDA,DISP=(MOD,PASS),DSNTYPE=LIBRARY
//SYSUT1 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))

```

(1) STEPLIB can be installation-dependent.

(2) SYSLIB can be installation-dependent.

Compile, link-edit, and run procedure (IGYWCLG)

IGYWCLG is a three-step cataloged procedure to compile, link-edit, and run a program.

The COBOL job step produces an object module that is input to the binder (linkage-editor). You can add other object modules. If the COBOL program refers to any data sets, you must also supply DD statements that define these data sets. You must supply the following DD statement, indicating the location of the source program, in the input stream:

```
//COBOL.SYSIN DD * (or appropriate parameters)
```

If the program uses copybooks, you must also supply a DD statement for SYSLIB or other libraries that you specify in COPY statements. For example:

```
//COBOL.SYSLIB DD DISP=SHR,DSN=DEPT88.BOBS.COBLIB
```

```

//IGYWCLG PROC LNGPRFX='IGY.V6R4M0',
// LIBPRFX='CEE',GOPGM=GO
//*
//* COMPILE, LINK EDIT AND RUN A COBOL PROGRAM
//*
//* PARAMETER DEFAULT VALUE USAGE
//* LNGPRFX IGY.V6R4M0 PREFIX FOR LANGUAGE DATA SET NAMES
//* LIBPRFX CEE PREFIX FOR LIBRARY DATA SET NAMES
//* GOPGM GO MEMBER NAME FOR LOAD MODULE
//*
//* CALLER MUST SUPPLY //COBOL.SYSIN DD . . .
//*
//* CALLER MUST ALSO SUPPLY //COBOL.SYSLIB DD . . . for COPY statements

```

```

//*
//COBOL EXEC PGM=IGYCRCTL,REGION=0M
//STEPLIB DD DSNAME=&LNGPRFX..SIGYCOMP,DISP=SHR (1)
// DD DSNAME=&LIBPRFX..SCEERUN,DISP=SHR
// DD DSNAME=&LIBPRFX..SCEERUN2,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN DD DSNAME=&&LOADSET,UNIT=SYSALLDA,
// DISP=(MOD,PASS),SPACE=(CYL,(1,1)),
//SYSUT1 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT2 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT3 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT4 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT5 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT6 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT7 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT8 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT9 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT10 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT11 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT12 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT13 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT14 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT15 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSMDECK DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//LKED EXEC PGM=IEWBLINK,COND=(8,LT,COBOL),REGION=0M
//SYSLIB DD DSNAME=&LIBPRFX..SCEELKEX,DISP=SHR (2)
// DD DSNAME=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN DD DSNAME=&&LOADSET,DISP=(OLD,DELETE)
// DD DDNAME=SYSIN
//SYSLMOD DD DSNAME=&&GOSET(&GOPGM),SPACE=(CYL,(1,1,1)),
// UNIT=SYSALLDA,DISP=(MOD,PASS),DSNTYPPE=LIBRARY
//SYSUT1 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//GO EXEC PGM=* . LKED.SYSLMOD,COND=((8,LT,COBOL),(4,LT,LKED)),
// REGION=0M
//STEPLIB DD DSNAME=&LIBPRFX..SCEERUN,DISP=SHR (1)
// DD DSNAME=&LIBPRFX..SCEERUN2,DISP=SHR
//SYSPRINT DD SYSOUT=*
//CEEDUMP DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*

```

(1) STEPLIB can be installation-dependent.

(2) SYSLIB can be installation-dependent.

In the procedure to run a program (GO statement), a valid *DDName* is up to 8 characters in length. In the ENVIRONMENT DIVISION of the COBOL program, there is a FILE CONTROL paragraph whose *assignment-name* should match the *DDName*. For example:

```
//GO.DDName DD DSN=data-set-name
```

Writing JCL to compile programs

If the cataloged procedures do not provide you with the flexibility that you need for more complex programs, write your own job control statements. The following example shows the general format of JCL used to compile a program.

```

//jobname JOB acctno,name,MSGCLASS=1 (1)
//stepname EXEC PGM=IGYCRCTL,PARM=(options) (2)
//STEPLIB DD DSNAME=IGY.V6R4M0.SIGYCOMP,DISP=SHR (3)
// DD DSNAME=SYS1.SCEERUN,DISP=SHR
// DD DSNAME=SYS1.SCEERUN2,DISP=SHR
//SYSUT1 DD UNIT=SYSALLDA,SPACE=(subparms) (4)
//SYSUT2 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSUT3 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSUT4 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSUT5 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSUT6 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSUT7 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSUT8 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSUT9 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSUT10 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSUT11 DD UNIT=SYSALLDA,SPACE=(subparms)

```



```

//SYSUT12 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSUT13 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSUT14 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSUT15 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSMDECK DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSPRINT DD SYSOUT=A (5)
//SYSLIN DD DSNAME=MYPROG,UNIT=SYSALLDA, (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 Enterprise COBOL compiler (IGYCRCTL) is to be invoked.

(3)

This DD statement defines the data set where the Enterprise COBOL compiler resides.

The Language Environment SCEERUN and SCEERUN2 data sets must be included in the concatenation (together with the compiler SIGYCOMP data set), unless the Language Environment data sets are available in the LNKLIST.

(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 compiler 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 (the object module) that receives output from the OBJECT compiler option.

(7)

The SYSIN DD statement defines the data set (source code) to be used as input to the job step.

You can use a mixture of files in the z/OS UNIX file system (PATH= ' *unix-directory-path* ') and traditional MVS data sets (DSN=*mvs-data-set-name*) in the compilation DD statements for the following data sets:

- Sources files
- Object files
- Listings
- ADATA files
- Debug files
- Executable modules

However, the compiler utility files (DD statements SYSUTx) and COPY libraries (DD statement SYSLIB) must be MVS data sets.

For more examples about the cataloged procedures in JCL, refer to [“Using a cataloged procedure” on page 256](#) and its following topics.

[“Example: user-written JCL for compiling” on page 262](#)

[“Example: sample JCL for a procedural DLL application” on page 569](#)

Related references

[MVS JCL Reference](#)

[MVS Program Management: User's Guide and Reference](#)

Example: user-written JCL for compiling

The following example shows a few possibilities for adapting the basic JCL.

```
//JOB1      JOB                               (1)
//STEP1    EXEC PGM=IGYCRCTL,PARM='OBJECT'   (2)
//STEPLIB  DD DSN=IGY.V6R4M0.SIGYCOMP,DISP=SHR
//         DD DSN=SYS1.SCEERUN,DISP=SHR
//         DD DSN=SYS1.SCEERUN2,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))
//SYSUT8   DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT9   DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT10  DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT11  DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT12  DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT13  DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT14  DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT15  DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSMDECK DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSPRINT DD SYSOUT=A
//SYSLIN   DD DSN=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 sole job step in the job. The EXEC statement also specifies that the generated object code should be placed on disk or tape (to be used 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 programs using traditional MVS data sets. You can use TSO commands or REXX execs to compile programs using z/OS UNIX 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 (SYSUT n) and the SYSIN and SYSPRINT data sets.

If you specify certain compiler options, you must allocate other data sets. For example, if you specify the TERMINAL compiler option, you must allocate the SYSTERM data set to receive compiler messages at your terminal.

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 at the READY prompt to request compilation:

```
CALL 'IGY.V6R4M0.SIGYCOMP(IGYCRCTL)'
```

You can specify the ALLOCATE and CALL commands on the TSO command line, or, if you are not using z/OS UNIX files, you can include them in a CLIST.

You can allocate z/OS UNIX files for all the compiler data sets except the SYSUTx utility data sets and the SYSLIB libraries. ALLOCATE statements have the following form:

```
Allocate File(SYSIN) Path('/u/myu/myap/std/prog2.cbl')
Pathopts(ORDONLY) Filedata(TEXT)
```

[“Example: ALLOCATE and CALL for compiling under TSO” on page 263](#)

[“Example: CLIST for compiling under TSO” on page 264](#)

Related references

[“Data sets used by the compiler under z/OS” on page 267](#)

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(SYSUT8) CYLINDERS SPACE(1 1)
[READY]
ALLOCATE FILE(SYSUT9) CYLINDERS SPACE(1 1)
[READY]
ALLOCATE FILE(SYSUT10) CYLINDERS SPACE(1 1)
[READY]
ALLOCATE FILE(SYSUT11) CYLINDERS SPACE(1 1)
[READY]
ALLOCATE FILE(SYSUT12) CYLINDERS SPACE(1 1)
[READY]
ALLOCATE FILE(SYSUT13) CYLINDERS SPACE(1 1)
[READY]
ALLOCATE FILE(SYSUT14) CYLINDERS SPACE(1 1)
[READY]
ALLOCATE FILE(SYSUT15) CYLINDERS SPACE(1 1)
[READY]
ALLOCATE FILE(SYSMDECK) 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.V6R4M0.SIGYCOMP(IGYCRCTL)' 'LIST,NOCOMPILE(S),OBJECT,FLAG(E,E),TERMINAL'
      (COBOL listings and messages)
      .
[READY]
FREE
FILE(SYSUT1,SYSUT2,SYSUT3,SYSUT4,SYSUT5,SYSUT6,SYSUT7,SYSUT8,SYSUT9,SYSUT10,SYSUT11,SYSUT12,
SYSUT13,SYSUT14,SYSUT15,SYSMDECK,SYSPRINT,SYSTEM,+
SYSIN,SYSLIN)
[READY]
```

Example: CLIST for compiling under TSO

The following example shows a CLIST for compiling under TSO. The FREE commands are not required. However, good programming practice dictates that you free files before you allocate them.

```
PROC 1 MEM
CONTROL LIST
FREE F(SYSUT1)
FREE F(SYSUT2)
FREE F(SYSUT3)
FREE F(SYSUT4)
FREE F(SYSUT5)
FREE F(SYSUT6)
FREE F(SYSUT7)
FREE F(SYSUT8)
FREE F(SYSUT9)
FREE F(SYSUT10)
FREE F(SYSUT11)
FREE F(SYSUT12)
FREE F(SYSUT13)
FREE F(SYSUT14)
FREE F(SYSUT15)
FREE F(SYSMDECK)
FREE F(SYSPRINT)
FREE F(SYSIN)
FREE F(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(1,1) CYL UNIT(SYSALLDA)
ALLOC F(SYSUT2) NEW SPACE(1,1) CYL UNIT(SYSALLDA)
ALLOC F(SYSUT3) NEW SPACE(1,1) CYL UNIT(SYSALLDA)
ALLOC F(SYSUT4) NEW SPACE(1,1) CYL UNIT(SYSALLDA)
ALLOC F(SYSUT5) NEW SPACE(1,1) CYL UNIT(SYSALLDA)
ALLOC F(SYSUT6) NEW SPACE(1,1) CYL UNIT(SYSALLDA)
ALLOC F(SYSUT7) NEW SPACE(1,1) CYL UNIT(SYSALLDA)
ALLOC F(SYSUT8) NEW SPACE(1,1) CYL UNIT(SYSALLDA)
ALLOC F(SYSUT9) NEW SPACE(1,1) CYL UNIT(SYSALLDA)
ALLOC F(SYSUT10) NEW SPACE(1,1) CYL UNIT(SYSALLDA)
ALLOC F(SYSUT11) NEW SPACE(1,1) CYL UNIT(SYSALLDA)
ALLOC F(SYSUT12) NEW SPACE(1,1) CYL UNIT(SYSALLDA)
ALLOC F(SYSUT13) NEW SPACE(1,1) CYL UNIT(SYSALLDA)
ALLOC F(SYSUT14) NEW SPACE(1,1) CYL UNIT(SYSALLDA)
ALLOC F(SYSUT15) NEW SPACE(1,1) CYL UNIT(SYSALLDA)
ALLOC F(SYSMDECK) NEW SPACE(1,1) CYL UNIT(SYSALLDA)
CALL 'IGY.V6R4M0.SIGYCOMP(IGYCRCTL)'
```

Related references

TSO/E Command Reference

Starting the compiler from an assembler program

You can start the Enterprise COBOL compiler from within an assembler program by using the ATTACH or the LINK macro by dynamic invocation. You must identify the compiler options and the ddnames of the data sets to be used during processing.

For example:

```
symbol {LINK|ATTACH} EP=IGYCRCTL,PARAM=(optionlist[,ddnamelist]),VL=1
```

EP

Specifies the symbolic name of the 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 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*.

optionlist

Specifies the address of a variable-length list that contains the COBOL options specified for compilation. This address must be written even if no list is provided.

The *optionlist* must begin on a halfword boundary. The 2 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 freeform, with each field separated from the next by a comma. No blanks or zeros should appear. The compiler recognizes only the first 100 characters.

ddnamelist

Specifies the address of a variable-length list that contains alternative ddnames for the data sets used during compiler processing. If standard ddnames are used, the *ddnamelist* can be omitted.

The *ddnamelist* must begin on a halfword boundary. The 2 high-order bytes contain a count of the number of bytes in the remainder of the list. Each name of less than 8 bytes must be left justified and padded with blanks. If an alternate ddname is omitted from the list, the standard name is assumed. If the name is omitted, 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 z/OS UNIX file system.

The following table shows the sequence of the 8-byte entries in the *ddnamelist*.

Alternative ddname 8-byte entry	Name for which alternative ddname is 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
16	SYSADATA
17	SYSJAVA
18	SYSDEBUG
19	SYSMDECK
20	DBRMLIB
21	SYSOPTF
22	SYSUT8

Alternative ddname 8-byte entry	Name for which alternative ddname is substituted
23	SYSUT9
24	SYSUT10
25	SYSUT11
26	SYSUT12
27	SYSUT13
28	SYSUT14
29	SYSUT15

VL

Specifies that the sign bit is to be set to 1 in the last fullword of the address parameter list.

When the compiler completes processing, it puts a return code in register 15.

Related tasks

[“Defining compiler input and output” on page 266](#)

Related references

[“Data sets used by the compiler under z/OS” on page 267](#)

[“Compiler options and compiler output under z/OS” on page 275](#)

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 269](#)

[“Defining a compiler-option data set \(SYSOPTF\)” on page 270](#)

[“Specifying source libraries \(SYSLIB\)” on page 270](#)

[“Defining the output data set \(SYSPRINT\)” on page 271](#)

[“Directing compiler messages to your terminal \(SYSTEM\)” on page 271](#)

[“Creating object code \(SYSLIN or SYSPUNCH\)” on page 271](#)

[“Defining an associated-data file \(SYSADATA\)” on page 272](#)

[“Defining the Java-source output file \(SYSJAVA\)” on page 272](#)

[“Defining the library-processing output file \(SYSMDECK\)” on page 273](#)

Related references

[“Data sets used by the compiler under z/OS” on page 267](#)

[“Compiler options and compiler output under z/OS” on page 275](#)

Data sets used by the compiler under z/OS

The following table lists the function, device requirements, and allowable device classes for each data set that the compiler uses.

Table 34. Compiler data sets

Type	ddname	Function	Required?	Device requirements	Allowable device classes	Can be in z/OS UNIX file system?
Input	SYSIN ¹	Reading source program	Yes	Card reader; intermediate storage	Any	Yes
	SYSOPTF	Reading compiler options	If OPTFILE is in effect	Card reader; intermediate storage; direct access	Any	Yes
	SYSLIB or other copy libraries ¹	Reading user source libraries (PDSs or PDSEs)	If program has COPY or BASIS statements	Direct access	SYSDA	No
Utility ²	SYSUT1, SYSUT2, SYSUT3, SYSUT4, SYSUT6	Work data set used by compiler during compilation	Yes	Direct access	SYSALLDA	No
	SYSUT5	Work data set used by compiler during compilation	If program has COPY, REPLACE, or BASIS statements	Direct access	SYSALLDA	No
	SYSUT7	Work data set used by compiler to create listing	Yes	Direct access	SYSALLDA	No
	SYSUT8, SYSUT9, SYSUT10, SYSUT11, SYSUT12, SYSUT13, SYSUT14, SYSUT15	Work data set used by compiler during compilation	Yes	Direct access	SYSALLDA	No

Table 34. **Compiler data sets** (continued)

Type	ddname	Function	Required?	Device requirements	Allowable device classes	Can be in z/OS UNIX file system?
Output	SYSPRINT ¹	Writing storage map, listings, and messages	Yes	Printer; intermediate storage	SYSSQ, SYSDA, standard output class A	Yes
	SYSTEM	Writing progress and diagnostic messages	If TERM is in effect	Output device; TSO terminal		Yes
	SYSPUNCH	Creating object code	If DECK is in effect	Card punch; direct access	SYSSQ, SYSDA	Yes
	SYSLIN	Creating object module data set as output from compiler and input to binder (linkage-editor)	If OBJECT is in effect	Direct access	SYSSQ, SYSDA	Yes
	SYSADATA ¹	Writing associated data file records	If ADATA is in effect	Output device		Yes
	SYSJAVA	Creating generated Java source file for a class definition	If compiling a class definition	(Must be a z/OS UNIX file)		Yes
	SYSUDUMP, SYSABEND, or SYSMDUMP	Writing dump	If DUMP is in effect (should be rarely used)	Direct access	SYSDA	Yes
	SYSDEBUG	Writing symbolic debug information tables to a data set separate from the object module	If TEST(. . . , SEP, . . .) is in effect	Direct access	SYSDA	Yes
	SYSMDECK	Processing for the MDECK option, or a work data set if NOMDECK is specified	Yes	Direct access	SYSALLDA	Yes

1. You can use the EXIT option to provide user exits from these data sets.
2. Utility data sets must be single volume, and cannot have DSNTYPE=LARGE (SYSUT1 - SYSUT15).

Related references

- “Logical record length and block size” on page 269
- “EXIT” on page 357

Logical record length and block size

For compiler data sets other than the work data sets (SYSUT n) and z/OS UNIX files, you can set the block size by 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 the following data sets:

- SYSDEBUG: You can specify any LRECL in the listed range, with 1024 recommended.
- SYSPRINT, SYSDEBUG: You can specify BLKSIZE=0, which results in a system-determined block size.

Data set	RECFM	LRECL (bytes)	BLKSIZE ¹
SYSDEBUG ²	F or FB	80 to 1024 ³	LRECL x n
SYSIN	F or FB	80	80 x n
SYSLIB or other copy libraries	F or FB	80	80 x n
SYSLIN	F or FB	80	80 x n
SYSMDECK	F or FB	80	80 x n
SYSOPTF	F or FB	80	80 x n
SYSPRINT ²	F or FB	133	133 x n
SYSPUNCH	F or FB	80	80 x n
SYSTEMR	F or FB	80	80 x n

1. n = blocking factor
2. If you specify BLKSIZE=0, the system determines the block size.
3. The default LRECL for SYSDEBUG is 1024.

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 by using the SYSIN DD statement as shown below.

```
//SYSIN DD DSNAME=dsname,UNIT=SYSSQ,VOLUME=(subparms),DISP=SHR
```

You can place your source code or BASIS statement directly in the input stream. To do so, use this SYSIN DD statement:

```
//SYSIN DD *
```

The source code or BASIS statement must follow the DD * statement. If another job step follows the compilation, the EXEC statement for that step must follow the /* statement or the last source statement.

Defining a compiler-option data set (SYSOPTF)

Define a data set that contains the compiler options for your COBOL program by coding the SYSOPTF DD statement as shown below.

```
//SYSOPTF DD DSNAME=dsname,UNIT=SYSDA,VOLUME=(subparms),DISP=SHR
```

To use a compiler-option data set, specify OPTFILE either as a compiler invocation option or in a PROCESS or CBL statement in your source program.

Within the SYSOPTF data set:

- Specify compiler options in free form between columns 2 and 72, using the same syntax as you use for invocation options or for compiler options in a PROCESS or CBL statement.
- Code an asterisk (*) in column 1 to cause a line to be treated as a comment.
- Optionally code sequence numbers in columns 73 through 80; those columns are ignored.

You can optionally place the compiler options directly in the input stream after the SYSOPTF DD statement if you compile using the OPTFILE option:

```
//COB      EXEC PGM=IGYCRCTL,PARM='OPTFILE'  
//SYSOPTF DD DATA,DLM=@@  
          SSRANGE  ARITH(COMPAT)  
          OPTIMIZE  
          . . .  
@@  
//SYSIN   DD . . .
```

You can concatenate multiple SYSOPTF DD statements if you have multiple compiler-option data sets:

```
//SYSOPTF DD DSNAME=dsname1, . . .  
//        DD DSNAME=dsname2, . . .
```

Compiler options that are in later data sets in the concatenation take precedence over options in earlier data sets in the concatenation.

Related references

[“Logical record length and block size” on page 269](#)

[“OPTFILE” on page 384](#)

Specifying source libraries (SYSLIB)

Use 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 BASIS statements 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 z/OS UNIX file system when you compile with JCL or under TSO.

Defining the output data set (SYSPRINT)

You can use ddname SYSPRINT to produce a listing. The listing includes the results of the default or requested options of the PARM parameter (that is, diagnostic messages and the object-code listing).

You can direct the output to a SYSOUT data set, a printer, a direct-access storage device, or a magnetic-tape device. For example:

```
//SYSPRINT DD SYSOUT=A
```

The SYSPRINT data set can be a sequential data set, a PDS or PDSE member, or a z/OS UNIX file. For details about how to specify the record format, record length, and block size of the SYSPRINT data set, see the related reference below.

Related references

[“Logical record length and block size” on page 269](#)

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.

```
ALLOC F(SYSTEM) DA(*)
```

You can define SYSTEM in various other ways, for example to a SYSOUT data set, a data set on disk, a file in the z/OS UNIX file system, 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 a z/OS UNIX file, or on tape. The compiler uses the file that you define in the SYSLIN or SYSPUNCH DD statement.

```
//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 binder (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 binder (linkage-editor) job step.

Your installation might use the DECK option and the SYSPUNCH DD statement. B is the standard output class for punch data sets:

```
//SYSPUNCH DD SYSOUT=B
```

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 383](#)

[“DECK” on page 350](#)

Defining an associated-data file (SYSADATA)

Define a SYSADATA file if you use the ADATA compiler option.

```
//SYSADATA DD DSNAME=dsname,UNIT=SYSDA
```

The SYSADATA file will be a sequential file that contains specific record types that have information about the program that is collected during compilation. The file can be a traditional MVS data set or a z/OS UNIX file.

Related references

[“ADATA” on page 335](#)

Defining the Java-source output file (SYSJAVA)

Add the SYSJAVA DD statement if you are compiling an OO program. The generated Java source file is written to the SYSJAVA ddname.

```
//SYSJAVA DD PATH='/u/userid/java/Classname.java',  
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),  
// PATHMODE=SIRWXU,  
// FILEDATA=TEXT
```

The SYSJAVA file must be in the z/OS UNIX file system.

Related tasks

[“Compiling OO applications in JCL or TSO/E” on page 295](#)

Defining the debug data set (SYSDEBUG)

When you compile from JCL or from TSO and specify the TEST(*...*,SEP,*...*) compiler option, the symbolic debug information tables are written to the data set that you specify in the SYSDEBUG DD statement.

```
//SYSDEBUG DD DSNAME=dsname,UNIT=SYSDA
```

The SYSDEBUG data set can be a sequential data set, a PDS or PDSE member, or an HFS file. For details about how to specify the record format, record length, and block size of the SYSDEBUG data set, see the related reference below about logical record length and block size.

Language Environment uses SYSDEBUG for its dump services. If the TEST | NOTEST(*...*,SEPARATE(DSNAME),*...*) compiler option is in effect, the SYSDEBUG dataset name is stored in the object program and is used as the default at run time. You can change the name of that data set at run time by using the SYSDEBUG COBOL debug file user exit, IGZIUXB. You can direct IBM z/OS Debugger to a renamed data set using the SET DEFAULT LISTINGS command, user exit EQAUEDAT, or the EQADEBUG DD statement.

The dataset name that you specify in DDNAME SYSDEBUG might be used by several IBM products, including Language Environment, IBM z/OS Debugger, Fault Analyzer, and Application Performance Analyzer. For details, see the documentation of those individual products.

Related tasks

Language Environment Customization (Modifying the COBOL debug file name)

IBM z/OS Debugger User's Guide (How does Debug Tool locate COBOL and PL/I separate debug files)

Related references

[“Logical record length and block size” on page 269](#)

[“TEST” on page 404](#)

Defining the library-processing output file (SYSMDECK)

The SYSMDECK DD allocation is required for all compilations. If you specify the MDECK compiler option, the SYSMDECK DD allocation must specify a permanent MVS data set or a z/OS UNIX file. However, if you use the NOMDECK option, SYSMDECK can alternatively be specified as a utility (temporary) data set.

The following example specifies a permanent dataset name:

```
//SYSMDECK DD DSNAME=dsname,UNIT=SYSDA
```

The following example specifies a z/OS UNIX file:

```
//SYSMDECK DD PATH='/path/filename.dek',  
//FILEDATA=TEXT,  
//PATHOPTS=(ORDWR,OCREAT,OTRUNC),  
//PATHMODE=(SIRUSR,SIWUSR)
```

The following example specifies a utility (temporary) data set:

```
//SYSMDECK DD UNIT=SYSALLDA
```

The SYSMDECK file will contain a copy of the updated input source after library processing, that is, the result of COPY, BASIS, REPLACE, EXEC SQL INCLUDE, and EXEC SQLIMS INCLUDE statements.

Related references

[“MDECK” on page 376](#)

Specifying compiler options under z/OS

The compiler is installed with default compiler options. While installing the compiler, the system programmer can fix compiler option settings to, for example, ensure better performance or maintain certain standards. You cannot override any compiler options that are fixed.

For options that are not fixed, you can override the default settings by specifying compiler options in any of these ways:

- Code them on the PROCESS or CBL statement in COBOL source.
- Include them when you start the compiler, either on the PARM parameter on the EXEC statement in the JCL or on the command line under TSO.
- Include them in a SYSOPTF data set, and specify the OPTFILE compiler option in either of the above ways.

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, OUTDD, SQL, and SQLIMS 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.

The precedence of options in a SYSOPTF data set depends on where you specify the OPTFILE compiler option. For example, if you specify OPTFILE in a PROCESS statement, the SYSOPTF options supersede the options that you specify in the compiler invocation. For further details, see the related reference below about the OPTFILE option.

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, 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 275](#)

[“Example: specifying compiler options under TSO” on page 275](#)

Related tasks

[“Defining a compiler-option data set \(SYSOPTF\)” on page 270](#)

[“Specifying compiler options in the PROCESS \(CBL\) statement” on page 274](#)

[“Specifying compiler options in a batch compilation” on page 278](#)

Related references

[“Compiler options and compiler output under z/OS” on page 275](#)

[“COBOL compiler options” on page 329](#)

[“Conflicting compiler options” on page 334](#)

[“OPTFILE” on page 384](#)

Specifying compiler options in the PROCESS (CBL) statement

Within a COBOL program, you can code most compiler options in PROCESS (CBL) statements. Code the statements before the IDENTIFICATION DIVISION header and before any comment lines or compiler-directing statements.



If you do not use a sequence field, you can start a PROCESS statement in column 1 or after. If you use a sequence field, the sequence number must start in column 1 and must contain six characters; the first character must be numeric. If used with a sequence field, PROCESS can start in column 8 or after.

You can use CBL as a synonym for PROCESS. CBL can likewise start in column 1 or after if you do not use a sequence field. If used with a sequence field, CBL can start in column 8 or after.

You must end PROCESS and CBL statements at or before column 72.

Use one or more blanks to separate a PROCESS or CBL statement from the first option in *options-list*. Separate options with a comma or a blank. Do not insert spaces between individual options and their suboptions.

You can code more than one PROCESS or CBL statement. If you do so, the statements must follow one another with no intervening statements. You cannot continue options across multiple PROCESS or CBL statements.

Your programming organization can inhibit the use of PROCESS (CBL) statements by using the default options module of the COBOL compiler. If PROCESS or CBL statements that are not allowed by the organization are found in a COBOL program, the COBOL compiler generates error diagnostics.

Related references

Reference format (*Enterprise COBOL for z/OS Language Reference*)

CBL (PROCESS) statement (*Enterprise COBOL for z/OS Language Reference*)

Example: specifying compiler options using JCL

The following example shows how to specify compiler options under z/OS 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 z/OS

When the compiler finishes processing your source program, it will have produced one or more outputs, depending on the compiler options that were in effect.

Compiler option	Compiler output	Type of output
ADATA	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 SYSDUMP DD statement); should be used rarely	Listing
EXPORTALL	Exported symbols for a DLL	Object
FLAG	List of errors that the compiler found in your program	Listing
LIST	Listing of object code in machine and assembler language	Listing
MAP (HEX) or MAP (DEC)	Map of the data items in your program	Listing
MDECK	Expansion of library-processing statements in your program	Library-processing side file
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 (1) or OPTIMIZE (2)	Optimized object code	Object
RENT	Reentrant object code	Object
SOURCE	Listing of your source program	Listing
SQL	SQL statements and host variable information for Db2 bind process	Database request module (DBRM)
TERMINAL	Progress and diagnostic messages sent to terminal	Terminal

Compiler option	Compiler output	Type of output
TEST (DWARF)	DWARF format debugging information in the object module, to enable interactive debugging	Object
TEST (NOSEP)	Information tables for IBM z/OS Debugger and for formatted dumps	Object
TEST (SEP)	Information tables for IBM z/OS Debugger and for formatted dumps	Separate debug file
NOTEST (DWARF)	Basic DWARF format diagnostic information, to enable application failure analysis tools	Object
VBREF	Cross-reference listing of statements 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 and included in the SYSPRINT data set. The database request module (DBRM) is the data set defined in DBRMLIB. The separate debug file is 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. You might also use the listings for diagnosis and debugging after the application is in production.

After compilation, fix any errors that the compiler found in your program. If no errors were detected, you can go to the next step in the process: binding (link-editing) your program. (If you used compiler options to suppress object code generation, you must recompile to obtain object code.)

Related tasks

Language Environment Programming Guide (Preparing to link-edit and run)

Related references

[“Messages and listings for compiler-detected errors” on page 281](#)
[“COBOL compiler options” on page 329](#)

Compiling multiple programs (batch compilation)

You can compile a sequence of separate COBOL programs by using a single invocation of the compiler. You can link the object program produced from this compilation into one program object or separate program objects, controlled by the NAME compiler option.

When you compile several programs as part of a batch job, you need to:

- Determine whether you want to create one or more program objects.
- 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 program objects, precede each set of objects with the NAME compiler option. When the compiler encounters the NAME option, the first program in the sequence and all subsequent programs until the next NAME compiler option is encountered are link-edited into a single program object. Then each successive program that is compiled with the NAME option is included in a separate program object.

Use the END PROGRAM marker to terminate each program in the sequence except the last program in the batch (for which the END PROGRAM marker is optional). Alternatively, you can precede each program in the sequence with a CBL or PROCESS statement.

If you omit the END PROGRAM marker 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” on page 277](#)

Related tasks

[“Specifying compiler options in a batch compilation” on page 278](#)

Related references

[“NAME” on page 377](#)

Example: batch compilation

The following example shows a batch compilation for three programs (PROG1, PROG2, and PROG3) and the creation of two program objects using one invocation of the IGYWCL cataloged procedure.

The following steps occur:

- PROG1 and PROG2 are link-edited together to form one program object that has the name PROG2. The entry point of this program object defaults to the first program in the program object, PROG1.
- PROG3 is link-edited by itself into a program object that has the name PROG3. Because it is the only program in the program object, 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=&&G0SET (1)
/*
//P2 EXEC PGM=PROG2
//STEPLIB DD DSN=&&G0SET,DISP=(SHR,PASS) (2)
. . . (3)
/*
//P3 EXEC PGM=PROG3
//STEPLIB DD DSN=&&G0SET,DISP=(SHR,PASS) (2)
. . . (4)
/*
//
```

(1)

The data-set name for the LKED step SYSLMOD is changed to the temporary name &&G0SET, without any member name.

(2)

The temporary data set &&G0SET is used as the STEPLIB for steps P2 and P3 to run the compiled programs. If the Language Environment library does not reside in shared storage, you must also add the library data set 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

Language Environment Programming Guide (IBM-supplied cataloged procedures)

Specifying compiler options in a batch compilation

You can specify compiler options for each program in the batch sequence either with a CBL or PROCESS statement that precedes the program, or upon 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 batch sequence. Compiler options are recognized in the following order of precedence, from highest to lowest:

1. Installation defaults that are fixed at your site
2. Values of the BUFSIZE, DEFINE, OUTDD, SQL, SQLIMS, and TEST 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 in the compiler invocation (JCL PARM or TSO CALL)
5. Installation defaults that are not fixed

If any program in the batch sequence requires the BUFSIZE, DEFINE, OUTDD, SQL, SQLIMS, or TEST 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 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 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” on page 278](#)

[“Example: LANGUAGE option in a batch compilation” on page 279](#)

Related references

[“DEFINE” on page 350](#)

Example: precedence of options in a batch compilation

The following example listing shows the precedence of compiler options for batch compilation.

```
PP 5655-EC6 IBM Enterprise COBOL for z/OS 6.4.0 PXXXXXX Date 05/31/2022. . .
```

```

Invocation parameters:
NOTERM
PROCESS(CBL) statements:
CBL CURRENCY,FLAG(I,I)
Options in effect: All options are installation defaults unless otherwise noted:
NOADATA
ADV
QUOTE
ARITH(COMPAT)
NOAWO
NOBLOCKØ
BUFSIZE(4096)
.
. CURRENCY          Process option PROGRAM 1
.
. FLAG(I,I)         Process option PROGRAM 1
.
.
. NOTERM            INVOCATION option
.
.
End of compilation for program 1
. . .

```

```

PP 5655-EC6 IBM Enterprise COBOL for z/OS 6.4.0 PXXXXXX Date 05/31/2022. . .
PROCESS(CBL) statements:
CBL APOST
Options in effect:
NOADATA
ADV
APOST              Process option PROGRAM 2
ARITH(COMPAT)
NOAWO
NOBLOCKØ
BUFSIZE(4096)
.
.
. NOCURRENCY        Installation default option for PROGRAM 2
.
. FLAG(I)           Installation default 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 EN), and the invocation option is XX, a nonexistent language.

```

CBL LANG(JP),FLAG(I,I),APOST (1)
IDENTIFICATION DIVISION. (2)
PROGRAM-ID. COMPILE1.
.
.
. END PROGRAM COMPILE1.
CBL LANGUAGE(YY) (3)
CBL LANGUAGE(JP),LANG(!) (4)
IDENTIFICATION DIVISION. (2)
PROGRAM-ID. COMPILE2.
.
.
. END PROGRAM COMPILE2.
IDENTIFICATION DIVISION.
PROGRAM-ID. COMPILE3.
.
.
. END PROGRAM COMPILE3.
CBL LANGUAGE(JP),LANGUAGE(YY) (5)
.
.

```

- (1) The installation default is EN. The invocation option was XX, a nonexistent language. EN is the language in effect.

- (2) After the CBL statement is scanned, JP is the language in effect.
- (3) CBL resets the language to EN. YY is ignored because it is superseded by JP.
- (4) !! is not alphanumeric and is discarded.
- (5) CBL resets the language to EN. YY supersedes JP but is nonexistent.

For the program COMPILER1, 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 COMPILER2, 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). 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 and PROCESS statements, the language in effect before COMPILER1 must be Japanese.

The program COMPILER3 has no CBL statement. It inherits the language in effect, Japanese (JP), from the previous compilation.

After compiling COMPILER3, 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, 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 occurred (LINEID). The text of a message tells you what the problem is. With this information, you can correct the source program.

Although you should try to correct errors, it is not always necessary to correct source code for every diagnostic message. You can leave a warning-level or informational-level message in a program without much risk, and you might decide that the recoding and compilation that are needed to remove the message are not worth the effort. Severe-level and error-level errors, however, indicate probable program failure and should be corrected.

In contrast with the four lower levels of severities, an unrecoverable (U-level) error might not result from a mistake in your source program. It could come from a flaw in the compiler itself or in the operating system. In such cases, the problem must be resolved, because the compiler is forced to end early and does not produce complete object code or a complete listing. If the message occurs for a program that has many S-level syntax errors, correct those errors and compile the program again. You can also resolve job set-up problems (such as missing data-set definitions or insufficient storage for compiler processing) by making changes to the compile job. If your compile job setup is correct and you have corrected the S-level syntax errors, you need to contact IBM to investigate other U-level errors.

After correcting the errors in your source program, recompile the program. If this second compilation is successful, proceed to the link-editing step. If the compiler still finds problems, repeat the above procedure until only informational messages are returned.

Related tasks

[“Generating a list of compiler messages” on page 281](#)

Related references

[“Messages and listings for compiler-detected errors” on page 281](#)

Generating a list of compiler messages

You can generate a complete listing of compiler diagnostic messages with their message numbers, severities, and text by compiling a program that has program-name ERRMSG.

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

```
Identification Division.  
Program-ID. ErrMsg.
```

Related tasks

[“Customizing compiler-message severities” on page 821](#)

Related references

[“Messages and listings](#)

[for compiler-detected errors” on page 281](#)

[“Format of compiler diagnostic messages” on page 281](#)

Messages and listings for compiler-detected errors

As the compiler processes your source program, it checks for COBOL language errors, and issues diagnostic messages. These messages are collated in the compiler listing (subject to the FLAG option).

Each message in the listing provides information about the nature of the problem, its severity, and the compiler phase that detected it. Wherever possible, the message provides specific instructions for correcting an error.

The messages for errors found during processing of compiler options, CBL and PROCESS statements, and BASIS, COPY, or REPLACE statements are displayed near the top of the listing.

The messages for compilation errors (ordered by line number) are displayed near the end of the listing for each program.

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

Related tasks

[“Correcting errors in your source program” on page 280](#)

[“Generating a list of compiler messages” on page 281](#)

Related references

[“Format of compiler diagnostic messages” on page 281](#)

[“Severity codes for compiler diagnostic messages” on page 282](#)

[“FLAG” on page 359](#)

Format of compiler diagnostic messages

Each message issued by the compiler has a source line number, a message identifier, and message text.

Each message has the following form:

```
nnnnnn IGYppxxxx-l message-text
```

nnnnnn

The number of the source statement of the last line that the compiler was processing. Source statement numbers are listed on the source printout of your program. If you specified the NUMBER option at compile time, the numbers are the original source program numbers. If you specified NONUMBER, the numbers are those generated by the compiler.

IGY

A prefix that identifies that the message was issued by the COBOL compiler.

pp

Two characters that identify which phase or subphase of the compiler detected the condition that resulted in a message. As an application programmer, you can ignore this information. If you are diagnosing a suspected compiler error, contact IBM for support and include this information.

xxxx

A four-digit number that identifies the message.

l

A character that indicates the severity level of the message: I, W, E, S, or U.

message-text

The message text; for an error message, a short explanation of the condition that caused the error.

Tip: If you used the FLAG option to suppress messages, there might be additional errors in your program.

Related references

[“Severity codes for compiler diagnostic messages” on page 282](#)
[“FLAG” on page 359](#)

Severity codes for compiler diagnostic messages

Conditions that the compiler can detect fall into five levels or categories of severity.

Level or category of message	Return code	Purpose
Informational (I)	0	To inform you. No action is required, and the program runs correctly.
Warning (W)	4	To indicate a possible error. The program probably runs correctly as written.
Error (E)	8	To indicate a condition that is definitely an error. The compiler attempted to correct the error, but the results of program execution might not be what you expect. You should correct the error.
Severe (S)	12	To indicate a condition that is a serious error. The compiler was unable to correct the error. The program does not run correctly, and execution should not be attempted. Object code might not be created.
Unrecoverable (U)	16	To indicate an error condition of such magnitude that the compilation was terminated.

The final return code at the end of compilation is generally the highest return code that occurred for any message during the compilation.

You can suppress compiler diagnostic messages or change their severities, however, which can have an effect upon the final compilation return code. For details, see the related information.

Related tasks

[“Customizing compiler-message severities” on page 821](#)

Related references

[“Processing of MSGEXIT” on page 819](#)

Chapter 15. Compiling under z/OS UNIX

Compile Enterprise COBOL programs under z/OS UNIX by using the `cob2` command. Under z/OS UNIX, you can compile any COBOL program that you can compile under z/OS. The object code generated by the COBOL compiler can run under z/OS.

As part of the compilation step, you define the files needed for the compilation, and specify any compiler options or compiler-directing statements that are necessary for your program and for the output that you want.

The main job of the compiler is to translate COBOL programs into language that the computer can process (object code). The compiler also lists errors in source statements and provides supplementary information to help you debug and tune programs.

Related tasks

[“Setting environment variables under z/OS UNIX” on page 283](#)

[“Specifying compiler options under z/OS UNIX” on page 284](#)

[“Compiling and linking with the `cob2` command” on page 285](#)

[“Compiling using scripts” on page 290](#)

[“Compiling, linking, and running OO applications under z/OS UNIX” on page 291](#)

Related references

[“Data sets used by the compiler under z/OS” on page 267](#)

[“Compiler options and compiler output under z/OS” on page 275](#)

Setting environment variables under z/OS UNIX

An *environment variable* is a name that is associated with a string of characters and that defines some variable aspect of the program environment. You use environment variables to set values that programs, including the compiler, need.

Set the environment variables for the compiler by using the `export` 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 that are significant to the z/OS UNIX shell. For example:

```
export COBOPT="TRUNC(OPT) XREF"
```

COBOL_INSTALL_DIR

The `cob2` utility and related files are normally installed under the HFS directory `/usr/lpp/IBM/cobol/igyv6r4`. If the `cob2` utility is installed somewhere else on your system, it is necessary to set the `COBOL_INSTALL_DIR` environment variable to that location in order to use the utility.

SYSLIB

Specify paths to directories to be used in searching for COBOL copybooks if you do not specify an explicit library-name in the `COPY` statement. Separate multiple paths with a colon. 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 in which you have not coded an explicit library-name, the compiler searches for copybooks in z/OS UNIX directories 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
4. In the locations specified in instances of the `COPYLOC` option that use the default library name or specify an explicit library name of `SYSLIB`

If the copybook is not found in any of the z/OS UNIX directories to be searched, and there are instances of the `COPYLOC` option specified without an explicit library name or specified with the library name `SYSLIB`, those locations are searched in the order that they were specified.

library-name

Specify the directory path from which to copy when you specify an explicit library-name in 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 z/OS UNIX” on page 284](#)

[“Compiling and linking with the cob2 command” on page 285](#)

[“Setting and accessing environment variables” on page 514](#)

Related references

Chapter 20, [“Compiler-directing statements,” on page 429](#)

[“COBOL compiler options” on page 329](#)

`COPY` statement (*Enterprise COBOL for z/OS Language Reference*)

Specifying compiler options under z/OS UNIX

The compiler is installed and set up with default compiler options. While installing the compiler, a system programmer can fix compiler option settings to ensure better performance or maintain certain standards. You cannot override any compiler options that your site has fixed.

For options that are not fixed, you can override the default settings by specifying compiler options in any of three ways:

- Code them on the `PROCESS` or `CBL` statement in your COBOL source.
- Specify the `-q` option of the `cob2` command.
- Set the `COBOPT` environment variable.

The compiler recognizes the options in the above order of precedence, from highest to lowest. 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`, `SQL`, `SQLIMS`, and `OUTDD` options in effect for the first program in a batch compilation
3. The values that you specify on `PROCESS` or `CBL` statements in 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

Restrictions:

- The SQL coprocessor can be used with a compile job initiated from z/OS UNIX. All the following conditions must be met for this to work:
 - The Db2 data set that contains the Db2 coprocessor services must be included in your STEPLIB, unless these services are in the LNKLST. Typically, this data set is called *xxxxxx.SDSNLOAD*. For example, for DB2® 11 it might be *DSNB10.SDSNLOAD*, but your installation might have changed the name.
 - The SQL compiler option must be specified.
 - The `-dbrmLib` option of `cob2` must be specified. Suppose that *file* is the name of the input COBOL file to compile:
 - Use `dbrmLib=xxx` to direct the database request module (DBRM) to an existing PDS data set, where a new member named *file* will be created.
 - Use `-dbrmLib` (without `=xxx`) to send the DBRM to an HFS file named *file.dbrm*.

The separate SQL precompiler does not run under z/OS UNIX.

- Do not use the `SQLIMS` compiler option under z/OS UNIX.
- The `OPTFILE` option is ignored when you compile using the `cob2` command under z/OS UNIX.

You can use the `COBOPT` environment variable, which provides a capability that is comparable to `OPTFILE`, instead.

Related tasks

[“Specifying compiler options in the PROCESS \(CBL\) statement” on page 274](#)

[“Setting environment variables under z/OS UNIX” on page 283](#)

[“Compiling and linking with the cob2 command” on page 285](#)

Related references

[“Conflicting compiler options” on page 334](#)

[“COBOL compiler options” on page 329](#)

Compiling and linking with the cob2 command

Use the `cob2` command to compile and link COBOL programs from the z/OS 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 z/OS 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 LNKLST, or if more than one level of IBM COBOL compiler is installed on your system, you can specify in the STEPLIB environment variable the compiler PDSE that you want to use. For example, the following statement specifies `IGY.V6R4M0.SIGYCOMP` as the compiler PDSE:

```
export STEPLIB=IGY.V6R4M0.SIGYCOMP
```

The `cob2` command implicitly uses the z/OS UNIX shell command `c89` for the link step. `c89` is the shell interface to the linker (the z/OS program management binder).

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

Only files with the suffix `.cbl` are passed to the compiler; `cob2` passes all other files to the linker.

The listing output that you request from the compilation of a COBOL source program *file.cbl* is written to *file.lst*. The listing output that you request from the linker is written to stdout.

The linker causes execution to begin at the first main program.

Related tasks

[“Creating a DLL under z/OS UNIX” on page 286](#)

[“Preparing OO applications under z/OS UNIX” on page 292](#)

UNIX System Services User's Guide

Related references

[“cob2 syntax and options” on page 287](#)

[“cob2 input and output files” on page 289](#)

UNIX System Services Command Reference

Creating a DLL under z/OS UNIX

To create a DLL from the z/OS UNIX shell, you must specify the cob2 option `-bdll`.

```
cob2 -o mydll -bdll mysub.cbl
```

When you specify `cob2 -bdll`:

- The COBOL compiler uses the compiler options `DLL`, `EXPORTALL`, and `RENT`, which are required for DLLs.
- The link step produces a DLL definition side file that 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 a suffix, that suffix 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 file later when you create a module that calls that DLL, specify the side file with any other object files (*file.o*) that you need to link. For example, the following command compiles `myapp1.cbl`, uses the DLL option to enable `myapp1.o` to reference DLLs, and links to produce the module `myapp1`:

```
cob2 -o myapp1 -qdll myapp1.cbl mydll.x
```

[“Example: using cob2 to compile and link under z/OS UNIX” on page 286](#)

Related tasks

[Chapter 30, “Creating a DLL or a DLL application,” on page 567](#)

[“Compiling programs to create DLLs” on page 568](#)

Related references

[“cob2 syntax and options” on page 287](#)

[“cob2 input and output files” on page 289](#)

Example: using cob2 to compile and link under z/OS UNIX

The following examples illustrate the use of `cob2`.

- To compile one file called `alpha.cbl`, enter:

```
cob2 -c alpha.cbl
```

The compiled file is named `alpha.o`.

- To compile two files called alpha.cbl and beta.cbl, enter:

```
cob2 -c alpha.cbl beta.cbl
```

The compiled files are named alpha.o and beta.o.

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

```
cob2 alpha.cbl beta.cbl -o gamma
```

This command creates alpha.o and beta.o, then links alpha.o, beta.o, and the COBOL libraries. If the link step is successful, it produces an executable program named gamma.

- To compile alpha.cbl with the LIST and NOADATA options, enter:

```
cob2 -qlist,noadata alpha.cbl
```

- To compile alpha.cbl with the SQL option so that the database request module (DBRM) is written to member "alpha" of the existing PDS data set USER.COBOL.DBRMLIB, enter:

```
cob2 -qsql alpha.cbl -o alpha -dbrm1ib=USER.COBOL.DBRMLIB
```

Note: The SQL coprocessor must be in your STEPLIB in order for this to work.

- To compile alpha.cbl with the SQL option so that the database request module (DBRM) is written to z/OS UNIX file alpha.dbrm, enter:

```
cob2 -qsql alpha.cbl -o alpha -dbrm1ib
```

Note: The SQL coprocessor must be in your STEPLIB in order for this to work, and you need to use the SQL coprocessor for Db2 12 with APAR PI88171 applied.

cob2 syntax and options

You can use the options listed below with the cob2 command. (Do not capitalize cob2.)



If you specify cob2 without any options or input files, the compiler manual page will be displayed.

-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 (except for the special cases noted below). The name and value 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 for *xxx* is *dll*, which specifies that the executable module is to be a DLL. This string 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`.

-dbrmlib=*xxx*

Specifies the location to be used for the generated database request module (DBRM). Only valid when the SQL compiler option is also specified.

- When *xxx* is not specified, the DBRM is written to a z/OS UNIX file. If the input file for the compile operation is named *file.cb1*, then the DBRM file will be named *file.dbrm*.
- When *xxx* is specified, *xxx* represents the name of an existing data set that will hold the generated DBRM. Note that the provided name *xxx* is used as-is with no extra qualification performed.

-e *xxx*

Specifies the name of the program to be used as the entry point of the module. The program must be one of the programs that will be included in the module. If you do not specify `-e`, the default entry point is the first program (*file.cb1*) 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.

-help

Displays the manual page of the compiler. If you specify `cob2 -help`, regardless of whether you provide input files, the compiler manual page is displayed and the compilation stops. This option has the same effect as `-?`.

-I*xxx*

Adds a path *xxx* to the directories to be searched for copybooks 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 in which you have not coded an explicit *library-name*, the compiler searches for copybooks in the following 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
4. In the locations specified in instances of the COPYLOC option that use the default library name or specify an explicit library name of SYSLIB

-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 in the `-L` option, then in the usual search order. (This option is lowercase *l*, not uppercase *I*.)

-M

Generates a make dependency file that contains the path and file names of all the z/OS UNIX-based copybook files for the source files that are input to the compiler. Here is sample output:

```
mkdepend.o: mkdepend.cbl
mkdepend.o: /home/usera/test/f171592/WSCOPY1.cpy
mkdepend.o: /home/usera/test/f171592/PDCOPY1.cpy
mkdepend.o: /home/usera/test/f171592/PDCOPY2.cpy
```

-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 quotation marks 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. Here is sample output:

```
cob2 -v -o mini -qssrange mini.cbl
compiler: ATTCRCTL PARM=RENT,TERM,SSRANGE /u/userid/cobol/mini.cbl
PP 5655-EC6 IBM Enterprise COBOL for z/OS 6.4.0 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.

-?

Displays the manual page of the compiler. If you specify cob2 -?, regardless of whether you provide input files, the compiler manual page is displayed and the compilation stops. This option has the same effect as -help.

Related tasks

[“Compiling and linking with the cob2 command” on page 285](#)

[“Creating a DLL under z/OS UNIX” on page 286](#)

[“Setting environment variables under z/OS UNIX” on page 283](#)

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)

If 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>
a.out	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 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 compiler option TEST(. . .,SEP,. . .)
<i>file.dbrm</i>	Database request module (DBRM)	Created by the compiler if you specify the <code>-dbrmLib</code> option alone, without xxx specified
<i>file.dek</i>	Extended COBOL source output from library processing	Created by the compiler if you specify compiler option MDECK
<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.u</i>	Make dependency file that contains the paths and file names of all the z/OS UNIX-based copybook files that are needed for the current compile	Created by the compiler
<i>file.x</i>	DLL definition side file	Created during the cob2 linking phase when creating <i>file.dll</i>
<i>class.java</i>	Java class definition (source)	Created when you compile a class definition

Related tasks

[“Compiling and linking with the cob2 command” on page 285](#)

Related references

[“ADATA” on page 335](#)

[“MDECK” on page 376](#)

[“TEST” on page 404](#)

UNIX System Services Command Reference

Compiling using scripts

If you use a shell script to automate cob2 tasks, you must code option syntax carefully to prevent the shell from passing invalid strings to cob2.

Code option strings in scripts as follows:

- Use an equal sign and colon rather than a left and right parenthesis, respectively, to specify compiler suboptions. For example, code `-qOPTIMIZE=1: ,XREF` instead of `-qOPTIMIZE(1) ,XREF`.
- Use an underscore rather than an apostrophe where a compiler option requires apostrophes for delimiting a suboption.
- Do not use blanks in the option string.

Chapter 16. Compiling, linking, and running OO applications

It is recommended that you compile, link, and run object-oriented (OO) applications in the z/OS UNIX environment. However, with certain limitations explained in the Related tasks, it is possible to compile, link, and run OO COBOL applications by using standard batch JCL or TSO/E commands.

Related tasks

[“Compiling, linking, and running OO applications under z/OS UNIX” on page 291](#)

[“Compiling, linking, and running OO applications in JCL or TSO/E” on page 295](#)

[“Using Java SDKs for z/OS” on page 298](#)

Compiling, linking, and running OO applications under z/OS UNIX

When you compile, link, and run object-oriented applications in a z/OS UNIX environment, application components reside in the z/OS UNIX file system. You compile and link them by using shell commands, and run them at a shell command prompt or with the BPXBATCH utility from JCL or TSO/E.

Related tasks

[“Compiling OO applications under z/OS UNIX” on page 291](#)

[“Preparing OO applications under z/OS UNIX” on page 292](#)

[“Running OO applications under z/OS UNIX” on page 293](#)

Compiling OO applications under z/OS UNIX

When you compile OO applications in a z/OS UNIX shell, use the `cob2` command to compile COBOL client programs and class definitions, and the `javac` command to compile Java class definitions to produce *bytecode* (suffix `.class`).

To compile COBOL source code that contains OO syntax such as `INVOKE` statements or class definitions, or that uses Java services, you must use these compiler options: `RENT`, `DLL`, `THREAD`, and `DBCS`. (The `RENT` and `DBCS` options are defaults.)

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

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

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

If a COBOL client program or class definition includes the file `JNI.cpy` by using a `COPY` statement, specify the `include` subdirectory of the COBOL install directory (typically `/usr/lpp/cobol/include`) in the search order for copybooks. You can specify the `include` subdirectory by using the `-I` option of the `cob2` command or by setting the `SYSLIB` environment variable.

Related tasks

[Chapter 15, “Compiling under z/OS UNIX,” on page 283](#)

[“Preparing OO applications under z/OS UNIX” on page 292](#)

[“Running OO applications under z/OS UNIX” on page 293](#)

[“Setting and accessing environment variables” on page 514](#)

[“Accessing JNI services” on page 707](#)

Related references

[“cob2 syntax and options” on page 287](#)

[“DBCS” on page 350](#)

[“DLL” on page 353](#)

[“RENT” on page 391](#)

[“THREAD” on page 408](#)

Preparing OO applications under z/OS UNIX

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

To prepare an OO COBOL client program for execution, link the object file with the following two DLL side files to create an executable module:

- `libjvm.x`, which is provided with your IBM Java Software Development Kit.
- `igzcjava.x`, which is provided in the `lib` subdirectory of the COBOL install directory in the z/OS UNIX file system. This DLL side file is also available as the member `IGZCJAVA` in the `SCEELIB` PDS (part of Language Environment).

To prepare a COBOL class definition for execution:

1. Link the object file using the two DLL side files mentioned above to create an executable DLL module.

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

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

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

Command	Input	Output
<code>cob2 -c -qdll,thread Classname.cbl</code>	<code>Classname.cbl</code>	<code>Classname.o</code> , <code>Classname.java</code>
<code>cob2 -bdll -o libClassname.so Classname.o /usr/lpp/java/IBM/J8.0/bin/j9vm/libjvm.x /usr/lpp/cobol/igyv6r4/lib/igzcjava.x</code>	<code>Classname.o</code>	<code>libClassname.so</code>
<code>javac Classname.java</code>	<code>Classname.java</code>	<code>Classname.class</code>

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

[“Example: compiling and linking a COBOL class definition under z/OS UNIX” on page 293](#)

Related tasks

Chapter 15, “Compiling under z/OS UNIX,” on page 283

[“REPOSITORY paragraph for defining a class” on page 672](#)

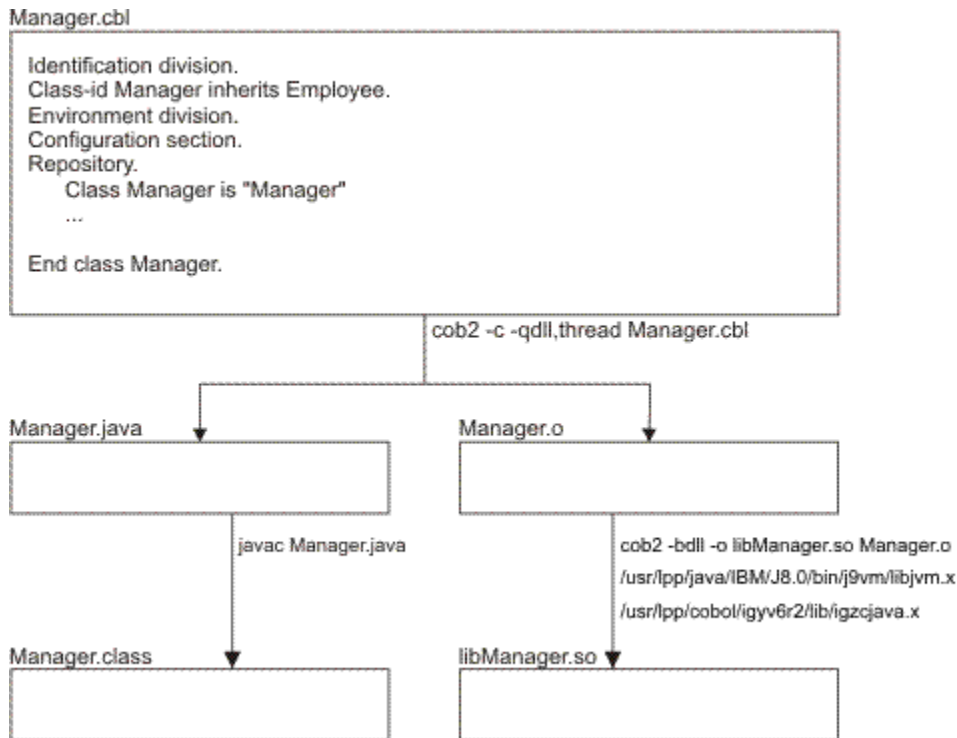
Related references

[“cob2 syntax and options” on page 287](#)

[“Object-oriented syntax, and Java 6 or later” on page 299](#)

Example: compiling and linking a COBOL class definition under z/OS UNIX

This example illustrates the commands that you use and the files that are produced when you compile and link a COBOL class definition, `Manager.cbl`, using z/OS UNIX shell commands.



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

Running OO applications under z/OS UNIX

It is recommended that you run object-oriented COBOL applications as z/OS UNIX applications. You must do so if an application begins with a Java program or the `main` factory method of a COBOL class.

Specify the directory that contains the DLLs for the COBOL classes in the `LIBPATH` environment variable. Specify the directory paths for the Java class files that are associated with the COBOL classes in the `CLASSPATH` environment variable as follows:

- For classes that are not part of a package, end the class path with the directory that contains the `.class` files.
- For classes that are part of a package, end the class path with the directory that contains the "root" package (the first package in the full package name).
- For a `.jar` file that contains `.class` files, end the class path with the name of the `.jar` file.

Separate multiple path entries with colons.

Note: If the `CLASSPATH` environment variable contains the filename wildcard (`*`), the wildcard cannot be expanded in the case where the JVM is initialized by the COBOL runtime.

Related tasks

[“Running OO applications that start with a main method” on page 294](#)

[“Running OO applications that start with a COBOL program” on page 294](#)

[“Running J2EE COBOL clients” on page 294](#)

[Chapter 25, “Running COBOL programs under z/OS UNIX,” on page 513](#)

[“Setting and accessing environment variables” on page 514](#)

[Chapter 39, “Writing object-oriented programs,” on page 667](#)

[“Structuring OO applications” on page 703](#)

Running OO applications that start with a main method

If the first routine of a mixed COBOL and Java application is the `main` method of a Java class or the `main` factory method of a COBOL class, run the application by using the `java` command and by specifying the name of the class that contains the `main` method.

The `java` command initializes the Java virtual machine (JVM). To customize the initialization of the JVM, specify options on the `java` command as in the following examples:

Purpose	Option
To set a system property	<code>-Dname=value</code>
To request that the JVM generate verbose messages about garbage collection	<code>-verbose:gc</code>
To request that the JVM generate verbose messages about class loading	<code>-verbose:class</code>
To request that the JVM generate verbose messages about native methods and other Java Native Interface activity	<code>-verbose:jni</code>
To set the initial Java heap size to <i>value</i> bytes	<code>-Xmsvalue</code>
To set the maximum Java heap size to <i>value</i> bytes	<code>-Xmxvalue</code>

For details about the options that the JVM supports, see the output from the `java -h` command, or see the Related references.

Related references

[IBM SDK for Java - Tools Documentation](#)

[WebSphere for z/OS: Applications \(Java Naming and Directory Interface \(JNDI\)\)](#)

Running OO applications that start with a COBOL program

If the first routine of a mixed COBOL and Java application is a COBOL program, run the application by specifying the program name at the command prompt. If a JVM is not already running in the process of the COBOL program, the COBOL run time automatically initializes a JVM.

To customize the initialization of the JVM, specify options by setting the `COBJVMINITOPTIONS` environment variable. Use blanks to separate options. For example:

```
export COBJVMINITOPTIONS="-Xms10000000 -Xmx20000000 -verbose:gc"
```

Related tasks

[“Using Java SDKs](#)

[for z/OS” on page 298](#)

[Chapter 25, “Running COBOL programs under z/OS UNIX,” on page 513](#)

[“Setting and accessing environment variables” on page 514](#)

Related references

[IBM SDK for Java - Tools Documentation](#)

[WebSphere for z/OS: Applications \(Java Naming and Directory Interface \(JNDI\)\)](#)

Running J2EE COBOL clients

You can use OO syntax in a COBOL program to implement a Java 2 Platform, Enterprise Edition (J2EE) client. You can, for example, invoke methods on enterprise beans that run in the WebSphere for z/OS environment.

Before you run a COBOL J2EE client, you must set the Java system property `java.naming.factory.initial` to access WebSphere naming services. For example:

```
export COBJVMINITOPTIONS
="-Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory"
```

[“Example: J2EE client written in COBOL” on page 719](#)

Compiling, linking, and running OO applications in JCL or TSO/E

It is recommended that you compile, link, and run applications that use OO syntax in the z/OS UNIX environment.

However, in limited circumstances it is possible to compile, prepare, and run OO applications by using standard batch JCL or TSO/E commands. To do so, you must follow the guidelines that are in the Related tasks. For example, you might follow this approach for applications that consist of a COBOL main program and subprograms that:

- Access objects that are all implemented in Java
- Access enterprise beans that run in a WebSphere server

Related tasks

[“Compiling OO applications in JCL or TSO/E” on page 295](#)

[“Preparing and running OO applications in JCL or TSO/E” on page 296](#)

[“Compiling, linking, and running OO applications under z/OS UNIX” on page 291](#)

Compiling OO applications in JCL or TSO/E

If you use batch JCL or TSO/E to compile an OO COBOL program or class definition, the generated object file is written, as usual, to the data set that has ddname SYSLIN or SYSPUNCH. You must use compiler options RENT, DLL, THREAD, and DBCS. (RENT and DBCS are defaults.)

If the COBOL program or class definition uses the JNI environment structure to access JNI callable services, add the SIGYLIB data set to the data set concatenation for the SYSLIB DD statement, and use a COPY statement of the form COPY JNI in the COBOL source.

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

When you compile a COBOL class definition, a Java source program that contains a class definition that corresponds to the COBOL class definition is generated in addition to the object file. Use the SYSJAVA ddname to write the generated Java source file to a file in the z/OS UNIX file system. For example:

```
//SYSJAVA DD PATH='/u/userid/java/Classname.java',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=SIRWXU,
// FILEDATA=TEXT
```

Do not edit this generated Java class definition in any way. If you change the COBOL class definition, you must regenerate both the object file and the Java class definition by recompiling the updated COBOL class definition.

Compile Java class definitions by using the `javac` command from a z/OS UNIX shell command prompt, or by using the BPXBATCH utility.

[“Example: compiling, linking, and running an OO application using JCL” on page 297](#)

Related tasks

[“Compiling with JCL” on page 255](#)

[“Compiling under TSO” on page 262](#)

[“Specifying source libraries \(SYSLIB\)” on page 270](#)

[“Defining the Java-source](#)

[output file \(SYSJAVA\)” on page 272](#)
[“Accessing JNI services” on page 707](#)
[“Compiling OO applications under z/OS UNIX” on page 291](#)
[“Preparing OO applications under z/OS UNIX” on page 292](#)

Related references

[“DBCS” on page 350](#)
[“DLL” on page 353](#)
[“RENT” on page 391](#)
[“THREAD” on page 408](#)
Appendix F, “JNI.cpy copybook,” on page 833
UNIX System Services User's Guide (The BPXBATCH utility)

Preparing and running OO applications in JCL or TSO/E

It is recommended that you run OO applications in a z/OS UNIX environment. To run OO applications from batch JCL or TSO/E, you should therefore use the BPXBATCH utility.

In limited circumstances, however, you can run an OO application by using standard batch JCL (EXEC PGM=COBPROG) or the TSO/E CALL command. To do so, follow these requirements when preparing the application:

- Structure the application to start with a COBOL program. (If an application starts with a Java program or with the main factory method of a COBOL class, you must run the application under z/OS UNIX, and the application components must reside in the z/OS UNIX file system.)
- **Link-edit considerations:** Link the program object for the COBOL program into a PDSE. COBOL programs that contain object-oriented syntax must be link-edited with AMODE 31.
- Ensure that the class files and DLLs associated with the COBOL or Java classes that are used by the application reside in the z/OS UNIX file system. You must name the class files and DLLs as described in the related task about preparing OO applications under z/OS UNIX.
- Specify INCLUDE control statements for the DLL side files libjvm.x and igzcpyjava.x when you bind the object deck for the main program. For example:

```
INCLUDE '/usr/lpp/java/IBM/J8.0/bin/j9vm/libjvm.x'  
INCLUDE '/usr/lpp/cobol/igyv6r4/lib/igzcpyjava.x'
```

- Create a file that contains the environment variable settings that are required for Java. For example, a file `/u/userid/javaenv` might contain the three lines shown below to set the PATH, LIBPATH, and CLASSPATH environment variables.

```
PATH=/bin:/usr/lpp/java/IBM/J8.0/bin  
LIBPATH=/lib:/usr/lib:/usr/lpp/java/IBM/J8.0/bin:/usr/lpp/java/IBM/J8.0/bin/j9vm  
CLASSPATH=./u/userid/applications
```

To customize the initialization of the JVM that will be used by the application, you can set the COBJVMINIOPTIONS environment variable in the same file. For example, to access enterprise beans that run in a WebSphere server, you must set the Java system property `java.naming.factory.initial`. For details, see the related task about running OO applications.

When you run an OO application that starts with a COBOL program by using standard batch JCL or the TSO/E CALL command, follow these guidelines:

- Use the `_CEE_ENVFILE` environment variable to indicate the location of the file that contains the environment variable settings required by Java. Set `_CEE_ENVFILE` by using the ENVAR runtime option.
- Specify the POSIX(ON) and XPLINK(ON) runtime option.
- Use DD statements to specify files in the z/OS UNIX file system for the standard input, output, and error streams for Java:
 - `JAVAIN DD for the input from statements such as c=System.in.read();`

- JAVAOUT DD for the output from statements such as `System.out.println(string);`
- JAVAERR DD for the output from statements such as `System.err.println(string);`
- Ensure that the SCEERUN2 and SCEERUN load libraries are available in the system library search order, for example, by using a STEPLIB DD statement.

[“Example: compiling, linking, and running an OO application using JCL” on page 297](#)

Related tasks

[“Preparing OO applications under z/OS UNIX” on page 292](#)

[“Running OO applications under z/OS UNIX” on page 293](#)

[“Structuring OO applications” on page 703](#)

UNIX System Services User's Guide (The BPXBATCH utility)

Language Environment Programming Guide (Running an application under batch)

Related references

XL C/C++ Programming Guide (_CEE_ENVFILE)

Language Environment Programming Reference (ENVAR)

Example: compiling, linking, and running an OO application using JCL

This example shows sample JCL that you could use to compile, link, and run a COBOL client that invokes a Java method.

The example shows:

- JCL to compile, link, and run an OO COBOL program, TSTHELLO
- A Java class definition, HelloJ, that contains a method that the COBOL program invokes
- A z/OS UNIX file, ENV, that contains the environment variable settings that Java requires

JCL for program TSTHELLO

```
//TSTHELLO JOB ,
// TIME=(1),MSGLEVEL=(1,1),MSGCLASS=H,CLASS=A,REGION=200M,
// NOTIFY=&SYSUID,USER=&SYSUID
//*
// SET COBPRFX='IGY.V6R4M0'
// SET LIBPRFX='CEE'
//*
//COMPILE EXEC PGM=IGYCRCTL,
//SYSLIN DD DSNNAME=&&OBJECT(TSTHELLO),UNIT=VIO,DISP=(NEW,PASS),
// SPACE=(CYL,(1,1,1))
//SYSPRINT DD SYSOUT=*
//STEPLIB DD DSN=&COBPRFX..SIGYCOMP,DISP=SHR
// DD DSN=&LIBPRFX..SCEERUN,DISP=SHR
// DD DSN=&LIBPRFX..SCEERUN2,DISP=SHR
//SYSUT1 DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT2 DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT3 DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT4 DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT5 DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT6 DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT7 DD UNIT=VIO,SPACE=(CYL,(1,1))
//SYSUT8 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT9 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT10 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT11 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT12 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT13 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT14 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSUT15 DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSMDECK DD UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSIN DD *
cbl dll,thread
Identification division.
Program-id. "TSTHELLO" recursive.
Environment division.
Configuration section.
Repository.
Class HelloJ is "HelloJ".
```

```

Data Division.
Procedure division.
    Display "COBOL program TSTHELLO entered"
    Invoke HelloJ "sayHello"
    Display "Returned from java sayHello to TSTHELLO"
    Goback.
End program "TSTHELLO".

/*
//LKED EXEC PGM=IEWL, PARM='RENT, LIST, LET, DYNAM(DLL), CASE(MIXED)'
//SYSLIB DD DSN=&LIBPRFX..SCEELKED, DISP=SHR
// DD DSN=&LIBPRFX..SCEELKEX, DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTEM DD SYSOUT=*
//SYSLMOD DD DSN=&&GOSET(TSTHELLO), DISP=(MOD, PASS), UNIT=VIO,
// SPACE=(CYL, (1, 1, 1)), DSNTYPE=LIBRARY
//SYSDEFS DD DUMMY
//OBJMOD DD DSN=&&OBJECT, DISP=(OLD, DELETE)
//SYSLIN DD *
INCLUDE OBJMOD(TSTHELLO)
INCLUDE '/usr/lpp/java/IBM/J8.0/bin/j9vm/libjvm.x'
INCLUDE '/usr/lpp/cobol/igyv6r4/lib/igzjava.x'

/*
//GO EXEC PGM=TSTHELLO, COND=(4, LT, LKED),
// PARM='/ENVAR("_CEE_ENVFILE=/u/userid/octest/tsthello/ENV")
// POSIX(ON) XPLINK(ON)'
//STEPLIB DD DSN=*.LKED.SYSLMOD, DISP=PASS
// DD DSN=&LIBPRFX..SCEERUN2, DISP=SHR
// DD DSN=&LIBPRFX..SCEERUN, DISP=SHR
//SYSOUT DD SYSOUT=*
//CEEDUMP DD SYSOUT=*
//SYSUDUMP DD DUMMY
//JAVAOUT DD PATH='/u/userid/octest/tsthello/javaout',
// PATHOPTS=(OWRONLY, OCREAT, OTRUNC),
// PATHMODE=(SIRUSR, SIWUSR, SIRGRP)
//JAVAERR DD PATH='/u/userid/octest/tsthello/javaerr',
// PATHOPTS=(OWRONLY, OCREAT, OTRUNC),
// PATHMODE=(SIRUSR, SIWUSR, SIRGRP)

```

Definition of class HelloJ

```

class HelloJ {
    public static void sayHello() {
        System.out.println("Hello World, from Java!");
    }
}

```

HelloJ.java is compiled with the javac command. The resulting .class file resides in the z/OS UNIX file system directory `u/userid/octest/tsthello`, which is specified in the **CLASSPATH** environment variable in the environment variable settings file.

Environment variable settings file, ENV

```

PATH=/bin:/usr/lpp/java/IBM/J8.0/bin
LIBPATH=/lib:/usr/lib:/usr/lpp/java/IBM/J8.0/bin:/usr/lpp/java/IBM/J8.0/bin/j9vm
CLASSPATH=./u/userid/octest/tsthello

```

The environment variable settings file also resides in directory `u/userid/octest/tsthello`, as specified in the **_CEE_ENVFILE** environment variable in the JCL.

Using Java SDKs for z/OS

The Java SDKs for z/OS are based on the XPLINK linkage convention defined by Language Environment.

If the application starts with a Java program or the main factory method of a COBOL class, the XPLINK environment is automatically started by the java command that starts the JVM and runs the application.

If an application starts with a COBOL program that invokes methods on COBOL or Java classes, you must specify the XPLINK(ON) runtime option so that the XPLINK environment is initialized. XPLINK(ON) is not recommended as a default setting, however; you should use XPLINK(ON) only for applications that specifically require it.

When you are running an application under z/OS UNIX, you can set the XPLINK(ON) option by using the _CEE_RUNOPTS environment variable as follows:

```
_CEE_RUNOPTS="XPLINK(ON)"
```

Exporting _CEE_RUNOPTS="XPLINK(ON)" so that it is in effect for the entire z/OS UNIX shell session is not recommended, however. Suppose for example that an OO COBOL application starts with a COBOL program called App1Driver. One way to limit the effect of the XPLINK option to the execution of the App1Driver application is to set the _CEE_RUNOPTS variable on the command-line invocation of App1Driver as follows:

```
_CEE_RUNOPTS="XPLINK(ON)" App1Driver
```

Related tasks

[“Running OO applications under z/OS UNIX” on page 293](#)

[“Setting and accessing environment variables” on page 514](#)

Related references

[“Object-oriented syntax, and Java 6 or later” on page 299](#)

[“Runtime environment variables” on page 515](#)

Language Environment Programming Reference (XPLINK)

XL C/C++ Programming Guide (_CEE_RUNOPTS)

Object-oriented syntax, and Java 6 or later

Enterprise COBOL 5.2 and later applications that use object-oriented syntax for Java interoperability are supported with Java 6 or later.

Earlier versions of Enterprise COBOL applications that use object-oriented syntax for Java interoperability were supported with Java SDK 1.4.2 and Java 5. To run these applications with Java 6 or later, do these steps:

1. Recompile and relink the applications using Enterprise COBOL 5.2 or later.
2. Recompile the generated Java class that is associated with each object-oriented COBOL class using the javac command from Java 6 or later.

Related tasks

[“Preparing OO applications under z/OS UNIX” on page 292](#)

Chapter 17. Compiling, linking, and running non-OO COBOL applications that interoperate with Java

Enterprise COBOL provides support for a more comprehensive and flexible form of COBOL/Java interoperability than that provided by the object-oriented (OO) COBOL framework.

Building non-OO applications that interoperate with Java

This section describes building non-OO COBOL applications that interoperate with Java using built-in Java interoperability features. Existing COBOL programs that are part of the application can be compiled as usual, though they may need new linking considerations. A utility program called `cjbuild` is provided to assist with building compiler-generated stub files into a DLL that is needed at run time by the application when executing the "glue code" that facilitates interoperability between COBOL and Java.

In general, building a COBOL application that interoperates with Java takes place in the following four stages:

1. Compile all user programs in the application. Programs that contain non-OO COBOL/Java interoperability features (that is, `JAVA-CALLABLE` directive, `JAVA-SHAREABLE` directive, or calls to static Java methods) must be linked in step 3. However, user programs that do not contain any non-OO COBOL/Java interoperability features can be linked as normal during this step, unless there is a dependency on programs that must be linked in step 3, in which case linking for these programs must be delayed until step 3 as well.

Note: When a COBOL program in the application contains one or more Java-interoperability constructs, compiler generated COBOL "stub programs" are automatically generated to facilitate interoperability with Java. The COBOL constructs that cause these stub programs to be generated are as follows:

- The `JAVA-CALLABLE` directive, which results in a COBOL native method stub program.
- The `JAVA-SHAREABLE` directive, which results in a working-storage initialization stub program, and also in a Java class that provides an interface for accessing all shareable `WORKING-STORAGE` data items in the COBOL program.
- `CALL` statements with a literal call target of the form `'Java.java-class-name.java-static-method-name'`, which results in a Java call stub program, one for each unique Java call target.

Any stub files that are produced will be compiled and linked in subsequent steps.

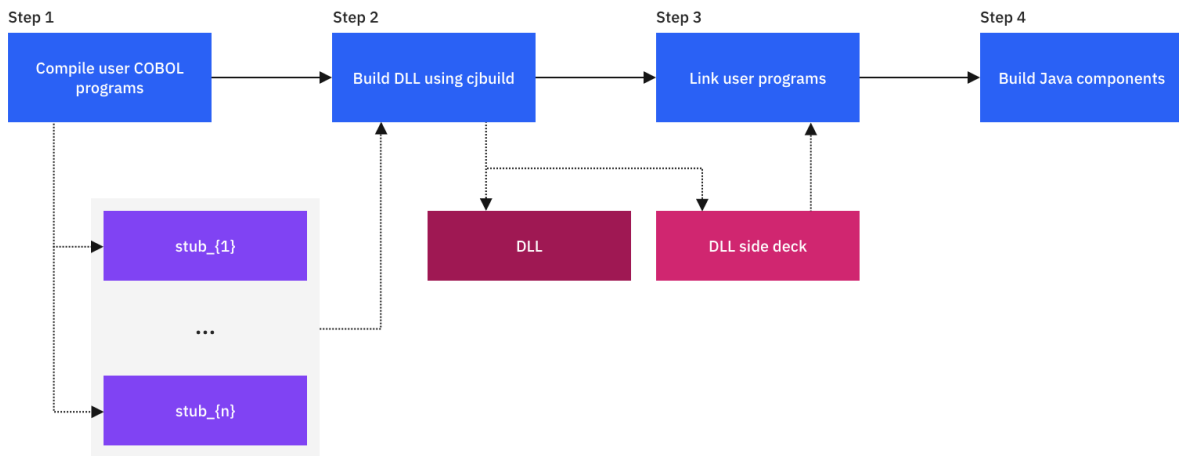
2. Use the `cjbuild` utility to build a DLL from all the COBOL native method stub programs, `WORKING-STORAGE` initialization stub programs, and Java call stub programs produced in step 1. A side deck for this DLL will also be generated and will be needed for linking in step 3.

Note: The `cjbuild` utility also generates several Java source files for classes that are needed by Java applications when calling COBOL native methods or accessing COBOL `WORKING-STORAGE` memory. The `cjbuild` utility takes care of building these files.

3. Link user COBOL programs. Programs that don't use non-OO COBOL/Java interoperability features can be linked normally and, if desired, can be linked during step 1, if there are no dependencies on programs that need to be linked in step 3. Programs that use non-OO COBOL/Java interoperability features, however, must be linked with the side deck produced for the DLL built in step 2.
4. Compile the user Java parts of the application.

The following figure describes the four stages of building a Java interoperable COBOL application:

Building a COBOL application that interoperates with Java



Related tasks

CALL statement (*Enterprise COBOL for z/OS Language Reference*)

JAVA-CALLABLE (*Enterprise COBOL for z/OS Language Reference*)

JAVA-SHAREABLE (*Enterprise COBOL for z/OS Language Reference*)

Introducing the COBPROD application

This section introduces a simple application called COBPROD that will be used in subsequent sections when describing how to build non-OO COBOL applications that interoperate with Java.

The COBPROD application consists of the following z/OS UNIX files:

- **cobprod.cbl**

Java-callable COBOL program that uses the JAVA-SHAREABLE directive to identify WORKING-STORAGE data items to be shared with Java.

- **addlist.cbl**

Java-callable COBOL program that takes an array of three `java.math.BigDecimal` objects, computes their sum and calls back to a Java static method to dump the result to the terminal, then returns the actual sum to the Java caller.

- **CobProd.java**

Java program that starts the application, calls out to COBOL native method COBPROD, retrieves information from the PROD-INFO table in COBPROD's WORKING-STORAGE memory, then calls COBOL native method 'addlist'.

The COBPROD application starts on the Java side with the "main" method of the CobProd class making a call to the COBOL native method COBPROD, which initializes a Java-shareable table of product information in its WORKING-STORAGE memory. When control returns to Java, some of the prices from the product information table are collected into an array and passed to COBOL native method 'addlist', which sums the prices and calls back to Java static method CobProd.printResult to dump the sum of the list to the terminal.

For details about the z/OS UNIX files, see [“Example: COBPROD application - building and running” on page 309](#).

Preparing to compile, bind, and run non-OO COBOL applications that interoperate with Java

This section discusses prerequisites for building and running non-OO COBOL programs that interoperate with Java.

In the discussion, compiling and linking will be done from z/OS UNIX using the `cob2` utility, and building the stub program DLL needed at run time by the application will be done from z/OS UNIX using the `cjbuild` utility. However, examples of how to send both the program objects and DLL to an MVS data set are provided.

Note: Both `cob2` and the `cjbuild` utility exist in the `bin` directory of your COBOL install directory in the z/OS UNIX file system. The default location is `/usr/lpp/cobol/igyvXrY`, where `X` is the compiler version and `Y` is the compiler release. This directory must be included in your UNIX `PATH` environment variable.

The `cjbuild` utility is provided as a UNIX only utility, but it can be launched from JCL using `BPX BATCH`.

Ensure that the `JAVA_HOME` environment variable is set to the Java SDK installation directory on your system. For example, if you use Java 8, the directory might be `/usr/lpp/java/IBM/J8.0` for AMODE 31 (31-bit) Java, and `/usr/lpp/java/IBM/J8.0_64` for AMODE 64 (64-bit) Java.

Note: You can choose any method to compile and link parts, such as JCL and z/OS UNIX command line. The z/OS UNIX method is used because UNIX is a natural environment for Java.

Step 1: Compiling user COBOL programs

This topic describes how to compile the user COBOL programs in a non-OO COBOL application that interoperates with Java.

In general, user COBOL programs in the application can be compiled in the same way that users normally compile their user COBOL programs. The exception is that any program that contains the `JAVA-CALLABLE` or `JAVA-SHAREABLE` directives, or uses a `CALL` statement to make a call to a static Java method, should now consider specifying the Java interoperability option `JAVAIOP`, which allows you to control various aspects of a Java interoperable program.

In particular, the `JAVAIOP` option allows you to do the following:

- Control where compiler generated stub files will be written to the z/OS UNIX file system during compilation with the `OUTPATH` suboption. These stub programs facilitate interoperability with Java.
Note: The z/OS UNIX directory provided to the `OUTPATH` suboption of `JAVAIOP` must exist prior to running the compiler.
- Control whether the interoperable application will target 31-bit Java by default or 64-bit Java with the `JAVA64` suboption. Note that the `JAVA64` suboption is only available when compiler option `LP(32)` is in effect. When `LP(64)` is in effect, 64-bit Java is required.
- Specify a string of Java Virtual Machine (JVM) initialization options with `JVMINITOPTIONS` suboption. This is only needed for applications that start on the COBOL side and only when compiling COBOL programs in the application that contain `CALL` statements to Java static methods, a situation that might result in COBOL needing to start the JVM at run time. Using `JAVAIOP` option with the `JVMINITOPTIONS` suboption is an alternative to setting the `COBJVMINITOPTIONS` environment variable at run time for the program, which is the method required by OO COBOL and also available for non-OO Java interoperability.

For the `COBPROD` application, the following commands build the COBOL files, instructing the compiler to send any generated stub files and Java files to the z/OS UNIX directory `${HOME}/jtoc/docs`.

```
cob2 -c cobprod.cb1 "-qjvaiop(OUTPATH('${HOME}/jtoc/docs'))"  
cob2 -c addlist.cb1 "-qjvaiop(OUTPATH('${HOME}/jtoc/docs'))"
```

Since you are compiling these programs with the default LP(32) option by default, the compiler assumes you are targeting 31-bit Java. To target 64-bit Java instead of 31-bit Java, you would simply change the compile commands to include the JAVA64 suboption of the JAVAIOPT option.

```
cob2 -c cobprod.cbl "-qjavaiop(OUTPATH('${HOME}/jtoc/docs'),java64)"
cob2 -c addlist.cbl "-qjavaiop(OUTPATH('${HOME}/jtoc/docs'),java64)"
```

Note: If this application starts on the Java side, there is no need to specify the JVMINIOPTIONS suboption, since it is Java that starts the JVM.

Related references

JAVAIOP

[“Example: COBPROD application - building and running” on page 309](#)

JAVA-CALLABLE (*Enterprise COBOL for z/OS Language Reference*)

JAVA-SHAREABLE (*Enterprise COBOL for z/OS Language Reference*)

Step 2: Running the cjbuild utility to build a DLL of Java stub programs

This topic describes how to run the cjbuild utility to build a DLL of Java stub programs.

When compiling the cobprod.cbl and addlist.cbl files in “[Step 1: Compiling user COBOL programs](#)” on [page 303](#), four stub files were produced in the specified z/OS UNIX output directory. These files must be built into a DLL that is used at run time by COBOL/Java interoperable applications.

The file names of the four files produced in the specified z/OS UNIX output directory are the following:

- COBPROD_java_native.cbl (COBOL native method stub for COBPROD)
- COBPROD_java_native_WS.cbl (WORKING-STORAGE initialization stub for COBPROD)
- addlist_java_native.cbl (COBOL native method stub for addlist)
- Java.CobProd.printResult.cbl (Java call stub for CobProd.printResult)

The cjbuild utility is provided to assist you in building these files into the necessary DLL. In general you don't need to be aware of these files. Instead, you only need to provide cjbuild with the name of an input file that contains the names of any COBOL programs that contain JAVA-CALLABLE, JAVA-SHAREABLE directives, or both, and also the names of Java static methods that are called from COBOL, where the name that appears in the file is the same name that appears in the CALL statement. In the COBPROD application, there are three such programs or methods: COBPROD, addlist, and Java.CobProd.printResult.

Note: The COBPROD application deliberately has one program that has a name that is not a literal and is case insensitive, and one program where the name *is* a literal and is case sensitive. This is for demonstration purposes. When providing the list of these programs to cjbuild, take the case into account.

In the COBPROD application example, let's assume that there is a file named "methods" that resides in the same directory on the z/OS UNIX file system that cjbuild is being run from, and let's further assume that it contains the names of the following COBOL programs and static Java methods:

methods file:

```
COBPROD
addlist
Java.CobProd.printResult
```

If we run the `cjbuild` utility from the directory of stub files, the following `cjbuild` command could be used to build a stub DLL that resides on the z/OS UNIX file system in directory `${HOME}/java/demo`:

```
cjbuild -v -c ${HOME}/jtoc/docs -j ${HOME}/java/demo \
-m PURE31 -d ${HOME}/java/demo \
-p enterprise.COBOL methods app1
```

The table lists a summary of the arguments passed to `cjbuild` in this example and their meanings:

<i>Table 43.</i>	
Augments	Meanings
-v	Verbose output. The actual commands to compile and link the DLL and build any related Java source files are displayed, for example, <code>cob2</code> , <code>javac</code> , etc. Verbose output is strongly recommended.
-c <code>\${HOME}/jtoc/docs</code>	Tells <code>cjbuild</code> where to find the repository of COBOL stub files generated by the compiler. Those files were written to whatever directory was indicated by the <code>OUTPATH</code> suboption of the <code>JAVAIOP</code> option used when building the corresponding COBOL files in “Step 1: Compiling user COBOL programs” on page 303 . Default: current directory
-j <code>\${HOME}/java/demo</code>	Tells <code>cjbuild</code> where to output the Java <code>.class</code> files generated by <code>cjbuild</code> when compiling Java files generated by either the compiler or <code>cjbuild</code> itself. Default: current directory
-m PURE31	Tells <code>cjbuild</code> that you want to build an application with an AMODE 31 COBOL program with LP (32) in effect and 31-bit Java. Alternatives are <code>MIX_31_64</code> and <code>PURE64</code> . Default: PURE31

Table 43. (continued)

Augments	Meanings
<p>-d \${HOME}/java/demo</p>	<p>Tells cjbuild where to output the DLL. Can be a z/OS UNIX path or an MVS data set name.</p> <p>To output the DLL to an MVS data set named USER . COBOL . LOAD as member LIBAPP1, use the following -d option:</p> <pre data-bbox="862 436 1471 485">-d "//COBOL.LOAD"</pre> <p>or</p> <pre data-bbox="862 554 1471 602">-d "'//USER.COBOL.LOAD'"</pre> <p>Note: According to MVS binder conventions, when the DLL is output to a data set, the corresponding side deck file is written to a data set whose name is the name of the DLL output data set with the last name part replaced with EXP. In this case, it would be USER . COBOL . EXP. However, cjbuild will also make a copy of the side deck in the current z/OS UNIX directory in case user build tools want to access it from there.</p> <p>Default: current directory</p>
<p>-p enterprise.COBOL</p>	<p>Tells cjbuild the package name to which the progs and strg classes are to belong. The progs and strg classes provide the interface from Java to COBOL native methods and COBOL working-storage items, respectively.</p> <p>Default: enterprise.COBOL</p>

To output the DLL to an MVS data set called USER . COBOL . LOAD, issue the following command:

```
cjbuild -v -c ${HOME}/jtoc/docs -j ${HOME}/java/demo \
-d "//USER.COBOL.LOAD" -m PURE31 methods app1
```

Note: When the DLL is output to an MVS data set, make sure that data set is included in your STEPLIB at run time; otherwise, the JVM won't be able to locate it. When the DLL is output to a z/OS UNIX directory, that directory must be specified to the JVM using its `java.library.path` property.

When cjbuild runs, three Java files are also created and compiled:

- cobol.java
- progs.java
- strg.java

These files contain the Java classes `progs` and `strg` that provide the interfaces that allow you to access COBOL native methods and working-storage items. These class names are qualified by whatever package name you specified using the `-p` option.

For example, in this case, since `-p` specifies `enterprise.COBOL` as the package name, you can access native methods and WORKING-STORAGE items as follows in their Java program:

- `enterprise.COBOL.progs.COBPROD`
- `enterprise.COBOL.strg.COBPROD`

Any COBOL program that contains the JAVA-SHAREABLE directive will also generate a Java file that defines classes that can be used from a Java application to interface directly with COBOL WORKING-STORAGE memory. These Java files will be identified from the corresponding COBOL program name in the methods file and will be automatically built by cjbuild. The resulting class files will be placed in a directory indicated by the -j option of cjbuild.

An example output of the cjbuild utility:

```
Running cjbuild COBOL/Java build tool
=====

NATIVE_METHODS_FILE = methods
APP_NAME             = app1
COBOL_DIR            = .
DLL_OUT_DIR         = .
JAVA_CLASS_DIR      = /home/user/java/demo
JAVA_SRC_DIR        = .
PACKAGE_NAME        = enterprise.COBOL
INTEROP_MODE        = PURE31

JAVA_HOME: /usr/lpp/java/IBM/J8.0
COBOL_INSTALL_DIR: /home/user/cobol/igyv6r4

User env STEPLIB: CEE.SCEERUN:CEE.SCEERUN2

Including file in app DLL: COBPROD_java_native_app1
Including file in app DLL: COBPROD_java_native_WS_app1
Including file in app DLL: addlist_java_native_app1
Including file in app DLL: Java.CobProd.printResult

# Compiling native method stub for COBPROD_java_native_app1
/home/user/cobol/igyv6r4/bin/cob2 -qlist -c /home/user/jtoc/docs/COBPROD_java_native_app1.cbl
-I /home/user/cobol/igyv6r4/include
# Command return code 0

# Compiling WS init stub for COBPROD_java_native_WS_app1
/home/user/cobol/igyv6r4/bin/cob2 -qlist -c /home/user/jtoc/docs/
COBPROD_java_native_WS_app1.cbl -I /home/user/cobol/igyv6r4/include
# Command return code 0

# Compiling native method stub for addlist_java_native_app1
/home/user/cobol/igyv6r4/bin/cob2 -qlist -c /home/user/jtoc/docs/addlist_java_native_app1.cbl
-I /home/user/cobol/igyv6r4/include
# Command return code 0

# Compiling stub for Java.CobProd.printResult
/home/user/cobol/igyv6r4/bin/cob2 -qlist -c /home/user/jtoc/docs/Java.CobProd.printResult.cbl
-I /home/user/cobol/igyv6r4/include
# Command return code 0

# Linking native method DLL
/home/user/cobol/igyv6r4/bin/cob2 -bdll -o ./libapp1.so COBPROD_java_native_app1.o
COBPROD_java_native_WS_app1.o addlist_java_native_app1.o Java.CobProd.printResult.o /usr/lpp/
java/IBM/J8.0/bin/j9vm/libjvm.x /home/user/cobol/igyv6r4/lib/igzcnj2.x
# Command return code 0

# Building java program: ./cobol.java
/usr/lpp/java/IBM/J8.0/bin/javac -d /home/user/java/demo -cp /home/user/java/demo ./cobol.java
# Command return code 0

# Building java program: ./progs.java
/usr/lpp/java/IBM/J8.0/bin/javac -d /home/user/java/demo -cp /home/user/java/demo ./progs.java
# Command return code 0

# Building java program: ./COBPRODWS.java
/usr/lpp/java/IBM/J8.0/bin/javac -d /home/user/java/demo -cp /home/user/java/demo ./
COBPRODWS.java
# Command return code 0

# Building java program: ./strg.java
/usr/lpp/java/IBM/J8.0/bin/javac -d /home/user/java/demo -cp /home/user/java/demo ./strg.java
# Command return code 0
```

Related references

[JAVAIOP](#)

[“cjbuild command reference](#)

[” on page 321](#)

[“Example: COBPROD application - building and running” on page 309](#)

[JAVA-CALLABLE \(Enterprise COBOL for z/OS Language Reference\)](#)

[JAVA-SHAREABLE \(Enterprise COBOL for z/OS Language Reference\)](#)

Step 3: Linking COBOL programs

This topic describes how to link user COBOL programs in non-OO COBOL applications that interoperate with Java.

For COBOL programs that make static calls to Java, these programs need to be linked against the side deck for the DLL produced by the cjbuild utility in [“Step 2: Running the cjbuild utility to build a DLL of Java stub programs” on page 304](#).

For the COBPROD application, this side deck is called `libapp1.x` and resides in the z/OS UNIX directory from which the cjbuild utility was invoked.

Note: If the `-d` option of cjbuild indicates that the output DLL is to be written to a data set, then the side deck should also be available in a data set with the same name as the output data set, except that the last name part is replaced with EXP. For example, if the DLL was written to data set `USER.COBOL.LOAD`, then the side deck file would be available in a data set called `USER.COBOL.EXP`. cjbuild will always guarantee that a copy of the side deck is available in the z/OS UNIX directory from which the cjbuild utility is run.

Use the following commands to link the COBOL programs `cobprod` and `addlist`, sending the resulting program objects to MVS data set `USER.COBOL.LOAD`. Run the commands from the directory of stub files:

```
cob2 cobprod.o -o "//COBOL.LOAD(cobprod)"
cob2 addlist.o -o "//COBOL.LOAD(addlist)" ./libapp1.x
```

Related references

[“Example: COBPROD application - building and running” on page 309](#)

Step 4: Building Java programs

This topic briefly discusses the building of Java files that are part of a non-OO COBOL application that interoperates with Java.

The Java files in the application should be built as they normally would be. However, if a Java class interacts with COBOL using this framework by either calling COBOL native methods or accessing COBOL `WORKING-STORAGE` directly, it is necessary to import the Java package that contains the COBOL classes generated by the cjbuild utility. This is the package name specified via the `-p` option of cjbuild and which defaults to `enterprise.COBOL`.

For example, this is the import statement for the default interoperability package name that will be added to the top of a Java file:

```
import enterprise.COBOL.*;
```

Troubleshooting

You can use the following methods to troubleshoot issues when building and running non-OO COBOL applications that interoperate with Java.

1. Make sure that `cob2` and `cjbuild` are in your `PATH` environment variable.

2. Make sure that JAVA_HOME is set in your environment and points to an appropriate Java JDK. For example, when running `build.sh` with no arguments to build a pure AMODE 31 application, try setting JAVA_HOME as follows:

```
export JAVA_HOME=/usr/lpp/java/IBM/J8.0
```

or:

```
export JAVA_HOME=/usr/lpp/java/IBM/J11.0
```

When running `build.sh 3164` or the makefile to build a mixed AMODE 31 COBOL and AMODE 64 Java application, try setting JAVA_HOME as follows:

```
export JAVA_HOME=/usr/lpp/java/IBM/J8.0_64
```

or:

```
export JAVA_HOME=/usr/lpp/java/IBM/J11.0_64
```

3. If you encounter linking errors (see below) during execution of the `cjbuild` utility, try setting the following environment variable:

```
export _C89_LSYSLIB=CEE.SCEELKED
```

If you have your runtime datasets installed under a different HLQ, use the corresponding SCEELKED for those dataset instead.

Below are the link error messages:

```
# Linking native method DLL
/home/userid/cobol/igyA6r5/bin/cob2 -bdll -o //COBOL.LOAD(libapp1) COBPROD_java_
native_app1.o COBPROD_java_native_WS_app1.o addlist_java_native_app1.o Java.CobP
rod.printResult.o /home/userid/jtoc/demo2/java_new5/J8.0_64/bin/j9vm/libjvm.x /h
ome/userid/cobol/igyv6r5/lib/igzcnj2.x
# Command return code 3:
IEW2456E 9207 SYMBOL IGZXLERR UNRESOLVED. MEMBER COULD NOT BE INCLUDED FROM
THE DESIGNATED CALL LIBRARY.
IEW2456E 9207 SYMBOL IGZXDER UNRESOLVED. MEMBER COULD NOT BE INCLUDED FROM THE
DESIGNATED CALL LIBRARY.
IEW2456E 9207 SYMBOL IGZXDDE UNRESOLVED. MEMBER COULD NOT BE INCLUDED FROM THE
DESIGNATED CALL LIBRARY.
IEW2638S 5384 AN EXECUTABLE VERSION OF MODULE *NULL* EXISTS AND CANNOT BE
REPLACED BY THE NON-EXECUTABLE MODULE JUST CREATED.
FSUM3065 The LINKEDIT step ended with return code 12.
```

Example: COBPROD application - building and running

The section describes how to build/run the COBPROD application from z/OS UNIX. A description of how to build and run the application is provided in the next section, "Sample JCL for building and running the COBPROD application". The COBPROD application is a contrived COBOL/Java interoperable application that is used for demonstration purposes throughout this section. A version of the COBPROD application is provided as a set of files available in the `demo/cobol_java_interop_sample` subdirectory of the COBOL install directory in the z/OS UNIX file system.

The COBPROD application consists of two COBOL programs, COBPROD and "addlist", and a Java class called CobProd. The application starts on the Java side, calls the COBPROD COBOL program to initialize a table of product information in WORKING-STORAGE, then accesses that information directly from Java upon return. The Java side then passes a list of prices to the "addlist" COBOL program which sums the list and displays the total price to the terminal by making a call back to a static Java method in the CobProd class called printResult.

Note: Passing large arrays of `java.math.BigDecimal` objects or `java.lang.String` objects to COBOL from Java can be expensive due to data type conversions and thus may not be appropriate when performance is critical. Passing a small array of `java.math.BigDecimal` objects is demonstrated here

to show the convenience of such an operation when it is used prudently or when performance is less of a factor.

Three interoperability features are demonstrated in the application:

- Java calling COBOL
- Java accessing COBOL WORKING-STORAGE directly
- COBOL calling a static Java method

The demo contains the following user source files:

cobprod.cbl

COBOL program that initializes WORKING-STORAGE with a table of product information that can be accessed from Java.

addlist.cbl

COBOL program that adds a list of prices together and dumps the resulting value to the terminal by making a call back to Java.

CobProd.java

Defines the Cobprod Java class that is the Java-based demo driver.

The following script files for building and running the COBPROD application are also provided:

build.sh

A shell script for building the COBPROD application. Instructions for building using the script are contained in the script itself.

Invoke `build.sh` at the z/OS UNIX command line in two different ways:

- `build.sh` (builds an AMODE 31 COBOL, AMODE 31 Java app)
- `build.sh 3164` (builds an AMODE 31 COBOL, AMODE 64 Java app)

makefile

As an alternative to the `build.sh` script, a makefile is provided to demonstrate how a makefile could be used to build an interoperable application for those that prefer makefiles. For the makefile build, a mixed AMODE application (AMODE 31 COBOL, AMODE 64 Java) is always built.

Initiate[®] a makefile-based build by issuing the following command at the z/OS UNIX command line in a directory that contains the makefile:

```
make
```

run.sh

A shell script for running the interoperable application. Works for applications built using either `build.sh` or `makefile`. See the script for details about how to run it.

Other miscellaneous files are provided:

methods

The input file of COBOL and/or Java methods that have associated stub files to be compiled by the `cjbuild` utility into a DLL.

README

Contains detailed information about building and running the COBPROD application.

cobprod.cbl

```
* Initialize a working-storage item with some
* product information to be shared with Java
identification division.
program-id. cobprod.
data division.
working-storage section.
* Product info table, shared with Java
01 prod-list-size pic s9(9) comp-5 value 5.
01 prod-info-data.
03 filler.
05 filler pic x(20) value 'chair'.
```

```

    05 filler pic s9(9)v9(2) value 29.99.
03 filler.
    05 filler pic x(20) value 'table'.
    05 filler pic s9(9)v9(2) value 45.98.
03 filler.
    05 filler pic x(20) value 'bed'.
    05 filler pic s9(9)v9(2) value 149.45.
03 filler.
    05 filler pic x(20) value 'blanket'.
    05 filler pic s9(9)v9(2) value 19.99.
03 filler.
    05 filler pic x(20) value 'sofa'.
    05 filler pic s9(9)v9(2) value 239.99.
>>java-shareable on
01 prod-info redefines prod-info-data.
    03 prod-list occurs 5 times.
        05 prod-name pic x(20).
        05 prod-price pic s9(9)v9(2).
>>java-shareable off
linkage section.
01 prod-list-size-out pic s9(9) comp-5.
>>java-callable
procedure division returning prod-list-size-out.
MainProgram.
* Populate table
    move prod-list-size to prod-list-size-out
    goback.
end program cobprod.

```

addlist.cbl

```

cbl pgmname(longmixed)
* Receives a list of three packed decimals and calculates
* and returns the sum
identification division.
program-id. 'addlist'.
data division.
working-storage section.
local-storage section.
01 idx pic s9(9) comp-5.
linkage section.
01 pricelist.
    03 price pic s9(9)v9(2) comp-3 occurs 3 times.
01 listsize pic s9(9) comp-5.
01 result pic s9(9)v9(2) comp-3.
>>java-callable
procedure division using by reference pricelist
    by value listsize
    returning result.
MainProgram.
* Compute sum of list
    display '<<COBOL: addlist: entered>>'

    compute result = 0

    perform varying idx from 1 by 1
        until idx > listsize
            compute result = result + price(idx)
    end-perform

    call 'Java.CobProd.printResult' using result

    display '<<COBOL: addlist: exited>>'
    goback.
end program 'addlist'.

```

CobProd.java

```

import enterprise.COBOL.*;
import java.math.*;

public class CobProd
{
    public static void main(String[] args)
    {
        BigDecimal[] prices = new BigDecimal[3];

        System.out.println("<<Java: CobProd.main: entered>>");
    }
}

```

```

System.out.println();

// Now access a product table from COBOL
System.out.println("");
System.out.println("Java: Access COBOL product info from COBPROD");

int tableSize = enterprise.COBOL.progs.COBPROD();
System.out.println("Java: prod table size = " + tableSize);
for (int i = 0; i < tableSize; i++)
{
    System.out.print("Java: Product name = " +
        enterprise.COBOL.strg.COBPROD.PROD_INFO.PROD_LIST[i].PROD_NAME.get());
    System.out.print(" Price = $");

System.out.println(enterprise.COBOL.strg.COBPROD.PROD_INFO.PROD_LIST[i].PROD_PRICE.get());
    if (i < 3)
    {
        prices[i] = enterprise.COBOL.strg.COBPROD.PROD_INFO.PROD_LIST[i].PROD_PRICE.get();
    }
}
System.out.println();

// Call out to COBOL to sum the list
System.out.println("Java: call COBOL addlist to add prices");
System.out.println();
BigDecimal result = enterprise.COBOL.progs.addlist(prices, 3);

System.out.println();
System.out.println("<<Java: CobProd.main: exited>>");
}

public static void printResult(BigDecimal bd)
{
    System.out.println("Java (called from COBOL): printResult: " +
        "total price is: $" + bd);
}
}

```

methods

```

COBPROD
addlist
Java.CobProd.printResult

```

build.sh

```

#!/bin/sh
#####
# Build script for the COBOL/Java interoperability demo COBPROD
#####
#
# Invocations:
# -----
#
# build.sh          - build pure 31-bit interoperable application
# build.sh 3164     - build mixed 31/64-bit interoperable application
#
# NOTE:
# -----
#
# 1) Make sure cjbuild and cob2 are in your PATH before running this script
#
# 2) Make sure your STEPLIB includes an Enterprise COBOL V6.4 or later
#    compiler and a runtime that is sufficient to run COBOL programs
#    compiled with the Enterprise COBOL V6.4 or later compiler
#
# 3) You must have an existing data set to receive COBOL program objects
#    and DLLs and it must be called <USER>.COBOL.LOAD, where <USER> is
#    your high-level qualifier
#
#####

OUT_DIR=./out
CLASSOUT_DIR=./class

# Process input arguments
if [[ $# > 0 ]]; then
    if [[ $1 = "3164" ]]; then
        JAVAMODE="JAVA64"
        CJMODE="MIX_31_64"
    fi
fi

```

```

else
    echo "Usage: build.sh [3164]"
    exit 1
fi
else
    JAVAMODE="NOJAVA64"
    CJMODE="PURE31"
fi

# Make the output directory
rm -fr ${OUT_DIR} ${CLASSOUT_DIR}
mkdir ${OUT_DIR} ${CLASSOUT_DIR}

# STEP 1: Compile user COBOL programs, specifying JAVAIOP option
cob2 -c cobprod.cbl "-qjavaiop(OUTPATH('${OUT_DIR}'),${JAVAMODE})"
cob2 -c addlist.cbl "-qjavaiop(OUTPATH('${OUT_DIR}'),${JAVAMODE})"

# STEP 2: Build stub file DLL using cjbuild
cjbuild -v -j $CLASSOUT_DIR -p enterprise.COBOL \
-m ${CJMODE} \
-c $OUT_DIR \
-d "//COBOL.LOAD" \
methods app1

# STEP 3: Link user programs (link with Java call stub, if necessary)
cob2 cobprod.o -o "//COBOL.LOAD(cobprod)" libapp1.x
cob2 addlist.o -o "//COBOL.LOAD(addlist)" libapp1.x

# STEP 4: Build Java parts of the application
export CLASSPATH=$CLASSOUT_DIR:$CLASSPATH

${JAVA_HOME}/bin/javac -d $CLASSOUT_DIR CobProd.java

```

run.sh

```

#!/bin/sh
#####
# Run script for the COBOL/Java interoperability demo COBPROD
#####
#
# NOTE:
# ----
#
# 1) Make sure your STEPLIB includes an Enterprise COBOL runtime
# that is sufficient to run programs compiled with the Enterprise
# COBOL V6.4 or later compiler.
#
# The dataset <USER>.COBOL.LOAD that received the program objects
# and DLL for your interoperable application during the build
# step must also be included in your STEPLIB
#
# 2) Make sure JAVA_HOME points to your 64-bit Java JDK
#
# e.g., JAVA_HOME=/usr/lpp/java/IBM/J8.0_64 or
#        JAVA_HOME=/usr/lpp/java/IBM/J11.0_64
#
# 3) If you modify this example so that cjbuild stores the DLL it
# produces to a z/OS UNIX directory, then you need to modify
# the last line of this script to the following:
#
#        ${JAVA_HOME}/bin/java -Djava.library.path=<dll_out_dir> CobProd
#
#####
export CLASSPATH=../class:${CLASSPATH}
export IBM_JAVA_OPTIONS="-XX:+Enable3164Interoperability"
export LIBPATH=${JAVA_HOME}/lib/s390x/j9vm:${JAVA_HOME}/lib/s390x/${JAVA_HOME}/lib/j9vm:${
${JAVA_HOME}/lib:${LIBPATH}
export _CEE_RUNOPTS="XPLINK(ON),POSIX(ON)"

${JAVA_HOME}/bin/java CobProd

```

makefile

```

#####
# Makefile for the COBOL/Java interoperability demo COBPROD (Mixed 31/64 mode)
#####
#
# NOTE:

```

```

# -----
#
# 1) Make sure cjbuild and cob2 are in your PATH before using this makefile
#
# 2) Make sure your STEPLIB is set to include an Enterprise V6.4 or later
#    compiler and runtime
#
# 3) Make sure JAVA_HOME points to your 64-bit Java JDK
#
#    e.g., JAVA_HOME=/usr/lpp/java/IBM/J8.0_64 or
#          JAVA_HOME=/usr/lpp/java/IBM/J11.0_64
#
# 4) You must have an existing data set to receive COBOL program objects
#    and DLLs and it must be called <USER>.COBOL.LOAD, where <USER> is
#    your high-level qualifier
#
#####

MYSTEPLIB=$(USER).COBOL.LOAD:$(STEPLIB)
OUT_DIR=./out
CLASSOUT_DIR=./class
MYCLASSPATH=$(CLASSOUT_DIR):$(CLASSPATH)

all: setup libapp1 cobprod addlist CobProd.class

setup:
    @rm -fr out class
    @mkdir -m 0755 out class

libapp1: cobprod.o addlist.o methods
    STEPLIB=$(MYSTEPLIB) cjbuild -v -j $(CLASSOUT_DIR) -p enterprise.COBOL -m MIX_31_64 -c
$(OUT_DIR) -d "//COBOL.LOAD" methods app1

cobprod: cobprod.o
    STEPLIB=$(MYSTEPLIB) cob2 cobprod.o -o "//COBOL.LOAD(cobprod)" libapp1.x

addlist: addlist.o libapp1
    STEPLIB=$(MYSTEPLIB) cob2 addlist.o -o "//COBOL.LOAD(addlist)" libapp1.x

cobprod.o: cobprod.cbl
    STEPLIB=$(MYSTEPLIB) cob2 -c cobprod.cbl "-qjvaiop(OUTPATH('$$(OUT_DIR)'),JAVA64)"

addlist.o: addlist.cbl
    STEPLIB=$(MYSTEPLIB) cob2 -c addlist.cbl "-qjvaiop(OUTPATH('$$(OUT_DIR)'),JAVA64)"

CobProd.class: CobProd.java
    CLASSPATH=$(MYCLASSPATH) $(JAVA_HOME)/bin/javac -d $(CLASSOUT_DIR) CobProd.java

clean:
    rm -fr *java_native* *WS.java *WS_base.java *.o *.class *.sig CEEDUMP.* Java.* *.x
    *.so Snap.* javacore.* jitdump.* *.lst
    rm -fr cobol.java progs.java strg.java
    rm -fr LIBAPP1* libapp1*
    rm -fr out
    rm -fr class

```

Related references

[Chapter 17, “Compiling, linking, and running non-OO COBOL applications that interoperate with Java,” on page 301](#)

[“Sample JCL for building and running the COBPROD application” on page 314](#)

Sample JCL for building and running the COBPROD application

This section contains sample JCL for building and running the COBPROD application. Previous sections describe the COBPROD demo application and how to build and run it using z/OS UNIX shell scripts and makefiles. Some users, however, may prefer to build and run non-OO COBOL/Java interoperable applications using JCL. The provided JCL is specifically for the COBPROD demo application but should be easily adaptable to any non-OO COBOL/Java interoperable application.

Much of the JCL is typical of any JCL to build and link COBOL applications. The main difference is that the build process contains a step for building the DLL for the interoperable application using the cjbuild tool. While cjbuild is a native z/OS UNIX program, the sample shows how it can be easily run from JCL using the BPXBATCH utility.

Things to consider before using the sample JCL to build and run the COBPROD application.

1. The JCL itself contains comprehensive comments at the beginning describing the changes that need to be made to the JCL to make it run successfully in your environment.
2. Follow the instructions in the JCL carefully, and ensure to meet the following important conditions:
 - All instances of `userid` in z/OS UNIX paths in the JCL are replaced by the user id of the account from which the JCL will be run (the user id on z/OS UNIX is typically just the lowercase version of your z/OS user id).
 - The various JCL path variables defined at the top of the JCL file (for example, `WORKDIR`, `ZFSCBDIR`, `OUTDIR`) are set appropriately for your environment.
 - A z/OS UNIX work directory location on your system is identified that can be used for the build job, where various listings and other artifact files produced by `cjbuild` will be created. The work directory and all its subdirectories will be created automatically by the sample JCL. There is no need to create them yourself first. You only need to define the work directory location via the `WORKDIR` JCL variable.
 - The `LNGPRFX` and `LIBPRFX` JCL variables are set appropriately to indicate your compiler and runtime levels respectively.
3. Before the JCL is run for the first time, you must ensure that the two COBOL user source files in the COBPROD application are copied to a data set with name `USERID.COBOL.COBOL` where `USERID` is the user id of the account from which the JCL will be run. If a differently named data set is desired, the JCL must be modified accordingly. The two user source files are:
 - `cobprod.cbl`
 - `addlist.cbl`

and they exist in the COBPROD application demo directory:

```
demo/cobol_java_interop_sample
```

which is located in the COBOL install directory in the z/OS UNIX file system, which typically has a path such as `/usr/lpp/cobol/igyv6r4`. Copying these files to the data set can be done using the following z/OS UNIX commands, assuming you run the commands from the `cobol_java_interop_sample` directory:

```
cp cobprod.cbl "'/USERID.COBOL.COBOL(COBPROD)'"
cp addlist.cbl "'/USERID.COBOL.COBOL(ADDLIST)'"
```

Note: In a production build environment, it is most likely that your user COBOL source files are already located in a data set. This extra copying is only needed for building these demo files which happen to reside in the COBOL install directory in the z/OS UNIX file system.

4. The provided JCL makes use of the `BPXATCH` utility to run z/OS UNIX commands from JCL. `BXPATCH` causes a z/OS UNIX shell to be started, which in turn causes the `.profile` file in your z/OS UNIX home directory to be “sourced” (that is, loaded into your environment and run). It is common in some z/OS UNIX environments for other non-default UNIX shells to be started at the end of the `.profile` file using the command:

```
exec <alternative-shell-program>
```

Note: If a non-default shell is not executed from your `.profile`, you may safely ignore the rest of this point.

A non-default shell is simply a shell program that is not the default shell set for your system, which is usually the `sh` UNIX command. This could cause a problem with `BXPATCH`. A reliable way to prevent such a problem is to ensure that your custom shell program is only run if you are running an interactive shell session. This can be achieved by guarding the `exec` command in your `.profile` file as follows:

```
if [[ $- == *i* ]]; then
```

```
    exec <alternative-shell-program>
fi
```

If this method does not work for some reason, the sample JCL sets a shell variable called `NO_CUSTOM_SHELL` that can be used by the user in their `.profile` to prevent the custom shell from running. The shell code would look like the following:

```
if [[ -z $NO_CUSTOM_SHELL ]]; then
    exec <alternative-shell-program>
fi
```

5. It is important to set the `JAVA_HOME` environment variable in the `.profile` file in the z/OS UNIX home directory of the user that is running the JCL. The line to add to the `.profile` file to ensure the 31-bit COBPROD application in the sample JCL runs successfully is:

```
export JAVA_HOME=/usr/lpp/java/IBM/J8.0
```

Note: Failure to do this could result in STEP 2 of the JCL failing to launch the `cjbuild` utility, which is a Java program.

Output from a successful run of the application

When the application runs successfully from the JCL, the job log should contain the following output:

```
<<Java: CobProd.main: entered>>

Java: Access COBOL product info from COBPROD
Java: prod table size = 5
Java: Product name = chair           Price = $29.99
Java: Product name = table          Price = $45.98
Java: Product name = bed            Price = $149.45
Java: Product name = blanket        Price = $19.99
Java: Product name = sofa           Price = $239.99

Java: call COBOL addlist to add prices

<<COBOL: addlist: entered>>
Java (called from COBOL): printResult: total price is: $225.42
<<COBOL: addlist: exited>>

<<Java: CobProd.main: exited>>
```

COBPROD JCL sample file

This sample JCL is available in member `IGYCJDMJ` of the Enterprise COBOL compiler's `SIGYSAMP` data set. It is also available as z/OS UNIX file `cobprod.jcl` in subdirectory `demo/cobol_java_interop_sample` of the COBOL install directory in the z/OS UNIX file system.

```
//USERID2 JOB 'JAVA/COBOL interop',
// MSGLEVEL=(1,1),MSGCLASS=S,CLASS=C,NOTIFY=&SYSUID,
// REGION=0M
//*****
//*                IBM Enterprise COBOL for z/OS                */
//*****
//*****
//*                LICENSED MATERIALS -- PROPERTY OF IBM.                *
//*                *
//*                5655-EC6                *
//*                COPYRIGHT IBM CORP. 2023                *
//*                *
//*                US GOVERNMENT USERS RESTRICTED RIGHTS - USE, DUPLICATION OR *
//*                DISCLOSURE RESTRICTED BY GSA ADP SCHEDULE CONTRACT WITH IBM *
//*                CORP.                *
//*                *
//*****
//*****
//* Sample JCL for building and running the COBPROD application
//* demonstrating non-00 COBOL/Java interoperability.
//*
```



```

//*****
//*
//* Prerequisites:
//*
//* 1) COBPROD demo source files cobprod.cbl and addlist.cbl from
//*    the COBOL install directory in the z/OS UNIX file system.
//*
//*    demo/cobol_java_interop_sample
//*
//*    must be copied to a PDS or PDSE data set with name
//*    userid.COBOL.COBOL.
//*
//*    Alternatively, the JCL could be modified to read the
//*    source files directly from the COBOL install directory in
//*    the z/OS UNIX file system in steps COBOL1 and COBOL2 below.
//*
//* 2) A PDSE data set named userid.COBOL.OBJECT must exist
//*    to receive object program files generated by the COBOL
//*    compiler.
//*
//* 3) A PDSE data set userid.COBOL.LOAD must exist to receive
//*    program object files. These program objects come from
//*    linking object files from user programs and also from
//*    linking the object files associated with stub programs
//*    into a DLL.
//*
//* 4) You must ensure the following JCL variables are set below,
//*    which control various build parameters. This includes the
//*    z/OS UNIX directory into which stub files will be generated
//*    by the compiler and the z/OS UNIX work directory to be
//*    used by the cjbuild utility. The cjbuild utility uses this
//*    directory to build the DLL needed for the interoperable
//*    application.
//*
//*    For this sample, we assume the z/OS UNIX directory
//*    structure to be used for building the application is as
//*    follows:
//*
//*    /home/userid/jtoc/cobprod
//*      - main work directory
//*    /home/userid/jtoc/cobprod/out
//*      - stub file output directory (compiler places stub
//*        files here)
//*    /home/userid/jtoc/cobprod/class
//*      - directory where cjbuild places any generated .class
//*        files
//*
//*    The following variables must be set:
//*
//*    JAVAHOME
//*    ZFSCBDIR
//*    OUTDIR
//*    INTMODE
//*    WORKDIR
//*    CLASSOUT
//*    JVSRCOUT
//*    PKGNAME
//*    DLLNAME
//*
//*    See below for detailed descriptions of how to set these
//*    variables. Example values are provided.
//*
//*    NOTE: z/OS UNIX directories for stub files and the work
//*    directory for cjbuild do not have to exist. The
//*    JCL below creates the directories if they do not
//*    exist prior to running this job (see STEP 0).
//*
//* 5) The contents of the "methods" file, used as input to the
//*    cjbuild utility, is specified in the instream data
//*    set STDPARM in STEP 2 that contains the UNIX commands
//*    that are required to run the cjbuild utility. To change
//*    the list of methods whose associated stub files should
//*    be included in the DLL, change the "echo" commands that
//*    are used to append each method name to the methods file.
//*
//*****
// EXPORT SYMLIST=*
//*****
//* Set compiler and run time library levels below
//*****
// SET LIBPRFX='CEE'
// SET LNGPRFX='COBOL.COMPILER'

```

```

//*****
//* JAVAHOME: Set this to your Java home directory. This will be used
//* to set the JAVA_HOME environment variable needed by
//* Java operations
//*****
// SET JAVAHOME='/usr/lpp/java/IBM/J8.0'
//*****
//* ZFSCBDIR: This should point to the location of your COBOL install
//* directory in the z/OS UNIX file system
//* (e.g., /usr/lpp/cobol/igyv6r4)
//*
//* The bin directory of this directory is where the cob2
//* and cjbuild utilities reside. They will be needed
//* when building the interoperable application
//*****
// SET ZFSCBDIR='/usr/lpp/cobol/igyv6r4'
//*****
//* OUTDIR: Set this to the z/OS UNIX directory that will receive
//* the stub files and other artifact files generated by
//* the COBOL compiler compiling programs that contain
//* the JAVA-CALLABLE and JAVA-SHAREABLE directives or
//* programs that make calls to Java using a CALL statement
//* of the form:
//*
//* CALL 'Java.class-name.static-method-name' ...
//*
//* The cjbuild utility (see STEP 2) will get the files from
//* this directory (see --coboldir option) so the directory
//* specified for OUTDIR should be used both in the OUTPATH
//* suboption of JAVAIOPIOP when compiling the user COBOL
//* program as well as the --coboldir option of cjbuild.
//*****
// SET OUTDIR='/home/userid/jtoc/cobprod/out'
//*****
//* INTMODE: Set this to the interoperability mode to be used
//* by your application (needed by the cjbuild utility)
//*
//* PURE31 - AMODE 31 COBOL, AMODE 31 Java
//* PURE64 - AMODE 64 COBOL, AMODE 64 Java
//* MIX_31_64 - AMODE 31 COBOL, AMODE 64 Java
//*
//* NOTE: if you specify MIX_31_64, you must add the
//* JAVA64 suboption to the JAVAIOPIOP option of
//* all related user COBOL programs.
//*****
// SET INTMODE=PURE31
//*****
//* WORKDIR: Set this to the z/OS UNIX directory to be used as a work
//* directory for building the interoperable application.
//* cjbuild will be run from this directory and basic
//* artifact files it generates (object files, listing,
//* etc.) go to this directory automatically.
//*
//* This is where the "methods" file will be located for
//* this sample.
//*****
// SET WORKDIR='/home/userid/jtoc/cobprod'
//*****
//* CLASSOUT: Set this to the directory you want any .class files
//* generated by cjbuild to go. These are the .class
//* files for Java accessor classes for JAVA-SHAREABLE
//* items, and also .class files for any Java files
//* generated by cjbuild, such as cobol.java, progs.java,
//* and strg.java.
//*****
// SET CLASSOUT='/home/userid/jtoc/cobprod/class'
//*****
//* JVSRCOUT: Set this to the directory you want any .java files
//* generated by cjbuild to go (cobol.java, progs.java,
//* strg.java).
//*****
// SET JVSRCOUT='/home/userid/jtoc/cobprod'
//*****
//* PKGNAME: Set this to the name of the Java package that you
//* want the 'progs' and 'strg' class to be a part of.
//* The default is enterprise.COBOL. This affects how
//* COBOL programs are invoked from Java. e.g., if the
//* package name is enterprise.COBOL., then you would
//* invoke a COBOL program called PROG1 as follows:
//*
//* enterprise.COBOL.progs.PROG1();
//*****

```

```

//      SET PKGNAME='enterprise.COBOL'
//*****
//* DLLNAME: The name of the DLL that cjbuild creates and which
//*           is needed at runtime when interoperable activities
//*           occur in the application.
//*
//*           If the DLL is targeted for a data set (which is
//*           the usual way, see --dlloutdir option of cjbuild)
//*           then the DLL will be stored as a member of the
//*           specified data set, and is called LIB<DLLNAME>.
//*
//*           If the DLL is targeted to the z/OS UNIX file system,
//*           it will be called lib<dllname>.so.
//*****
//      SET DLLNAME=APP1
//*****
//PROCLIB JCLLIB ORDER=(&LNGPRFX..SIGYPROC)
//*****
//*****
//* STEP 0: Ensure required z/OS UNIX directories exist
//*****
//*****
//CRDIRS EXEC PGM=BPXBATCH
//STDPARM DD *,SYMBOLS=EXECSYS
SH mkdir -p &OUTDIR;
mkdir -p &WORKDIR;
mkdir -p &CLASSOUT;
mkdir -p &JVSRCOUT
/*
//STDOUT DD SYSOUT=*
//STDERR DD SYSOUT=*
//STDENV DD *
NO_CUSTOM_SHELL=1
/*
//*****
//*****
//* STEP 1: Compile user COBOL programs, specifying JAVAIOPT option
//*****
//*****
// SET INFILE='COBPROD'
//*****
// IF (CRDIRS.RUN AND (CRDIRS.RC EQ 0)) THEN
//COBOL1 EXEC IGYWC,
// LNGPRFX=&LNGPRFX,
// LIBPRFX=&LIBPRFX,
// PARM.COBOLE='OPTFILE'
//SYSLIB DD DSN=&SYSUID..COBOL.COBOLE,DISP=SHR
//SYSIN DD DSN=&SYSUID..COBOL.COBOLE(&INFILE),DISP=SHR
//SYSLIN DD DSN=&SYSUID..COBOL.OBJECT(&INFILE),DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSOPTF DD *,SYMBOLS=EXECSYS
//JAVAIOPT(OUTPATH('&OUTDIR'))
/*
//SYSOUT DD SYSOUT=*
// ENDIF
//*****
// SET INFILE='ADDLIST'
//*****
// IF (COBOL1.COBOLE.RUN AND (COBOL1.COBOLE.RC EQ 0)) THEN
//COBOL2 EXEC IGYWC,
// LNGPRFX=&LNGPRFX,
// LIBPRFX=&LIBPRFX,
// PARM.COBOLE='OPTFILE'
//SYSLIB DD DSN=&SYSUID..COBOL.COBOLE,DISP=SHR
//SYSIN DD DSN=&SYSUID..COBOL.COBOLE(&INFILE),DISP=SHR
//SYSLIN DD DSN=&SYSUID..COBOL.OBJECT(&INFILE),DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSOPTF DD *,SYMBOLS=EXECSYS
//JAVAIOPT(OUTPATH('&OUTDIR'))
/*
//SYSOUT DD SYSOUT=*
// ENDIF
//*****
//* STEP 2: Build stub file DLL using cjbuild
//*****
//*****
// IF (COBOL2.COBOLE.RUN AND (COBOL2.COBOLE.RC EQ 0)) THEN
//CJBUILD EXEC PGM=BPXBATCH
//STDPARM DD *,SYMBOLS=EXECSYS
SH export JAVA_HOME=&JAVAHOME;
PATH=./bin:/usr/bin:/usr/local/bin:&ZFSCBDIR/bin;
cd &WORKDIR;

```

```

echo COBPROD > methods;
echo addlist >> methods;
echo Java.CobProd.printResult >> methods;

&ZFSCBDIR/bin/cjbuild
-v
--coboldir &OUTDIR
--dlloutdir "//&SYSUID..COBOL.LOAD"
--pkgname &PKGNAME
--javaclassdir &CLASSOUT
--javasrcdir &JVSRCOUT
methods &DLLNAME
/*
//STEPLIB DD DSN=&LNGPRFX..SIGYCOMP,DISP=SHR
//          DD DSN=&LIBPRFX..SCEERUN,DISP=SHR
//          DD DSN=&LIBPRFX..SCEERUN2,DISP=SHR
//STDOUT DD SYSOUT=*
//STDERR DD SYSOUT=*
//STDENV DD *,SYMBOLS=EXECSYS
NO_CUSTOM_SHELL=1
_C89_LSYSLIB=&LIBPRFX..SCEELKED
/*
// ENDIF
//*****
//*****
//* STEP 3: Bind (Link) user programs. User programs may call out to
//* programs in the DLL produced by cjbuild in STEP 2
//* thus should be linked with the side deck produced in
//* that step
//*****
//*****
// SET INFILE='COBPROD'
//*****
// IF (CJBUILD.RUN AND (CJBUILD.RC EQ 0)) THEN
//LKED1 EXEC PGM=IEWL,
// PARM='RENT,LIST,LET,DYNAM(DLL),CASE(MIXED)'
//SYSLIB DD DSN=&LIBPRFX..SCEELKED,DISP=SHR
//          DD DSN=&LIBPRFX..SCEELKEX,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTEM DD SYSOUT=*
//SYSLMOD DD DSN=&SYSUID..COBOL.LOAD(&INFILE),DISP=SHR
//SYSDEFSD DD DUMMY
//OBJMOD DD DSN=&SYSUID..COBOL.OBJECT(&INFILE),DISP=SHR
//SIDEDK DD DSN=&SYSUID..COBOL.EXP(LIB&DLLNAME),DISP=SHR
//SYSLIN DD *
// INCLUDE OBJMOD
// INCLUDE SIDEDK
/*
// ENDIF
//*****
// SET INFILE='ADDLIST'
//*****
// IF (LKED1.RUN AND (LKED1.RC EQ 0)) THEN
//LKED2 EXEC PGM=IEWL,
// PARM='RENT,LIST,LET,DYNAM(DLL),CASE(MIXED)'
//SYSLIB DD DSN=&LIBPRFX..SCEELKED,DISP=SHR
//          DD DSN=&LIBPRFX..SCEELKEX,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTEM DD SYSOUT=*
//SYSLMOD DD DSN=&SYSUID..COBOL.LOAD(&INFILE),DISP=SHR
//SYSDEFSD DD DUMMY
//OBJMOD DD DSN=&SYSUID..COBOL.OBJECT(&INFILE),DISP=SHR
//SIDEDK DD DSN=&SYSUID..COBOL.EXP(LIB&DLLNAME),DISP=SHR
//SYSLIN DD *
// INCLUDE OBJMOD
// INCLUDE SIDEDK
/*
// ENDIF
//*****
//*****
//* STEP 4: Builds Java application
//*****
//*****
// IF (LKED2.RUN AND (LKED2.RC EQ 0)) THEN
//BLDJAVA EXEC PGM=BPXBATSL
//STDPARM DD *,SYMBOLS=EXECSYS
PGM &JAVAHOME/bin/javac
-d &CLASSOUT
&ZFSCBDIR/demo/cobol_java_interop_sample/CobProd.java
/*
//STDOUT DD SYSOUT=*
//STDERR DD SYSOUT=*

```

```

//STDENV DD *,SYMBOLS=EXECSYS
JAVA_HOME=&JAVAHOME
CLASSPATH=.:&CLASSOUT
PATH=./bin:/usr/bin:/usr/local/bin:&ZFSCBDIR/bin
/*
// ENDIF
//*****
//*****
//* STEP 5: Run COBOL/Java interoperable application
//*****
//*****
// IF (BLDJAVA.RUN AND (BLDJAVA.RC EQ 0)) THEN
//GO EXEC PGM=BPXBATSL
//STDPARM DD *,SYMBOLS=EXECSYS
PGM &JAVAHOME/bin/java CobProd
/*
//STEPLIB DD DSN=&SYSUID..COBOL.LOAD,DISP=SHR
// DD DSN=&LNGPRFX..SIGYCOMP,DISP=SHR
// DD DSN=&LIBPRFX..SCEERUN,DISP=SHR
// DD DSN=&LIBPRFX..SCEERUN2,DISP=SHR
//STDOUT DD SYSOUT=*
//STDERR DD SYSOUT=*
//STDENV DD *,SYMBOLS=EXECSYS
JAVA_HOME=&JAVAHOME
PATH=./bin:/usr/bin:/usr/local/bin:&ZFSCBDIR/bin
_C89_LSYSLIB=&LIBPRFX..SCEELKED
CLASSPATH=.:&CLASSOUT
_CEE_RUNOPTS=XPLINK(ON),POSIX(ON)
/*
// ENDIF

```

Related references

[Chapter 17, “Compiling, linking, and running non-OO COBOL applications that interoperate with Java,” on page 301](#)

AMODE considerations for running non-OO COBOL/Java interoperable applications

This topic discusses AMODE-related considerations for non-OO COBOL/Java interoperable applications.

This topic discusses AMODE-related considerations for non-OO COBOL/Java interoperable applications.

For mixed AMODE applications (that is, AMODE 31 COBOL and 64-bit Java applications) running on Java 8, set LIBPATH as follows:

```
export LIBPATH=${JAVA_HOME}/lib/s390x/j9vm:${JAVA_HOME}/lib/s390x/:${LIBPATH}
```

For mixed AMODE applications running on Java 11, set LIBPATH as follows:

```
export LIBPATH=${JAVA_HOME}/lib:${JAVA_HOME}/lib/j9vm:${LIBPATH}
```

In both cases, set the following environment variable at run time for mixed AMODE Java interoperable COBOL applications:

```
export IBM_JAVA_OPTIONS="-XX:+Enable3164Interoperability"
```

cjbuild command reference

The cjbuild utility is provided to assist with building non-OO COBOL applications that interoperate with Java. When the COBOL files of such an application are compiled, stub files are produced that facilitate the communication between COBOL and Java. These stub files must be built into a DLL that is available when the application runs.

Note: The cjbuild utility must be run from the z/OS UNIX command line or from JCL using the BPXBATCH utility. However, the DLL that is produced by the utility can be targeted to an MVS data set or a z/OS UNIX directory.

cjbuild command usage and parameter descriptions

Execute the following command to run the cjbuild utility:

```
cjbuild [-hv] [-c <cobol-dir>] [-d <dll-out-dir>] [-j <java-class-dir>] [-s <java-src-dir>] [-m  
<PURE31|PURE64|MIX_31_64>] [-p <package-name>] <methods-file> <app-name>
```

where:

<methods-file>

The methods file is a file that contains a list of COBOL programs that have been compiled with the JAVA-CALLABLE and/or JAVA-SHAREABLE directives and a list of static Java methods that are called from COBOL programs in the application using a CALL statement of the form:

```
CALL 'Java.<java-class-name>.<java-static-method-name>'
```

These programs have associated stub files located in the directory indicated by the `--coboldir` option that will be compiled into a DLL for the application.

Programs compiled with the JAVA-CALLABLE directive will be grouped into a class called 'progs' in the specified package (see `-p` option), and data items that fall under the scope of a JAVA-SHAREABLE directive in any of the listed COBOL programs will be grouped into a class called 'strg' in the specified package.

For example, the contents of the methods file for an application consisting of two Java-callable COBOL programs COBPROG1 and COBPROG2, one of which makes a call to a static Java method 'Java.com.acme.MyClass.myMethod' would look as follows:

```
COBPROG1  
COBPROG2  
Java.com.acme.MyClass.myMethod
```

<app-name>

Used to create the name of the DLL that contains the stub code needed at run time by the interoperable application (both Java and COBOL may need this DLL).

For example, if the app name is `appl`, then the DLL will be called `libapp1.so`, if the argument to the `-d` option is a z/OS UNIX directory, or will be called `LIBAPP1`, if the argument to the `-d` option is an MVS PDS data set.

See the following table for the options in cjbuild command reference:

Options	Descriptions
<code>--coboldir <cobol-dir></code> <code>-c <cobol-dir></code>	Path to COBOL artifact directory. This directory contains files generated by the compiler when compiling COBOL programs that contain the JAVA-CALLABLE and/or JAVA-SHAREABLE directives or CALL statements with a literal call target of the form 'Java.java-class-name.java-static-method-name!'. Default: current directory

Table 44. Options and descriptions (continued)

Options	Descriptions
<pre>--dlloutdir <dll-out-dir> -d <dll-out-dir></pre>	<p>Path where the DLL of stub code is to be placed. For a PDS, use the path of the form:</p> <pre>//<DATASET-NAME></pre> <p>For example:</p> <pre>//COBOL.LOAD or //'USER.COBOL.LOAD'</pre> <p>Default: current directory</p>
<pre>--help -h</pre>	<p>Displays verbose help.</p> <p>Default: N/A</p>
<pre>--javaclassdir <java-class-dir> -j <java-class-dir></pre>	<p>Path to location of output class files for generated Java code. This path should be in your <i>CLASSPATH</i> environment variable.</p> <p>Default: current directory</p>
<pre>--javasrcdir <java-src-dir> -s <java-src-dir></pre>	<p>Path to location of output Java source files.</p> <p>Default: current directory</p>
<pre>--mode <PURE31 PURE64 MIX_31_64> -m <PURE31 PURE64 MIX_31_64></pre>	<p>Java/COBOL interoperability mode.</p> <p>Specify PURE31 for AMODE 31 COBOL/31-bit Java application.</p> <p>Specify PURE64 for AMODE 64 COBOL/64-bit Java application.</p> <p>Specify MIX_31_64 for AMODE 31 COBOL/64-bit Java application.</p> <p>Note: You must build COBOL files with JAVAIOP (JAVA64) option for MIX_31_64 mode applications.</p> <p>Default: PURE31</p>
<pre>--pkgname <package-name> -p <package-name></pre>	<p>Name of package with which these native methods will be associated. Each application should have its own package name.</p> <p>For example:</p> <pre>com.acme.COBOL.myapp1</pre> <p>Three classes will be created as part of this package:</p> <pre>com.acme.COBOL.myapp1.cobol com.acme.COBOL.myapp1.progs com.acme.COBOL.myapp1.strig</pre> <p>Default: enterprise.COBOL</p>

Table 44. Options and descriptions (continued)

Options	Descriptions
--verbose -v	Verbose output. Shows compile/link commands executed. Default: N/A

Related tasks

Chapter 17, “[Compiling, linking, and running non-OO COBOL applications that interoperate with Java](#),” on page 301

“[Example: COBPROD application - building and running](#)” on page 309

Related references

JAVA-CALLABLE (*Enterprise COBOL for z/OS Language Reference*)

JAVA-SHAREABLE (*Enterprise COBOL for z/OS Language Reference*)

CALL statement (*Enterprise COBOL for z/OS Language Reference*)

Chapter 18. Compiling, binding, and running COBOL AMODE 64 applications

You can compile, bind, and run Enterprise COBOL AMODE 64 applications under z/OS environment using JCL, TSO commands, CLISTs, or ISPF panels. The process is almost the same as the corresponding tasks for COBOL AMODE 31 applications. See related tasks for key differences and limitations.

Related tasks

[“Compiling and binding AMODE 64 programs under z/OS” on page 325](#)

[“DLL considerations” on page 326](#)

[“Running COBOL AMODE 64 applications” on page 327](#)

[“Specifying COBOL only runtime options” on page 327](#)

Compiling and binding AMODE 64 programs under z/OS

You can compile Enterprise COBOL AMODE 64 applications under z/OS using job control language (JCL), TSO commands, CLISTs, or ISPF panels. The process is the same as the corresponding tasks for COBOL AMODE 31 applications.

The key differences are as follows:

- Specify the LP(64) compiler option
- Include the Language Environment system library SCEEBND2 in the SYSLIB DD of the Bind step
- Include the Language Environment system library PDSE member SCEELIB(CELQV004) in the SYSLIN DD of the Bind step
- Include the Binder option RENT
- If you specify an explicit ENTRY statement in the Bind step for LP(64) COBOL programs, it must specify ENTRY CELQSTRT. In other words, the entry point for LP(64) programs is not the name in the PROGRAM-ID statement, it is CELQSTRT.

When 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.

For illustrative purpose, the following JCL code fragment shows how to modify an existing compile and bind step JCL to build COBOL AMODE 64 programs:

```
//COMPILE EXEC PGM=IGYCRCTL,REGION=0M,  
//   PARM='LP(64)'                               <- add LP(64) option  
// ...  
//BIND EXEC. PGM=IEWBLINK,REGION=0M,  
//   PARM='RENT,DYNAM(DLL)'                       <- add RENT and DYNAM(DLL) options  
//SYSLIB DD. DSN=CEE.SCEEBND2,DISP=SHR           <- include .SCEEBND2  
// ...  
//SYSLIN DD ...  
//   DD DSN=CEE.SCEELIB(CELQV004),DISP=SHR       <- include member CELQV004  
// ...
```

You can also use the following cataloged procedures.

1. IGYQC - A single-step cataloged procedure for compiling a COBOL AMODE 64 program
2. IGYQCB - A two-step cataloged procedure for compiling and binding a COBOL AMODE 64 program
3. IGYQCBG - A three-step cataloged procedure for compiling, binding, and running a COBOL AMODE 64 program

DLL considerations

When LP (64) is specified, DLL style CALLS are generated by the COBOL compiler; the DLL and DYNAM options are ignored.

You can statically bind the module to other LP (64) modules.

When bind into a DLL to be called at run-time, you must use the EXPORTALL option to expose the program names that callers of the DLL uses.

In either static or run-time case, you must always specify DYNAM(DLL) in the bind.

Related tasks

[“DLL” on page 353](#)

[“DYNAM” on page 356](#)

[“EXPORTALL” on page 359](#)

Compiling and binding AMODE 64 programs under z/OS UNIX

The setup and preparations for compiling COBOL AMODE 64 programs with z/OS UNIX are the same as the corresponding setup and preparations under AMODE 31. See [Chapter 15, “Compiling under z/OS UNIX,” on page 283](#) for more information.

The compilation is done using the -q64 option of the cob2 command. For example, the following command compiles and binds the COBOL source file hello.cb1 in the current directory. The binder output is written to the default executable file a.out.

```
cob2 -q64 hello.cb1
```

You can explicitly specify the binder output file by using the -o option. The following cob2 command generates the executable file hello.exe.

```
cob2 -q64 hello.cb1 -o hello.exe
```

Other compiler options can be specified by using the -q option in the same way as using cob2 command with AMODE 31. For example, the following command invokes the COBOL compiler by using SOURCE and LIST options:

```
cob2 -q64 -q"SOURCE,LIST" hello.cb1
```

Note that -q64 is the short form of the compiler option LP (64) in the cob2 command line. It must be specified on its own in the cob2 command line, separate from other compiler option specifications. Instead of -q64, you can use LP (64) directly with other compiler options:

```
cob2 -q"LP(64),SOURCE,LIST" hello.cb1
```

Correspondingly, the -q32 is the short form of the compiler option LP (32). If cob2 is invoked without explicitly specifying -q64 or -q32, the default is -q32.

Environment variables

In addition to the environment variables supported with AMODE 31, the following two additional environment variables are supported for AMODE 64:

_C89_L6SYSLIB

Specifies the library dataset concatenations of SYSLIB DD to be used by the bind step. Dataset names are separated by colons. For example, the following environment variable setting

```
export _C89_L6SYSLIB=MYSAMPLE.SCEEEND2:CEE.SCEEEND2
```

gives the following SYSLIB DD concatenation in the bind step:

```
//SYSLIB DD DSN=MYSAMPLE.SCEEBND2,DISP=SHR
// DD DSN=CEE.SCEEBND2,DISP=SHR
```

By default, if `_C89_L6SYSLIB` is not specified, the following equivalent setting is used:

```
export _C89_L6SYSLIB=CEE.SCEEBND2
```

`_C89_L6SYSIX`

Specifies the side-deck dataset concatenations of SYSLIN to be used by the bind step. Dataset names are separated by colons. For example, the following environment variable setting

```
export _C89_L6SYSIX="MYSAMPLE.SCEELIB(CELQV004):CEE.SCEELIB(CELQV004)"
```

gives the following SYSLIN DD concatenation in the bind step:

```
//SYSLIN DD DSN=MYSAMPLE.SCEELIB(CELQV004),DISP=SHR
// DD DSN=CEE.SCEELIB(CELQV004),DISP=SHR
```

By default, if `_C89_L6SYSIX` is not specified, the following equivalent setting is used:

```
export _C89_L6SYSIX="CEE.SCEELIB(CELQV004)"
```

Running COBOL AMODE 64 applications

When preparing to run an application that contains COBOL AMODE 64 programs, ensure that the SCEERUN2 and SCEERUN Language Environment load libraries are available in the system library search order, for example, by using a STEPLIB DD statement.

Note: Users can compile AMODE 64 COBOL applications on z/OS 2.2 or above, but must be on z/OS 2.3 or above to run AMODE 64 COBOL applications.

Specifying COBOL only runtime options

AMODE 64 COBOL only runtime options can be specified using the IGZOPTS DD statements, or by setting the environment variable `_IGZ_RUNOPTS`.

Below shows an example of setting the CBLQDA and DEBUG options:

```
//...
//IGZOPTS DD *
//DEBUG,CBLQDA(ON)
/*
//*
```

The option syntax is the same as the corresponding specifications in AMODE 31 using the Language Environment CEEOPTS DD statements.

When running the program under z/OS UNIX environment, you can specify the COBOL only runtime options using the `_IGZ_RUNOPTS` environment variable. The option syntax is the same as their corresponding specifications in AMODE 31 using the LE environment variable `_CEE_RUNOPTS`. The example below specifies the DEBUG and CBLQDA options:

```
export _IGZ_RUNOPTS="DEBUG,CBLQDA(ON)"
```

AMODE 64 COBOL only runtime options can only be specified using these two methods. For more information about how to specify COBOL runtime options, see [“Specifying COBOL runtime options” on page 422](#).

Note: The CBLOPTS runtime option is not supported in AMODE 64. When specifying Language Environment runtime option on the parameter string of the main program, runtime options must be

specified before program arguments. This behavior is consistent with other Language Environment high-level languages. The example below shows the JCL for invoking a COBOL main program with program argument '20190901' and runtime option POSIX(ON):

```
//GOSTEP EXEC PGM=COBMAIN,PARM='POSIX(ON) / 20190901'
```

Chapter 19. COBOL compiler and runtime options

This section provides detailed information on the options available for the COBOL compiler and COBOL runtime environment.

COBOL compiler options

You can direct and control your compilation by using compiler options or by using compiler-directing statements (compiler directives).

Compiler options affect the aspects of your program that are listed in the table below. The linked-to information for each option provides the syntax for specifying the option and describes the option, its parameters, and its interaction with other parameters.

Table 45. Compiler options

Aspect of your program	Compiler option	Default	Option abbreviations
Source language	“APOST/QUOTE” on page 337	QUOTE	APOST Q
	“ARITH” on page 339	ARITH (COMPAT)	AR (C E)
	“CICS” on page 342	NOCICS	None
	“CODEPAGE” on page 343	CODEPAGE (1140)	CP (<i>ccsid</i>)
	“CURRENCY” on page 348	NOCURRENCY	CURR NOCURR
	“DBCS” on page 350	DBCS	None
	“NSYMBOL” on page 377	NSYMBOL (NATIONAL)	NS (DBCS NAT)
	“NUMBER” on page 378	NONUMBER	NUM NONUM
	“QUALIFY” on page 390	QUALIFY (COMPAT)	QUA (C E)
	“SEQUENCE” on page 395	SEQUENCE	SEQ NOSEQ
	“SQL” on page 398	NOSQL	None
	“SQLCCSID” on page 399	SQLCCSID	SQLC NOSQLC
	“SQLIMS” on page 400	NOSQLIMS	None
	“SUPPRESS” on page 403	SUPPRESS	SUPP
	“WORD” on page 415	NOWORD	WD NOWD
“XMLPARSE” on page 415	XMLPARSE (XMLSS)	XP (X) XP (C)	
Date processing	“INTDATE” on page 365	INTDATE (ANSI)	None

Table 45. **Compiler options** (continued)

Aspect of your program	Compiler option	Default	Option abbreviations
Maps and listings	CONDCOMP	CONDCOMP (NOSKIPSRC)	CC (SKIPS NOSKIPS)
	“LANGUAGE” on page 370	LANGUAGE (ENGLISH)	LANG (EN UE JA JP)
	“LINECOUNT” on page 371	LINECOUNT (60)	LC
	“LIST” on page 372	NOLIST	None
	“MAP” on page 374	NOMAP	None
	“OFFSET” on page 383	NOOFFSET	OFF NOOFF
	“SOURCE” on page 397	SOURCE	S NOS
	“SPACE” on page 398	SPACE (1)	None
	“TERMINAL” on page 403	NOTERMINAL	TERM NOTERM
	“VBREF” on page 412	NOVBREF	None
	“XREF” on page 416	XREF (FULL)	X NOX
Object deck generation	“COMPILE” on page 345	NOCOMPILE (S)	C NOC
	“COPYRIGHT” on page 347	NOCOPYRIGHT	CPYR NOCPYR
	“DECK” on page 350	NODECK	D NOD
	“NAME” on page 377	NONAME, or NAME (NOALIAS) if only NAME is specified	None
	“OBJECT” on page 383	OBJECT	OBJ NOOBJ
	“PGMNAME” on page 388	PGMNAME (COMPAT)	PGMN (CO LU LM)
	“SERVICE” on page 395	NOSERVICE	SERV NOSERV

Table 45. **Compiler options** (continued)

Aspect of your program	Compiler option	Default	Option abbreviations
Object code control	“ADV” on page 336	ADV	None
	“AFP” on page 336	AFP (NOVOLATILE)	None
	“ARCH” on page 337	ARCH(10)	None
	“AWO” on page 340	NOAWO	None
	“BLOCK0” on page 340	NOBLOCK0	None
	“DEFINE” on page 350	NODEFINE	DEF NODEF
	“DISPSIGN” on page 352	DISPSIGN (COMPAT)	DS(S C)
	“DLL” on page 353	NODLL	None
	“EXPORTALL” on page 359	NOEXPORTALL	EXP NOEXP
	“FASTSRT” on page 359	NOFASTSRT	FSRT NOFSRT
	“HGPR” on page 362	HGPR (PRESERVE)	None
	“INITIAL” on page 364	NOINITIAL	None
	“INLINE” on page 365	INLINE	INL NOINL
	“INVDATA” on page 366	NOINVDATA	NOINV INVDFNC NOFNC, CS NOCS)
	“JAVAIOP” on page 369	NOJAVAIOP	JIOP NOJIOP
	“LP” on page 373	LP(32)	None
	“MAXPCF” on page 375	MAXPCF(100000)	None
	“NUMCHECK” on page 378	NONUMCHECK	NC NONC
	“NUMPROC” on page 382	NUMPROC (NOPFD)	None
	“OPTIMIZE” on page 385	OPTIMIZE(0)	OPT(n)
	“OUTDD” on page 386	OUTDD (SYSOUT)	OUT
	“PARMCHECK” on page 387	NOPARMCHECK	PC NOPC
	“TRUNC” on page 409	TRUNC (STD)	None
	“TUNE” on page 411	TUNE(10) if ARCH is not specified. The default TUNE level must match the ARCH level if ARCH is specified.	None
“VLR” on page 412	VLR(STD)	VLR(C S)	
“ZONECHECK” on page 417	NOZONECHECK	NOZC ZC(MSG) ZC(ABD)	
“ZONEDATA” on page 418	ZONEDATA (PFD)	ZD(PFD) ZD(MIG) ZD(NOPFD)	
“ZWB” on page 421	ZWB	None	

Table 45. **Compiler options** (continued)

Aspect of your program	Compiler option	Default	Option abbreviations
Virtual storage usage	“BUFSIZE” on page 341	4096	BUF
	“DATA” on page 349	DATA (31)	None
	“DYNAM” on page 356	NODYNAM	DYN NODYN
	“RENT” on page 391	RENT	None
	“RMODE” on page 392	AUTO	None
	“STGOPT” on page 402	NOSTGOPT	SO NOSO
Debugging and diagnostics	“DIAGTRUNC” on page 352	NODIAGTRUNC	DTR NODTR
	“DUMP” on page 355	NODUMP	DU NODU
	“FLAG” on page 359	FLAG(I, I)	F NOF
	“FLAGSTD” on page 360	NOFLAGSTD	None
	“INITCHECK” on page 363	NOINITCHECK	IC NOIC
	“RULES” on page 393	NORULES	RULES (ENDP, EVENP, LXPRF, SLCKB, OOM) RULES (NOENDP, NOEVENP, NOLXPRF, NOSLCKB, NOOOM, NOUNRA NOUNRS)
	“SSRANGE” on page 401	NOSSRANGE	SSR (ZLEN NOZLEN, MSG ABD) NOSSR
	“TEST” on page 404	NOTEST	None
Other	“ADATA” on page 335	NOADATA	None
	“COPYLOC” on page 346	NOCOPYLOC	CPLC
	“EXIT” on page 357	NOEXIT	NOEX EX (INX NOINX, LIBX NOLIBX, PRTX NOPRTX, ADX NOADX, MSGX NOMSGX)
	“MDECK” on page 376	NOMDECK	NOMD MD MD (C NOC)
	“OPTFILE” on page 384	None	None
	“SMARTBIN” on page 396	SMARTBIN when LP (32) is in effect. SMARTBIN is not supported when LP (64) is in effect.	None
	“THREAD” on page 408	NOTHREAD	None
	“VSAMOPENFS” on page 414	VSAMOPENFS (COMPAT)	VS (C S)

Installation defaults: The default compiler options that were set up when your compiler was installed are in effect for your program unless you override those options. (In some installations, certain compiler options are fixed so that you cannot override them. If you have problems with the default options, contact

your system administrator.) To determine which are the default options, run a test compilation without specifying any compiler options. The output listing lists the default options in effect at your site.

Nonoverridable options: In some installations, certain compiler options are fixed so that you cannot override them. If you have problems with those options, contact your system administrator.

Option specification: Compiler options and suboptions are not case sensitive.

Performance considerations: The AFP, ARCH, ARITH, AWO, BLOCK0, DYNAM, FASTSRT, HGPR, MAXPCF, NUMCHECK, NUMPROC, OPTIMIZE, PARMCHECK, RENT, SQLCCSID, SSRANGE, STGOPT, TEST, THREAD, TRUNC, TUNE, ZONECHECK, and INVDATA compiler options can affect runtime performance.

Related tasks

[Chapter 14, “Compiling under z/OS,” on page 255](#)

[“Compiling under TSO” on page 262](#)

[Chapter 15, “Compiling under z/OS UNIX,” on page 283](#)

[Chapter 49, “Tuning your program,” on page 757](#)

Related references

[“Conflicting compiler options” on page 334](#)

[Chapter 20, “Compiler-directing statements,” on page 429](#)

[“Option settings for 85 COBOL Standard conformance” on page 333](#)

[“Performance-related compiler options” on page 764](#)

Option settings for 85 COBOL Standard conformance

Compiler options and runtime options are required for conformance with the 85 COBOL Standard.

The following compiler options are required:

- ADV
- DYNAM
- NAME (ALIAS) or NAME (NOALIAS)
- NOBLOCK0
- NOCICS
- NODLL
- NOEXPORTALL
- NOFASTSRT
- NOTHREAD
- NOWORD
- NUMPROC (NOPFD)
- PGMNAME (COMPAT) or PGMNAME (LONGUPPER)
- QUALIFY (COMPAT)
- QUOTE
- TRUNC (STD)
- VLR (STANDARD)
- VSAMOPENFS (SUCC)
- ZWB

You can use the FLAGSTD compiler option to flag nonconforming elements such as IBM extensions.

The following runtime options are required:

- AIXBLD
- CBLQDA (ON)
- TRAP (ON)

Related references

Language Environment Programming Reference

Conflicting compiler options

The Enterprise COBOL compiler can encounter conflicting compiler options in either of two ways: both the positive and negative form of an option are specified at the same level in the hierarchy of precedence, or mutually exclusive options are specified at the same level in the hierarchy.

When conflicting options are specified at the same level in the hierarchy (such as specifying both DECK and NODECK in a PROCESS or CBL statement), the option specified last takes effect.

If you specify mutually exclusive compiler options at the same level, the compiler generates an error message and forces one of the options to a nonconflicting value. For example, if you specify both OFFSET and LIST in a 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 a JCL statement but LIST in a PROCESS statement, LIST takes effect because the options coded in the PROCESS statement and any options forced on by an option coded in the PROCESS statement have higher precedence.

Specified	Ignored ¹	Forced on ¹
AFP(VOLATILE)	LP(64)	LP(32)
CICS	DYNAM	NODYNAM
	LP(64)	LP(32)
	NORENT	RENT
DATA(24)	LP(64)	LP(32)
DLL	DYNAM	NODYNAM
	NORENT	RENT
EXPORTALL	NODLL	DLL
	DYNAM	NODYNAM
	NORENT	RENT
HGPR(NOPRESERVE)	LP(64)	LP(32)
LP(64)	AFP(VOLATILE)	AFP(NOVOLATILE)
	CICS	NOCICS
	DATA(24)	DATA(31)
	HGPR(NOPRESERVE)	HGPR(PRESERVE)
	NODLL	DLL
	NORENT	RENT
	RMODE(24)	RMODE(ANY)
	SMARTBIN	NOSMARTBIN
	SQLIMS	NOSQLIMS
	THREAD	NOTHREAD

Table 46. **Mutually exclusive compiler options** (continued)

Specified	Ignored ¹	Forced on ¹
NORENT	RMODE (ANY)	RMODE (24)
	LP (64)	LP (32)
NSYMBOL (NATIONAL)	NODBCS	DBCS
OBJECT	DECK	NODECK
OFFSET	LIST	NOLIST
PGMNAME (LM LU)	NAME	NONAME
RMODE (24)	LP (64)	LP (32)
SMARTBIN	LP (64)	LP (32)
SOURCE (HEX)	NUMBER	NONUMBER
	SEQUENCE	NOSEQUENCE
SQLIMS	LP (64)	LP (32)
TEST	NOOBJECT and NODECK	OBJECT and NODECK
THREAD	INITIAL	NOINITIAL
	NORENT	RENT
	LP (64)	LP (32)
WORD	FLAGSTD	NOFLAGSTD
1. Unless in conflict with a fixed installation default option.		

Related tasks

[“Specifying compiler options under z/OS” on page 273](#)

[“Specifying compiler options in a batch compilation” on page 278](#)

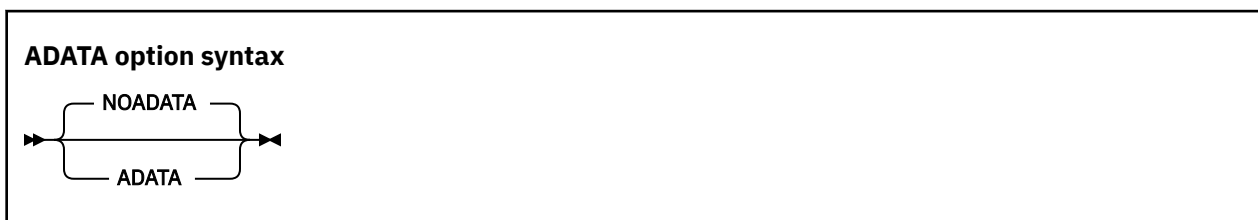
[“Specifying compiler options under z/OS UNIX” on page 284](#)

Related references

[“OPTFILE” on page 384](#)

ADATA

Use ADATA when you want the compiler to create a SYSADATA file that contains records of additional compilation information.



Default is: NOADATA

Abbreviations are: None

On z/OS, the SYSADATA file is written to ddname SYSADATA.

The size of the SYSADATA file generally grows with the size of the associated program.

Option specification: You cannot specify the ADATA option in a PROCESS (or CBL) statement. You can specify it only in one of the following ways:

- In the PARM parameter of JCL
- As a cob2 command option
- As an installation default
- In the COBOPT environment variable

Related references

[“Setting environment variables under z/OS UNIX” on page 283](#)

[“cob2 syntax and options” on page 287](#)

[Appendix G, “COBOL SYSADATA file contents,” on page 837](#)

ADV

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.



Default is: ADV

Abbreviations are: None

Use NOADV if you already adjusted record length to include 1 byte for the printer control character.

AFP

The AFP option controls the compiler usage of the Additional Floating Point (AFP) registers that are provided by IBM z/Architecture processors.



Default is: AFP (NOVOLATILE)

Abbreviations are: None

The Enterprise COBOL compiler generates code that uses the full complement of 16 floating point registers (FPR) provided by an IBM z/Architecture processor. These FPRs are as follows:

- Original FPRs, which are numbered 0, 2, 4, and 6
- AFP registers, which are numbered 1, 3, 5, 7, and 8-15

AFP (VOLATILE)

If you specify AFP (VOLATILE), the AFP registers 8-15 are considered volatile, which means that they might be changed by a called subprogram. Therefore, the COBOL compiler generates extra code to protect the values in these registers.

AFP (NOVOLATILE)

If you specify AFP (NOVOLATILE), the AFP registers 8-15 are considered nonvolatile, which means that they are known to be unchanged or preserved by every called subprogram. Therefore, the

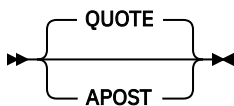
compiler can generate more efficient code sequences for programs with floating point operations. It is the normal z/OS architecture convention.

AMODE 64 considerations: When the LP (64) compiler option is in effect, the AFP (VOLATILE) option is not supported. If the AFP (VOLATILE) option is specified explicitly, a warning message is issued and the setting is discarded.

APOST/QUOTE

Use APOST if you want the figurative constant [ALL] QUOTE or [ALL] QUOTES to represent one or more apostrophe (') characters. Use QUOTE if you want the figurative constant [ALL] QUOTE or [ALL] QUOTES to represent one or more quotation mark (") characters.

APOST/QUOTE option syntax



Default is: QUOTE

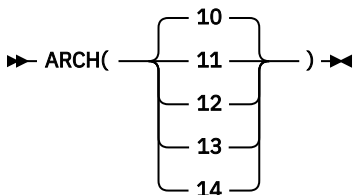
Abbreviations are: Q (for QUOTE) | None (for APOST)

Delimiters: You can use either quotation marks (") or apostrophes (') as literal delimiters regardless of whether the APOST or QUOTE option is in effect. The delimiter character used as the opening delimiter for a literal must be used as the closing delimiter for that literal.

ARCH

The ARCH option specifies the machine architecture for which the executable program instructions are to be generated.

ARCH option syntax



Default is: ARCH(10)

Abbreviations are: None

If you specify a higher ARCH level, the compiler generates code that uses newer and faster instructions. Your application might abend if it runs on a processor with an architecture level lower than what you specified with the ARCH option. Use the ARCH level that matches the lowest machine architecture where your application runs, including any disaster recovery (DR) machines.

Current supported architecture levels and groups of models are as follows:

10

Produces code that uses instructions available on the 2827-xxx (IBM zEnterprise® EC12) and 2828-xxx (IBM zEnterprise BC12) models in IBM z/Architecture mode.

Specifically, these ARCH(10) machines and their follow-ons add instructions supported by the following facilities:

- Execution-Hint Facility

- Load-and-Trap Facility
- Miscellaneous-Instructions-Extension Facility
- Transactional-Execution Facility
- Enhanced Decimal Floating Point Facility that enables more efficient conversions between zoned decimal data items and decimal floating point data items. Instead of converting zoned decimal data items to packed decimal data items to perform arithmetic when conditions permit it and the optimization level is greater than 0, the compiler converts zoned decimal data items directly to decimal floating point data items, and then back again to zoned decimal data items after the computations are complete.

11

Produces code that uses instructions available on 2964-xxx (IBM z13[®]) and 2965-xxx (IBM z13s[®]) models in IBM z/Architecture mode.

Specifically, these ARCH(11) machines and their follow-ons add instructions supported by the following facilities:

- Enhanced Decimal Floating Point Facility that enables more efficient conversions between packed-decimal data items and decimal floating point intermediate result data items when the surrounding conditions are optimal and the optimization level is greater than 0.
- Exploitation of the Vector Extension Facility (SIMD) instructions for some INSPECT REPLACING and INSPECT TALLYING statements.

To use the Vector Extension Facility (SIMD) instructions, the code must be executed on a machine running on z/OS 2.2, or z/OS 2.1 with the PTFs for APARs OA43803 and PI12412 installed.

12

Produces code that uses instructions available on 3906-xxx (IBM z14) and 3907-xxx (IBM z14 ZR1) models in IBM z/Architecture mode.

Specifically, these ARCH(12) machines and their follow-ons add instructions supported by the Vector Packed Decimal Facility, which accelerates packed and zoned decimal computation by storing intermediate results in vector registers instead of in memory.

13

Produces code that uses instructions available on the 8561-xxx (IBM z15) and 8562-xxx (IBM z15 T02) models in IBM z/Architecture mode.

Specifically, these ARCH(13) machines and their follow-ons add instructions supported by the following facilities:

- Vector Packed Decimal Enhancement Facility
- Vector-Enhancements Facility 2
- Miscellaneous Instruction-Extensions-Facility 3
- Aligned vector load/store hints

14

Produces code that uses instructions available on the 3931-xxxx (IBM z16) model in IBM z/Architecture mode.

Specifically, this ARCH(14) machine and its follow-ons add instructions supported by the new Vector Packed Decimal Enhancement Facility 2. This new facility adds performance improvements for COBOL programs that contain one or more of the following types of statements:

- Exponentiation operations on packed or zoned decimal data items where the exponent is declared with one or more fractional digits
- Arithmetic statements involving mixed decimal and floating-point data items
- Statements using numeric-edited data items

Note: A higher ARCH level includes the facilities of the lower ARCH levels. For example, ARCH(14) includes all the facilities of the lower ARCH levels.

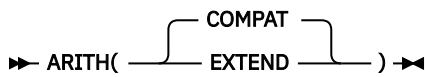
For more information about these facilities, see *IBM z/Architecture Principles of Operation*.

Related references
“TUNE” on page 411

ARITH

ARITH affects the maximum number of digits that you can code for numeric items, and the number of digits used in fixed-point intermediate results.

ARITH option syntax



Default is: ARITH(COMPAT)

Abbreviations are: AR(C | E)

When you specify ARITH(EXTEND):

- The maximum number of digit positions that you can specify in the PICTURE clause for packed-decimal, external-decimal, and numeric-edited data items is raised from 18 to 31.
- The maximum number of digits that you can specify in a fixed-point numeric literal is raised from 18 to 31. You can use numeric literals with large precision anywhere that numeric literals are currently allowed, including:
 - Operands of PROCEDURE DIVISION statements
 - VALUE clauses (for numeric data items with large-precision PICTURE)
 - Condition-name values (on numeric data items with large-precision PICTURE)
- The maximum number of digits that you can specify in the arguments to NUMVAL, NUMVAL -C and NUMVAL -F is raised from 18 to 31.
- The maximum value of the integer argument to the FACTORIAL function is 29.
- Intermediate results in arithmetic statements use *extended mode*.

When you specify ARITH(COMPAT):

- The maximum number of digit positions in the PICTURE clause for packed-decimal, external-decimal, and numeric-edited data items is 18.
- The maximum number of digits in a fixed-point numeric literal is 18.
- The maximum number of digits in the arguments to NUMVAL, NUMVAL -C and NUMVAL -F is 18.

Note: NUMVAL uses long-precision floating point calculations to convert the digits in the argument to the numeric output. At most, 15 decimal digits can be converted accurately to long-precision floating point (as described in the related reference below about conversions and precision). Therefore, if the argument to NUMVAL, NUMVAL -C, or NUMVAL -F has more than 15 digits, it is recommended that you specify the ARITH(EXTEND) compiler option, so that the result will accurately represent the input argument.

- The maximum value of the integer argument to the FACTORIAL function is 28.
- Intermediate results in arithmetic statements use *compatibility mode*.

ARITH(EXTEND) example

This example shows how to use ARITH(EXTEND):

```
CBL ARITH(EXTEND)
IDENTIFICATION DIVISION.
PROGRAM-ID.     SAMPLE
DATE-COMPILED.
ENVIRONMENT DIVISION.
```

```

DATA DIVISION.
FILE SECTION.
WORKING-STORAGE SECTION.

01 WS-CHAR          PIC X(30).
01 WS-NUM           PIC 9(30).
*> Note this will get a syntax error with ARITH(COMPAT)

PROCEDURE DIVISION.

    MOVE '651310406401912040024657817701' TO WS-CHAR
    MOVE WS-CHAR TO WS-NUM

    DISPLAY 'THE CHAR VALUE: ' WS-CHAR
    DISPLAY 'THE NUM  VALUE: ' WS-NUM

GOBACK.

```

The output of the example:

```

THE CHAR VALUE: 651310406401912040024657817701
THE NUM  VALUE: 651310406401912040024657817701

```

Related concepts

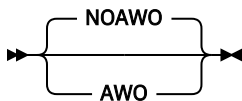
Appendix A, “Intermediate results and arithmetic precision,” on page 779

[“Conversions and precision” on page 52](#)

AWO

If you specify AWO, an implicit APPLY WRITE-ONLY clause is activated for all QSAM files in the program that have blocked variable-length records.

AWO option syntax



Default is: NOAWO

Abbreviations are: None

Related tasks

[“Optimizing buffer and device space” on page 10](#)

Related references

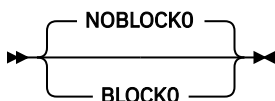
[“BLOCK0” on page 340](#)

APPLY WRITE-ONLY clause (*Enterprise COBOL for z/OS Language Reference*)

BLOCK0

Use BLOCK0 to change the compiler default for QSAM files from unblocked to blocked (as if BLOCK CONTAINS 0 were specified) and thus gain the benefit of system-determined blocking for output files.

BLOCK0 option syntax



Default is: NOBLOCK0

Abbreviations are: None

Specifying BLOCK0 activates an implicit BLOCK CONTAINS 0 clause for each file in the program that meets the following three criteria:

- The FILE-CONTROL paragraph either specifies ORGANIZATION SEQUENTIAL or omits the ORGANIZATION clause.
- The FD entry does not specify RECORDING MODE U.
- The FD entry does not specify a BLOCK CONTAINS clause.

Files for which the resulting BLOCK CONTAINS 0 clause is in effect have a blocking factor that is determined at run time from the data definition or from the data-set characteristics.

Interaction of the APPLY WRITE-ONLY clause and the AWO compiler option with BLOCK0:

- If NOBLOCK0 is in effect, and the file description of a file that meets the three criteria listed above specifies APPLY WRITE-ONLY, the compiler issues an error message because APPLY WRITE-ONLY applies only to blocked files. But if BLOCK0 is in effect, the result is that the file is blocked, and the APPLY WRITE-ONLY clause is therefore accepted.
- AWO applies to any QSAM files that have blocked variable-length records. If BLOCK0 is in effect, the result is that more files might be blocked than if NOBLOCK0 were in effect; thus AWO might apply to more files than it otherwise would.

Specifying BLOCK0 for existing programs might result in a change of behavior, and in some cases produce undesirable results for files opened as INPUT. For example:

- The OPEN INPUT statement fails for files for which no block size can be determined.
- Programs that continue after handling nonzero FILE STATUS codes for files opened as INPUT might abnormally terminate when executing subsequent I/O statements on those files.

For these reasons, after compiling with BLOCK0 you should investigate and test the effects on your program.

For recommendations about blocking, see the related reference from the *Enterprise COBOL for z/OS Migration Guide* (in the information about migrating from CMPR2 to NOCMPR2).

Related tasks

[“Optimizing buffer and device space” on page 10](#)

[“Setting block sizes” on page 174](#)

Related references

[“AWO” on page 340](#)

[APPLY WRITE-ONLY clause \(*Enterprise COBOL for z/OS Language Reference*\)](#)

[BLOCK CONTAINS clause \(*Enterprise COBOL for z/OS Language Reference*\)](#)

[Enterprise COBOL for z/OS Migration Guide](#)

(Recommendation for DCB= parameters of JCL)

BUFSIZE

Use BUFSIZE to allocate an amount of main storage to the buffer for each compiler work data set. Usually, a large buffer size improves the performance of the compiler.

BUFSIZE option syntax

►► BUFSIZE($\overbrace{\hspace{1.5cm}}^{nnnn}$ $\underbrace{\hspace{1.5cm}}_{nnnK}$) ◀◀

Default is: 4096

Abbreviations are: BUF

nnnn specifies a decimal number that must be at least 256.

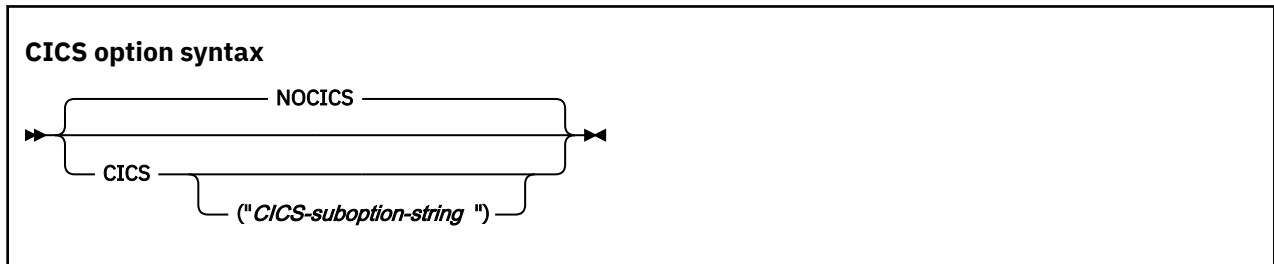
nnnK specifies a decimal number in 1 KB increments, where 1 KB = 1024 bytes.

BUFSIZE cannot exceed the track capacity for the device used, nor can it exceed the maximum allowed by data management services.

Note: The BUFSIZE compiler option can only be specified for the first program in a sequence of programs (batch compilation). If you specify BUFSIZE in a CBL or PROCESS statement for a program other than the first program in the sequence, you will receive a compiler diagnostic message.

CICS

The CICS compiler option enables the integrated CICS translator and lets you specify CICS suboptions. You must use the CICS option if your COBOL source program contains EXEC CICS or EXEC DLI statements and the program has not been processed by the separate CICS translator.



Default is: NOCICS

Abbreviations are: None

Use the CICS option only to compile CICS programs. Programs compiled with the CICS option will not run in a non-CICS environment.

If you specify the NOCICS option, any CICS statements found in the source program are diagnosed and discarded.

Use either quotation marks or apostrophes to delimit the string of CICS suboptions.

You can partition a long CICS suboption string into multiple suboption strings in multiple CBL or PROCESS statements. The CICS suboptions are concatenated in the order of their appearance. For example:

```
//STEP1 EXEC IGYWC, . . .
// PARM.COBOL='CICS("string1")'
//COBOL.SYSIN DD *
    CBL CICS('string2')
    CBL CICS("string3")
    IDENTIFICATION DIVISION.
    PROGRAM-ID. DRIVER1.
    . . .
```

The compiler passes the following suboption string to the integrated CICS translator:

```
"string1 string2 string3"
```

The concatenated strings are delimited with single spaces as shown. If multiple instances of the same CICS suboption are found, the last specification of that suboption in the concatenated string prevails. The compiler limits the size of the concatenated suboption string to 4 KB.

AMODE 64 considerations: When the LP (64) compiler option is in effect, the CICS option is not supported. If the CICS option is specified explicitly, a warning message is issued and the setting is discarded.

Note: The CICS compiler option can only be specified for the first program in a sequence of programs (batch compilation). If you specify CICS in a CBL or PROCESS statement for a program other than the first program in the sequence, you will receive a compiler diagnostic message.

Related concepts

[“Integrated CICS translator” on page 488](#)

Related tasks

[“Compiling with the CICS option” on page 487](#)

[“Separating CICS suboptions” on page 488](#)

Defining translator options (*Developing CICS Applications*)

Related references

[“Conflicting compiler options” on page 334](#)

CODEPAGE

Use CODEPAGE to specify the coded character set identifier (CCSID) for an EBCDIC code page for processing compile-time and runtime COBOL operations that are sensitive to character encoding.

CODEPAGE option syntax

► CODEPAGE(*ccsid*) ◄

Default is: CODEPAGE(1140)

Abbreviations are: CP(*ccsid*)

ccsid must be an integer that represents a valid CCSID for an EBCDIC code page.

The default CCSID 1140 is the equivalent of CCSID 37 (COM EUROPE EBCDIC), but additionally includes the euro symbol.

ccsid specifies these encodings:

- The encoding for alphanumeric, national, UTF-8, and DBCS literals in a COBOL source program
- The default encoding of the content of alphanumeric and DBCS data items at run time
- The encoding for DBCS user-defined words when processed by an XML GENERATE statement to create XML element and attribute names
- The default encoding of an XML document created by an XML GENERATE statement if the receiving data item for the document is alphanumeric
- The default encoding assumed for an XML document in an alphanumeric data item when the document is processed by an XML PARSE statement

The CODEPAGE *ccsid* is used when code-page-sensitive operations are performed at compile time or run time, and an explicit CCSID that overrides the default code page is not specified. Such operations include:

- Conversion of literal values to Unicode
- Conversion of alphanumeric data to and from national (Unicode) data and UTF-8 (Unicode) data as part of move operations, comparison, or the intrinsic functions DISPLAY-OF and NATIONAL-OF
- Object-oriented language such as INVOKE statements or class definitions and method definitions
- XML parsing
- XML generation
- Processing of DBCS names as part of XML generation at run time
- Processing of SQL string host variables if the SQLCCSID option is in effect
- Processing of source code for EXEC SQL statements
- Processing of source code for EXEC SQLIMS statements

However, the encoding of the following items in a COBOL source program is not affected by the CODEPAGE compiler option:

- Data items that have USAGE NATIONAL

These items are always encoded in UTF-16 in big-endian format, CCSID 1200.

- Data items that have USAGE UTGF-8

These items are always encoded in UTF-8 format, CCSID 1208.

- Characters from the basic COBOL character set (see the table of these characters in the related reference below about characters)

Though the encoding of the basic COBOL characters default currency sign (\$), quotation mark ("), and the lowercase Latin letters varies in different EBCDIC code pages, the compiler always interprets these characters using the EBCDIC code page 1140 encoding. In particular, the default currency sign is always the character with value X'5B' (unless changed by the CURRENCY compiler option or the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph), and the quotation mark is always the character with value X'7F'.

Some COBOL operations can override the CODEPAGE *ccsid* by using an explicit encoding specification, for example:

- DISPLAY-OF and NATIONAL-OF intrinsic functions that specify a code page as the second argument
- XML PARSE statements that specify the WITH ENCODING phrase
- XML GENERATE statements that specify the WITH ENCODING phrase

Additionally, you can use the CURRENCY compiler option or the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph to override:

- The default currency symbol used in the PICTURE character-strings for numeric-edited data items in your source program
- The currency sign value used in the content of numeric-edited data items at run time

DBCS code pages:

Compile your COBOL program using the CODEPAGE option with the *ccsid* set to one of the EBCDIC multibyte character set (MBCS) CCSIDs shown in the table below if the program contains any of the following items:

- User-defined words formed with DBCS characters
- DBCS (USAGE DISPLAY-1) data items
- DBCS literals

All of the CCSIDs in the table below identify mixed code pages that refer to a combination of SBCS and DBCS coded character sets. These are also the CCSIDs that are supported for mixed data by Db2.

National language	MBCS CCSID	SBCS CCSID component	DBCS CCSID component
Japanese (Katakana-Kanji)	930	290	300
Japanese (Katakana-Kanji with euro)	1390	8482	16684
Japanese (Katakana-Kanji)	5026	290	4396
Japanese (Latin-Kanji)	939	1027	300
Japanese (Latin-Kanji with euro)	1399	5123	16684
Japanese (Latin-Kanji)	5035	1027	4396
Korean	933	833	834
Korean	1364	13121	4930
Simplified Chinese	935	836	837
Simplified Chinese	1388	13124	4933
Traditional Chinese	937	28709	835

Note: If you specify the TEST option, you must set the CODEPAGE option to the CCSID that is used for the COBOL source program. In particular, programs that use Japanese characters in DBCS literals or DBCS user-defined words must be compiled with the CODEPAGE option set to a Japanese codepage CCSID.

Note for Db2 users: IBM recommends that you use the COBOL CCSID value the same as Db2 DSNHDECP value and/or the value for precompiler CCSID option.

Related concepts

[“COBOL and Db2 CCSID determination” on page 498](#)

Related tasks

[“Using currency signs” on page 64](#)

[Chapter 37, “Processing XML input,” on page 611](#)

[Chapter 38, “Producing XML output,” on page 651](#)

Related references

[“CURRENCY” on page 348](#)

[“SQLCCSID” on page 399](#)

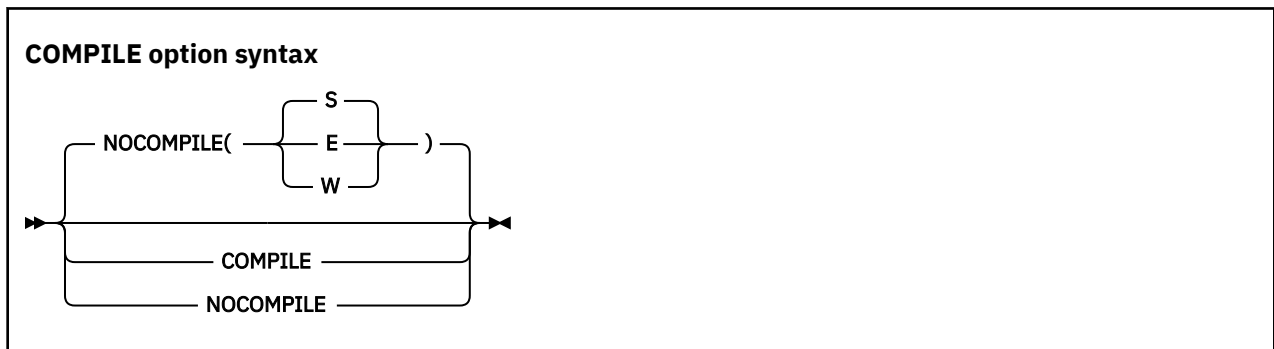
[“TEST” on page 404](#)

[“The encoding of XML documents” on page 628](#)

Characters (*Enterprise COBOL for z/OS Language Reference*)

COMPILE

Use the COMPILE option only if you want to force full compilation even in the presence of serious errors. All diagnostics and object code will be generated. Do not try to run the object code if the compilation resulted in serious errors: the results could be unpredictable or an abnormal termination could occur.



Default is: NOCOMPILE(S)

Abbreviations are: C | NOC

Use NOCOMPILE without any suboption to request a syntax check (only diagnostics produced, no object code). If you use NOCOMPILE without any suboption, several compiler options will have no effect because no object code will be produced, for example: DECK, LIST, OBJECT, OFFSET, OPTIMIZE, SSRANGE, and TEST.

Use NOCOMPILE with suboption W, E, or S for conditional full compilation. Full compilation (diagnosis and object code) will stop when the compiler finds an error of the level you specify (or higher), and only syntax checking will continue.

Related tasks

[“Finding coding errors” on page 438](#)

Related references

[“Messages and listings for compiler-detected errors” on page 281](#)

CONDCOMP

CONDCOMP affects the behavior of conditional compilation directives and controls how conditional code will be displayed in the listing.

CONDCOMP option syntax

```
CONDCOMP( NOSKIPSRC | SKIPSRC )
```

Default is: CONDCOMP(NOSKIPSRC)

Abbreviations are: CC(SKIPS | NOSKIPS)

CONDCOMP(NOSKIPSRC)

If CONDCOMP(NOSKIPSRC) is in effect, all source lines and comments bounded by conditional compilation directives will be displayed in the listing. Source lines in false branches of IF and EVALUATE directives will be displayed as comments.

CONDCOMP(SKIPSRC)

If CONDCOMP(SKIPSRC) is in effect, source lines and comments in the false branches of conditional compilation directives, IF and EVALUATE directives, will not be shown in the listing. In addition, compiled source lines and comments will also be omitted from the MDECK output file and SYSMDECK.

Related references

[Example: conditional compilation output](#)

Conditional compilation (*Enterprise COBOL for z/OS Language Reference*)

COPYLOC

Use the COPYLOC compiler option to add either a PDSE (or PDS) dataset or z/OS UNIX directory as an additional location to be searched for copy members during the library phase. The location specified by the COPYLOC option is added to the end of the order of locations to search for copy members. For details, see COPY member search order in the *Enterprise COBOL for z/OS Language Reference*.

COPYLOC is introduced in Enterprise COBOL V6.2 with the PTF for APAR PI91584 installed.

COPYLOC option syntax

```
COPYLOC ( NOCOPYLOC | SYSLIB ( library-name ) , DSN ( dataset-name ) , PATH ( unix-directory-name ) )
```

Default is: NOCOPYLOC

Abbreviations are: CPLC | NOCPLC

library-name

The library name that the copy location is to be associated with. When *library-name* is not specified, the default is SYSLIB, which is the library name assumed in COPY statements that do not include an explicit library name.

dataset-name

The name of a PDS or PDSE dataset in which the compiler should search for copy members when processing COPY statements that refer to library *library-name*.

unix-directory-name

The name of a z/OS UNIX directory in which the compiler should search for copy members when processing COPY statements that refer to library *library-name*. The specified path must not exceed 64 characters. To specify a lowercase path, which is standard for z/OS UNIX, the path should be surrounded in quotes. Otherwise, the path name will be converted to uppercase.

Multiple instances of the COPYLOC option are supported. There is no limit on the number of z/OS UNIX directories that can be specified, but there is a limit of 256 datasets that can be specified for searching. Copy locations will be searched in the order that they are specified via the COPYLOC option. This provides users the ability to mix PDSE (or PDS) locations and z/OS UNIX directories in the search.

If the NOCOPYLOC option is specified, any previous instances of the COPYLOC option are ignored.

To control the searching of copy member locations exclusively using the COPYLOC option, you should avoid using any of the existing methods for indicating copybook locations, such as, allocating datasets to a ddname in JCL, or specifying the -I option of the cob2 command. If the compiler is invoked from cob2, you should also avoid keeping copy members in the current directory, because the current directory will always be searched before the COPYLOC locations are being searched.

Tip: You might find it convenient to control searching of copy members exclusively using the COPYLOC option, especially when the compiler is invoked from the cob2 command.

When COPYLOC options are specified in CBL statements, they can be used only on the first program of a batch program. Therefore, if a file has multiple COBOL programs in it, there can be CBL statements with COPYLOC options preceding the first program, but not the other programs. The COPYLOC options specified for the first program (and COPYLOC options specified in PARM of JCL or COPYLOC options specified as cob2 command options under z/OS UNIX) apply to all programs in a file. The copy locations specified in COPYLOC options found in CBL cards will be searched after copy locations found in COPYLOC options specified as invocation parameters.

Example

```
COPYLOC(MYLIB,DSN(USERID.COBOL.COPYLIB1))
COPYLOC(MYLIB,PATH('/home/userid/copylib1'))
COPYLOC(MYLIB,DSN(USERID.COBOL.COPYLIB2))
```

For COPY statements that reference library name MYLIB explicitly, if the preceding options were specified in a single invocation of the compiler and a copy member could not be found in the locations specified in JCL (or in locations indicated for a cob2 compile), additional searching for the copy member will be done in the following places sequentially:

1. *USERID.COBOL.COPYLIB1* dataset
2. z/OS UNIX directory */home/userid/copylib1*
3. *USERID.COBOL.COPYLIB2* dataset

related references

COPY statement (*Enterprise COBOL for z/OS Language Reference*)

COPY member search order (*Enterprise COBOL for z/OS Language Reference*)

ALLOWCOPYLOC (*Enterprise COBOL Customization Guide*)

COPYRIGHT

Use COPYRIGHT to place a string in the object module if the object module is generated. If the object is linked into a program object, the string is loaded into memory with that program object.



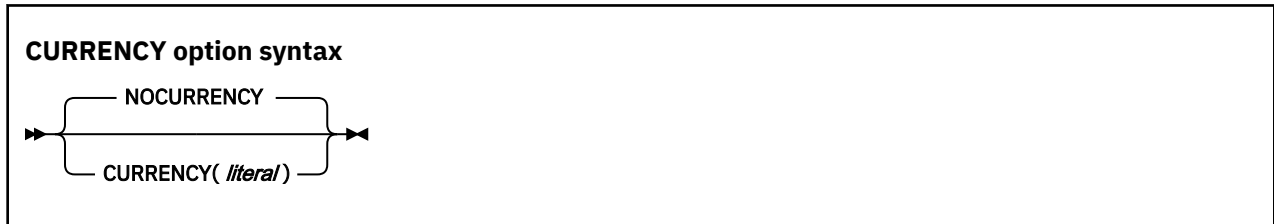
Default is: NOCOPYRIGHT

Abbreviations are: CPYR | NOCPYR

The *copyright string* is limited to 64 characters in length.

CURRENCY

You can use the CURRENCY option to provide an alternate default currency symbol to be used for a COBOL program. (The default currency symbol is the dollar sign (\$).)



Default is: NOCURRENCY

Abbreviations are: CURR | NOCURR

NOCURRENCY specifies that no alternate default currency symbol will be used.

To change the default currency symbol, specify CURRENCY (*literal*), where *literal* is a valid COBOL alphanumeric literal (optionally a hexadecimal literal) that represents a single character. The literal must not be from the following list:

- Digits zero (0) through nine (9)
- Uppercase alphabetic characters A, B, C, D, E, G, N, P, R, S, U, V, X, Z, or their lowercase equivalents
- The space
- Special characters * + - / , . ; () " =
- A figurative constant
- A null-terminated literal
- A DBCS literal
- A national literal

If your program processes only one currency type, you can use the CURRENCY option as an alternative to the CURRENCY SIGN clause for indicating the currency symbol you will use in the PICTURE clause of your program. If your program processes more than one currency type, you should use the CURRENCY SIGN clause with the WITH PICTURE SYMBOL phrase to specify the different currency sign types.

If you use both the CURRENCY option and the CURRENCY SIGN clause in a program, the CURRENCY option is ignored. Currency symbols specified in the CURRENCY SIGN clause or clauses can be used in PICTURE clauses.

When the NOCURRENCY option is in effect and you omit the CURRENCY SIGN clause, the dollar sign (\$) is used as the PICTURE symbol for the currency sign.

Delimiter: You can delimit the CURRENCY option literal with either quotation marks or apostrophes, regardless of the APOST | QUOTE compiler option setting.

Related tasks

[“Using currency signs” on page 64](#)

DATA

The DATA option affects whether storage for dynamic data areas and other dynamic runtime storage is obtained from above or below the 16 MB line.



Default is: DATA(31)

Abbreviations are: None

For reentrant programs, the DATA compiler option and the HEAP runtime option control whether storage for dynamic data areas (such as WORKING-STORAGE and FD record areas) is obtained from below the 16 MB line (DATA(24)) or from unrestricted storage (DATA(31)). (DATA does not affect the location of LOCAL-STORAGE data; the STACK runtime option controls that location instead, along with the AMODE of the program.)

Specify DATA(24) for programs that run in 31-bit addressing mode and that pass data arguments to programs in 24-bit addressing mode. Doing so ensures that the data will be addressable by the called program.

External data and QSAM buffers: The DATA option interacts with other compiler options and runtime options that affect storage and its addressability. See the related information for details.

The DATA compiler option setting influences how ALLOCATE acquires storage:

- If DATA(24) is in effect and the LOC 31 phrase of the ALLOCATE statement is not specified, ALLOCATE acquires storage from below the 16 MB line.
- If DATA(31) is in effect and the LOC 24 phrase of the ALLOCATE statement is not specified, ALLOCATE will attempt to acquire storage from above the 16 MB line.

AMODE 64 considerations: The DATA(24) compiler option is not supported when LP(64) is in effect. If the DATA(24) option is specified explicitly, a warning message is issued and the setting is discarded.

The LOCAL-STORAGE section is allocated from stack storage, which is managed by Language Environment. LE allocates stack storage above the 2 GB bar in a 64-bit enclave.

Related concepts

[“Storage and its addressability” on page 39](#)

Related tasks

[“Making programs reentrant” on page 548](#)

Language Environment Programming Guide (Using runtime options)

Related references

[“Allocation of buffers for QSAM files” on page 186](#)

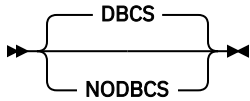
[“RENT” on page 391](#)

ALLOCATE statement (*Enterprise COBOL for z/OS Language Reference*)

DBCS

Using DBCS causes the compiler to recognize X'0E' (SO) and X'0F' (SI) as shift codes for the double-byte portion of an alphanumeric literal.

DBCS option syntax



Default is: DBCS

Abbreviations are: None

With DBCS in effect, the double-byte portion of the literal is syntax-checked and the literal remains category alphanumeric.

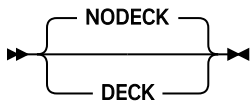
Related references

[“Conflicting compiler options” on page 334](#)

DECK

Use DECK to produce object code in the form of 80-column records. If you use the DECK option, be certain that SYSPUNCH is defined in your JCL for compilation.

DECK option syntax



Default is: NODECK

Abbreviations are: D | NOD

Related tasks

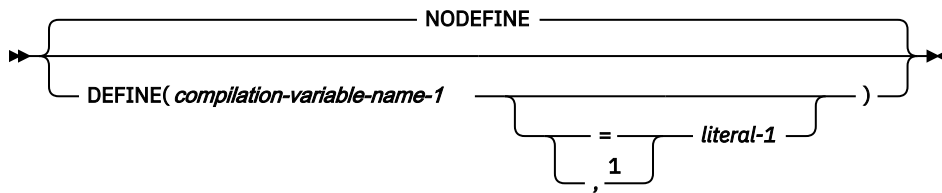
[“Creating object code \(SYSLIN or SYSPUNCH\)” on page 271](#)

DEFINE

Use the DEFINE compiler option to assign a literal value to a compilation variable that is defined in the program by using the DEFINE directive with the PARAMETER phrase. The literal value provided for the compilation variable in the DEFINE option is sometimes referred to as a "parameter value" for the corresponding compilation variable. Compilation variables can be used within any of the conditional compilation directives, including DEFINE, EVALUATE, and IF. When a conditional compilation variable appears in a conditional compilation directive, it is treated as a symbolic reference to the literal value it currently represents.

The DEFINE compiler option provides a way for you to assign values to compilation variables from outside the program source. If that is not needed, it is sufficient to use the DEFINE directive within program source to define compilation variables.

DEFINE option syntax



Notes:

- ¹ You cannot use "=" if you are invoking the COBOL compiler from the z/OS UNIX shell via the cob2 command.

Default is: NODEFINE

Abbreviations are: DEF | NODEF

compilation-variable-name-1

The name of a compilation variable to be referenced in conditional compilation directives in the program. If no corresponding DEFINE directive with PARAMETER phrase exists for *compilation-variable-name-1* in the program, any instances of the DEFINE compiler option specified for that compilation variable are ignored. *compilation-variable-name-1* is formed according to the rules of a data-name user-defined word, except that DBCS characters are not allowed in the name. For details, see *User-defined words* in the *Enterprise COBOL for z/OS Language Reference*.

literal-1

The literal value that *compilation-variable-name-1* will represent symbolically in conditional compilation-related directives in the program. *literal-1* must be one of the following items:

- An alphanumeric literal, which can be specified as a regular alphanumeric literal ('abcd') or as a hex literal (x'F1F2F3'). National literals, UTF-8 literals, DBCS literals, and null-terminated alphanumeric literals (Z literals) are not supported.
- An integer literal.
- A boolean literal (only B'0' and B'1' are supported).

If *literal-1* is not specified, a value of B'1' will be assigned to the compilation variable. For example, if you specify:

```
DEFINE (foo)
```

foo will be assigned the value B'1'.

Multiple instances of the DEFINE option can be specified to define a value for multiple different compilation variables. If a single conditional compilation variable is defined more than once, the last definition of the variable will be used as the value of the corresponding conditional compilation variable. If NODEFINE appears after previous instances of the DEFINE option, the definitions for all conditional compilation variables are cancelled.

When DEFINE options are specified in CBL statements, they can be used only on the first program of a batch program. Therefore, if a file has multiple COBOL programs in it, there can be CBL statements with DEFINE options preceding the first program, but not the other programs. The DEFINE options specified for the first program (and DEFINE options specified in PARM of JCL or DEFINE options specified as cob2 command options under z/OS UNIX) apply to all programs in a file.

Related references

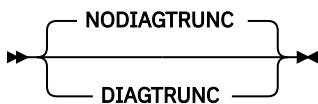
Conditional compilation (*Enterprise COBOL for z/OS Language Reference*)

DEFINE (*Enterprise COBOL for z/OS Language Reference*)

DIAGTRUNC

DIAGTRUNC causes the compiler to issue a severity-4 (Warning) diagnostic message for MOVE statements that have numeric receivers when the receiving data item has fewer integer positions than the sending data item or literal. In statements that have multiple receivers, the message is issued separately for each receiver that could be truncated.

DIAGTRUNC option syntax



Default is: NODIAGTRUNC

Abbreviations are: DTR | NODTR

The diagnostic message is also issued for implicit moves associated with statements such as these:

- INITIALIZE
- READ . . . INTO
- RELEASE . . . FROM
- RETURN . . . INTO
- REWRITE . . . FROM
- WRITE . . . FROM

The diagnostic message is also issued for moves to numeric receivers from alphanumeric data-names or literal senders, except when the sending field is reference modified.

There is no diagnostic message for COMP-5 receivers, nor for binary receivers when you specify the TRUNC(BIN) option.

Related concepts

[“Formats for numeric data” on page 47](#)

[“Reference modifiers” on page 110](#)

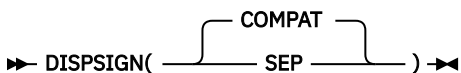
Related references

[“TRUNC” on page 409](#)

DISPSIGN

The DISPSIGN option controls output formatting for DISPLAY of signed numeric items.

DISPSIGN option syntax



Default is: DISPSIGN(COMPAT)

Abbreviations are: DS(C | S)

DISPSIGN(COMPAT)

If you specify DISPSIGN(COMPAT), formatting for displayed values of signed numeric items is compatible with prior versions of Enterprise COBOL. Overpunch signs are generated in some cases.

DISPSIGN(SEP)

If you specify DISPSIGN(SEP), the displayed values for signed binary, signed packed-decimal, or overpunch signed zoned-decimal items are always formatted with a leading separate sign.

The following example shows the DISPLAY output with the DISPSIGN (COMPAT) option or the DISPSIGN (SEP) option specified:

Table 48. DISPLAY output with the DISPSIGN (COMPAT) option or the DISPSIGN (SEP) option specified:

Data items	DISPLAY output with the DISPSIGN (COMPAT) option specified	DISPLAY output with the DISPSIGN (SEP) option specified
Unsigned binary	111	111
Positive binary	111	+111
Negative binary	11J	-111
Unsigned packed-decimal	222	222
Positive packed-decimal	222	+222
Negative packed-decimal	22K	-222
Zoned-decimal unsigned	333	333
Zoned-decimal trailing positive	33C	+333
Zoned-decimal trailing negative	33L	-333
Zoned-decimal leading positive	C33	+333
Zoned-decimal leading negative	L33	-333

DLL

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 or class definitions.

Note: The DLL option can be overridden for particular CALL statements by using the CALLINTERFACE directive.

DLL option syntax

Default is: NODLL

Abbreviations are: None

Link-edit considerations: COBOL programs that are compiled with the DLL option must be link-edited with the RENT and AMODE 31 link-edit options.

NODLL instructs the compiler to generate an object module that is not enabled for DLL usage.

AMODE 64 considerations: When the LP (64) compiler option is in effect, the DLL and NODLL options are effectively ignored because object files generated using LP (64) are always DLL enabled. They can be linked as DLL or non-DLL.

The EXPORTALL option is supported. Use this option to export symbols from programs when building DLLs.

Example

This example shows how to build, link and run two COBOL programs, PROG1 and SUB1, on z/OS UNIX. Program SUB1 is built as a DLL program and linked as a DLL called `mylibdll.so` so that resides in the current z/OS UNIX directory. Program PROG1 is essentially a client of DLL `mylibdll.so` and makes a call to program SUB1 in the DLL.

Note: In practice, a DLL such as `mylibdll.so` would typically consist of many COBOL programs linked together into a DLL.

The example consists of the following files:

buildrun.sh

z/OS UNIX shell script that builds and runs the example programs

prog1.cbl

COBOL program PROG1, which makes a call to a DLL program

sub1.cbl

COBOL program SUB1, which becomes part of a DLL `mylibdll.so`

The `buildrun.sh` script is as follows:

```
#!/bin/sh

# Remove existing artifact files
rm -f *.o *.lst *.so *.x prog1

# Build SUB1 as a DLL program and link it into a DLL called mylibdll.so,
# which will be written to the current directory
cob2 sub1.cbl -o libmydll.so "-qdll,exportall,rent,list" -bdll

# Build PROG1 with the DLL compiler option as it will be making a call
# to program SUB1 which resides in DLL mylibdll.so, but PROG1 itself
# is not linked as a DLL -- it is linked as a non-DLL program.
# Note that PROG1 must be linked with the side deck mylibdll.x produced
# in the previous step in order for the symbol SUB1 to be resolved
cob2 prog1.cbl -o prog1 libmydll.x "-qdll,list"

# Set LIBPATH to include the current directory so that it will be
# found at runtime when PROG1 is executed
export LIBPATH=.:$LIBPATH

echo ""

# Run PROG1
./prog1
```

Running the `buildrun.sh` script produces the following output:

```
PP 5655-EC6 IBM Enterprise COBOL for z/OS 6.4.0 in progress ...
End of compilation 1, program SUB1, no statements flagged.
PP 5655-EC6 IBM Enterprise COBOL for z/OS 6.4.0 in progress ...
End of compilation 1, program PROG1, no statements flagged.

Entered PROG1
Calling DLL routine SUB1 from PROG1
Entered DLL routine SUB1
Exited DLL routine SUB1
Exited PROG1
```

After PROG1 and SUB1 are compiled and linked, the following files exist in the current z/OS UNIX directory:

```
buildrun.sh  libmydll.x  prog1.cbl  prog1.o  sub1.lst
libmydll.so  prog1      prog1.lst  sub1.cbl  sub1.o
```

`prog1.cbl`:

```
identification division.
program-id. prog1.
data division.
```

```

working-storage section.
procedure division.
MainProgram.
    display "Entered PROG1"
    display "Calling DLL routine SUB1 from PROG1"
    call 'SUB1'
    display "Exited PROG1"
    goback.
end program prog1.

```

sub1.cbl:

```

identification division.
program-id. sub1.
data division.
working-storage section.
procedure division.
MainProgram.
    display 'Entered DLL routine SUB1'
    display 'Exited DLL routine SUB1'
    goback.
end program sub1.

```

Related tasks

[“Making dynamic calls” on page 536](#)

Related references

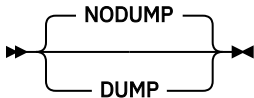
[“Conflicting compiler options” on page 334](#)

CALLINTERFACE (*Enterprise COBOL for z/OS Language Reference*)

DUMP

Use DUMP to produce a system dump at compile time for an internal compiler error.

DUMP option syntax



Default is: NODUMP

Abbreviations are: DU | NODU

Not for general use: The DUMP option should be used only 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 SYSDUMP.

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 compiled with OPTIMIZE (1 | 2), 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 (1 | 2) 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 does not represent a compiler error. Using the NODUMP option will allow the compiler to issue message IGYOP3124-W and continue processing.

Related tasks

[Language Environment Debugging Guide \(Understanding abend codes\)](#)

Related references

[“Conflicting compiler options” on page 334](#)

DYNAM

Use DYNAM to cause nonnested, separately compiled programs invoked through the CALL *literal* statement to be loaded for CALL, and deleted for CANCEL, dynamically at run time.

Note: The DYNAM option can be overridden for particular CALL statements by using the CALLINTERFACE directive.

CALL *identifier* statements always result in a runtime load of the target program and are not affected by this option.

DYNAM option syntax



Default is: NODYNAM

Abbreviations are: DYN | NODYN

Restriction: The DYNAM compiler option must not be used in the following cases:

- COBOL programs that are processed by the CICS translator or the CICS compiler option
- COBOL programs that have EXEC SQL statements and are run under CICS or Db2 call attach facility (CAF)

If your COBOL program calls programs that have been linked as dynamic link libraries (DLLs), you must not use the DYNAM option. You must instead compile the program with the NODYNAM and DLL options.

Related tasks

[“Making both static and dynamic calls” on page 540](#)

[“Choosing the DYNAM or NODYNAM compiler option” on page 502](#)

Related references

[“Conflicting compiler options” on page 334](#)

CALLINTERFACE (*Enterprise COBOL for z/OS Language Reference*)

EXIT

Use the EXIT option to provide user-supplied modules in place of various compiler functions.

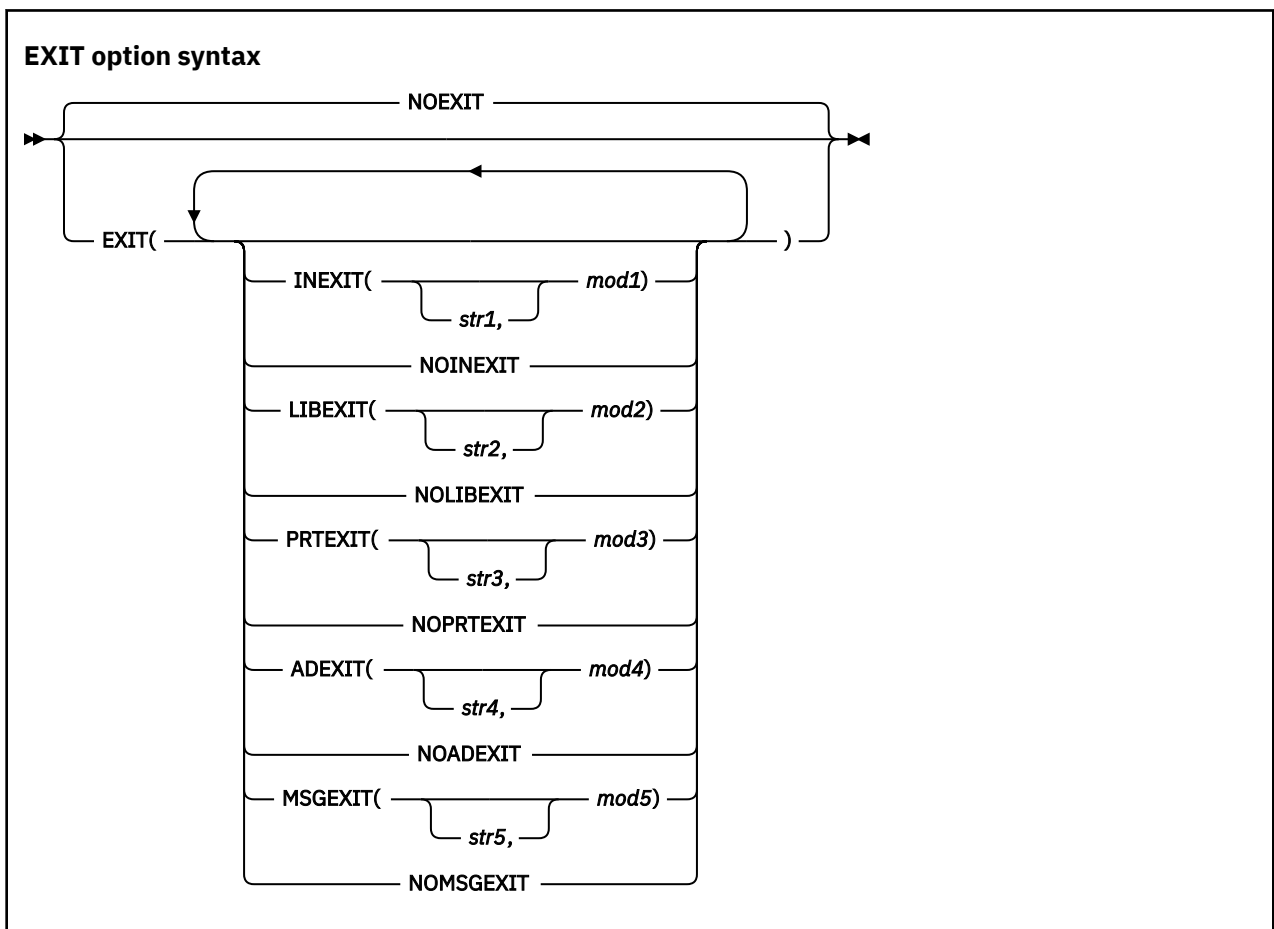
For compiler input, use the INEXIT and LIBEXIT suboptions to provide modules in place of SYSIN and SYSLIB (or copy library), respectively.

Note: If INEXIT or LIBEXIT is used, the 'Source' view for your debug session will not be available. If you choose the 'Show Source' feature of 'Switch View' with Debug for z/OS, it will not work because there is no source file.

For compiler output, use the PRTEXTIT suboption to provide a module in place of SYSPRINT.

To provide a module that will be called for each SYSADATA record immediately after the record has been written out to the file, use the ADEXIT suboption.

To customize compiler messages (change their severity or suppress them, including converting FIPS (FLAGSTD) messages to diagnostic messages to which you assign a severity), use the MSGEXIT suboption. The module that you provide to customize the messages will be called each time the compiler issues a diagnostic message or a FIPS message.



Default is: NOEXIT

Abbreviations are: NOEX | EX (INX | NOINX, LIBX | NOLIBX, PRTX | NOPRTX, ADX | NOADX, MSGX | NOMSGX)

You can specify the suboptions in any order, and can separate them by either commas or spaces. If you specify both the positive and negative form of a suboption, the form specified last takes effect. If you specify the same suboption more than once, the last one specified takes effect.

If you specify the EXIT option without specifying at least one suboption, NOEXIT will be in effect.

You can specify the EXIT option only at invocation in the JCL PARM field (under TSO/E, 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 program object (where *mod1* is the module name) instead of SYSIN.

LIBEXIT(['str2'],mod2)

The compiler obtains copybooks from a user-supplied program object (where *mod2* is the module name) instead of *library-name* or SYSLIB. For use with either COPY or BASIS statements.

PRTEXT(['str3'],mod3)

The compiler passes printer-destined output to the user-supplied program object (where *mod3* is the module name) instead of SYSPRINT.

ADEXIT(['str4'],mod4)

The compiler passes the SYSADATA output to the user-supplied program object (where *mod4* is the module name).

MSGEXIT(['str5'],mod5)

The compiler passes the message number, and passes the default severity of a compiler diagnostic message, or the category (as a numeric code) of a FIPS compiler message, to the user-supplied program object (where *mod5* is the module name).

The names *mod1*, *mod2*, *mod3*, *mod4*, and *mod5* can refer to the same module.

The suboptions *str1*, *str2*, *str3*, *str4*, and *str5* are character strings that are passed to the program object. These strings are optional. They can be up to 64 characters in length, and you must enclose them in a pair of apostrophes (' '). You can use any character in the strings, but any included apostrophes must be doubled (" "). Lowercase characters are folded to uppercase.

If one of *str1*, *str2*, *str3*, *str4*, or *str5* is specified, that string is passed to the appropriate user-exit module in the following format, where LL is a halfword (on a halfword boundary) that contains the length of the string.

LL *string*

[“Example: MSGEXIT user exit” on page 823](#)

Compiler exit modules that are specified on the EXIT option can be implemented either in an assembler language or in a high-level programming language such as COBOL. However, when exits are written in a Language Environment conforming programming language or Language Environment conforming assembler language, the exit must be reentrant. In addition, compiler exits must NOT be compiled with the LP(64) compiler option.

The Enterprise COBOL compiler automatically manages a preinitialized Language Environment at compile time, and calls compiler exits within this environment. Therefore, the following rules apply:

- Compiler exits are run as subprograms instead of main programs.
- Compiler exits must not include logic for explicitly initializing or terminating Language Environment. In particular, exits must not use the RTEREUS runtime option, the IGZERRE callable service, or the CEEPIPI callable service for environment management.
- Compiler exits must not use the STOP RUN statement.

Related references

[“Conflicting compiler options” on page 334](#)

[“FLAGSTD” on page 360](#)

[Appendix E, “EXIT compiler option,” on page 811](#)

EXPORTALL

Use EXPORTALL to instruct the compiler to automatically export the PROGRAM-ID name and each alternate entry-point name from each program definition when the object deck is link-edited to form a DLL.

EXPORTALL option syntax



Default is: NOEXPORTALL

Abbreviations are: EXP | NOEXP

With these symbols exported from the DLL, the exported program and entry-point names can be called from programs in the root program object, in other DLL program objects in the application, and from programs that are linked into that DLL.

Specification of the EXPORTALL option requires that the RENT linker option also be used.

NOEXPORTALL instructs the compiler to not export any symbols. In this case the programs are accessible only from other routines that are link-edited into the same program object as the COBOL program definition.

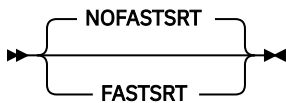
Related references

[“Conflicting compiler options” on page 334](#)

FASTSRT

Use FASTSRT to let IBM DFSORT, or an equivalent product, perform sort input and output instead of Enterprise COBOL. It applies only to sorting files by using the format 1 SORT (that is, file SORT) statement.

FASTSRT option syntax



Default is: NOFASTSRT

Abbreviations are: FSRT | NOFSRT

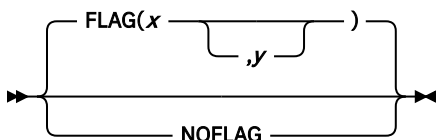
Related tasks

[“Improving sort performance with FASTSRT” on page 232](#)

FLAG

Use FLAG(*x*) to produce diagnostic messages at the end of the source listing for errors of a severity level *x* or above.

FLAG option syntax



Default is: FLAG(I, I)

Abbreviations are: F | NOF

x and y can be either I, W, E, S, or U.

Use FLAG(x, y) to produce diagnostic messages for errors of severity level x or above at the end of the source listing, with error messages of severity y and above to be embedded directly in the source listing. The severity coded for y must not be lower than the severity coded for x . To use FLAG(x, y), you must also specify the SOURCE compiler option.

Error messages in the source listing are set off by the embedding of the statement number in an arrow that points to the message code. The message code is followed by the message text. For example:

```
000413      MOVE CORR WS-DATE TO HEADER-DATE
==000413==>  IGYPS2121-S      " WS-DATE " was not defined as a data-name. . . .
```

When FLAG(x, y) is in effect, most messages of severity y and above are embedded in the listing after the line that caused the message. Messages with the IGYCB prefix will never be embedded in the source. (See the related reference below for information about messages for exceptions.)

Use NOFLAG to suppress error flagging. NOFLAG does not suppress error messages for compiler options.

Embedded messages

- Embedding level-U messages is not recommended. The specification of embedded level-U messages is accepted, but does not produce any messages in the source.
- The FLAG option does not affect diagnostic messages that are produced before the compiler options are processed.
- Diagnostic messages that are produced during processing of compiler options, CBL or PROCESS statements, or BASIS, COPY, or REPLACE statements are not embedded in the source listing. All such messages appear at the beginning of the compiler output.
- Diagnostic messages with the IGYCB prefix are not embedded in the source listing. All such messages appear at the end of the compiler output, regardless of the setting of the FLAG option.
- Messages that are produced during processing of the *CONTROL or *CBL statement are not embedded in the source listing.

Related references

[“Messages and listings for compiler-detected errors” on page 281](#)

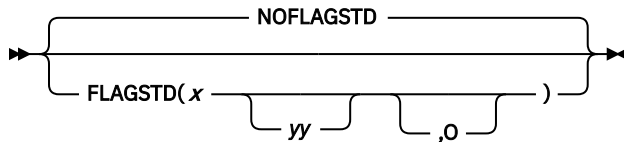
FLAGSTD

Use FLAGSTD to specify the level or subset of the 85 COBOL Standard to be regarded as conforming, and to get informational messages about the 85 COBOL Standard elements that are included in your program.

You can specify any of the following items for flagging:

- A selected Federal Information Processing Standard (FIPS) COBOL subset
- Any of the optional modules
- Obsolete language elements
- Any combination of subset and optional modules
- Any combination of subset and obsolete elements
- IBM extensions (these are flagged any time that FLAGSTD is specified, and identified as "nonconforming nonstandard")

FLAGSTD option syntax



Default is: NOFLAGSTD

Abbreviations are: None

`x` specifies the subset of the 85 COBOL Standard to be regarded as conforming:

M

Language elements that are not from the minimum subset are to be flagged as "nonconforming standard."

I

Language elements that are not from the minimum or the intermediate subset are to be flagged as "nonconforming standard."

H

The high subset is being used and elements will not be flagged by subset. Elements that are IBM extensions will be flagged as "nonconforming Standard, IBM extension."

`yy` specifies, by a single character or combination of any two, the optional modules to be included in the subset:

D

Elements from debug module level 1 are not flagged as "nonconforming standard."

N

Elements from segmentation module level 1 are not flagged as "nonconforming standard."

S

Elements from segmentation module level 2 are not flagged as "nonconforming standard."

If S is specified, N is included (N is a subset of S).

`O` (the letter) specifies that obsolete language elements are flagged as "obsolete."

The informational messages appear in the source program listing, and identify:

- The element as "obsolete," "nonconforming standard," or "nonconforming nonstandard" (a language element that is both obsolete and nonconforming is flagged as obsolete only)
- The clause, statement, or header that contains the element
- The source program line and beginning location of the clause, statement, or header that contains the element
- The subset or optional module to which the element belongs

FLAGSTD requires the standard set of reserved words.

In the following example, the line number and column where a flagged clause, statement, or header occurred are shown with the associated message code and text. After that is a summary of the total number of flagged items and their type.

LINE	COL	CODE	FIPS MESSAGE TEXT
		IGYDS8211	Comment lines before "IDENTIFICATION DIVISION": nonconforming nonstandard, IBM extension to ANS/ISO 1985.
11.14		IGYDS8111	"GLOBAL clause": nonconforming standard, ANS/ISO 1985 high subset.
59.12		IGYPS8169	"USE FOR DEBUGGING statement": obsolete element in ANS/ISO 1985.
FIPS MESSAGES TOTAL			
			STANDARD
			NONSTANDARD
			OBSOLETE
		3	1
			1
			1

You can convert FIPS informational messages into diagnostic messages, and can suppress FIPS messages, by using the MSGEXIT suboption of the EXIT compiler option. For details, see the related reference about the processing of MSGEXIT, and see the related task.

Related tasks

[“Customizing compiler-message severities” on page 821](#)

Related references

[“Conflicting compiler options” on page 334](#)

[“Processing of MSGEXIT” on page 819](#)

HGPR

The HGPR option controls the compiler usage of the 64-bit registers provided by IBM z/Architecture processors.



Default is: HGPR(PRESERVE)

Abbreviations are: None

The Enterprise COBOL compiler uses the 64-bit width of the IBM z/Architecture General Purpose Registers (GPRs). HGPR stands for "High-halves of 64-bit GPRs", which means the use of native 64-bit instructions.

HGPR(PRESERVE)

If you specify HGPR(PRESERVE), the compiler preserves the high halves of the 64-bit GPRs that a program is using, by saving them in the prolog for the function and restoring them in the epilog. The PRESERVE suboption is necessary only if the caller of the program is not Enterprise COBOL, Enterprise PL/I, or z/OS XL C/C++ compiler-generated code.

HGPR(NOPRESERVE)

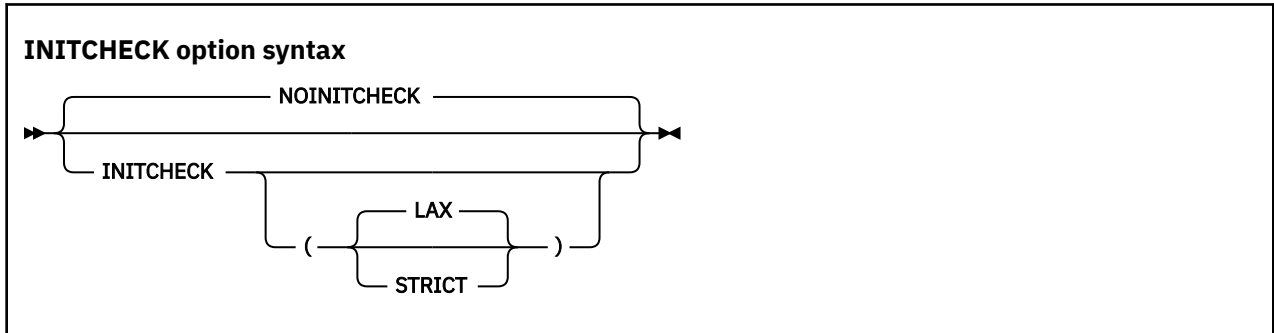
If you specify HGPR(NOPRESERVE), the compiler omits preserving the high-halves of the 64-bit GPRs that a program is using, which improves performance.

Related tasks

[“Using COBOL user exit routines with DFSORT” on page 237](#)

INITCHECK

Use the INITCHECK option to have the compiler check for uninitialized data items and issue warning messages when they are used without being initialized.



Default is: NOINITCHECK

Suboption default is: INITCHECK(LAX) if INITCHECK is specified with no suboption

Abbreviations are: IC | NOIC

NOINITCHECK

If NOINITCHECK is in effect, the compiler will not issue any warning messages for uninitialized data items.

INITCHECK(LAX | STRICT)

If INITCHECK or INITCHECK(LAX) is in effect, the compiler will check for uninitialized data items and issue a warning message when a data item is used without being initialized. However, if a data item is initialized on at least one logical path to a statement, no warning message will be issued.

If INITCHECK(STRICT) is in effect, the compiler will still check for uninitialized data items and issue a warning message when a data item is used without being initialized. However, unlike INITCHECK(LAX), INITCHECK(STRICT) will issue a warning message about uninitialized data for a data item used in a statement unless the data item is initialized on all logical paths to the statement.

Here is a sample program to illustrate the behavior differences between specifying INITCHECK(LAX) versus INITCHECK(STRICT). Y and Z represent some data items, with no value clauses:

```
PROCEDURE DIVISION.  
  IF Y > 5  
    MOVE 2 TO Z  
  END-IF  
  DISPLAY Z
```

Z is initialized on one path to the DISPLAY statement but not the other, so if INITCHECK(LAX) is in effect, a warning message will be issued for Y only, while INITCHECK(STRICT) will also issue a warning message for Z.

Restrictions:

- The INITCHECK option analyzes data items in the WORKING-STORAGE SECTION and LOCAL-STORAGE SECTION only. In particular, it does not analyze data items in the LINKAGE SECTION or FILE SECTION.
- The INITCHECK analysis does not track external or global data items.
- The INITCHECK analysis does not track individual elements in tables independently. Instead, if one element of a table is initialized, all corresponding elements of the table are considered to be initialized. This applies to both fixed-length and variable-length tables.
- The INITCHECK analysis does not track the initialization of items if it happens through a pointer. For example, if a pointer to an uninitialized data item is created by using ADDRESS-OF, and that data item is initialized through that pointer, the INITCHECK analysis might also issue a warning message.

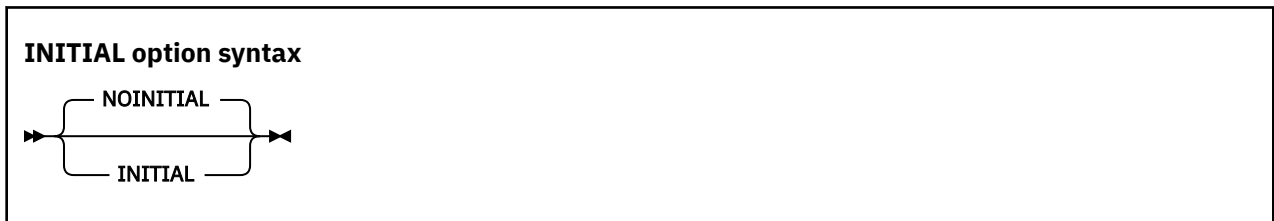
- For uninitialized data items being passed BY REFERENCE, no warning messages will be issued. However, the INITCHECK analysis will warn about uninitialized data items being passed BY CONTENT and BY VALUE.
- The INITCHECK option does not track individual bytes of reference-modified data items accurately. Instead, if a data item is accessed by using a reference modification, this data item is considered to be initialized.
- If a data item is in a group with other items that have had their address taken, for example, as the result of being an SQL host variable, then that data item will also be considered to have its address taken, and the set of all address taken data items is always considered to be set by any call to an external function.

Notes:

- All of the INITCHECK analyses occur at compile time only.
- The INITCHECK option has no effect on the behavior or performance of the program after it has been compiled.
- Use of the INITCHECK option might increase compile time and memory consumption.
- The INITCHECK option reports and prints only the first uninitialized data item in a group. Subsequent data items that are also uninitialized will not be printed.
- INITCHECK is more accurate when used with OPT(1) or OPT(2), but it is also helpful when used with OPT(0).

INITIAL

The INITIAL compiler option causes a program and all of its nested programs to behave as if the IS INITIAL clause was specified on the PROGRAM-ID paragraph.



Default is: NOINITIAL

Abbreviations are: None

INITIAL

INITIAL causes a program and all of its nested programs to behave as if the IS INITIAL clause was specified on the PROGRAM-ID paragraph.

Note: INITIAL and the IS INITIAL clause have no effect on data items that do not have VALUE clauses.

NOINITIAL

NOINITIAL will have no effect on programs that already have IS INITIAL on the PROGRAM-ID paragraph in the source.

Related tasks **Related tasks**

[“Setting a program to an initial state” on page 4](#)

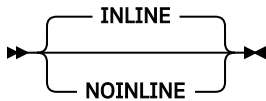
Related references

[“Conflicting compiler options” on page 334](#)

INLINE

The `INLINE` compiler option controls whether the inlining of procedures (paragraphs or sections) referenced by `PERFORM` statements in the source program is allowed.

INLINE option syntax



Default is: `INLINE`

Abbreviations are: `INL` | `NOINL`

INLINE

`INLINE` is a potential performance-boosting option.

Specifying `INLINE` causes the compiler to allow the inlining of procedures referenced by `PERFORM` statements in the source program when `OPTIMIZE(1)` or `OPTIMIZE(2)` is in effect. Whether or not to inline procedures in a specific `PERFORM` block is determined by the compiler.

*Inlining*¹ can be prevented for specific paragraphs by using the `>>INLINE ON` and `>>INLINE OFF` directives in the program source code when memory usage is a concern and there is a desire to reduce the overall size of the executable, such as in a CICS region.

Note:

1. The word *inlining* here implies that the compiler might choose to replace the `PERFORM` of a procedure (paragraph or section) with a copy of that procedure's code. By inserting the procedure code at the location of the `PERFORM`, the compiler saves the overhead of branching logic to and from the procedure.

NOINLINE

Specifying `NOINLINE` causes no inlining of procedures referenced by `PERFORM` statements, regardless of the optimization level in effect. `NOINLINE` can be specified in cases where there is a desire to reduce the overall size of the executable when memory usage is a concern, such as in a CICS region. `NOINLINE` cannot be overridden by using the `>>INLINE ON` directive.

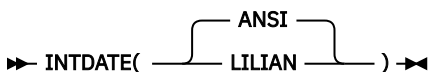
Related references

`INLINE` directive (*Enterprise COBOL for z/OS Language Reference*)

INTDATE

`INTDATE(ANSI)` instructs the compiler to use the 85 COBOL Standard starting date for integer dates used with date intrinsic functions. Day 1 is Jan 1, 1601. `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.

INTDATE option syntax



Default is: `INTDATE(ANSI)`

Abbreviations are: None

With `INTDATE(LILIAN)`, the date intrinsic functions return results that are compatible with the Language Environment date callable services.

Usage note: When INTDATE (LILIAN) is in effect, CEECBLDY is not usable because you have no way to turn an ANSI integer into a meaningful date by using either intrinsic functions or callable services. If you code a CALL *literal* statement with CEECBLDY as the target of the call when INTDATE (LILIAN) in effect, the compiler diagnoses this and converts the call target to CEEDAYS.

Related tasks

[“Using date callable services” on page 59](#)

INVDATA

The INVDATA option tells the compiler whether the data in USAGE DISPLAY and PACKED-DECIMAL data items is valid, and if not, what the behavior of the compiler should be.

Because most users have valid data in their USAGE DISPLAY and USAGE PACKED-DECIMAL data items, they should use NOINVDATA, even if they use NUMPROC (NOPFD). Even if you find that your programs are processing invalid data at run time with the NUMCHECK compiler option, you should change your programs to avoid processing invalid data and use NOINVDATA.

Note: The goal of the INVDATA option is to provide a behavior that is as compatible as possible with the behavior of programs compiled with COBOL 4 or earlier versions in cases of invalid numeric data. When discrepancies are found, this option will be updated in favor of making the behavior more closely match the behavior of COBOL 4 or earlier versions.

When the INVDATA option is in effect, the compiler will avoid performing known optimizations that might produce a different result than COBOL 4 or earlier versions when a zoned decimal or packed decimal data item has invalid digits or an invalid sign code, or when a zoned decimal data item has invalid zone bits.

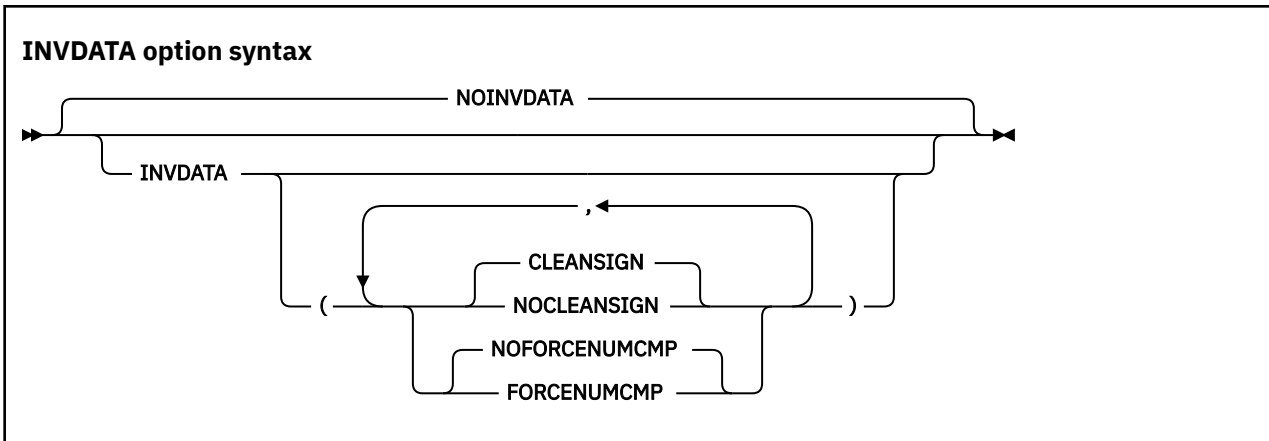
The following table provides a quick reference on how to set the INVDATA and NUMPROC options when migrating to COBOL 6.2 or later versions from earlier versions of COBOL, depending on the default value of the NUMPROC option that was used in the earlier version of COBOL and whether or not you have invalid data.

COBOL versions	Invalid data present?	NUMPROC/ZONEDATA used in COBOL 6.1 or earlier versions	INVDATA and NUMPROC settings in COBOL 6.2 or later versions
Pre-COBOL 5	No	NUMPROC (MIG)	NOINVDATA, NUMPROC (NOPFD)
Pre-COBOL 5	No	NUMPROC (NOPFD)	NOINVDATA, NUMPROC (NOPFD)
Pre-COBOL 5	No	NUMPROC (PFD)	NOINVDATA, NUMPROC (PFD)
Pre-COBOL 5	Yes	NUMPROC (MIG)	INVDATA (FORCENUMCMP, NOCLEANSIGN), NUMPROC (NOPFD)
Pre-COBOL 5	Yes	NUMPROC (NOPFD)	INVDATA (NOFORCENUMCMP, CLEANSTGN), NUMPROC (NOPFD) or INVDATA, NUMPROC (NOPFD)
Pre-COBOL 5	Yes	NUMPROC (PFD)	INVDATA (NOFORCENUMCMP, CLEANSTGN), NUMPROC (PFD) or INVDATA, NUMPROC (PFD)
COBOL 5 or later	No	ZONEDATA (PFD)	NOINVDATA

Table 49. Setting INVDATA and NUMPROC options when migrating from earlier COBOL versions (continued)

COBOL versions	Invalid data present?	NUMPROC/ZONEDATA used in COBOL 6.1 or earlier versions	INVDATA and NUMPROC settings in COBOL 6.2 or later versions
COBOL 5 or later	Yes	ZONEDATA (NOPFD)	INVDATA (NOFORCENUMCMP, CLEANSIGN) or simply INVDATA
COBOL 5 or later	Yes	ZONEDATA (MIG)	INVDATA (FORCENUMCMP, CLEANSIGN) ¹

1. INVDATA (FORCENUMCMP, NOCLEANSIGN) is a closer representation of the pre-COBOL 5 NUMPROC (MIG) behavior than INVDATA (FORCENUMCMP, CLEANSIGN) when invalid data is present. If you are not satisfied with the behavior of ZONEDATA (MIG) in COBOL 5 or later versions when invalid data is present, then consider using INVDATA (FORCENUMCMP, NOCLEANSIGN) to more closely mimic the pre-COBOL 5 NUMPROC (MIG) behavior when invalid data is present.



Default is: NOINVDATA

The default is INVDATA (NOFORCENUMCMP, CLEANSIGN) when INVDATA is specified without suboptions.

Abbreviations are:

- INVD | NOINVD
- CS | NOCS
- FNC | NOFNC

NOINVDATA

When the NOINVDATA option is in effect, the compiler assumes that all data in USAGE DISPLAY and PACKED-DECIMAL data items is valid, and generates the most efficient code possible. For example, the compiler might generate a string comparison to avoid numeric conversion.

INVDATA (NOFORCENUMCMP | FORCENUMCMP)

NOFORCENUMCMP is the default when INVDATA is specified and neither FORCENUMCMP nor NOFORCENUMCMP is specified.

When the INVDATA (FORCENUMCMP) option is in effect, the compiler generates instructions to do comparisons of zoned decimal data items that ignore the zone bits of each digit. For example, the zoned decimal value is converted to packed-decimal with a PACK instruction before the comparison.

When the INVDATA (NOFORCENUMCMP) option is in effect, the compiler generates instructions for numeric comparisons or an alphanumeric comparison of zoned decimal data in the same manner as COBOL 4 or earlier versions does when you use NUMPROC (NOPFD | PFD) with COBOL 4 or earlier versions:

- In the cases where COBOL 4 or earlier versions considered the zone bits, the compiler generates an alphanumeric comparison which will also consider the zone bits of each digit in zoned decimal data items. The zoned decimal value remains as zoned decimal.
- In the cases where COBOL 4 or earlier versions ignored the zone bits, the compiler generates numeric comparisons that ignore the zone bits of each digit in zoned decimal data items. The zoned decimal value is converted to packed-decimal with a PACK instruction before the comparison.

In order for the compiler to handle zone bits in the same way as COBOL 4 or earlier versions do when generating comparisons of zoned decimal data, the NUMPROC suboption used in COBOL 6 must match the NUMPROC suboption used in COBOL 4 or earlier versions:

- To get COBOL 4 or earlier versions NUMPROC (NOPFD) behavior in COBOL 6, use INVDATA (NOFORCENUMCMP) (or simply INVDATA) and NUMPROC (NOPFD) in COBOL 6.
- To get the COBOL 4 or earlier versions NUMPROC (PFD) behavior in COBOL 6, use INVDATA (NOFORCENUMCMP) (or simply INVDATA) and NUMPROC (PFD) in COBOL 6.

Note: The sign code must be a valid sign code according to the NUMPROC option setting. In addition, the low-order byte must have a valid zone (x'F') with either SIGN IS LEADING or SIGN IS SEPARATE.

INVDATA(CLEANSIGN | NOCLEANSIGN)

CLEANSIGN is the default when INVDATA is specified and neither CLEANSIGN nor NOCLEANSIGN is specified.

When the INVDATA (CLEANSIGN) option is in effect, the compiler generates code to clean the sign nibble of USAGE DISPLAY and USAGE PACKED-DECIMAL data items on input to compare, add, subtract, multiply, and divide operations.

Note: CLEANSIGN does not apply to USAGE DISPLAY items defined with SIGN IS SEPARATE.

When the INVDATA (NOCLEANSIGN) option is in effect, the compiler avoids generating code to clean the sign nibble of USAGE DISPLAY and USAGE PACKED-DECIMAL data items on input to compare, add, subtract, multiply, and divide operations, increasing the probability of a SOC7 abend when one of the operands of the operation contains an invalid sign nibble.

Note: The INVDATA option affects the behavior of MOVE statements, comparisons, and computations for USAGE DISPLAY or PACKED-DECIMAL data items that could contain invalid digits, an invalid sign code, or invalid zone bits.

In the following example, you can see a data item with an invalid zone bit 4 in the zone bits in the middle of data item VALUE1, forced in by REDEFINES:

```

77 VALUE0    PIC X(4) VALUE '00 0'.           *>  x'F0F040F0'
77 VALUE1    REDEFINES VALUE0 PIC 9(4).
PROCEDURE DIVISION.
  IF VALUE1 = ZERO
    DISPLAY 'INVDATA(FORCENUMCMP) is in effect ' VALUE1
  ELSE
    DISPLAY 'INVDATA(NOFORCENUMCMP) is in effect ' VALUE1
  END-IF

```

In this example,

- With COBOL 4 or earlier versions, the test is true if the NUMPROC (MIG) option is used, and false for NUMPROC (NOPFD | PFD).
- With 6.2 or later versions:
 - When you use NOINVDATA, the test is true at OPT (0) and false at OPT (1 | 2).
 - When you use INVDATA (NOFORCENUMCMP), the test is false at any OPT setting.

In all, to ease your migration to COBOL 6:

- If your digits, sign codes, and zone bits are valid, use NOINVDATA, and if you used NUMPROC (PFD) or NUMPROC (NOPFD) in COBOL 4 or earlier versions, then use the same NUMPROC setting when you use

COBOL 6; if you used NUMPROC (MIG) in COBOL 4 or earlier versions, then use NUMPROC (NOPFD) when you use COBOL 6.

- If you have invalid digits, invalid sign codes, or invalid zone bits in your data, change your programs or systems so that your programs do not have invalid data in numeric data items at run time. Once you have corrected your programs or systems, you can use the preferred NOINVDATA option. Only if you cannot contain this work and must continue to run with invalid data, consider the following choices for INVDATA:
 - If you are migrating from COBOL 4 or earlier versions and used NUMPROC (MIG) with COBOL 4, then use INVDATA (FORCENUMCMP , NOCLEANSIGN) and NUMPROC (NOPFD) with COBOL 6.2 or later versions to most closely mimic the previous NUMPROC (MIG) behavior when invalid data is present.
 - If you are migrating from COBOL 4 or earlier versions and used NUMPROC (NOPFD) with COBOL 4, then use INVDATA (NOFORCENUMCMP , CLEAN SIGN) (or simply INVDATA) and NUMPROC (NOPFD) with COBOL 6.
 - If you are migrating from COBOL 4 or earlier versions and used NUMPROC (PFD) with COBOL 4, then use INVDATA (NOFORCENUMCMP , CLEAN SIGN) (or simply INVDATA) and NUMPROC (PFD) with COBOL 6.

Notes:

- If you completed migration from COBOL 4 or earlier versions to COBOL 5 or 6 in the past and used the deprecated ZONEDATA (MIG) option in COBOL 5 or 6 and are satisfied with the behavior, use INVDATA (FORCENUMCMP , CLEAN SIGN) instead of ZONEDATA (MIG).
- It is not always possible to entirely match the behavior of the old compiler even with these options when faced with clearly invalid data. For example, even for comparisons, INVDATA (NOFORCENUMCMP) does not give the same result in all cases as COBOL 4.

Performance considerations: NOINVDATA gives better runtime performance than INVDATA does. INVDATA disables some of the optimizations that NUMPROC (PFD) can give.

related tasks

[“Checking for incompatible data \(numeric class test\)” on page 53](#)

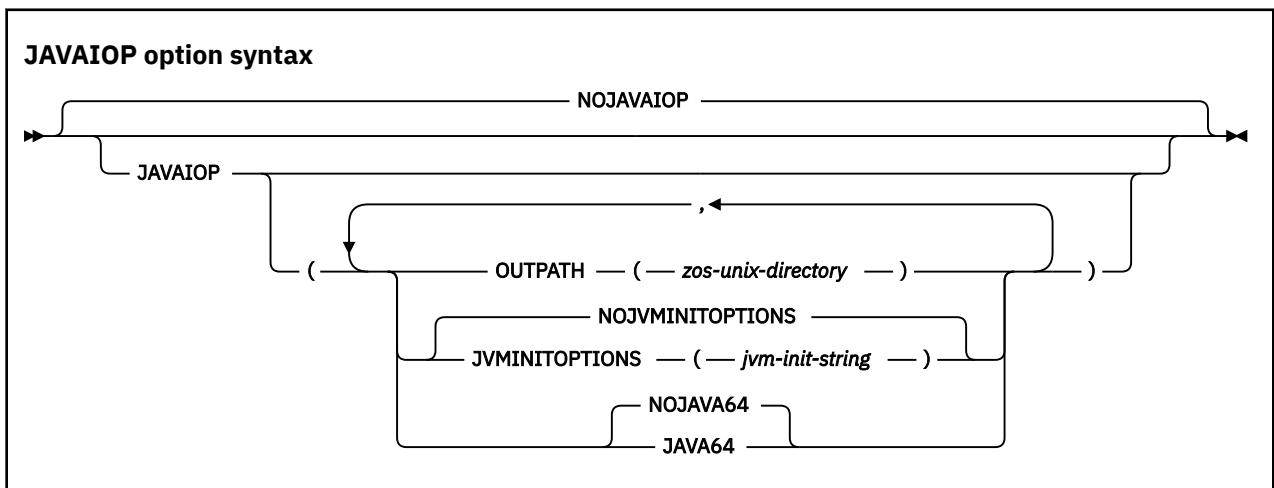
related references

[“NUMCHECK” on page 378](#)

[“NUMPROC” on page 382](#)

JAVAIOP

The JAVAIOP option controls the behavior of COBOL programs that interoperate with Java through the JAVA-CALLABLE or JAVA-SHAREABLE directives or by calling Java static methods using the CALL statement.



Default is: NOJAVAIOP

Abbreviations are:

- JIOP = JAVA IOP
- NOJIOP = NOJAVA IOP
- OP = OUTPATH
- JVMI = JAVAINITOPTIONS
- NOJVMI = NOJAVAINITOPTIONS

OUTPATH(*zos-unix-directory*)

The default is the current working directory for the userid under which the compiler is running. If the compiler is not invoked from z/OS UNIX, the current working directory is the home directory for the userid under which the compiler is running. *zos-unix-directory* must exist prior to running the compiler.

The OUTPATH suboption is used to specify the z/OS UNIX directory where the artifact files will be generated by COBOL programs that contain one of the following:

- The JAVA-CALLABLE directive
- The JAVA_SHAREABLE directive
- A CALL statement with a literal call target of the form '*Java.java-class-name.java-static-method-name*'.

Artifact files that are generated include the following:

- The COBOL native method stub for programs that contain the JAVA-CALLABLE directive
- WORKING-STORAGE initialization stubs that are generated for programs that use the JAVA-SHAREABLE directive
- Java call stub files generated by the compiler when the COBOL application contains CALL statements with a literal call target of the form '*Java.java-class-name.java-static-method-name*'

The maximum supported length of *zos-unix-directory* is 1023 characters.

NOJAVAINITOPTIONS | JAVAINITOPTIONS(*jvm-init-string*)

jvm-init-string specifies the initialization string that is to be used to initialize the Java virtual machine (JVM). This is only relevant in scenarios where the application starts on the COBOL side and then makes a call to Java, which requires the COBOL side of the application to start the JVM.

The default is `-Djava.library.path=.`, which indicates that the DLL that contains COBOL native method stubs and working-storage initialization stubs are located in the current directory where the Java application is running.

The maximum length of *jvm-init-string* is 256 characters.

NOJAVA64 | JAVA64

The default is NOJAVA64. Specifying the JAVA64 suboption when LP(32) is in effect indicates that the COBOL program will interoperate with 64-bit Java. As a result, Java object references will be treated as 8 bytes instead of 4 bytes.

Note: The JAVA IOP compiler option can only be specified for the first program in a sequence of programs (batch compilation). If you specify JAVA IOP in a CBL or PROCESS statement for a program other than the first program in the sequence, you will receive a compiler diagnostic message.

Related tasks

CALL statement (*Enterprise COBOL for z/OS Language Reference*)

JAVA-CALLABLE (*Enterprise COBOL for z/OS Language Reference*)

JAVA-SHAREABLE (*Enterprise COBOL for z/OS Language Reference*)

LANGUAGE

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).

LANGUAGE option syntax

►► LANGUAGE(*name*) ◄◄

Default is: LANGUAGE (ENGLISH)

Abbreviations are: LANG (EN | UE | JA | JP)

name specifies the language for compiler output messages. Possible values for the LANGUAGE option are shown in the table below.

Table 50. Values of the LANGUAGE compiler option

Name	Abbreviation ¹	Output language
ENGLISH	EN	Mixed-case English (the default)
JAPANESE ³	JA, JP	Japanese, using the Japanese character set
UENGLISH ^{2, 3}	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.
3. To change to uppercase English or Japanese compiler messages, in addition to using the LANGUAGE compiler option, you must also set the Language Environment runtime option NATLANG at compile time. We recommend using CEEOPTS DD in the compile JCL.

For example, to change messages to Japanese, use the LANGUAGE (JA) compiler option and also specify the NATLANG LE runtime option at compile time:

```
//CEEOPTS DD *
           NATLANG (JPN)
/*
```

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 runtime option allows you to control the national language to be used for the runtime environment, including error messages, month names, and day-of-the-week names. The LANGUAGE compiler option and the NATLANG runtime option act independently of each other. You can use them together with neither taking precedence over the other.

LINECOUNT

Use LINECOUNT(*nnn*) to specify the number of lines to be printed on each page of the compilation listing, or use LINECOUNT(0) to suppress pagination.

LINECOUNT option syntax

►► LINECOUNT(*nnn*) ◄◄

Default is: LINECOUNT (60)

Abbreviations are: LC

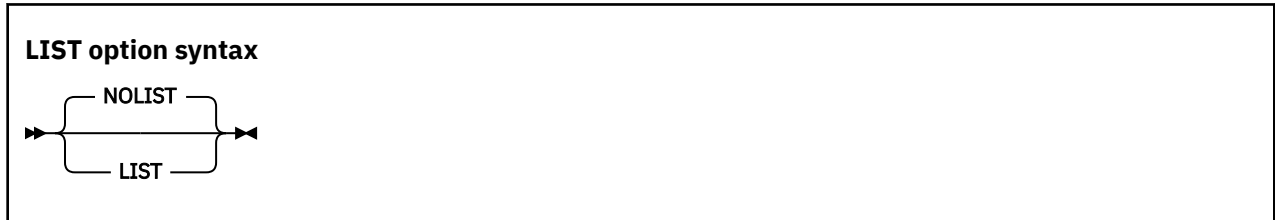
nnn must be an integer between 10 and 255, or 0.

If you specify `LINECOUNT(0)`, no page ejects are generated in the compilation listing.

The compiler uses three lines of *nnn* for titles. For example, if you specify `LINECOUNT(60)`, 57 lines of source code are printed on each page of the output listing.

LIST

Use the LIST compiler option to produce a listing of the assembler-language expansion of your source code.



Default is: NOLIST

Abbreviations are: None

These items will also be written to the output listing:

- Constant area
- Program prolog areas (PPA1, PPA2, PPA3, PPA4)
- Time stamp, compiler version, and build level information
- Compiler options and program information
- Base locator table
- External symbols dictionary
- Initial heap storage maps
- Stack storage maps

The output is generated if:

- You specify the `COMPILE` option, or the `NOCOMPILE(x)` option is in effect and an error of 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` (or `*CBL`) `LIST` or `NOLIST` statements in the `PROCEDURE DIVISION`. Source statements that follow a `*CONTROL NOLIST` statement are not included in the listing until a subsequent `*CONTROL LIST` statement switches the output back to normal `LIST` format.

Related tasks

[“Getting listings” on page 443](#)

Related references

[“Conflicting compiler options” on page 334](#)

`*CONTROL (*CBL) statement` (*Enterprise COBOL for z/OS Language Reference*)

LP

Use the LP compiler option to indicate whether a AMODE 31 (31-bit) or AMODE 64 (64-bit) program should be generated with the related language features enabled.

LP option syntax

A diagram showing the LP option syntax. It consists of the text "LP(" followed by a horizontal line. Above the line is the number "32" and below the line is the number "64". A bracket connects the two numbers. The line ends with a closing parenthesis ")", followed by a right-pointing arrowhead.

Default is: LP(32)

Abbreviations are: None

LP(32)

Indicates that an AMODE 31 (31-bit) program should be generated with the related language features enabled.

LP(64)

Indicates that an AMODE 64 (64-bit) program should be generated with the related language features enabled.

Option specification: You can specify the LP option in the ways that you specify other compiler options. However, if you specify the option in a CBL or PROCESS statement, you can only specify the LP option for the first program in a sequence of programs (batch compilation). You cannot change the value of the option for subsequent programs in the batch.

Runtime consideration: Currently, Language Environment does not support mixing AMODE 64 and AMODE 31 programs in the same application. If one program is compiled with LP(64), all programs within the application should also be compiled with LP(64). For static CALLs, the binder will issue a message if it encounters mixing addressing modes during external name resolution. For dynamic CALLs, you would receive a run time error for a CALL between programs if one is AMODE 64 and one is AMODE 31 or 24.

When using the LP(64) compiler option, the compilation process includes a component that runs in POSIX(ON) mode. This implies that there must be an OMVS Segment established in RACF (or equivalent in RACF alternatives) for each user executing the compiler with this option.

Restrictions for programs with LP(64):

- Programs compiled with LP(64) cannot contain XML GENERATE or XML PARSE statements, JSON GENERATE or JSON PARSE statements, object-oriented COBOL statements, ALTER statements, GO TO. statements, and DISPLAY . . . UPON SYSPUNCH statements.
- AMODE 64 programs cannot run in CICS or IMS.
- Compiler exits (using the EXIT compiler option) must NOT be compiled with the LP(64) compiler option.
- When LP(64) is in effect, both the DLL and NODLL options are ignored without warning, and NODLL will always appear in the listing.

Note: With LP(64), some compiler options are not applicable. For more information, see [“Using compiler options to compile AMODE 64 programs”](#) on page 522.

related references

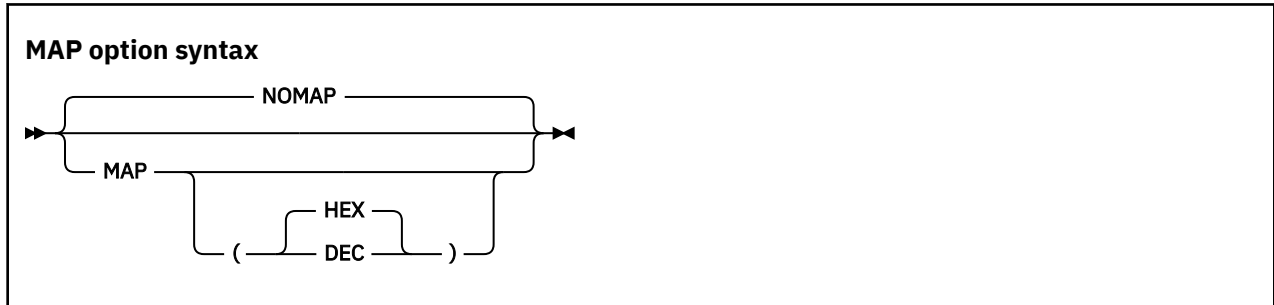
[“Specifying compiler options under z/OS”](#) on page 273

[“Specifying compiler options in the PROCESS \(CBL\) statement”](#) on page 274

[“Compiling multiple programs \(batch compilation\)”](#) on page 276

MAP

Use the MAP option to create a listing of the DATA DIVISION items and all implicitly declared items. You can also specify whether hexadecimal or decimal offsets are shown for MAP output in the listing.



Default is: NOMAP

Suboption default is: MAP (HEX) if MAP is specified with no suboption

Abbreviations are: None

HEX

If you specify MAP (HEX), data item offsets within groups will be in hexadecimal notation.

DEC

If you specify MAP (DEC), data item offsets within groups will be in decimal notation.

The output includes the following items:

- DATA DIVISION map
- Nested program structure map, and program attributes
- Size of the program's WORKING-STORAGE and LOCAL-STORAGE 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 that follow *CONTROL NOMAP are not included in the listing until a *CONTROL MAP statement switches the output back to normal MAP format. For example:

```
*CONTROL NOMAP          *CBL NOMAP
  01 A                   01 A
  02 B                   02 B
*CONTROL MAP            *CBL MAP
```

When the MAP (HEX | DEC) option is in effect, you also get an embedded MAP report in the source code listing. The condensed MAP information is shown to the right of data-name definitions in the WORKING-STORAGE SECTION, FILE SECTION, LOCAL-STORAGE SECTION, and LINKAGE SECTION of the DATA DIVISION. When both XREF data and an embedded MAP summary are on the same line, the embedded MAP summary is listed first.

[“Example: MAP output” on page 449](#)

Related concepts

[Chapter 21, “Debugging,” on page 433](#)

Related tasks

[“Getting listings” on page 443](#)

Related references

*CONTROL (*CBL) statement (*Enterprise COBOL for z/OS Language Reference*)

MAXPCF

Use the MAXPCF option to specify a maximum program complexity factor value. The program complexity factor (PCF) is computed by the compiler and the computed value is in the listing file. If the PCF of your program exceeds the maximum value, the compiler will automatically reduce the optimization level to speed up the compilation and use less storage. Therefore, when you compile a suite of programs, you do not have to specify an OPTIMIZE option value for each program.

MAXPCF option syntax

►► MAXPCF(*n*) ◄◄

Default is: MAXPCF(100000)

Abbreviations are: None

n must be an integer of 0 - 999999.

The aspects of the program taken into consideration when computing the complexity factor include:

- The number of COBOL statements in the PROCEDURE DIVISION, including generated statements from the CICS, SQL or SQLIMS options, and the expansion of COPY and REPLACE statements
- Initialization operations for WORKING-STORAGE or LOCAL-STORAGE data items with value clauses
- Operations for variable-length groups or subgroups in the DATA DIVISION, which compute their size at run time

Note: PCF is not a metric to measure how complex a program is. It is merely a count of COBOL items that can cause problems for optimization when there are a lot of them. To measure program complexity, you should use something like the [Metrics](#) feature provided by IBM Developer for z/OS.

For large and complex programs, you can use the MAXPCF option to set a threshold on the program complexity that the compiler attempts to optimize. Lower the MAXPCF value to reduce the optimization level, hence the compiler needs less memory and compilation time. Raise the MAXPCF value to attempt to optimize the programs at the cost of longer compilation time.

If you specify MAXPCF(0), no limit is enforced on the complexity of the program, and the MAXPCF option has no effect.

If you specify MAXPCF(*n*) and *n* is not zero, when the program complexity factor exceeds *n*, any specification of OPTIMIZE(1) or OPTIMIZE(2) is reset to OPTIMIZE(0), and a warning message is generated.

If the COBOL source file contains a sequence of source programs (a batch compile), the MAXPCF limit is applied on a per program basis.

Notes:

- If the OPTIMIZE(1) or OPTIMIZE(2) option is set at installation time as a fixed, nonoverridable option, then MAXPCF(*n*) with a nonzero *n* is an option conflict. In this case, the OPTIMIZE option takes precedence and the MAXPCF(0) option is forced on.
- If you attempt to optimize a program larger than the default threshold by raising the value of MAXPCF to *n* where *n* is greater than the default, or by specifying MAXPCF(0), the compiler might take excessive time to compile or fail to compile because of insufficient memory.

Related references

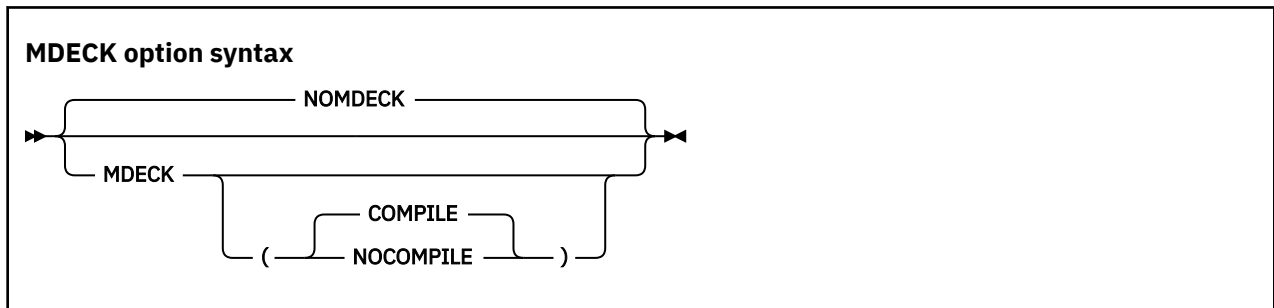
[“OPTIMIZE” on page 385](#)

MDECK

The MDECK compiler option specifies that a copy of the updated input source after library processing (that is, the result of COPY, BASIS, REPLACE, EXEC SQL INCLUDE, EXEC SQLIMS INCLUDE, and conditional compilation directive statements) is written to a file.

If Enterprise COBOL is running under z/OS UNIX, the MDECK output is written in the current directory to a file that has the same name as the COBOL source file and a suffix of .dek. For Enterprise COBOL running under TSO or batch, the MDECK output is written to the data set defined by the SYSDMDECK DD allocation, which must specify an MVS data set that has RECFM F or FB and an LRECL of 80 bytes.

Note: When compiling under z/OS TSO or batch, the COBOL compiler requires the SYSDMDECK data set allocation for all compilations, no matter if you specify the MDECK or NOMDECK option. The SYSDMDECK DD allocation can be specified as either a temporary dataset or a permanent cataloged/uncataloged data set.



Default is: NOMDECK

Abbreviations are: NOMD | MD | MD(C | NOC)

Option specification:

You cannot specify the MDECK option in a PROCESS (or CBL) statement. You can specify it only in one of the following ways:

- In an OPTFILE (as long as the OPTFILE is not specified in a PROCESS or CBL statement)
- In the PARM parameter of JCL
- As a cob2 command option
- As an installation default
- In the **COBOPT** environment variable

Suboptions:

- When MDECK(COMPILE) is in effect, compilation continues normally after library processing and generation of the MDECK output file have completed, subject to the settings of the COMPILE | NOCOMPILE, DECK | NODECK, and OBJECT | NOOBJECT compiler options.
- When MDECK(NOCOMPILE) is in effect, compilation is terminated after library processing has completed and the expanded source program file has been written. The compiler does no further syntax checking or code generation regardless of the settings of the COMPILE, DECK, and OBJECT compiler options.

If you specify MDECK with no suboption, MDECK(COMPILE) is implied.

Contents of the MDECK output file:

If you use the MDECK option with programs that contain EXEC CICS, EXEC SQL, or EXEC SQLIMS statements, these EXEC statements are included in the MDECK output as is. However, if you compile using the SQL or SQLIMS option, the corresponding EXEC SQL INCLUDE or EXEC SQLIMS INCLUDE statements are expanded in the MDECK output.

CBL, PROCESS, *CONTROL, and *CBL card images are passed to the MDECK output file in the proper locations.

For a batch compilation (multiple COBOL source programs in a single input file), a single MDECK output file that contains the complete expanded source is created.

Any SEQUENCE compiler-option processing is reflected in the MDECK file.

COPY statements are included in the MDECK file as comments.

Related tasks

[“Starting the compiler from an assembler program” on page 264](#)

[“Defining the library-processing output file \(SYSMDECK\)” on page 273](#)

Related references

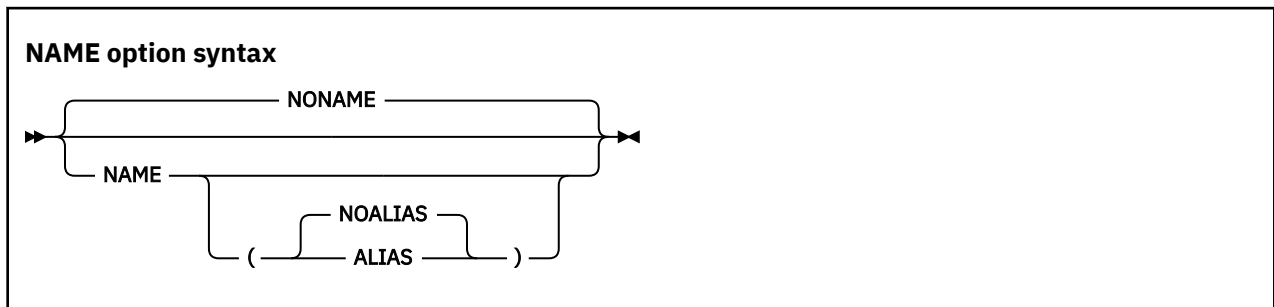
[“Conflicting compiler options” on page 334](#)

[Chapter 20, “Compiler-directing statements,” on page 429](#)

NAME

Use NAME to generate a link-edit NAME card for each object module. You can also use NAME to generate names for each program object when you are doing batch compilations.

When NAME is specified, a NAME card is appended to each object module that is created. Program object names are formed using the rules for forming module names from PROGRAM-ID statements.



Default is: NONAME, or NAME (NOALIAS) if only NAME is specified

Abbreviations are: None

If you specify NAME (ALIAS), and your program contains ENTRY statements, a link-edit ALIAS card is generated for each ENTRY statement.

Related references

[PROGRAM-ID paragraph \(*Enterprise COBOL for z/OS Language Reference*\)](#)

NSYMBOL

The NSYMBOL option controls the interpretation of the N symbol used in literals and PICTURE clauses, indicating whether national or DBCS processing is assumed.



Default is: NSYMBOL (NATIONAL)

Abbreviations are: NS(NAT | DBCS)

With NSYMBOL (NATIONAL):

- Data items defined with a PICTURE clause that consists only of the symbol N without the USAGE clause are treated as if the USAGE NATIONAL clause is specified.
- Literals of the form N" . . ." or N' . . .' are treated as national literals.

With NSYMBOL (DBCS):

- Data items defined with a PICTURE clause that consists only of the symbol N without the USAGE clause are treated as if the USAGE DISPLAY-1 clause is specified.
- Literals of the form N" . . ." or N' . . .' are treated as DBCS literals.

The NSYMBOL (DBCS) option provides compatibility with previous releases of IBM COBOL, and the NSYMBOL (NATIONAL) option makes the handling of the above language elements consistent with the 2002 COBOL Standard in this regard.

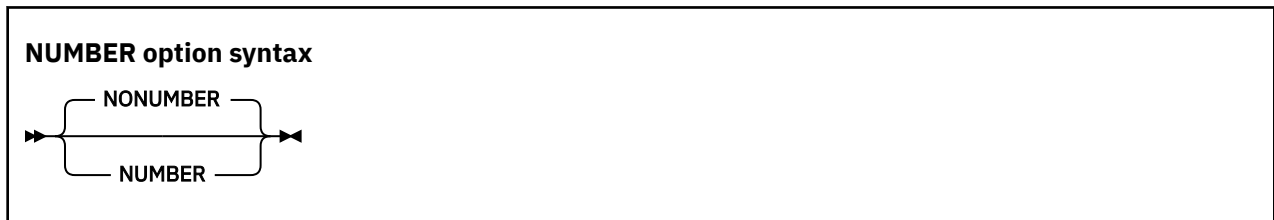
NSYMBOL (NATIONAL) is recommended for applications that use Unicode data or object-oriented syntax for Java interoperability.

Related references

[“Conflicting compiler options” on page 334](#)

NUMBER

Use the NUMBER compiler option if you have line numbers in your source code and want those numbers to be used in error messages and SOURCE, MAP, LIST, and XREF listings.



Default is: NONNUMBER

Abbreviations are: NUM | NONUM

If you request NUMBER, the compiler checks columns 1 through 6 to make sure that they contain only numbers and that the numbers are in numeric collating sequence. (In contrast, SEQUENCE checks the characters in these columns according to EBCDIC collating sequence.) When a line number is found to be out of sequence, the compiler assigns to it a line number with a value one higher than the line number of the preceding statement. The compiler flags the new value with two asterisks and includes in the listing a message indicating an out-of-sequence error. Sequence-checking continues with the next statement, based on the newly assigned value of the previous line.

If you use COPY statements and NUMBER is in effect, be sure that your source program line numbers and the copybook line numbers are coordinated.

If you are doing a batch compilation and NUMBER is 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 NONNUMBER if you do not have line numbers in your source code, or if you want the compiler to ignore the line numbers you do have in your source code. With NONNUMBER in effect, the compiler generates line numbers for your source statements and uses those numbers as references in listings.

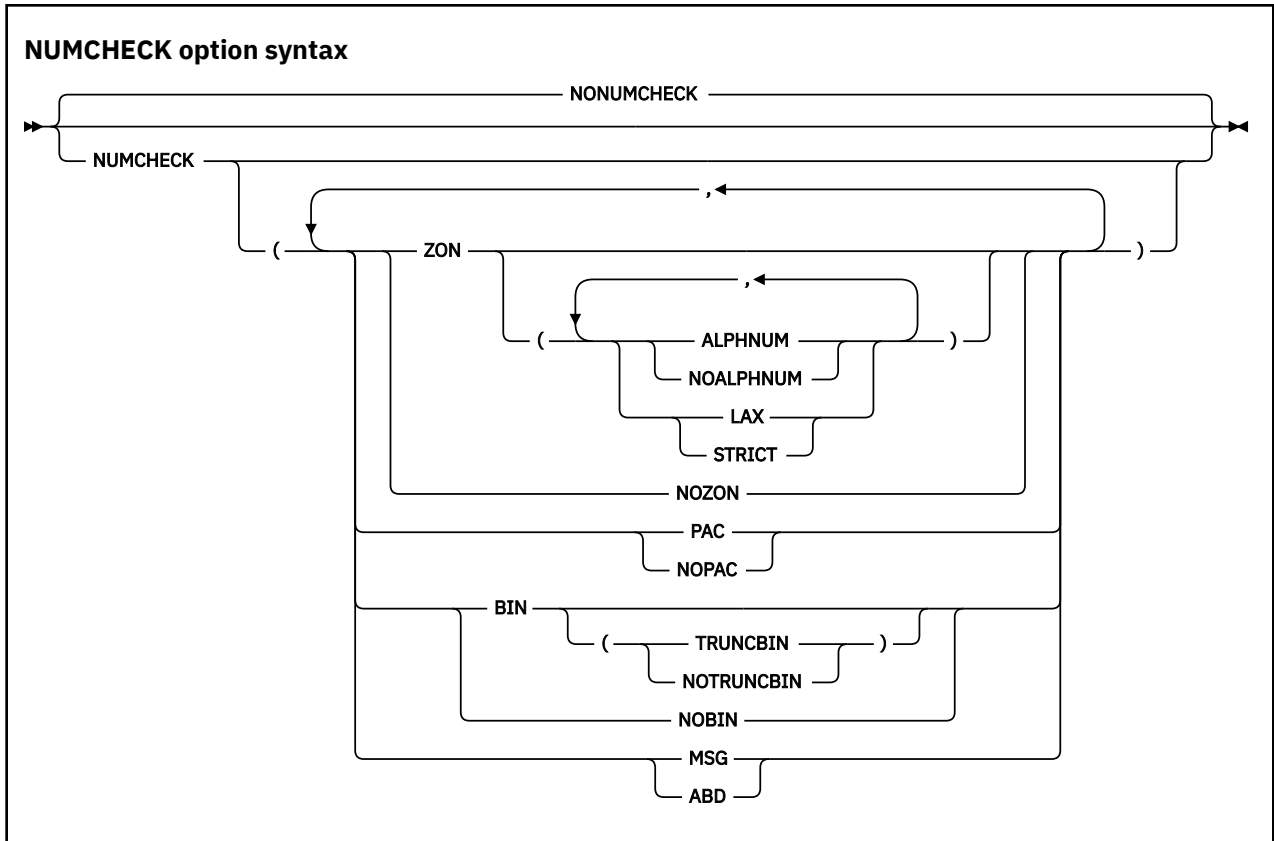
NUMCHECK

The NUMCHECK compiler option tells the compiler whether to generate extra code to validate data items when they are used as sending data items. For zoned decimal (numeric USAGE DISPLAY) and packed decimal (COMP-3) data items, the compiler generates implicit numeric class tests for each sending field. For alphanumeric senders whose contents are being moved to a numeric receiver, the compiler treats the sender as a numeric integer so NUMCHECK generates an implicit numeric class test for each alphanumeric sender. For binary data items, the compiler generates SIZE ERROR checking to see whether the data item has more digits than its PICTURE clause allows.

The NUMCHECK option is updated to remove redundant checks for invalid data, thus improving runtime performance. There may be fewer runtime messages than before.

The analysis done to remove redundant checks is more involved at OPT (1 | 2) than at OPT (0). OPT (0) does a simpler form of the analysis to keep compilation time as low as possible. There may be fewer messages at higher OPT levels.

When the compiler is able to determine at compile time that a check will always find invalid data, a compile-time message is produced and the runtime check is removed. (See MSG | ABD below.)



Default is: NONUMCHECK

Suboption defaults are:

- If no suboption is specified, defaults are ZON (ALPHNUM , STRICT) , PAC , BIN (TRUNCBIN) , and MSG. For example, NUMCHECK has the same effect as NUMCHECK (ZON (ALPHNUM , STRICT) , PAC , BIN (TRUNCBIN) , MSG) .
- If no datatype suboption is specified, default datatype suboptions are ZON (ALPHNUM , STRICT) , PAC , and BIN (TRUNCBIN) . For example, NUMCHECK (ABD) has the same effect as NUMCHECK (ZON (ALPHNUM , STRICT) , PAC , BIN (TRUNCBIN) , ABD) .
- If only one datatype suboption is specified, defaults are NOZON , NOPAC , NOBIN , and MSG. For example, NUMCHECK (BIN) has the same effect as NUMCHECK (NOZON , NOPAC , BIN (TRUNCBIN) , MSG) .
- If all datatype suboptions are specified with NO, then the listing will show NONUMCHECK. For example, NUMCHECK (NOZON , NOPAC , NOBIN) has the same effect as NONUMCHECK.

Abbreviations are: NONC | NC

ZON (ALPHNUM | NOALPHNUM , LAX | STRICT) | NOZON

The default is ZON (ALPHNUM , STRICT) when ZON is specified with no suboptions.

Specifying ZON (ALPHNUM) causes the compiler to generate code for an implicit numeric class test for zoned decimal (numeric USAGE DISPLAY) data items that are used as sending data items in COBOL statements.

Specifying ZON (NOALPHNUM) causes the compiler to generate code for an implicit numeric class test for zoned decimal (numeric USAGE DISPLAY) data items that are used as sending data items

in COBOL statements, except when they are used in a comparison with an alphanumeric data item, alphanumeric literal, or alphanumeric figurative constant.

Receivers are not checked, unless they are both a sender and a receiver, such as data item B in the following sample statements:

```
ADD A TO B
```

```
DIVIDE A INTO B
```

```
COMPUTE B = A + B
```

```
INITIALIZE B REPLACING ALPHANUMERIC BY B
```

This checking is done before the data is used in each statement:

- If the data is NOT NUMERIC, either a warning message for NUMCHECK(ZON,MSG) or a terminating message for NUMCHECK(ZON,ABD) is issued.
- If the data is NUMERIC, the external behavior of the statement is the same as NUMCHECK(NOZON), other than being slower.

Specifying ZON(LAX) causes the compiler to be more tolerant of invalid data in a zoned decimal data item. Three cases are considered by the compiler as follows:

- An unsigned zoned decimal data item redefines a signed trailing overpunch zoned decimal data item such that the last byte of the unsigned item overlaps the last byte of the signed item. In this case, the unsigned redefining item is treated as a signed zoned decimal item for the purposes of the NUMCHECK checking.

Notes:

- The signed zoned decimal item that is redefined must be a level-01 or level-77 item. The unsigned zoned decimal item can be a level-01 or level-77 item or can be a subordinate item in a group.
- The unsigned zoned decimal item does not need to overlap the entire signed zoned decimal item. It is only necessary for the last byte of each item to overlap. For example:

```
01 NUM1 PIC S9(8).  
01 NUM2 REDEFINES NUM1.  
   03 NUM2-PART1 PIC 9(4).  
   03 NUM2-PART2 PIC 9(2).  
   03 NUM2-PART3 PIC 9(2).
```

In this case, data item NUM2-PART3 will be treated by NUMCHECK as a signed zoned decimal data item because its last byte overlaps the last byte of NUM1, which is a signed trailing overpunch zoned decimal item. Thus, the following values of NUM2-PART3 are all considered valid:

- x'F1F2F3F4F5F6F7F8'
- x'F1F2F3F4F5F6F7C8'
- x'F1F2F3F4F5F6F7D8'

- A zoned decimal data item redefines a numeric-edited data item that may contain leading spaces, as indicated by the Z symbol in the numeric-edited item's PICTURE string, and the leading bytes of the zoned decimal data item overlap some or all of the leading bytes of the numeric-edited item. In this case, NUMCHECK will tolerate spaces in the leading bytes of the zoned decimal data item that overlap those bytes of the numeric-edited item that permit spaces.

Notes:

- The numeric-edited item that is redefined must be a level-01 or level-77 item. The zoned decimal item can be a level-01 or level-77 item or can be a subordinate item in a group.

- If the zoned decimal item is signed, it must be signed trailing overpunch.
- The first byte of the zoned decimal item must overlap the first byte of the numeric-edited item to be considered eligible for this treatment, but the zoned decimal item does not need to overlap the entire numeric-edited item. For example:

```
01 NUMED PIC ZZ99.99.
01 NUM REDEFINES NUMED.
03 INTVAL PIC 9(4).
03 FILLER PIC X.
03 DECVAL PIC 9(2).
```

In this case, NUMCHECK tolerates spaces in the first two bytes of INTVAL because it overlaps the first two bytes of NUMED which are defined with the Z symbol in its PICTURE string. Thus, the following values of INTVAL are all considered valid:

- x'F1F2F3F4'
- x'40F1F2F3'
- x'4040F1F2'

Note that for performance reasons, mixes of spaces and non-spaces are tolerated in the leading bytes, thus x'F140F1F2' is also considered valid.

- A zoned decimal data item is moved to a zoned or alphanumeric data item. In this case, NUMCHECK will not check the sender of the move. However, if the sender is subsequently used in a numeric context, it will be checked.

If ZON (STRICT) is specified, NUMCHECK does not consider any data items that a zoned decimal data item might redefine, and strict checking of the zoned decimal data is performed as usual.

Note: The ZON (LAXREDEF | STRICTREDEF) option is deprecated but is tolerated for compatibility, and it is replaced by the ZON (LAX | STRICT) option.

PAC | NOPAC

Specifying PAC causes the compiler to generate code for an implicit numeric class test for packed decimal (COMP-3) data items that are used as sending data items in COBOL statements. For packed decimal data items that have an even number of digits, the unused bits are checked for ones.

Restriction: For CALL statements, NUMCHECK (ZON) and NUMCHECK (PAC) check BY CONTENT data items that are zoned decimal or packed decimal, but they do not check BY REFERENCE parameters. (Neither zoned decimal nor packed decimal data items can be specified in a BY VALUE phrase.)

BIN (TRUNCBIN | NOTRUNCBIN) | NOBIN

Specifying BIN causes the compiler to generate code similar to ON SIZE ERROR to test if binary data items contents are bigger than the PICTURE clause. This extra code will be generated only for binary data items that are used as sending data items, and COMP-5 data items will not get this ON SIZE ERROR code generated.

When BIN (TRUNCBIN) is in effect, the checking code is generated for binary data items, even when the TRUNC (BIN) compiler option is in effect. Note that BIN (TRUNCBIN) is the default when no suboption for BIN is specified.

When BIN (NOTRUNCBIN) is in effect, the checking code is not generated for binary items when the TRUNC (BIN) compiler option is in effect.

Note: BIN (NOTRUNCBIN) is useful for users who want to make NUMCHECK (. . . , BIN, . . .) a fixed option in their default options, but do not want the checking to be done for modules that are compiled with the TRUNC (BIN) option in effect.

MSG | ABD

Determines whether the message issued for invalid data is a warning level message to continue processing or a terminating level message to cause an abend:

- If MSG is in effect, a runtime warning message with the line number, data item name, data item content, and program name is issued. Also, the affected statements will still be executed.

- If ABD is in effect, a terminating message is issued that causes an abend.

When the compiler is able to determine at compile time that a check will always find invalid data, a compile-time error-level message is produced and the check is removed regardless of whether MSG or ABD is in effect.

Performance considerations: NUMCHECK is much slower than NONUMCHECK, depending on how many zoned decimal (numeric USAGE DISPLAY) data items, packed decimal (COMP-3) data items, and binary data items are used in a COBOL program.

Since COBOL 6.2 with service applied, performance of NUMCHECK has been improved. However, performance is still best when specifying NONUMCHECK, and will be better at a higher OPT level.

Note: ZONECHECK is deprecated but is tolerated for compatibility, and it is replaced by NUMCHECK(ZON(ALPHNUM)).

Related tasks

[“Checking for incompatible data \(numeric class test\)” on page 53](#)

Related references

[“NUMPROC” on page 382](#)

[“TRUNC” on page 409](#)

[“ZONECHECK” on page 417](#)

[“INVDATA” on page 366](#)

NUMPROC

Use NUMPROC(NOPFD) if your internal decimal and zoned decimal data might use nonpreferred signs.



Default is: NUMPROC(NOPFD)

Abbreviations are: None

The compiler accepts any valid sign configuration: X'A', X'B', X'C', X'D', X'E', or X'F'. NUMPROC(NOPFD) is the recommended option in most cases.

Performance considerations: NUMPROC(PFD) improves the performance of processing internal decimal and zoned decimal data. Use this option however only if your numeric data agrees exactly with the following IBM system standards:

- **Zoned decimal, unsigned:** High-order 4 bits of the sign byte contain X'F'.
- **Zoned decimal, signed overpunch:** High-order 4 bits of the sign byte contain X'C' if a number is positive or 0, and X'D' if it is not.
- **Zoned decimal, separate sign:** Separate sign contains the character '+' if a number is positive or 0, and '-' 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 a number is positive or 0, and X'D' if it is not.

Data produced by COBOL arithmetic statements conforms to the IBM system standards described above. 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. Use NUMPROC(NOPFD) if a COBOL program calls programs written in PL/I or FORTRAN.

Sign representation is affected not only by the NUMPROC option, but also by the NUMCLS installation option.

Related tasks

[“Checking for incompatible data \(numeric class test\)” on page 53](#)

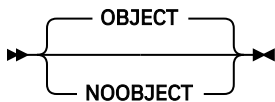
Related references

[“Sign representation of zoned and packed-decimal data” on page 53](#)

OBJECT

Use OBJECT to write the generated object code to a file to be used as input for the binder.

OBJECT option syntax



Default is: OBJECT

Abbreviations are: OBJ | NOOBJ

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 output to 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 that your installation guidelines recommend.

Related references

[“Conflicting compiler options” on page 334](#)

OFFSET

Use OFFSET to produce a condensed PROCEDURE DIVISION listing.

OFFSET option syntax



Default is: NOOFFSET

Abbreviations are: OFF | NOOFF

With OFFSET, the condensed PROCEDURE DIVISION listing will contain line numbers, statement references, and the location of each block of instructions generated for a statement. The optimizer might inline paragraphs, move code around, or indeed place it after the body of the program if little used, such as the error message formatting code. As a result, there might be more than one entry in the OFFSET table of a given statement.

These items will also be written to the output listing:

- Constant area
- Program prolog areas (PPA1, PPA2, PPA3, PPA4)
- Time stamp and compiler version information

- Compiler options and program information
- Base locator table
- External symbols dictionary
- Initial heap storage maps
- Stack storage maps

Note: Due to the out of line code used for error message formatting, Language Environment generated offsets, as indicated in "From compile unit {name} at entry point {name} at compile unit offset {offset}...", might be outside the offset range of the program. In these cases, refer to the statement number in the COBOL message (IGZnnnns) to locate the problem.

Related references

[“Conflicting compiler options” on page 334](#)

[“Example: OFFSET compiler output” on page 477](#)

OPTFILE

Use OPTFILE to enable the specifying of COBOL compiler options in a data set. Using a compiler-option data set circumvents the 100-character limit on options specified in a JCL PARM string.

OPTFILE option syntax

►► OPTFILE ◄◄

Default is: None

Abbreviations are: None

You can specify OPTFILE as a compiler invocation option or in the PROCESS or CBL statement in your COBOL source program. OPTFILE cannot be specified as an installation default.

OPTFILE is ignored if you compile using the cob2 command in the z/OS UNIX environment. (In that environment, the COBOPT environment variable provides a capability that is comparable to OPTFILE.)

If OPTFILE is in effect, compiler options are read from the data set that you identify in a SYSOPTF DD statement. A SYSOPTF data set must have RECFM F or FB and an LRECL of 80 bytes. For further details about the format of a SYSOPTF data set, see the related task below about defining a compiler-option data set.

The precedence of options in the SYSOPTF data set is determined by where you specify the OPTFILE option. For example, if you specify OPTFILE in the invocation PARM string, an option specified later in the PARM string supersedes any option specified in the SYSOPTF data set that conflicts with it.

(Conceptually, OPTFILE in an options specification is replaced with the options that are in the SYSOPTF data set; then the usual rules about precedence of compiler options and conflicting compiler options apply.)

If you start the COBOL compiler from within an assembler program, you can use the alternate ddname list to specify a ddname to be used instead of SYSOPTF to identify the compiler-option data set.

Related tasks

[“Starting the compiler from an assembler program” on page 264](#)

[“Defining a compiler-option data set \(SYSOPTF\)” on page 270](#)

[“Specifying compiler options under z/OS” on page 273](#)

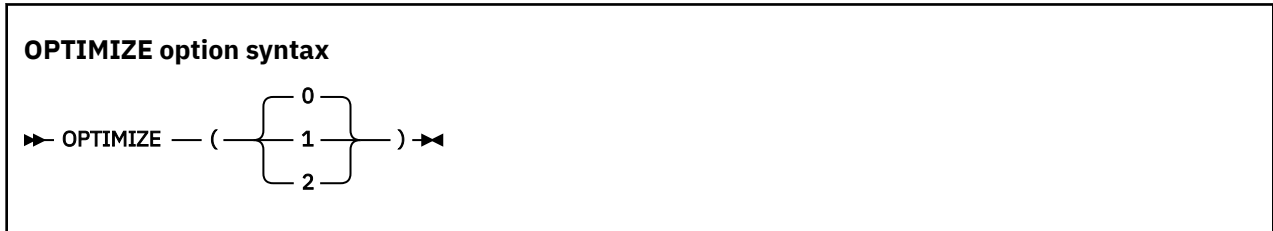
[Chapter 15, “Compiling under z/OS UNIX,” on page 283](#)

Related references

[“Conflicting compiler options” on page 334](#)

OPTIMIZE

Use OPTIMIZE to reduce the run time of your object program. Optimization might also reduce the amount of storage your object program uses.



Default is: OPTIMIZE (0)

Abbreviations are: OPT (0), OPT (1), or OPT (2)

Optimizations are performed under the assumption that the program and data are valid, given the compiler options. For example, external decimal data that has USAGE DISPLAY must be valid unless ZONEDATA(MIG | NOPFD) is used to allow invalid zone bits. Digits and sign codes must be valid no matter what options are used. If the program or data is invalid, programs might behave differently at different levels of optimization or between different versions of Enterprise COBOL.

- OPTIMIZE (0) specifies limited optimizations, which result in the shortest compilation time. When the TEST option is specified, full debug capabilities are available.
- OPTIMIZE (1) specifies optimizations that improve application runtime performance. Optimizations at this level include basic inlining, strength reduction, simplification of complex operations into equivalent simpler operations, removal of some unreachable code and block rearrangement. Also, OPTIMIZE (1) includes some intrablock optimizations such as common subexpression elimination and value propagation. When the TEST option is specified, most debug capabilities are available.
- OPTIMIZE (2) specifies further optimizations, which include more aggressive simplifications and instruction scheduling. Also, some interblock optimizations such as global value propagation and loop invariant code motion are included. When the TEST option is specified, some debug capabilities are available.

When OPTIMIZE (1) or OPTIMIZE (2) is used without the TEST compiler option, care must be taken with user-written condition handlers registered via the Language Environment service CEEHDLR. In particular, if a condition handler accesses data items that are not defined local to the condition handler program themselves (for example, data items defined in the application as EXTERNAL), such data items must be defined with the VOLATILE clause to ensure that the handler uses the latest value of the data item, or the condition handler program can be compiled with the TEST compiler option. The use of the VOLATILE clause is preferred over the use of the TEST option because the use of the TEST option can reduce optimization for the entire program, while VOLATILE localizes the reduced optimization. For more information about the VOLATILE clause, see VOLATILE clause in the *Enterprise COBOL for z/OS Language Reference*.

Note: For best performance, IBM recommends that you use the OPT (2) compiler option.

When OPTIMIZE (1) or OPTIMIZE (2) is in effect, specifying INLINE causes the compiler to consider inlining procedures for PERFORM statements. For details, see [“INLINE” on page 365](#).

Note: Since Enterprise COBOL 5, the NOOPTIMIZE, OPTIMIZE, OPTIMIZE (STD), and OPTIMIZE (FULL) options are removed but are tolerated for compatibility. If one of those options is specified, it is mapped to the new option or options as follows:

Removed options	New options
NOOPTIMIZE	OPTIMIZE (0)
OPTIMIZE	OPTIMIZE (2)

Table 51. Mapping of removed options to new options (continued)

Removed options	New options
OPTIMIZE (STD)	OPTIMIZE (2)
OPTIMIZE (FULL)	OPTIMIZE (2) and STGOPT

Related concepts

[“Optimization” on page 762](#)

Related tasks

[“Writing routines for handling errors” on page 252](#)

Related references

[“Conflicting compiler options” on page 334](#)

[“INLINE” on page 365](#)

[“MAXPCF” on page 375](#)

[“TEST” on page 404](#)

[“STGOPT” on page 402](#)

VOLATILE clause (*Enterprise COBOL for z/OS Language Reference*)

OUTDD

Use OUTDD to specify that you want DISPLAY output that is directed to the system logical output device to go to a specific ddname.

You can specify a file in the z/OS UNIX file system with the ddname named in OUTDD. To understand where output is directed when this ddname is not allocated, see the related task about displaying data.

OUTDD option syntax

►► OUTDD(*ddname*) ◄◄

Default is: OUTDD(SYSOUT)

Abbreviations are: OUT

If the OUTDD compiler option and the Language Environment MSGFILE runtime option specify the same ddname (both default to SYSOUT), DISPLAY output to the system logical output device is written using Language Environment message facilities.

Notes:

- The OUTDD option has no effect under CICS.
- The OUTDD compiler option can only be specified for the first program in a sequence of programs (batch compilation). If you specify OUTDD in a CBL or PROCESS statement for a program other than the first program in the sequence, you will receive a compiler diagnostic message.

Related tasks

[“Displaying data on the system logical output device” on page 36](#)

[“Coding COBOL programs to run under CICS” on page 483](#)

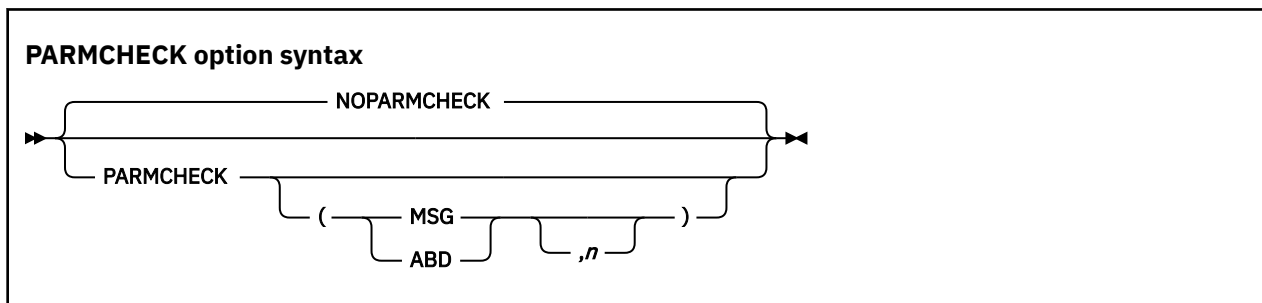
Related references

Language Environment Programming Reference (MSGFILE)

PARMCHECK

The PARMCHECK option tells the compiler to generate an extra data item following the last item in WORKING-STORAGE. This buffer data item is then used at run time to check whether a called subprogram corrupted data beyond the end of WORKING-STORAGE.

When a calling program is compiled with PARMCHECK, the compiler generates a buffer following the last data item in the WORKING-STORAGE section. At run time, before each call, the buffer is set to ALL x'AA'. After each call, the buffer is checked to see whether it was changed. The PARMCHECK option can help with your migration from COBOL 4 and earlier compilers to COBOL 6 and later compilers, and can also be used to clean up and check for good programming practices.



Default is: NOPARMCHECK

Abbreviations are: NOPC | PC

Suboption defaults are:

- If no suboption is specified, default is PARMCHECK (MSG , 100).
- If only MSG or ABD is specified, default is PARMCHECK (MSG | ABD , 100). For example, PC (ABD) = PC (ABD , 100).
- If only *n* is specified, default is PARMCHECK (MSG , *n*). For example, PC (5000) = PC (MSG , 5000).

MSG | ABD

Determines whether the message issued for subprogram corruption of data is a warning level message to continue processing or a terminating level message to cause an abend:

- If MSG is in effect, a runtime warning message with the name of the parameter, the line number of the CALL statement, and the program name is issued. Also, this check is done after the affected CALL statement is executed.
- If ABD is in effect, a similar message is issued, but with a terminating level that causes an abend.

n

The size in bytes of the buffer to be added after the last item in WORKING-STORAGE. Must be an integer in the range of 1 to 9999.

Performance considerations: PARMCHECK will cause the compiler to generate slower code for programs with CALL statements. NOPARMCHECK should be in effect for good performance.

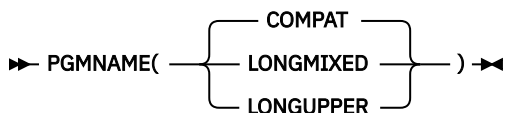
Related references

CALL statement (*Enterprise COBOL for z/OS Language Reference*)

PGMNAME

The PGMNAME option controls the handling of program-names, entry-point names, and user-defined function-names.

PGMNAME option syntax



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.

PGMNAME controls the handling of names used in the following contexts:

- Program-names defined in the PROGRAM- ID paragraph.
- Program entry-point names in the ENTRY statement.
- Program-name references in:
 - CALL statements that reference nested programs, statically linked programs, or DLLs.
 - SET *procedure-pointer* or *function-pointer* statements that reference statically linked programs or DLLs.
 - CANCEL statements that reference nested programs.
- User-defined function-names defined in the FUNCTION- ID paragraph if the AS phrase is not specified.
- *literal-1* of the AS phrase of the user-defined function-name if the AS phrase is specified.

Note: If the user-defined function-name is specified with the AS phrase, *literal-1* of the AS phrase is the externalized name, not the user-defined function-name itself.

PGMNAME(COMPAT)

With PGMNAME (COMPAT), program-names and user-defined function-names are handled in a manner compatible with older versions of COBOL compilers:

- The program-name can be up to 30 characters in length.
- All the characters used in the name must be alphabetic, digits, the hyphen, or the underscore, except that if the program-name is a literal and is in the outermost program, then the literal can also contain the extension characters @, #, and \$, and the first character can be an underscore.
- At least one character must be alphabetic.
- The hyphen cannot be used as the first or last character.

External program-names and external user-defined function-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, and is not an underscore, it is converted as follows:
 - 1-9 are translated to A-I.
 - 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, or user-defined function-names that are specified in the `FUNCTION-ID` paragraph as COBOL user-defined words, must follow the normal COBOL rules for forming a user-defined word:

- The program-name can be up to 30 characters in length.
- All the characters used in the name must be alphabetic, digits, the hyphen, or the underscore.
- At least one character must be alphabetic.
- The hyphen cannot be used as the first or last character.
- The underscore cannot be used as the first character.

When a program-name is specified as a literal, in either a definition or a reference, then:

- The program-name can be up to 160 characters in length.
- All the characters used in the name must be alphabetic, digits, the hyphen, or the underscore.
- At least one character must be alphabetic.
- The hyphen cannot be used as the first or last character.
- The underscore can be used in any position.

External program-names and external user-defined function-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, and is not an underscore, it is converted as follows:
 - 1-9 are translated to A-I.
 - Anything else is translated to J.

Names of nested programs 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.

If you want to use a `PROGRAM-ID` that is longer than eight characters, enclose it in single quotation marks or apostrophes (') and use `PGMNAME(LONGMIXED)`.

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 user for a program-name can contain any character in the range `X'41'-X'FE'`.

With `PGMNAME(LONGMIXED)`, all user-defined function definitions must be specified with the `AS` phrase in the `FUNCTION-ID` paragraph. *literal-1* of the `AS` phrase is processed as is, without truncation, translation, or folding to uppercase.

Usage notes

- The following elements 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 in `COPY` statements).
 - Dynamic calls.

Dynamic calls are resolved with truncation of the program-name to eight characters, folding to uppercase, and translation of embedded hyphens or a leading digit.

- CANCEL of nonnested programs. Name resolution uses the same mechanism as for a dynamic call.
- **Link-edit considerations:** COBOL programs that are compiled with the PGMNAME (LONGUPPER) or PGMNAME (LONGMIXED) option must be link-edited in AMODE 31.
- 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.
- When using the extended character set supported by PGMNAME (LONGMIXED), be sure to use names that conform to the binder (linkage-editor) or system conventions that apply, depending on the mechanism used to resolve the names.

Using characters such as commas or parentheses is not recommended, because these characters are used in the syntax of binder (linkage-editor) control statements.

Related references

PROGRAM-ID paragraph (*Enterprise COBOL for z/OS Language Reference*)

FUNCTION-ID paragraph (*Enterprise COBOL for z/OS Language Reference*)

QUALIFY

QUALIFY affects qualification rules and controls whether to extend qualification rules so that some data items that cannot be referenced under COBOL Standard rules can be referenced.

QUALIFY option syntax

►► QUALIFY(COMPAT | EXTEND) ◄◄

Default is: QUALIFY (COMPAT)

Abbreviations are: QUA (C | E)

QUALIFY (COMPAT)

If QUALIFY (COMPAT) is in effect, references to data items must be unique.

QUALIFY (EXTEND)

If QUALIFY (EXTEND) is in effect, qualification rules are extended so that some references that are not unique by COBOL standard rules can be unique. If every level in the containing hierarchy of names is specified, the set of qualifiers is called a *complete set of qualifiers*. If there is only one data item with a specific complete set of qualifiers, the reference resolves to that data item, even if the same set of qualifiers can match with another reference as an incomplete set of qualifiers.

Example

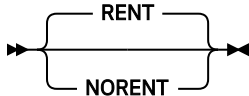
```
01 A.
  02 B.
    03 C PIC X.
    03 A PIC X.
  02 C PIC X.
.
.
.
  Move space to C of A      *> Refers to 02 level C (unique only with
QUALIFY(EXTEND))
  Move space to A          *> Refers to 01 level A (unique only with
QUALIFY(EXTEND))
  Move space to C of B of A *> Refers to 03 level C (unique by COBOL standard
rules)
  Move space to C of B     *> Refers to 03 level C (unique by COBOL standard
rules)
```

RENT

A program compiled as RENT is generated as a reentrant object program. A program compiled as NORENT is generated as a nonreentrant object program.

Either a reentrant or a nonreentrant program can be invoked as a main program or as a subprogram.

RENT option syntax



Default is: RENT

Abbreviations are: None

DATA and RMODE settings: The RENT option interacts with other compiler options that affect storage and its addressability. Use the DATA (24 | 31) option for reentrant programs to control whether dynamic data areas are allocated in unrestricted storage or in storage obtained from below 16 MB. Compile programs with RENT if they will be run in virtual storage addresses above 16 MB.

Execution of nonreentrant programs above 16 MB is not supported. Programs compiled with NORENT must be RMODE 24.

The setting of the DATA option does not affect programs compiled with NORENT.

For information about which Enterprise COBOL programs need to be reentrant, see the related task about making programs reentrant.

Link-edit considerations: If all programs in a program object are compiled with RENT, it is recommended that the program object be link-edited with the RENT binder (linkage-editor) option. Use the REUS binder (linkage-editor) option instead if the program object will also contain any non-COBOL programs that are only serially reusable.

If any program in a program object is not reentrant, the program object must not be link-edited with the RENT or REUS link-edit attributes. The NOREUS binder (linkage-editor) option is needed to ensure that the CANCEL statement will guarantee a fresh copy of the program on a subsequent CALL.

AMODE 64 considerations: The LP (64) compiler option implies RENT. If the user explicitly specifies NORENT, an informational message is issued and the setting is ignored.

Related concepts

[“Storage and its addressability” on page 39](#)

Related tasks

[“Making programs reentrant” on page 548](#)

DB2 Application Programming and SQL Guide (Using reentrant code)

Related references

[“Conflicting compiler options” on page 334](#)

[“DATA” on page 349](#)

[“RMODE” on page 392](#)

RMODE

The RMODE setting influences the RMODE (residency mode) of your generated object program.



Default is: AUTO

Abbreviations are: None

A program compiled with the RMODE (AUTO) option will have RMODE 24 if NORENT is specified, or RMODE ANY if RENT is specified. RMODE AUTO is compatible with older compilers such as VS COBOL II, which produced RMODE 24 for programs compiled with NORENT, and RMODE ANY for programs compiled with RENT.

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 must also be compiled with the RENT option. The program will have the RMODE ANY attribute.

If the NORENT option is specified, the RMODE (24) or RMODE (AUTO) compiler option must be specified. Overriding the module RMODE with a binder option or control statement is not supported.

DATA and RENT: The RMODE option interacts with other compiler options and runtime options that affect storage and its addressability. For information about passing data between programs with different modes, see the related concept about storage and its addressability.

Link-edit considerations: If the object code that COBOL generates has an attribute of RMODE 24, you must link-edit the code into an RMODE 24 module. If the object code that COBOL generates has an attribute of RMODE ANY, you can link-edit the code into either an RMODE ANY or RMODE 24 module.

AMODE 64 considerations: The LP (64) compiler option implies RMODE (ANY). If the user explicitly specifies RMODE (24), an informational message is issued and the setting is ignored.

Related concepts

[“Storage and its addressability” on page 39](#)

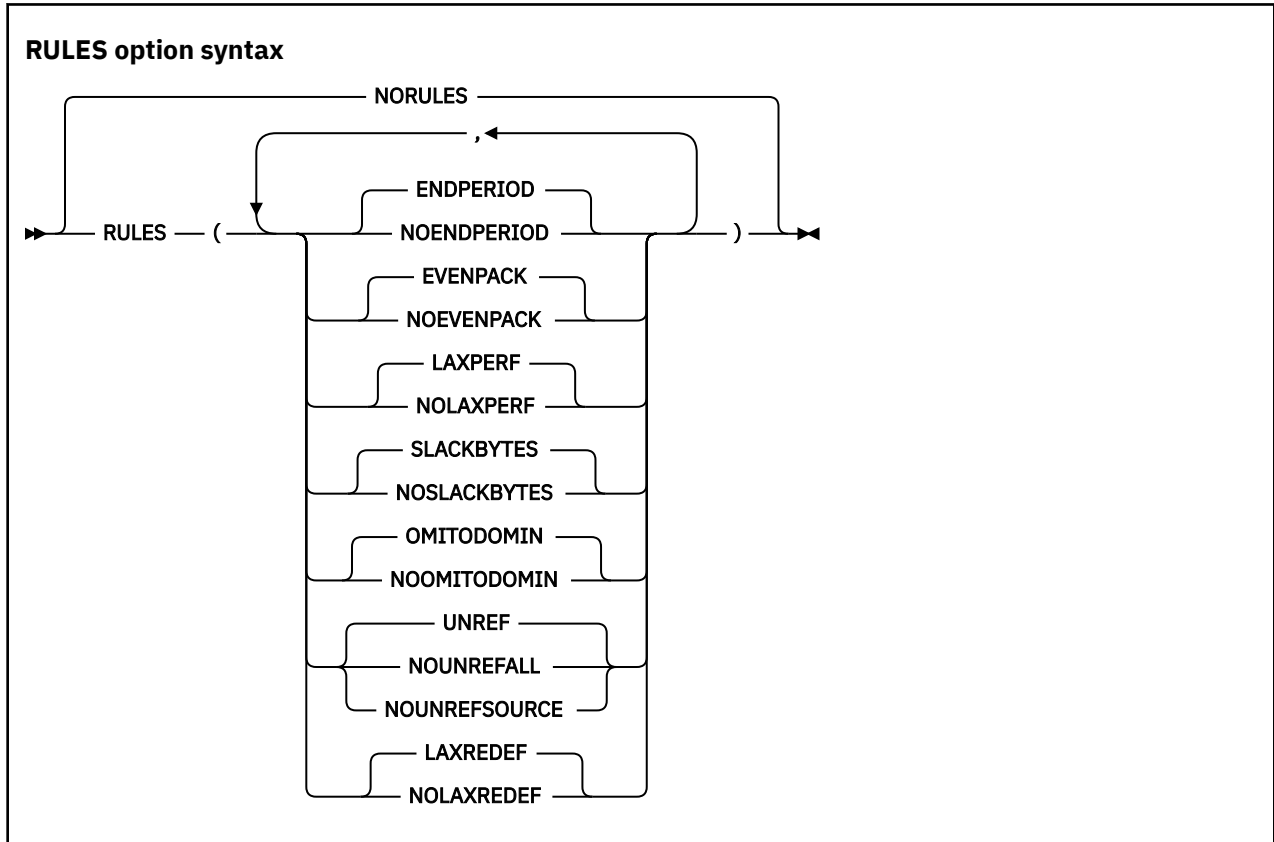
Related references

[“Allocation of buffers for QSAM files” on page 186](#)

[“Conflicting compiler options” on page 334](#)

RULES

You can use the RULES option to request information about your program from the compiler to improve the program by flagging certain types of source code at compile time.



Default is: NORULES

Abbreviations are:

- ENDP = ENDPERIOD
- EVENP = EVENPACK
- LXPRF = LAXPERF
- SLCKB = SLACKBYTES
- OOM = OMITODOMIN
- NOUNRA = NOUNREFALL
- NOUNRS = NOUNREFSOURCE
- LXRDF=LAXREDEF

You can specify the following suboptions for RULES:

ENDPERIOD | NOENDPERIOD

The default is ENDPERIOD. Specifying NOENDPERIOD causes the compiler to issue warning messages when the scope of a conditional statement is terminated by a period instead of an explicit scope terminator END-*.

EVENPACK | NOEVENPACK

The default is EVENPACK. Specifying NOEVENPACK causes the compiler to issue warning messages for any USAGE PACKED-DECIMAL (COMP-3) data items that have an even number of digits because those data items whose unused bits are not zero can lead to an unexpected program behavior.

Notes:

- RULES (NOEVENPACK) helps you identify USAGE PACKED-DECIMAL (COMP-3) data items that have unused extra space reserved for them. However, it is not necessary to change those data items to have an odd number of digits, it is only a slightly better way of programming.
- The compiler does not issue messages for even-digit PACKED-DECIMAL data items if the name starts with DFH, DSN, EYU, or SQL, that is, data items generated for/by CICS and Db2.

LAXPERF | NOLAXPERF

The default is LAXPERF. Specifying NOLAXPERF suboption causes the compiler to issue warning messages for usage of inefficient COBOL features. These features might include USAGE DISPLAY numeric data items in arithmetic statements, large amounts of space padding in MOVE statements, inefficient compiler options, and other cases.

SLACKBYTES | NOSLACKBYTES

The default is SLACKBYTES. Specifying NOSLACKBYTES causes the compiler to issue warning messages for any SYNCHRONIZED data items that cause the compiler to add slack bytes, either slack bytes within records or slack bytes between records. Each data item that causes slack bytes to be added gets a compiler diagnostic.

OMITODOMIN | NOOMITODOMIN

The default is OMITODOMIN. Specifying NOOMITODOMIN causes the compiler to issue warning messages for any OCCURS DEPENDING ON clauses that are specified without *integer-1* (the minimum number of occurrences).

For more information about the OCCURS DEPENDING ON clause, see *Variable-length tables* in the *Enterprise COBOL for z/OS Language Reference*.

UNREF | NOUNREFALL | NOUNREFSOURCE

The default is UNREF, which means that no reporting of unreferenced data items occurs.

When NOUNREFALL is specified, all level-01 and level-77 data items in the FILE SECTION, WORKING-STORAGE SECTION, LOCAL-STORAGE SECTION, and LINKAGE SECTION that are unreferenced, including no subordinate items referenced when the item is a group, are reported, regardless of whether the definition of the data item appears directly in the user source program or was included in the program from a copy member.

When NOUNREFSOURCE is specified, all level-01 and level-77 data items in the FILE SECTION, WORKING-STORAGE SECTION, LOCAL-STORAGE SECTION, and LINKAGE SECTION that are unreferenced, including no subordinate items referenced when the item is a group, are reported only if the definition of the data item appears directly in the user source program.

Notes:

- In COBOL, the definition of a single group item can spread across different files. When this occurs, and if the definition of the level-01 data item of the group is in the main source file, then those data items that are unreferenced will be reported when NOUNREFSOURCE is in effect.
- Data items with the name prefix DFH, DSN, EYU, or SQL (that is, data items generated for/by CICS and Db2) will not be reported when NOUNREFALL or NOUNREFSOURCE is in effect.

LAXREDEF | NOLAXREDEF

The default is LAXREDEF. Specifying NOLAXREDEF will cause the compiler to issue warning messages when a data item is redefined by a smaller item on any level. The default option, LAXREDEF, causes the compiler to issue such warnings only for non-level-01 data items.

If the RULES option is specified with no suboptions, the default is RULES (ENDPERIOD, EVENPACK, LAXPERF, SLACKBYTES, OMITODOMIN, UNREF, LAXREDEF).

Notes:

- It is not necessary to specify all of the suboptions for RULES. If a suboption is not specified, the default value of that suboption takes effect. For example, if you specify RULES (NOENDP, NOSLCKB), RULES (NOENDP, EVENP, LXPRF, NOSLCKB, OMITODOMIN, UNREF, LXRDF) takes effect.
- If there are multiple RULES specifications for a compilation, the suboptions are additive, which means they are accumulated. For example, if you specify RULES (ENDP), you will get RULES (ENDP) in effect

as well as whatever RULES suboptions that were previously specified. If there is no RULES specification for a suboption, the default will take effect. If a subsequent RULES specification references a previously specified suboption, then the last specification takes effect.

- You can optionally use the RULES option with the MSGEXIT suboption of the EXIT compiler option to enforce local coding standards. For example, if you want to ensure that no programmers use periods instead of explicit scope delimiters to delimit conditional statements, you can change the severity of the ENDPERIOD message from Warning level (RC=4) to Severe level (RC=12). For a sample of how to modify the severity of this and other RULES messages, see the sample MSGEXIT in SIGYSAMP called IGYMSGXT.

Related references

SYNCHRONIZED clause (Enterprise COBOL for z/OS Language Reference)

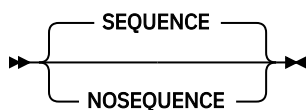
[“STGOPT” on page 402](#)

SEQUENCE

When you use SEQUENCE, the compiler examines columns 1 through 6 to check that the source statements are arranged in ascending order according to their EBCDIC collating sequence. The compiler issues a diagnostic message if any statements are not in ascending order.

Source statements with blanks in columns 1 through 6 do not participate in this sequence check and do not result in messages.

SEQUENCE option syntax



Default is: SEQUENCE

Abbreviations are: SEQ | NOSEQ

If you use COPY statements with the SEQUENCE option in effect, be sure that your source program's sequence fields and the copybook sequence fields are coordinated.

If you use NUMBER and SEQUENCE, the sequence is checked according to numeric, rather than EBCDIC, collating sequence.

If you are doing a batch compilation and SEQUENCE is in effect, all programs in the batch compilation are 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.

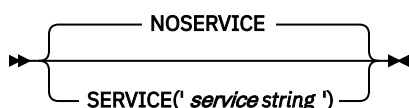
Related tasks

[“Finding line sequence problems” on page 438](#)

SERVICE

Use SERVICE to place a string in the object module if the object module is generated. If the object module is linked into a program object, the string is loaded into memory with this program object. If the Language Environment dump includes a traceback, this string is included in that traceback.

SERVICE option syntax



Default is: NOSERVICE

Abbreviations are: SERV | NOSERV

The *service string* is limited to 64 characters in length.

SMARTBIN

Use SMARTBIN to instruct the compiler to generate modules containing additional binary metadata that enables them to be optimized by IBM Automatic Binary Optimizer (ABO) for z/OS 2.2.

SMARTBIN option syntax



Default is: SMARTBIN when LP(32) is in effect.

Note: SMARTBIN is not supported when LP(64) is in effect.

The SMARTBIN option enables IBM Automatic Binary Optimizer 2.2 to optimize modules created by IBM Enterprise COBOL 6.4.

When SMARTBIN is in effect, the additional binary metadata is placed in a NOLOAD segment of the module. To generate the SMARTBIN metadata, compile times may increase by up to 21% at OPT(0) and 2-3% at OPT(1) and OPT(2) on an IBM z15[®] (or later) machine with zEnterprise Data Compression (zEDC) enabled (hardware compression turned on). This is in comparison to an increase of up to 33% at OPT(0) and 10% at OPT(1) and OPT(2) on an IBM z15[™] (or later) machine without zEDC enabled (hardware compression turned off). The additional metadata will also increase the size of the module on disk, requiring larger load libraries, but will not increase the size in memory when the program is running since it is not loaded. The size increase on disk will be approximately 2 times to 3 times the size of the original binary.

SMARTBIN is the default when LP(32) is in effect. You can change the option to NOSMARTBIN, however, without the additional binary metadata, COBOL modules built with Enterprise COBOL 6.4 will be ineligible for ABO optimization and you would need to recompile and test your modules in the future to maximize benefit from IBM Z hardware improvements. If you use ABO or plan to in the future, the SMARTBIN option is recommended.

Notes:

- IBM Automatic Binary Optimizer for z/OS 2.2 can optimize CSECTs within program modules that were generated by the following COBOL compilers:
 - Enterprise COBOL for z/OS 6
Newly eligible as of ABO 2.2. ABO 2.2 can only optimize COBOL 6.4 modules if the SMARTBIN compiler option is in effect. ABO 2.2 can optimize COBOL 6.1, 6.2, and 6.3 modules without requiring special COBOL compiler options to be set.
 - Enterprise COBOL for z/OS 5
Newly eligible as of ABO 2.2. ABO 2.2 can optimize COBOL 5 modules without requiring special COBOL compiler options to be set.
 - Enterprise COBOL for z/OS 4
 - Enterprise COBOL for z/OS 3
 - COBOL for OS/390 & VM 2
 - COBOL for MVS & VM 1.2
 - COBOL/370 1.1
 - VS COBOL II 1.4.0

– VS COBOL II 1.3.x

- COBOL modules that have been processed by CA-Optimizer cannot be optimized by ABO. For these types of modules, it is recommended to use ABO to optimize the original module created by the COBOL compiler before it was processed by CA-Optimizer.

Refer to the [IBM Automatic Binary Optimizer for z/OS product page](#) for additional information on ABO benefits.

Related concepts

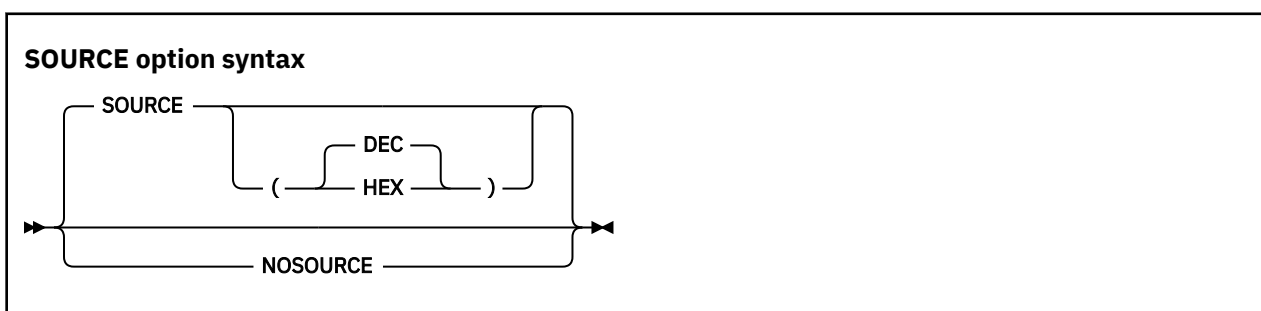
[“Storage and its addressability” on page 39](#)

Related references

[“LP” on page 373](#)

SOURCE

Use SOURCE to get a listing of your source program. This listing will include any statements embedded by PROCESS or COPY statements.



Default is: SOURCE (DEC)

Abbreviations are: S | NOS

You must specify SOURCE if you want embedded messages in the source listing. The compiler will generate sequence numbers for the listing of the source in your program.

Use NOSOURCE to suppress the source code from the compiler output listing.

DEC

If SOURCE (DEC) is in effect, the line numbers for the listing of the source will be in decimal format. SOURCE with no suboption has the same effect as SOURCE (DEC).

HEX

If SOURCE (HEX) is in effect, the line numbers for the listing of the source will be in hexadecimal format.

You can specify SOURCE (HEX) as a compiler invocation option or in the PROCESS or CBL statement in your COBOL source program.

SOURCE (HEX) cannot be specified as an installation default.

Note:

- If SOURCE (HEX) is in effect, then compiler options NUMBER and SEQUENCE cannot be used.
- Even with SOURCE (HEX), any runtime messages will still refer to decimal line numbers, so that when locating the source of a runtime message, the user will have to convert the decimal line number from the runtime message (or ABEND) into hexadecimal to find the relevant line in the source listing.

If you want to limit the SOURCE output, use *CONTROL SOURCE or NOSOURCE statements in your PROCEDURE DIVISION. Source statements that follow a *CONTROL NOSOURCE statement are not included in the listing until a subsequent *CONTROL SOURCE statement switches the output back to normal SOURCE format.

The following example, shows a source listing with embedded diagnostic message:

```
00006C          Check-results section.
00006D          If ec = 0 then

== 00006D> IGYPS0086-I "EC" is reserved in ISO/IEC FCD 1989:2002, Programming language COBOL,
          and may be reserved in a future release of this compiler.
```

[“Example: MAP output” on page 449](#)

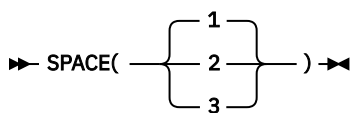
Related references

*CONTROL (*CBL) statement (*Enterprise COBOL for z/OS Language Reference*)

SPACE

Use SPACE to select single-, double-, or triple-spacing in your source code listing.

SPACE option syntax



Default is: SPACE (1)

Abbreviations are: None

SPACE has meaning only when the SOURCE compiler option is in effect.

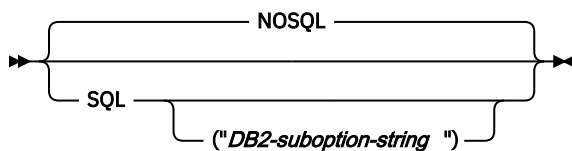
Related references

[“SOURCE” on page 397](#)

SQL

Use the SQL compiler option to enable the Db2 coprocessor and to specify Db2 suboptions.

SQL option syntax



Default is: NOSQL

Abbreviations are: None

If a COBOL source program contains SQL statements (EXEC SQL statements), you must specify the SQL option when LP (64) is in effect, and Db2 coprocessor must be available to the compiler during the compilation. The Db2 precompiler is not supported in LP(64).

When you use the SQL option, the Db2 coprocessor writes the database request module (DBRM) to ddname DBRMLIB.

If you specify the NOSQL option, any SQL statements found in the source program are diagnosed and discarded.

Use either quotation marks or apostrophes to delimit the string of Db2 suboptions.

You can partition a long suboption string into multiple suboption strings in multiple CBL statements. For example:

```
//STEP1 EXEC IGYWC, . . .
// PARM.COBOL='SQL("string1")'
//COBOL.SYSIN DD *
    CBL SQL("string2")
    CBL SQL('string3')
    IDENTIFICATION DIVISION.
    PROGRAM-ID. DRIVER1.
    . . .
```

The Db2 suboptions are concatenated in the order of their appearance. Thus in the example above, 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 suboption string to 4 KB.

Note: The SQL compiler option can only be specified for the first program in a sequence of programs (batch compilation). If you specify SQL in a CBL or PROCESS statement for a program other than the first program in the sequence, you will receive a compiler diagnostic message.

Related concepts

[“Db2 coprocessor” on page 493](#)

[“COBOL and Db2 CCSID determination” on page 498](#)

Related tasks

[“Compiling with the SQL option” on page 497](#)

[“Separating Db2 suboptions” on page 498](#)

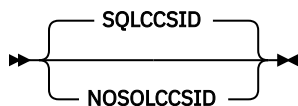
Related references

[“Conflicting compiler options” on page 334](#)

SQLCCSID

Use the SQLCCSID compiler option to control whether the CODEPAGE compiler option will influence the processing of SQL statements in your COBOL programs.

SQLCCSID option syntax



Default is: SQLCCSID

Abbreviations are: SQLC | NOSQLC

When the LP(64) compiler option is in effect, the SQLCCSID option is supported, which behaves in the same way as in LP(32).

The SQLCCSID option has an effect only if you use the integrated Db2 coprocessor (SQL compiler option).

If SQLCCSID is in effect, the setting of the CODEPAGE compiler option will influence the processing of SQL statements within your COBOL programs when you use the integrated Db2 coprocessor.

If NOSQLCCSID is in effect:

1. The CODEPAGE compiler option will only be used as the encoding for string literals and the COBOL application source that includes converted SQL statements;
2. Db2 (character string) host variables will not be affected by the CODEPAGE compiler option. Instead, the encoding for Db2 (character string) host variables will come from the CCSID value found in the DSNHDECP file, which means Db2 (via DSNHDECP) determines the encoding of the Db2 data (host variables).

Related concepts

[“Db2 coprocessor” on page 493](#)

[“COBOL and Db2 CCSID determination” on page 498](#)

Related tasks

[“Programming with the SQLCCSID or NOSQLCCSID option” on page 500](#)

Related references

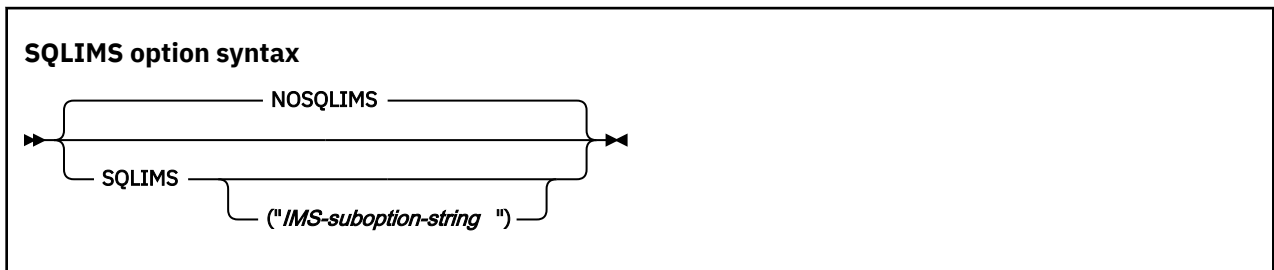
[“Code-page determination for string host variables in SQL statements” on page 499](#)

[“CODEPAGE” on page 343](#)

[“SQL” on page 398](#)

SQLIMS

Use the SQLIMS compiler option to enable the IMS SQL coprocessor and to specify Information Management System (IMS) suboptions. You must specify the SQLIMS option if a COBOL source program contains SQLIMS statements (EXEC SQLIMS statements).



Default: NOSQLIMS

Abbreviation: None

If you specify the NOSQLIMS option, any SQLIMS statements that are found in the source program are diagnosed and discarded.

Use either quotation marks or apostrophes to delimit the string of IMS suboptions.

You can partition a long suboption string into multiple suboption strings in multiple CBL statements. For example:

```
//STEP1 EXEC IGYWC, . . .
// PARM.COBOL='SQLIMS("string1")'
//COBOL.SYSIN DD *
  CBL SQLIMS("string2")
  CBL SQLIMS('string3')
  IDENTIFICATION DIVISION.
  PROGRAM-ID. DRIVER1.
  . . .
```

The IMS suboptions are concatenated in the order of their appearance. Thus in the preceding example, the compiler passes the following suboption strings to the IMS SQL coprocessor:

```
"string1 string2 string3"
```

The concatenated strings are delimited with single spaces as shown. If multiple instances of the same IMS suboption are found, the last specification of each suboption takes effect. The compiler limits the length of the concatenated IMS suboption string to 4 KB.

When the LP(64) compiler option is in effect, the SQLIMS option is not supported. If the option is specified explicitly by the user, a diagnostic message is emitted.

Note: The SQLIMS compiler option can only be specified for the first program in a sequence of programs (batch compilation). If you specify SQLIMS in a CBL or PROCESS statement for a program other than the first program in the sequence, you will receive a compiler diagnostic message.

Related concepts

[“IMS SQL coprocessor” on page 505](#)

Related tasks

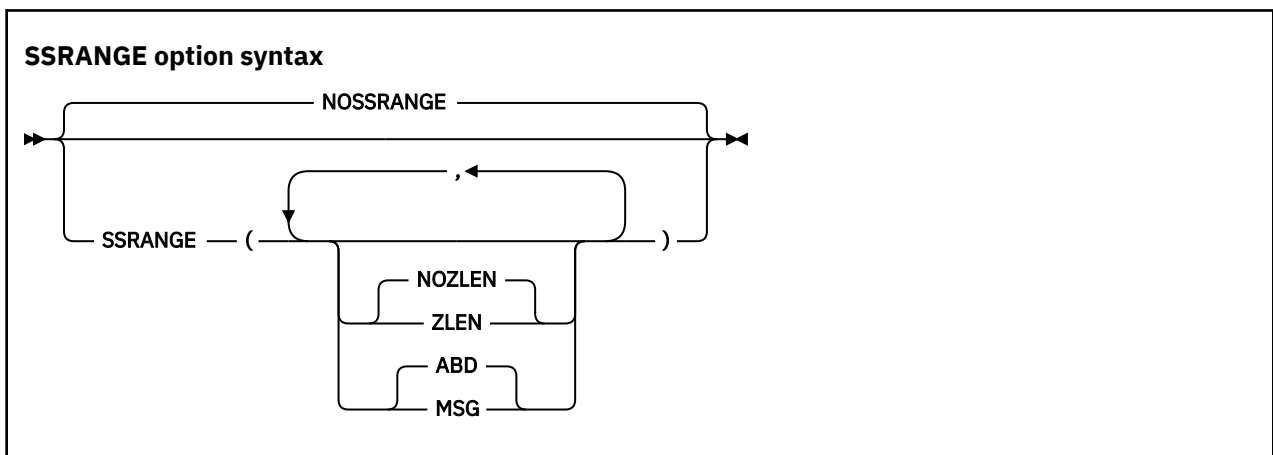
[“Compiling with the SQLIMS option” on page 507](#)
[“Separating IMS suboptions” on page 508](#)

Related references

[“Conflicting compiler options” on page 334](#)

SSRANGE

Use SSRANGE to generate code that checks for out-of-range storage references.



Default is: NOSSRANGE

Suboption default is: NOZLEN , ABD if only SSRANGE is specified.

Abbreviations are: SSR | NOSSR

SSRANGE generates code that checks whether subscripts, including ALL subscripts, or indexes try to reference areas outside the region of their associated tables. Each subscript or index is not individually checked for validity. Instead, the effective address is checked to ensure that it does not reference outside the table.

If you specify SSRANGE with no suboptions, it will be accepted as a specification of SSRANGE (NOZLEN , ABD).

Note: If the SSRANGE option is in effect, range checks will be generated by the compiler and the checks will always be conducted at run time. You cannot disable the compiled-in range checks at run time by specifying the runtime option CHECK(OFF).

Variable-length items are also checked to ensure that references are within their maximum defined length.

Reference modification expressions for non-UTF-8 data items and function values are checked to ensure that:

- The starting position is greater than or equal to 1.
- The starting position is not greater than the current length of the subject data item.

- The starting position and length value (if specified) do not reference an area beyond the end of the subject data item.
- The length value (if specified) is greater than or equal to 1.

The ZLEN and NOZLEN suboptions control how the compiler checks reference modification lengths:

- If ZLEN is in effect, the compiler will generate code to ensure that reference modification lengths are greater than or equal to zero. Zero-length reference modification specifications will not get an SSRANGE error at run time.
- If NOZLEN is in effect, the compiler will generate code to ensure that reference modification lengths are greater than or equal to 1. Zero-length reference modification specifications will get an SSRANGE error at run time. This is compatible with how SSRANGE behaved in previous COBOL versions.

The MSG and ABD suboptions control the runtime behavior of the COBOL program when a range check fails.

- If MSG is in effect and a range check fails, a runtime warning message will be issued. Also, the affected statements will still be executed. The program will continue and might potentially identify other out-of-range conditions.
- If ABD is in effect and a range check fails, the first out-of-range condition will result in a runtime error message and the program will ABEND. You can find the next potential out-of-range condition by fixing the first out-of-range condition and then recompiling and running the program again. To identify all other potential out-of-range conditions, you might need to repeat this process several times.

For unbounded groups or their subordinate items, checking is done only for reference modification expressions. Subscripted or indexed references to tables subordinate to an unbounded group are not checked.

Related concepts

[“Reference modifiers” on page 110](#)

Related tasks

[“Checking for valid ranges” on page 439](#)

STGOPT

The STGOPT option controls storage optimization.



Default is: NOSTGOPT

Abbreviations are: S0 | NOS0

If you specify STGOPT, the compiler might discard any or all of the following data items, and does not allocate storage for them.

- Unreferenced LOCAL-STORAGE and WORKING-STORAGE level-77 and level-01 elementary data items
- Level-01 group items if none of their subordinate items are referenced
- Unreferenced special registers

Note: The STGOPT option is ignored for data items that have the VOLATILE clause. For details, see VOLATILE clause in the *Enterprise COBOL for z/OS Language Reference*.

The compiler will not generate code to initialize discarded data items to the values in their VALUE clauses.

In addition, with STGOPT, data items in the LOCAL - STORAGE SECTION can be reordered in memory to optimize performance.

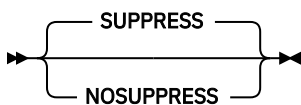
When NOSTGOPT is in effect, the storage for all data items, including unreferenced data items, is allocated by the compiler, data items are never reordered to improve performance, and all data items defined with a VALUE clause are guaranteed to be initialized, even if they are unreferenced.

You can also use the RULES (UNREF | NOUNREFALL | NOUNREFSOURCE) option to control whether to issue warning messages for unreferenced data items. For details, see [“RULES”](#) on page 393.

SUPPRESS

Use the NOSUPPRESS option to ignore the SUPPRESS phrase of all COPY statements in a program so that copybook information can appear in the listing. The copybook information can be used by debuggers, tools, and so on, without users needing to modify their source code.

SUPPRESS option syntax



Default is: SUPPRESS

Abbreviations are: SUPP | NOSUPP

NOSUPPRESS

Ignores the SUPPRESS phrase of COPY statements.

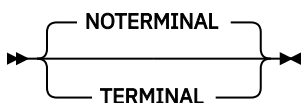
SUPPRESS

Enables the SUPPRESS phrase of COPY statements.

TERMINAL

Use TERMINAL to send progress and diagnostic messages to the SYSTERM ddname.

TERMINAL option syntax



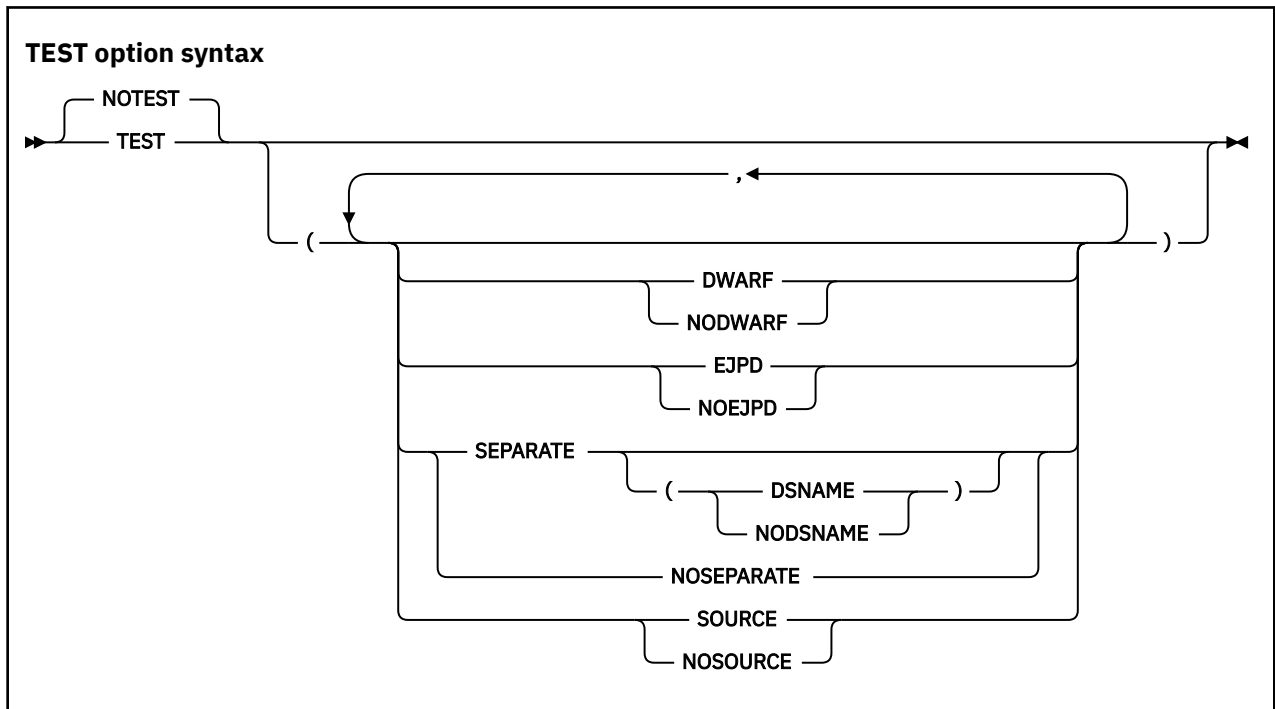
Default is: NOTERMINAL

Abbreviations are: TERM | NOTERM

Use NOTERMINAL if you do not want this additional output.

TEST

Use TEST to produce object code that enables debugging with problem determination tools such as IBM z/OS Debugger and Fault Analyzer. With TEST, you can also enable the inclusion of symbolic variables in the formatted dumps that are produced by Language Environment.



Default is: NOTEST (NODWARF, NOSOURCE, NOSEPARATE)

Suboption defaults are:

- NODWARF, NOSOURCE, NOSEPARATE when NOTEST is specified with no suboptions
- NOEJPD, DWARF, SOURCE, NOSEPARATE when TEST is specified with no suboptions

Abbreviations are: None

Suboption abbreviation are:

- NOSO | SO for SOURCE | NOSOURCE
- NOSEP | SEP for SEPARATE | NOSEPARATE

DWARF | NODWARF

If TEST (DWARF) is in effect, complete DWARF diagnostic information is included in the object program, or a separate debug file when the SEPARATE suboption is in effect. This option enables the best usability for application failure analysis tools, such as CEEDUMP and IBM Fault Analyzer.

When NOTEST (DWARF) is in effect, the debugging information is a subset of the DWARF information that is available with TEST (DWARF). The DWARF diagnostic information that is produced when NOTEST (DWARF) is in effect cannot be used with IBM z/OS Debugger. Consider using NOTEST (DWARF) when use of the debugger is not needed and you want to avoid the performance implications of the TEST option while having improved usability for application failure analysis tools, such as CEEDUMP and IBM Fault Analyzer.

Debugging information generated by the compiler is in the industry-standard DWARF format. For more information about DWARF, see [About Common Debug Architecture](#) in the *DWARF/ELF Extensions Library Reference*.

If NODWARF is in effect, DWARF diagnostic information is not included in the object program, or written to a separate debug file.

Notes:

- SOURCE and SEPARATE are not allowed with NODWARF.
- If you specify the DWARF suboption of TEST or NOTEST, you must set the CODEPAGE option to the CCSID that is used for the COBOL source program. Programs that use Japanese characters in DBCS literals or DBCS user-defined words must be compiled with the CODEPAGE option set to a Japanese codepage CCSID. For more information, see [“CODEPAGE” on page 343](#).

EJPD | NOEJPD

EJPD and NOEJPD control enablement of the IBM z/OS Debugger commands JUMPTO and GOTO in production debugging sessions. EJPD and NOEJPD only take effect if you specify the TEST option and a non-zero OPTIMIZE level (OPTIMIZE (1) or OPTIMIZE (2)).

If you specify TEST (EJPD) and a non-zero OPTIMIZE level:

- The JUMPTO and GOTO commands are enabled.
- The amount of program optimization is reduced. Optimization is done within statements, but most optimizations do not cross statement boundaries.

If you specify TEST (NOEJPD) and a non-zero OPTIMIZE level:

- The JUMPTO and GOTO commands are not enabled, but you can use JUMPTO and GOTO if you use the SET WARNING OFF IBM z/OS Debugger command. In this scenario, JUMPTO and GOTO will have unpredictable results.
- The normal amount of program optimization is done.

Note: EJPD is not allowed with NOTEST.

SOURCE | NOSOURCE

If you specify SOURCE, the DWARF debugging information generated by the compiler includes the expanded source code.

Note: SOURCE is not allowed if NODWARF is specified.

If you specify NOSOURCE, the generated DWARF debugging information does not include the expanded source code. You will not be able to debug using the IBM z/OS Debugger with TEST (NOSOURCE).

SEPARATE [(DSNAME | NODSNAME)] | NOSEPARATE

The default is SEPARATE (NODSNAME) when SEPARATE is specified with no suboptions.

Specify SEPARATE or SEPARATE (NODSNAME) to control program object size on disk (NOSEPARATE does not affect the size of the loaded program object) while retaining debugging capability. Generated DWARF debugging information is written to the SYSDEBUG data set instead of to the object program. See the [section below](#) about controlling module size while retaining debugging capability.

If SEPARATE (DSNAME) is in effect, the SYSDEBUG dataset name used during compilation is stored in the object program. This name will be used as the default at run time when DWARF information is required. The dataset name can be overridden by using the SYSDEBUG COBOL debug file user exit, IGZIUXB. Note that when NODSNAME is in effect, the IGZIUXB user exit provides the only mechanism of locating the DWARF debugging information for the program.

Notes:

- SEPARATE is not allowed if NODWARF is specified.
- Support for debugging DWARF debugging information in the SYSDEBUG data set requires IBM z/OS Debugger 14.1 and later:
 - IBM Debug for z Systems 14.1 (5655-Q50) (formerly IBM Debug Tool for z/OS) or later
 - IBM Developer for z Systems 14.1 (5724-T07) or later
 - IBM Application Delivery Foundation for z Systems 3.1 (5655-AC6) or later

Specify NOSEPARATE to include generated DWARF debugging information in the object program.

Controlling module size while retaining debugging capability:

The DWARF suboption of TEST causes the compiler to generate debug information tables that IBM z/OS Debugger uses to resolve data-names, paragraph-names, and the like. This information can take a lot of

storage. You can choose either to compile this information into the object program or to write it to the separate SYSDEBUG data set:

- For smaller executables, use the SEPARATE suboption and keep the separate debugging files for use during IBM z/OS Debugger sessions.
- To avoid having to manage separate debugging files, compile with the NOSEPARATE suboption; note that this suboption results in larger object programs on DASD. The size when loaded into virtual storage is not increased by use of the NOSEPARATE option.

If you invoke the COBOL compiler from JCL or TSO and you specify NOTEST | TEST (. . . , SEPARATE , . . .), the DWARF debugging information is written to the data set that you specify in the SYSDEBUG DD statement. For details about coding that statement and about the SYSDEBUG data set, see the related information below about defining the debug data set and about logical record length and block size.

When you invoke the COBOL compiler from the z/OS UNIX shell and you specify NOTEST | TEST (. . . , SEPARATE , . . .), the DWARF debugging information is written to *file.dbg* in the current directory, where *file* is the name of the COBOL source file.

Performance versus debugging capability:

You can control the amount of debugging capability that you get and the program performance, as follows:

- For the best performance, but with some restrictions on debugging, compile using a non-zero OPTIMIZE level, STGOPT and TEST (NOEJPD).
 - The IBM z/OS Debugger commands JUMPTO and GOTO are not supported. However, you can still use JUMPTO and GOTO if you use the SET WARNING OFF IBM z/OS Debugger command. In this scenario, JUMPTO and GOTO will have unpredictable results.
 - Except for the DESCRIBE ATTRIBUTES command, IBM z/OS Debugger commands cannot refer to any data item that was discarded from a program by the STGOPT option.
 - The IBM z/OS Debugger command AT CALL *entry-name* is not supported.
- For some reduction in program performance from the production-debugging scenario above, but to enable predictable behavior for the IBM z/OS Debugger commands JUMPTO and GOTO, specify a non-zero OPTIMIZE level and TEST (EJPD).

The restrictions above about referring to items discarded by the STGOPT option, and about the AT CALL command also apply when you use a non-zero OPTIMIZE level and TEST (EJPD).

- For slowest performance but maximum debugging capability, specify OPTIMIZE (0), NOSTGOPT and TEST.

The OPTIMIZE (0) option causes the compiler to generate slower code, but all IBM z/OS Debugger commands are supported.

Language Environment:

The TEST option specified with any of its suboptions, and NOTEST with DWARF, can improve your formatted dumps from Language Environment by adding these two features to the dumps:

- A line number that indicates the failing statement, rather than just an offset
- The values of the program variables (if DWARF is in effect)

With DWARF, the dump will have program variables and the line number of the failing statement. With NODWARF, the dump will not have program variables nor the line number of the failing statement.

Enterprise COBOL uses the Language Environment-provided dump services to produce dumps that are consistent in content and format with those that are produced by other Language Environment-conforming member languages.

Whether Language Environment produces a dump for unhandled conditions depends on the setting of the runtime option TERMTHDACT. If you specify TERMTHDACT (DUMP), a dump is generated when a condition of severity 2 or greater goes unhandled.

Note: IBM z/OS Debugger is a component of the following products:

- IBM Developer for z/OS Enterprise Edition (included in IBM Application Delivery Foundation for z/OS)
- IBM z/OS Debugger (formerly IBM Debug for z Systems and Debug Tool)
- IBM Developer for z/OS

To find out which IBM debug product best suits your needs, see <https://www.ibm.com/docs/developer-for-zos/latest?topic=debugger-overview-zos>.

LLA/VLF managed programs where DWARF diagnostic information is included:

When DWARF diagnostic information is included in the object program because of the TEST (NOSEPARATE) or other equivalent options and the program is staged to VLF, the information cannot be extracted from the program. The execution JCL requires the use of the IGZLOAD DD, the STEPLIB DD, or both to specify the location of any LLA/VLF managed libraries from which the application will load programs that have been staged to VLF. Otherwise, the COBOL variable information and the statement numbers in the traceback will be missing in the CEEDUMP and a diagnostic message will appear in the CEEDUMP. For more information, see *Diagnostic messages in CEEDUMPs for IBM Enterprise COBOL for z/OS* at <https://www.ibm.com/support/pages/diagnostic-messages-ceedumps-ibm-enterprise-cobol-zos>.

For each program staged to VLF, the following steps are taken to extract the DWARF diagnostic information:

1. The COBOL Runtime checks for the allocation of the IGZLOAD DD. It then checks if the library is associated with the IGZLOAD DD contains a member, for example, a COBOL program, whose name matches the program staged to VLF.
2. If the previous step was unsuccessful because the IGZLOAD DD was not allocated or the member was not found, the same process is attempted using the STEPLIB DD.
3. If the member is found, and the checksum that is stored with the DWARF diagnostic information retrieved from the member matches the checksum of the compile unit in the executing program, the information is extracted from the member.

Note that if duplicate members exist in the IGZLOAD DD concatenation or in the STEPLIB DD concatenation respectively, the first occurrence in the DD concatenation is used to obtain the checksum.

A mismatch can occur if a revised copy of the program is placed into a LLA/VLF managed library, but the environment-appropriate refresh for the staged LLA/VLF program has not been completed.

For example, you can allocate the IGZLOAD DD with DSN=USER.COBOL.LOADLIB when all of the following conditions are met:

- USER.COBOL.LOADLIB is in LNKLST.
- USER.COBOL.LOADLIB (PGM1) is staged to VLF.
- PGM1 is executed from USER.COBOL.LOADLIB in LNKLST.

```
//G01      EXEC  PGM=PGM1,                * CALLED FROM LNKLST
//STEPLIB  DD    DSN=CEE1.SCEERUN,DISP=SHR
//IGZLOAD  DD    DSN=USER.COBOL.LOADLIB,DISP=SHR * CONTAINS PGM1
```

If a program is LLA/VLF managed and staged to VLF, and is executed from one of the data sets in LNKLST, there is a need to allocate the IGZLOAD DD. However, if the program resides in one of the data sets in the STEPLIB DD, there is no need to allocate the IGZLOAD DD.

Related concepts

DWARF/ELF Extensions Library Reference (About Common Debug Architecture)

Related tasks

[“Defining the debug data set \(SYSDEBUG\)” on page 272](#)

[“Using the debugger” on page 443](#)

[“Suppressing information in CEEDUMP processing \(IGZ1OPT\)” on page 479](#)
Generating a Language Environment dump with TERMTHDACT
(*Language Environment Debugging Guide*)
Special considerations while using the TEST runtime option
(*IBM z/OS Debugger User's Guide*)

Related references

[“Logical record length and block size” on page 269](#)
[“cob2 input and output files” on page 289](#)
[“Conflicting compiler options” on page 334](#)
[“OPTIMIZE” on page 385](#)
TEST | NOTEST
(*Language Environment Programming Reference*)

THREAD

THREAD indicates that a COBOL program is to be enabled for execution in a Language Environment enclave that has multiple POSIX threads or PL/I tasks.



Default is: NOTHREAD

Abbreviations are: None

A program that has been compiled using the THREAD option can also be used in a nonthreaded application. However, if a COBOL program is to be run in a threaded application, all the COBOL programs in the Language Environment enclave must be compiled using the THREAD option.

NOTHREAD indicates that the COBOL program is not to be enabled for execution in an enclave that has multiple POSIX threads or PL/I tasks.

Programs that are compiled using compilers earlier than Enterprise COBOL are treated as if compiled using NOTHREAD.

If the THREAD option is in effect, the following elements are not supported. If encountered, they are diagnosed as errors:

- ALTER statement
- DEBUG-ITEM special register
- GO TO statement without procedure-name
- INITIAL phrase in PROGRAM-ID clause or INITIAL compiler option
- Nested programs
- RERUN
- Segmentation module
- SORT or MERGE statements
- STOP *literal* statement
- USE FOR DEBUGGING statement

Additionally, some language constructs have different semantics than in the nonthreaded case.

Although threaded applications are subject to a number of programming and environment restrictions, the use of a program in nonthreaded applications is not so restricted. For example, a program compiled using the THREAD option can run in the CICS and IMS environments, can run AMODE 24, and can call and

be called by other programs that are not enabled for multithreading, provided that the application does not contain multiple POSIX threads or PL/I tasks at run time.

Programs compiled using the THREAD option are supported in the reusable environment that is created by calling the Language Environment preinitialization routine CEEPIPI. But a reusable environment created by using the RTEREUS runtime option is not supported for programs compiled using the THREAD option.

Performance consideration: If you use the THREAD option, you can expect some runtime performance degradation due to the overhead of serialization logic that is automatically generated.

AMODE 64 considerations: The THREAD option is ignored when LP(64) is in effect. If the user explicitly specifies the THREAD option, an informational message is issued.

Related tasks

Chapter 31, “Preparing COBOL programs for multithreading,” on page 577

Related references

“Conflicting compiler options” on page 334

TRUNC

TRUNC affects the way that binary data is truncated during moves and arithmetic operations.



Default is: TRUNC(STD)

Abbreviations are: None

TRUNC has no effect on COMP-5 data items; COMP-5 items are handled as if TRUNC(BIN) is in effect regardless of the TRUNC suboption specified.

TRUNC(STD)

TRUNC(STD) applies only to USAGE BINARY receiving fields in MOVE statements and arithmetic expressions. When TRUNC(STD) is in effect, the final result of an arithmetic expression, or the sending field in the MOVE statement, is truncated to the number of digits in the PICTURE clause of the BINARY receiving field.

TRUNC(OPT)

TRUNC(OPT) is a performance option. When TRUNC(OPT) is in effect, the compiler assumes that data conforms to PICTURE specifications in USAGE BINARY receiving fields in MOVE statements and arithmetic expressions. The results are manipulated in the most optimal way, either truncating to the number of digits in the PICTURE clause, or to the size of the binary field in storage (halfword, fullword, or doubleword).

Tips:

- Use the TRUNC(OPT) option only if you are sure that the data being moved into the binary areas will not have a value with larger precision than that defined by the PICTURE clause for the binary item. Otherwise, unpredictable results could occur. This truncation is performed in the most efficient manner possible; therefore, the results are dependent on the particular code sequence generated. It is not possible to predict the truncation without seeing the code sequence generated for a particular statement.

TRUNC(BIN)

The TRUNC(BIN) option applies to all COBOL language that processes USAGE BINARY data. When TRUNC(BIN) is in effect, all binary items (USAGE COMP, COMP-4, or BINARY) are handled as native hardware binary items, that is, as if they were each individually declared USAGE COMP-5:

- BINARY receiving fields are truncated only at halfword, fullword, or doubleword boundaries.
- BINARY sending fields are handled as halfwords, fullwords, or doublewords when the receiver is numeric; TRUNC (BIN) has no effect when the receiver is not numeric.
- The full binary content of fields is significant.
- DISPLAY will convert the entire content of binary fields with no truncation.

Recommendations: TRUNC (BIN) is the recommended option for programs that use binary values set by other products. Other products, such as 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 use TRUNC (OPT) with CICS programs provided that your data conforms to the PICTURE clause for your BINARY data items.

USAGE COMP-5 has the effect of applying TRUNC (BIN) behavior to individual data items. Therefore, you can avoid the performance overhead of using TRUNC (BIN) for every binary data item by specifying COMP-5 on only some of the binary data items, such as those data items that are passed to non-COBOL programs or other products and subsystems. The use of COMP-5 is not affected by the TRUNC suboption in effect.

Large literals in VALUE clauses: When you use the compiler option TRUNC (BIN), numeric literals specified in VALUE clauses for binary data items (COMP, COMP-4, or BINARY) can generally contain a value of magnitude up to the capacity of the native binary representation (2, 4, or 8 bytes) rather than being limited to the value implied by the number of 9s in the PICTURE clause.

Note: When TRUNC (BIN) and NUMCHECK (BIN) are both in effect and an error message or an abend is generated, if you intend to switch to TRUNC (STD|OPT) later for better performance, you must correct the data; if not, you can turn off NUMCHECK (BIN) to reduce the execution time of the application and avoid an error message or an abend.

TRUNC example 1

```
01 BIN-VAR      PIC S99 USAGE BINARY.
   .
   .
   .
   MOVE 123451 to BIN-VAR
```

The following table shows values of the data items after the MOVE statement.

Data item	Decimal	Hex	Display
Sender	123451	00 01 E2 3B	123451
Receiver TRUNC (STD)	51	00 33	51
Receiver TRUNC (OPT)	-7621	E2 3B	2J
Receiver TRUNC (BIN)	-7621	E2 3B	762J

A halfword of storage is allocated for BIN-VAR. The result of this MOVE statement if the program is compiled with the TRUNC (STD) option is 51; the field is truncated to conform to the PICTURE clause.

If you compile the program with TRUNC (BIN), the result of the MOVE statement is -7621. The reason for the unusual result is that nonzero high-order digits are truncated. Here, the generated code sequence would merely move the lower halfword quantity X'E23B' to the receiver. Because the new truncated value overflows into the sign bit of the binary halfword, the value becomes a negative number.

It is better not to compile this MOVE statement with TRUNC (OPT), because 123451 has greater precision than the PICTURE clause for BIN-VAR. With TRUNC (OPT), the results are again -7621. This is because the best performance was obtained by not doing a decimal truncation.

TRUNC example 2

```
01 BIN-VAR      PIC 9(6) USAGE BINARY
```

```
MOVE 1234567891 to BIN-VAR
```

The following table shows values of the data items after the MOVE statement.

Data item	Decimal	Hex	Display
Sender	1234567891	49 96 02 D3	1234567891
Receiver TRUNC (STD)	567891	00 08 AA 53	567891
Receiver TRUNC (OPT)	567891	53 AA 08 00	567891
Receiver TRUNC (BIN)	1234567891	49 96 02 D3	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 47](#)

Related tasks

[“Compiling with the CICS option” on page 487](#)

Related references

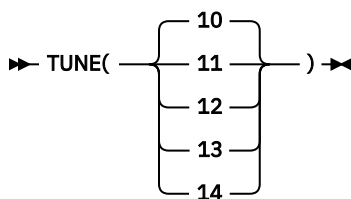
[“NUMCHECK” on page 378](#)

VALUE clause (*Enterprise COBOL for z/OS Language Reference*)

TUNE

The TUNE option specifies the architecture for which the executable program will be optimized.

TUNE option syntax



Default is: The default TUNE level matches the ARCH level if ARCH is specified. If ARCH is not specified, both ARCH and TUNE default to 10.

Abbreviations are: None

Current supported architecture levels and groups of models are as follows:

10

Generates code that is optimized for the 2827-xxx (IBM zEnterprise EC12) and 2828-xxx (IBM zEnterprise BC12) models in IBM z/Architecture mode.

11

Generates code that is optimized for the 2964-xxx (IBM z13) and 2965-xxx (IBM z13s) models in IBM z/Architecture mode.

12

Generates code that is optimized for the 3906-xxx (IBM z14) and 3907-xxx (IBM z14 ZR1) models in IBM z/Architecture mode.

13

Generates code that is optimized for the 8561-xxx (IBM z15) and 8562-xxx (IBM z15 T02) models in IBM z/Architecture mode.

14

Generates code that is optimized for the 3931-xxx (IBM z16) model in IBM z/Architecture mode.

The TUNE option specifies the architecture for which the executable program will be optimized. The TUNE level controls how the compiler selects the available machine instructions, while staying within the restrictions of the ARCH level in effect. The TUNE option does so to provide the highest performance possible on the given TUNE architecture, choosing from the instructions allowed by the given ARCH level.

Note: TUNE impacts performance only; it does not impact the processor model on which you will be able to run your application.

Select TUNE to match the architecture of the machine where your application will run most often. The TUNE level must always be greater or equal to the ARCH level because you will want to tune an application for a machine on which it can run. The compiler enforces this by adjusting TUNE up rather than ARCH down. TUNE does not specify where an application can run and it affects optimization only.

For example, if you have production machines that are z15 and disaster recovery (DR) machines that are z14, you should compile with ARCH (12) and TUNE (13). In this way, your programs will run well on the DR machines and run as fast as possible in production.

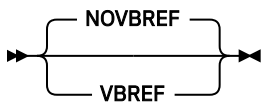
Note: If the specified TUNE level is lower than the specified ARCH level, the compiler adjusts the TUNE level to match the ARCH level.

Related references

[“ARCH” on page 337](#)

VBREF

Use VBREF to get a cross-reference between all statements used in the source program and the line numbers in which they are used. VBREF also produces a summary of the number of times each statement was used in the program.

VBREF option syntax

Default is: NOVBREF

Abbreviations are: None

Use NOVBREF for more efficient compilation.

VLR

The VLR option affects the file status returned from READ statements for variable-length records when the length of record returned is inconsistent with the record descriptions. It eases your migration from

earlier versions to Enterprise COBOL 6, if your programs have READ statements that result in a record length conflict.

VLR option syntax

►► VLR(——— STANDARD ———) ◄◄
 COMPAT

Default is: VLR(STANDARD)

Abbreviations are: VLR(C | S)

After the execution of a READ statement:

- If the number of character positions in the record that is read is less than the minimum size specified by the record description entries for the file, the portion of the record area that is to the right of the last valid character read is undefined.
- If the number of character positions in the record that is read is greater than the maximum size specified by the record description entries for the file, the record is truncated on the right to the maximum size.

In either of these cases, the READ statement is successful, and the file status is set to either 00 (hiding the record length conflict condition) or 04 (indicating that a record length conflict has occurred), depending on the VLR compiler option setting.

VLR(COMPAT)

VLR(COMPAT) checks the size of the read record against the "VERYING IN SIZE FROM min TO max" declaration of the FD clause. If you specify VLR(COMPAT), you get the status value of 00 when READ statements encounter a record length conflict.

Note: This setting can hide I/O problems that can arise with the wrong length read situation. Use the VLR(COMPAT) option with caution, and check for correct READ statements.

VLR(STANDARD)

VLR(STANDARD) checks the size of the read record against the declaration of the FD level 01 clause. If you specify VLR(STANDARD), you get the status value of 04 when READ statements encounter a record length conflict.

You can add code to test for FS=04 to avoid accessing undefined data in a record and also avoid getting protection exceptions for attempting to reference a part of the record that was truncated.

The following example shows how VLR option checks the size of the read record against the declaration of the FD clause and FD level 01 clause.

```
FD MYDD
  block contains 0 records
  record varying in size from 10 to 80
  recording mode V.
01 REC-20
  02 PIC X(20).
01 REC-50.
  02 PIC X(50).
```

Table 52. Length of record read and file status

Length of record read	=5 < min varying < min level 01	= 15 >= min varying < min level 01	= 40 >= min varying <= max varying >= min level 01 <= max level 01	= 70 >= min varying <= max varying > max level 01
COBOL 4	FS=04	FS=00	FS=00	FS=00
COBOL 6 VLR (COMPAT)	FS=04	FS=00	FS=00	FS=00
COBOL 6 VLR (STANDARD)	FS=04	FS=04	FS=00	FS=04

Using VLR (STANDARD) can result in more reliable code and fewer I/O problems because the file status will tell you when a "wrong length READ" might occur. A new compiler message, MSGIGYP3178, can also help you avoid I/O problems by telling you if a program has a possibility of a "wrong length READ". This message can be used to assist with migration from VLR (COMPAT) to VLR (STANDARD) by indicating the possible "wrong length READ" that you can solve by correcting the File Definition (FD). You can also raise the severity of the message so that the program must be corrected in order to run. To do this, use the MSGEXIT suboption of the EXIT compiler option to change the severity of message MSGIGYP3178 from I (RC=0) to S (RC=12), E (RC=8), or W (RC=4). If you are not interested in seeing this message, you can suppress the message completely.

Related references

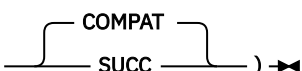
["EXIT" on page 357](#)

Variable length records - wrong length READ
(Enterprise COBOL for z/OS Migration Guide)

VSAMOPENFS

The VSAMOPENFS option affects the user file status reported from successful VSAM OPEN statements that require verified file integrity check.

VSAMOPENFS option syntax

► VSAMOPENFS() ◄

Default is: VSAMOPENFS (COMPAT)

Abbreviations are: VS(C | S)

COMPAT

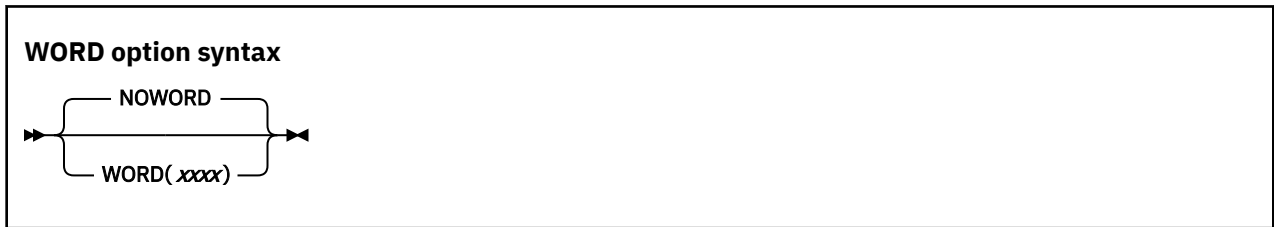
If you specify VSAMOPENFS (COMPAT), the statement returns the file status 97 when a VSAM OPEN statement is successfully verified. This is compatible with pre-COBOL 6 runtime behavior.

SUCC

If you specify VSAMOPENFS (SUCC), the statement returns the file status 00 when a VSAM OPEN statement is successfully verified. This allows users to simply check for 0 in the first digit of the returned file status, as they usually do with other successful operations.

WORD

Use WORD(*xxxx*) to specify that an alternate reserved-word table is to be used during compilation.



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.

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.

Enterprise 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).

Related tasks

[“Compiling with the CICS option” on page 487](#)

Related references

[“Conflicting compiler options” on page 334](#)

[“CICS reserved-word table” on page 490](#)

XMLPARSE

Use XMLPARSE to select the parser to be used for processing XML input, and, therefore, the XML processing capabilities that are available to your program.



Default is: XMLSS

Abbreviations are: XP(X | C)

If you specify the XMLPARSE(XMLSS) option, XML PARSE statements are processed using the z/OS XML System Services parser. The following XML parsing capabilities are available only if you specify XMLPARSE(XMLSS):

- Validation of XML input documents against an XML schema (by using the VALIDATING phrase of the XML PARSE statement)
- Enhanced namespace processing (special registers XML-NAMESPACE, XML-NNAMESPACE, XML-NAMESPACE-PREFIX, and XML-NNAMESPACE-PREFIX)
- Automatic conversion of document fragments to Unicode UTF-16 (by using the RETURNING NATIONAL phrase of the XML PARSE statement)

- Specification of the encoding of the input document (by using the ENCODING phrase of the XML PARSE statement)
- Direct parsing of XML documents encoded in UTF-8
- Parsing of XML documents, a buffer of XML at a time
- Offloading of XML parsing to System z Application Assist Processors (zAAPs)

If you specify the XMLPARSE (COMPAT) option, XML PARSE statements are processed using the XML parser that is a built-in component of the COBOL library. The XML PARSE statement results and operational behaviors are then compatible with those obtained with COBOL 3, and also with COBOL 4 when XMLPARSE (COMPAT) was used, and the advanced features described above for XMLPARSE (XMLSS) are not available.

Related tasks

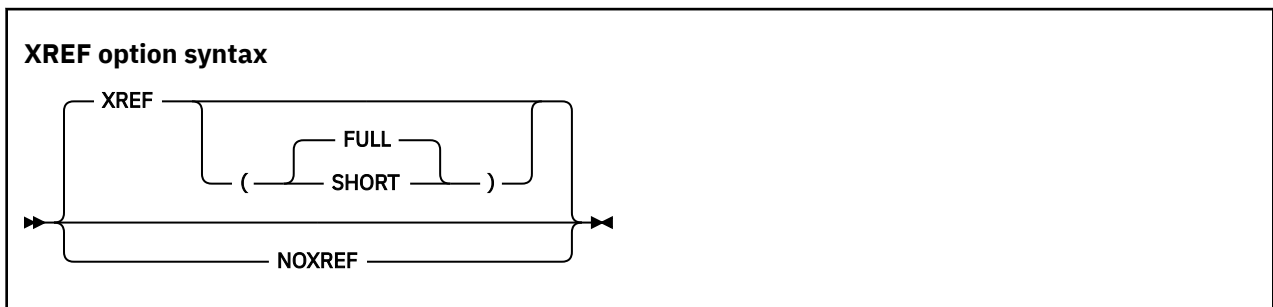
Chapter 37, “Processing XML input,” on page 611

Related references

XML PARSE statement (*Enterprise COBOL for z/OS Language Reference*)
z/OS XML System Services User's Guide and Reference

XREF

Use XREF to produce a sorted cross-reference listing.



Default is: XREF (FULL)

Abbreviations are: X | NOX

You can choose XREF, XREF (FULL), or XREF (SHORT). If you specify XREF without any suboptions, XREF (FULL) will be in effect.

A section of the listing shows all the program-names, data-names, and procedure-names that are referenced in your program, and the line numbers where those names are defined. External program-names are identified.

[“Example: XREF output: data-name cross-references” on page 473](#)

[“Example: XREF output: program-name cross-references” on page 474](#)

A section is also included that cross-references COPY or BASIS statements in the program with the data sets or files from which associated copybooks were obtained.

[“Example: XREF output: COPY/BASIS cross-references” on page 475](#)

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; they are shown before the EBCDIC data-names and procedure-names unless the DBCSXREF installation option is selected with a DBCS ordering program. In that case, DBCS data-names and procedure-names are in the order specified by the DBCS ordering program.

If you use XREF and SOURCE, data-name and procedure-name cross-reference information is printed on the same line as the original source. Line-number references or other information appears on the right-hand side of the listing page. On the right of source lines that reference an intrinsic function, the

letters IFN are printed with the line number of the locations where the function arguments are defined. Information included in the embedded references lets you know if an identifier is undefined (UND) or defined more than once (DUP), if items are implicitly defined (IMP) (such as special registers or figurative constants), or if a program-name is external (EXT).

If you use XREF and NOSOURCE, you get only the sorted cross-reference listing.

XREF (SHORT) prints only the explicitly referenced data items in the cross-reference listing.

XREF (SHORT) applies to DBCS data-names and procedure-names as well as to single-byte names.

NOXREF suppresses this listing.

Usage notes

- Group names used in a MOVE CORRESPONDING statement are in the XREF listing. The elementary names in those groups are also listed.
- In the data-name XREF listing, line numbers that are preceded by the letter M indicate that the data item is explicitly modified by a statement on that line.
- XREF listings take additional storage.
- If there is more than one data set in your SYSLIB concatenation, in some cases the COPY/BASIS cross-reference might be incomplete or missing. This loss can occur if XREF is set only in a CBL or PROCESS statement, and XREFOPT=NO is set as an installation default or NOXREF is coded in your JCL PARM parameter.

To ensure that the COPY/BASIS cross-reference is complete, either verify with your system programmer that XREFOPT=FULL or XREFOPT=SHORT is your installation default, or code the XREF option in your JCL PARM parameter.

Related concepts

[Chapter 21, “Debugging,” on page 433](#)

Related tasks

[“Getting listings” on page 443](#)

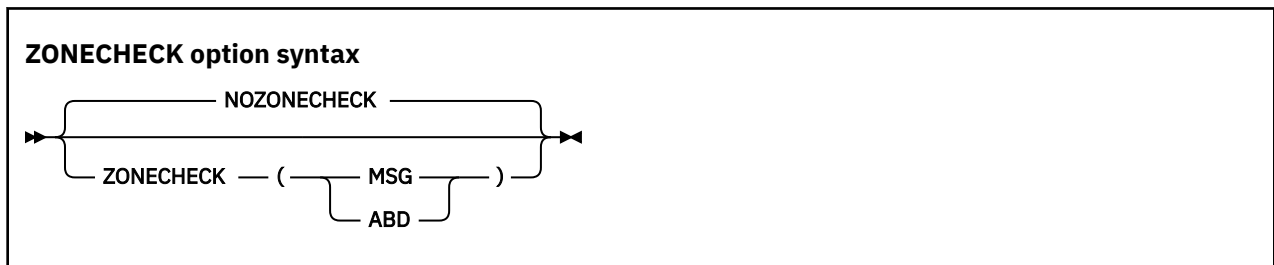
Related references

Language Environment Debugging Guide (COBOL compiler options)

ZONECHECK

Use the ZONECHECK option to have the compiler generate IF NUMERIC class tests for zoned decimal data items that are used as sending data items.

Note: ZONECHECK is deprecated but is tolerated for compatibility, and it is replaced by NUMCHECK(ZON(ALPHNUM)).



Default is: NOZONECHECK

Abbreviations are: NOZC | ZC (MSG) | ZC (ABD)

MSG

The MSG suboption requests that an IF NUMERIC test be done on every use of zoned decimal data items as senders, and if the data is invalid (for example, NOT NUMERIC), a runtime warning message with the line number, data item name, data item content, and program name is issued. Also, the

affected statements will still be executed. ZONECHECK (MSG) is treated as if NUMCHECK (ZON, MSG) were in effect.

ABD

The ABD suboption requests that an IF NUMERIC test be done on every use of zoned decimal data items as senders, and if the data is invalid (for example, NOT NUMERIC), a terminating message is issued that causes an abend. ZONECHECK (ABD) is treated as if NUMCHECK (ZON, ABD) were in effect.

Both ZONECHECK (MSG) and ZONECHECK (ABD) result in the compiler generating an implicit numeric class test for each zoned decimal data item that is referenced as a sender in a COBOL statement. Receivers are not checked, unless they are both a sender and a receiver, such as data item B in the following sample statements:

```
ADD A TO B
```

```
DIVIDE A INTO B
```

```
COMPUTE B = A + B
```

```
INITIALIZE B REPLACING ALPHANUMERIC BY B
```

This checking is done before the data is used in each statement:

- If the data is NOT NUMERIC, either a warning message for ZONECHECK (MSG) or a terminating message for ZONECHECK (ABD) is issued.
- If the data is NUMERIC, the external behavior of the statement is the same as NOZONECHECK, other than being slower.

Restriction: For CALL statements, ZONECHECK checks BY CONTENT and BY VALUE data items that are numeric USAGE DISPLAY only, but it does not check BY REFERENCE parameters.

Performance considerations: ZONECHECK (MSG) and ZONECHECK (ABD) are much slower than NOZONECHECK, depending on how many zoned decimal data items are used in COBOL statements in a program.

Related tasks

[“Checking for incompatible data \(numeric class test\)” on page 53](#)

Related references

[“NUMCHECK” on page 378](#)

[“NUMPROC” on page 382](#)

[“ZONEDATA” on page 418](#)

ZONEDATA

The ZONEDATA option tells the compiler whether the data in USAGE DISPLAY and PACKED-DECIMAL data items is valid, and if not, what the behavior of the compiler should be.

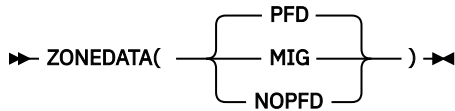
Since most users have valid data in their USAGE DISPLAY and USAGE PACKED-DECIMAL data items, most users should use ZONEDATA (PFD), even if they use NUMPROC (NOPFD). Even if you find that your programs are processing invalid data at run time (using the NUMCHECK compiler option), you should change your programs to avoid processing invalid data, and use ZONEDATA (PFD).

Note: The ZONEDATA option is deprecated but is tolerated for compatibility, and it is replaced by the INVDATA option.

When the ZONEDATA option is specified, it is mapped to the equivalent INVDATA option as follows:

```
ZONEDATA (PFD)      = NOINVDATA
ZONEDATA (NOPFD)   = INVDATA (NOFORCENUMCMP, CLEANSIGN) (or simply INVDATA)
ZONEDATA (MIG)     = INVDATA (FORCENUMCMP, CLEANSIGN)
```

ZONEDATA option syntax



Default is: ZONEDATA(PFD)

Abbreviations are: ZD(PFD) | ZD(MIG) | ZD(NOPFD)

Each digit of a valid zoned decimal number is represented by a single byte from X'F0' through X'F9'. The 4 high-order bits of each byte are zone bits, and the 4 low-order bits of each byte contain the value of the digit. The 4 high-order bits of the low-order byte for SIGN TRAILING represent the sign of the item. The sign is in the high-order byte with SIGN LEADING, or in a separate byte for SIGN IS SEPARATE.

ZONEDATA(PFD)

When the ZONEDATA(PFD) option is in effect, the compiler assumes that all data in USAGE DISPLAY and PACKED-DECIMAL data items is valid, and generates the most efficient code possible to make numeric comparisons. For example, the compiler might generate a string comparison to avoid numeric conversion.

ZONEDATA(MIG)

When the ZONEDATA(MIG) option is in effect, the compiler generates instructions to do numeric comparisons that ignore the zone bits of each digit in zoned decimal data items. For example, the zoned decimal value is converted to packed-decimal with a PACK instruction before the comparison. The compiler will also avoid performing known optimizations that might produce a different result than COBOL 4 (or earlier versions) when a zoned decimal or packed decimal data item has invalid digits or an invalid sign code, or when a zoned decimal data item has invalid zone bits.

ZONEDATA(NOPFD)

When the ZONEDATA(NOPFD) option is in effect, the compiler generates instructions for numeric comparisons or an alphanumeric comparison of zoned decimal data in the same manner as COBOL 4 (or earlier versions) does when using NUMPROC(NOPFD | PFD) with COBOL 4 (or earlier versions):

- In the cases where COBOL 4 (or earlier versions) considered the zone bits, the compiler generates an alphanumeric comparison which will also consider the zone bits of each digit in zoned decimal data items. The zoned decimal value remains as zoned decimal.
- In the cases where COBOL 4 (or earlier versions) ignored the zone bits, the compiler generates numeric comparisons that ignore the zone bits of each digit in zoned decimal data items. The zoned decimal value is converted to packed-decimal with a PACK instruction before the comparison.

In order for the compiler to handle zone bits in the same way as COBOL 4 (or earlier versions) did when generating comparisons of zoned decimal data, the NUMPROC suboption used in COBOL 6 must match the NUMPROC suboption used in COBOL 4 (or earlier versions):

- To get the COBOL 4 (or earlier versions) NUMPROC(NOPFD) behavior in COBOL 6, use ZONEDATA(NOPFD) and NUMPROC(NOPFD) in COBOL 6.
- To get the COBOL 4 (or earlier versions) NUMPROC(PFD) behavior in COBOL 6, use ZONEDATA(NOPFD) and NUMPROC(PFD) in COBOL 6.

The compiler will also avoid performing known optimizations that might produce a different result than COBOL 4 (or earlier versions) when a zoned decimal or packed decimal data item has invalid digits or an invalid sign code, or when a zoned decimal data item has invalid zone bits.

Note: The sign code must be a valid sign code according to the NUMPROC compiler option setting. In addition, the low-order byte must have a valid zone (x'F') for unsigned and signed with either SIGN IS LEADING or SIGN IS SEPARATE.

Note: The ZONEDATA option affects the behavior of MOVE statements, comparisons, and computations for USAGE DISPLAY or PACKED-DECIMAL data items that could contain invalid digits, an invalid sign code, or invalid zone bits.

In the following example, you can see a data item with an invalid zone bit 4 in the zone bits in the middle of data item VALUE1, forced in by REDEFINES:

```
77 VALUE0 PIC X(4) VALUE '00 0'.          *> x'F0F040F0'  
77 VALUE1 REDEFINES VALUE0 PIC 9(4).  
PROCEDURE DIVISION.  
  IF VALUE1 = ZERO  
    DISPLAY 'ZONEDATA(MIG) is in effect ' VALUE1  
  ELSE  
    DISPLAY 'ZONEDATA(NOPFD | PFD) is in effect ' VALUE1  
  END-IF
```

In this example:

- With COBOL 4 (or earlier versions) or earlier versions, the test is true if the NUMPROC (MIG) option is used, and false for NUMPROC (NOPFD | PFD).
- With COBOL 5 or later versions:
 - When using ZONEDATA (PFD), the test is true at OPT(0) and false at OPT(1 | 2).
 - When using ZONEDATA (NOPFD), the test is false at any OPT setting.

In all, to ease your migration to COBOL 6:

- If your digits, sign codes, and zone bits are valid, use ZONEDATA (PFD) and the same NUMPROC setting that you used with COBOL 4 (or earlier versions) when using COBOL 6.
- If you have invalid digits, invalid sign codes, or invalid zone bits in your data, change your programs or systems so that your programs do not have invalid data in numeric data items at run time.

Once you have corrected your programs or systems, you can use the preferred ZONEDATA (PFD) option. Only if you cannot contain this work and must continue to run with invalid data, consider the following choices for ZONEDATA:

- If you used NUMPROC (MIG) with COBOL 4 (or earlier versions), use ZONEDATA (MIG) and NUMPROC (NOPFD) with COBOL 6.
- If you used NUMPROC (NOPFD) with COBOL 4 (or earlier versions), use ZONEDATA (NOPFD) and NUMPROC (NOPFD) with COBOL 6.
- If you used NUMPROC (PFD) with COBOL 4 (or earlier versions), use ZONEDATA (NOPFD) and NUMPROC (PFD) with COBOL 6.

Note: It is not always possible to entirely match the behavior of the old compiler even with these options when faced with clearly invalid data. For example, even for compares, ZONEDATA (NOPFD) isn't going to give the same result in all cases as COBOL 4.

Performance considerations: ZONEDATA (PFD) gives better runtime performance than ZONEDATA (NOPFD | MIG) does. ZONEDATA (NOPFD | MIG) disables some of the optimizations that NUMPROC (PFD) can give.

Related tasks

[“Checking for incompatible data \(numeric class test\)” on page 53](#)

Related references

[“NUMCHECK” on page 378](#)

[“NUMPROC” on page 382](#)

[“ZONECHECK” on page 417](#)

ZWB

If you compile using ZWB, the compiler removes the sign from a signed zoned decimal (DISPLAY) field before comparing this field to an alphanumeric elementary field during execution.

ZWB option syntax



Default is: ZWB

Abbreviations are: None

If the zoned decimal item is a scaled item (that is, it contains the symbol P in its PICTURE string), comparisons that use the decimal item are not affected by ZWB. Such items always have their sign removed before the comparison is made to an alphanumeric field.

ZWB affects how a program runs. The same COBOL program can produce different results depending on the setting of this option.

Use NOZWB if you want to test input numeric fields for SPACES.

COBOL runtime options

This section lists the COBOL runtime options.

COBOL runtime options	Purposes
AIXBLD NOAIXBLD	Invokes the access method services (AMS) for VSAM indexed and relative data sets to complete the file and index definition procedures for COBOL programs. It builds the alternate index at runtime.
CBLOPTS	Specifies the relative order in which program arguments and runtime options are expected. The default CBLOPTS(ON) implies program arguments are first followed by a "/" and then runtime options.
CBLPSHPOP	Controls if CICS PUSH HANDLE and CICS POP HANDLE are used.
CBLQDA	Controls COBOL QSAM dynamic allocation.
CHECK	Enables range checking if compiled with SSRANGE. Ignored for applications compiled with Enterprise COBOL 5 and later.
DEBUG NODEBUG	Activates the COBOL batch debugging features specified by the USE FOR DEBUGGING declarative.
FLOW	Controls the FLOW output produced by OS/VS COBOL programs.
RTEUSE	Implicitly initializes the runtime environment to be reusable.
SIMVRD NOSIMVRD	Specifies if your COBOL programs use a VSAM KSDS to simulate variable-length relative organization data sets.

<i>Table 53. COBOL runtime options (continued)</i>	
COBOL runtime options	Purposes
UPSI	Sets the eight UPSI switches on or off for applications that use COBOL programs.
Amode3164	Must be specified when running mixed AMODE COBOL programs.
SKIPDWARF=ON OFF	Suppresses information in CEEDUMP processing. SKIPDWARF=OFF is the default. See “Suppressing information in CEEDUMP processing (IGZ1OPT)” on page 479.
QSAMBUFFATL=ON OFF	Allocates QSAM buffers above the line (ATL). QSAMBUFFATL=OFF is the default. See "Requesting QSAM buffers above the line (IGZ3OPT)" in the <i>Enterprise COBOL Migration Guide</i> .
QSAMBUFFINITCHAR=DEFAULT SPACE BINZERO	Buffers initialization character. QSAMBUFFINITCHAR=DEFAULT is the default. See "Controlling initialization of QSAM buffer (IGZ4OPT)" in the <i>Enterprise COBOL Migration Guide</i> .
VSAMDYNAMICDIR=ON OFF	Allows you to select the read logic path for VSAM KSDS/RRDS files that use the Dynamic access mode. VSAMDYNAMICDIR=OFF is the default. See “VSAM dynamic access read option VSAMDYNAMICDIR” on page 424.
DISABLEUOPTREPORT=ON OFF	COBOL Runtime option report is generated by default if you specify the LE runtime option RPTOPTS(ON). This report can be turned off with the use of this option. DISABLEUOPTREPORT=OFF is the default.

For more details about these options, see [Using the Language Environment runtime options](#) in the *z/OS Language Environment Programming Reference*.

Related references

"Runtime options that affect runtime performance" in the *Performance Tuning Guide*

Specifying COBOL runtime options

This section describes the ways to specify COBOL runtime options available in the COBOL runtime environment.

- Via the Language Environment (LE) options method. For example, PARM= or CEEOPTS DD statement. For details, see [How to specify runtime options](#) in the *z/OS Language Environment Programming Guide*.
- Via the IGZOPTS DD statement or the environment variable _IGZ_RUNOPTS. For details about how to specify runtime options via this method, see [“Specifying COBOL only runtime options”](#) on page 327.
- Via a customizable COBOL runtime option control block called IGZUOPT for AMODE 31 or IGZQUOPT for AMODE 64. See [“Building IGZUOPT and IGZQUOPT runtime option control block”](#) on page 425.

Each runtime option can be specified via a different method. The following tables show how a specific runtime option can be specified.

<i>Table 54. AMODE 31 COBOL runtime options</i>			
	LE runtime options	IGZOPTS DD / _IGZ_RUNOPTS	IGZUOPT
AIXBLD NOAIXBLD	X		
CBLOPTS	X		
CBLPSHPOP	X		
CBLQDA	X		
CHECK	X		
DEBUG NODEBUG	X		
FLOW NOFLOW	X		
RTEUSE NORTEREUS	X		
SIMVRD NOSIMVRD	X		
UPSI	X		
AMODE3164		X	
SKIPDWARF			X
QSAMBUFFATL			X
QSAMBUFFINITCHAR			X
VSAMDYNAMICDIR			X
DISABLEUOPTREPORT			X

<i>Table 55. AMODE 64 COBOL runtime options</i>			
	LE runtime options	IGZOPTS DD / _IGZ_RUNOPTS	IGZQUOPT
AIXBLD NOAIXBLD		X	
CBLQDA		X	
DEBUG NODEBUG		X	
SIMVRD NOSIMVRD		X	
UPSI		X	
AMODE3164		X	
SKIPDWARF			X
QSAMBUFFATL			X
QSAMBUFFINITCHAR			X
VSAMDYNAMICDIR			X
DISABLEUOPTREPORT			X

Table 55. AMODE 64 COBOL runtime options (continued)

	LE runtime options	IGZOPTS DD / _IGZ_RUNOPTS	IGZQUOPT
<p>Note: The following runtime options are not supported for AMODE 64:</p> <ul style="list-style-type: none"> • CBLOPTS • CBLPSHPOP • CHECK • FLOW • RTEUSE 			

VSAM dynamic access read option VSAMDYNAMICDIR

To change the dynamic access method of VSAM to a direct read-by-key request to the VSAM engine, you must specify the COBOL runtime option VSAMDYNAMICDIR. This alternate logic path takes advantage of IBM's hardware disk systems that make use of zHyperLink. zHyperLink improves application response time, which cuts I/O-sensitive workload response times significantly. If you do not change the dynamic access method, the default runtime behavior is to point to a record by key and then read records sequentially.

Note: You can take advantage of zHyperLink by changing the dynamic access method of VSAM. If you do not have zHyperLink installed, do not use VSAMDYNAMICDIR. To determine if you have zHyperLink available, see [zHyperLink I/O](#) in the IBM z/OS documentation.

Important: You can control which applications make use of this option by following the instructions in [“Building IGZUOPT and IGZQUOPT runtime option control block”](#) on page 425.

To use this feature, create the IGZUOPT (for AMODE 31) or IGZQUOPT (for AMODE 64) COBOL runtime option module that will use the VSAMDYNAMICDIR=ON option.

When the IGZUOPT or IGZQUOPT module is built, specify the VSAMDYNAMICDIR option with the following syntax:

```
IGZUOPT VSAMDYNAMICDIR=ON | OFF
```

If VSAMDYNAMICDIR is not specified, the default is VSAMDYNAMICDIR=OFF.

For more details, see [“Building IGZUOPT and IGZQUOPT runtime option control block”](#) on page 425.

See the sample JCL IGZ5OPT in the Language Environment sample data set .SCEESAMP to learn how to turn on VSAMDYNAMICDIR.

Disabling COBOL runtime options report option DISABLEUOPTREPORT

A COBOL runtime options report is generated if the LE runtime option RPTOPTS(ON) is specified. If you need to suppress the COBOL runtime options report, you can create the IGZUOPT (for AMODE 31) or IGZQUOPT (for AMODE 64) COBOL runtime option module that will use DISABLEUOPTREPORT=ON.

When the IGZUOPT or IGZQUOPT module is built, specify the DISABLEUOPTREPORT option with the following syntax:

```
IGZUOPT DISABLEUOPTREPORT=ON | OFF
```

If DISABLEUOPTREPORT is not specified, the default is DISABLEUOPTREPORT=OFF.

See the sample JCL IGZ6OPT in the Language Environment sample data set .SCEESAMP to learn how to submit a JCL job that alters this option.

Building IGZUOPT and IGZQUOPT runtime option control block

The IGZUOPT (for AMODE 31) and IGZQUOPT (for AMODE 64) modules refer to load modules that you build and provide during COBOL program execution time. These modules are assembler programs that use the assembler macro IGZXOPT.

The IGZXOPT macro is an assembler macro that is used in an assembler program that must be assembled and then linked into a load module called either IGZUOPT or IGZQUOPT applications. This load module, IGZUOPT or IGZQUOPT, must be available in a load module PDS that appears in a STEPLIB DD statement for your running application. The COBOL runtime searches for this module when needed and based on the module's settings, sets runtime option control flags. If no such module is found, IBM defaults will prevail.

Important: You should only add this option control block load module into the STEPLIB stream for applications that are meant to use the specified options. In particular, when you use the VSAMDYNAMICDIR option, you should be aware of which applications should take advantage of this feature because it has impact on performance.

JCL samples for COBOL IGZUOPT runtime options

The compiler dataset .SCEESAMP contains JCL samples that build the IGZUOPT COBOL runtime control block. The JCL performs an ASSEMBLE and a LINK step. The assembler compiles the sample assembler code that uses the IGZXOPT macro to define custom COBOL runtime options. The LINK step builds the load module that is required by the COBOL application at run time.

Options	Samples
SKIPDWARF	SCEESAMP(IGZ1OPT)
QSAMBUFFFATL	SCEESAMP(IGZ3OPT)
QSAMBUFFINITCHAR	SCEESAMP(IGZ4OPT)
VSAMDYNAMICDIR	SCEESAMP(IGZ5OPT)
DISABLEUOPTREPORT	SCEESAMP(IGZ6OPT)

Related tasks

[“Building IGZUOPT and IGZQUOPT runtime option control block” on page 425](#)

[“Suppressing information in CEEDUMP processing \(IGZ1OPT\)” on page 479](#)

["Requesting QSAM buffers above the line \(IGZ3OPT\)" in the Migration Guide](#)

["Controlling initialization of QSAM buffer \(IGZ4OPT\)" in the Migration Guide](#)

[“VSAM dynamic access read option VSAMDYNAMICDIR” on page 424](#)

[“Disabling COBOL runtime options report option DISABLEUOPTREPORT” on page 424](#)

COBOL runtime options report

With the PTF for APAR PH56036 for AMODE 31 or PH56037 for AMODE 64 installed, the COBOL runtime options report is automatically generated if you specify the LE runtime option RPTOPTS(ON). This report appears in addition to the LE runtime option report.

If you do not need this report to be generated, you must create an IGZUOPT (for AMODE 32) or IGZQUOPT (for AMODE 64) module that specifies the COBOL customization runtime option DISABLEUOPTREPORT=ON.

Only runtime options entered by IGZOPTS DD, _IGZ_RUNOPTS or the IGZUOPT or IGZQUOPT module are listed in the COBOL runtime options report.

Note: The COBOL runtime options report indicates the following states under the LAST WHERE SET column:

- "From LE Option setting": No IGZUOPT module was found. All settings are defaulted to the base LE option settings.

- "IGZUOPT": The IGZUOPT module was found, and the setting was taken from that module.
- "DD:IGZOPTS": The runtime option was specified or found via the IGZOPTS DD statement.
- "_IGZ_RUNOPTS": The runtime option was specified or found via the _IGZ_RUNOPTS environment variable.

Examples

The following table has example scenarios for the runtime options:

Numbers	Scenarios	Results
Example 1	No IGZUOPT module was found. RPTOPTS(ON), aixbld, and DEBUG were specified.	Both COBOL and LE runtime options reports are generated.
Example 2	IGZUOPT module was found. RPTOPTS(ON), aixbld, DEBUG, and DISABLEUOPTREPORT=ON were specified.	Only the LE runtime options report is generated.

Example 1 report output is as follows:

```
COBOL Run-time Options Report ...
This report supplements Language Environment Options Report
```

```
LAST WHERE SET          OPTION
-----
From LE Option setting   NOAMODE3164
From LE Option setting   NOSKIPDWARF
From LE Option setting   NOQSAMBUFATL
From LE Option setting   BUFFINITCHAR(DEFAULT)
From LE Option setting   NOVSAMDYNAMICDIR
From LE Option setting   NODISABLEUOPTREPORT
```

```
End of COBOL Run-time Option Report
```

```
Options Report for Enclave ...
Language Environment V02 R04.00
```

```
LAST WHERE SET          OPTION
-----
IBM-supplied default    ABPERC(NONE)
IBM-supplied default    ABTERMENC(ABEND)
Invocation command      AIXBLD
IBM-supplied default    ALL31(ON)
...
IBM-supplied default    COUNTRY(US)
Invocation command      DEBUG
...
```

Example 2 output is as follows:

```
Options Report for Enclave ...
Language Environment V02 R04.00
```

```
LAST WHERE SET          OPTION
-----
IBM-supplied default    ABPERC(NONE)
IBM-supplied default    ABTERMENC(ABEND)
Invocation command      AIXBLD
IBM-supplied default    ALL31(ON)
...
IBM-supplied default    COUNTRY(US)
Invocation command      DEBUG
...
```

Table 58. Examples for AMODE 64 runtime scenarios

Numbers	Scenarios	Results
Example 3	IGZUOPT module was found. VSAMDYNAMICDIR=ON was specified.	Both COBOL and LE runtime options reports are generated.
Example 4	IGZUOPT module found. DISABLEUOPTREPORT=ON was specified	Only the LE runtime options report is generated.

Example 3 report output is as follows:

```
COBOL Run-time Options Report ...
This report supplements Language Environment Options Report
```

```
LAST WHERE SET          OPTION
-----
From LE Option setting  NOAIXBLD
From LE Option setting  NOSIMVRD
From LE Option setting  NODEBUG
From LE Option setting  CBLQDA(OFF)
From LE Option setting  NOAMODE3164
IGZUOPT                 NOSKIPDWARF
IGZUOPT                 NOQSAMBUFATL
IGZUOPT                 BUFFINITCHAR(DEFAULT)
IGZUOPT                 VSAMDYNAMICDIR
IGZUOPT                 NODISABLEUOPTREPORT
```

End of COBOL Run-time Option Report

```
Options Report for Enclave PGMA ...
Language Environment V02 R04.00
```

```
LAST WHERE SET          OPTION
-----
IBM-supplied default    CEEDUMP(60,SYSOUT=*,FREE=END,SPIN=UNALLOC)
PARMLIB(CEEPRM0W)       DYNDUMP(*USERID,NODYNAMIC,NOTDUMP)
IBM-supplied default    ENVAR("")
...
```

Example 4 report output is as follows:

```
Options Report for Enclave PGMA ...
Language Environment V02 R04.00
```

```
LAST WHERE SET          OPTION
-----
IBM-supplied default    CEEDUMP(60,SYSOUT=*,FREE=END,SPIN=UNALLOC)
PARMLIB(CEEPRM0W)       DYNDUMP(*USERID,NODYNAMIC,NOTDUMP)
IBM-supplied default    ENVAR("")
...
```


Chapter 20. Compiler-directing statements

Several statements help you to direct the compilation of your program.

These are the compiler-directing statements:

BASIS statement

This extended source program library statement provides a complete COBOL program as the source for a compilation. For rules of formation and processing, see the description of *text-name* for the COPY statement.

CALLINTERFACE directive

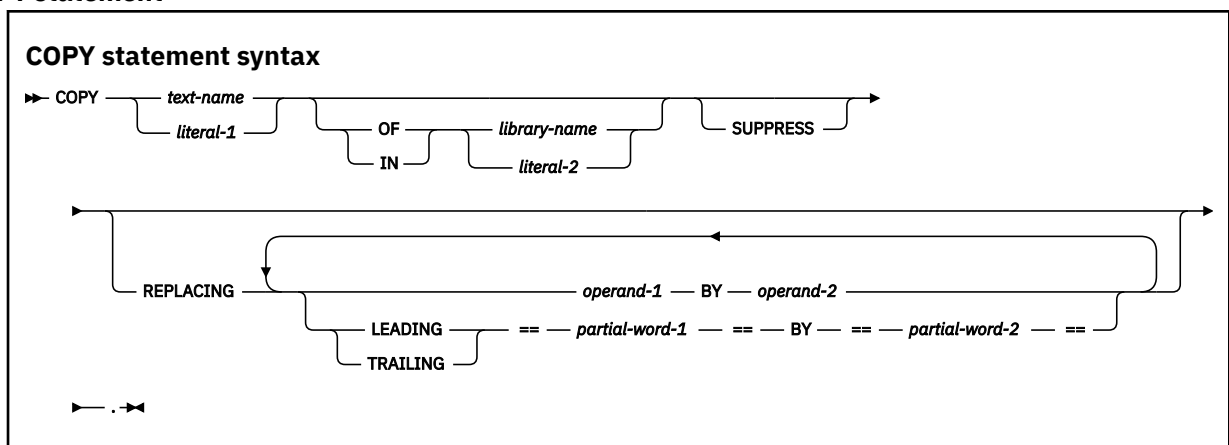
The CALLINTERFACE directive specifies the interface convention for CALL and SET statements. The convention specified stays in effect until another CALLINTERFACE directive is encountered in the source.

The CALLINTERFACE directive can be used only in the procedure division and its effect is limited to the current compilation unit.

***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.

Neither *text-name* nor *library-name* need to be unique within a program. They can be identical to other user-defined words in the program, except that they cannot contain the underscore.

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.

Compiling with JCL:

text-name, *library-name*, and *literal-1* and *literal-2* are processed as follows:

- The name (which can be from 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 any 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 issued.
- If the first character is numeric, then the characters 1-9 are translated to A-I, zero (0) is converted to J, and a warning message is issued.

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 DSN=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 in your DD statement for SYSLIB. Alternatively, define multiple DD statements and include the IN/OF phrase in your COPY statements.

The maximum block size for the copy library depends on the device on which your data set resides.

Compiling in the z/OS UNIX shell:

When you compile using the cob2 command, copybooks are included from the z/OS UNIX file system. *text-name*, *library-name*, and *literal-1* and *literal-2* are processed as follows:

- User-defined words are folded to uppercase. Literals are not folded. Because UNIX is case sensitive, if your file-name is lowercase or mixed case, you must specify it as a literal.
- If *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"
```

- If *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 that contains the copybook.

If an environment variable of that name is not defined, the copybook is searched for under the following names, in this order:

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*

- If *library-name* is a literal, it is treated as the actual path, relative or absolute, from which to copy file *text-name*.
- If *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 copybook 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
```

If *library-name* is an environment variable that identifies the path from which copybooks are to be copied, use an `export` command to define *library-name*, as in this example:

```
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 (:).

If *library-name* is omitted and *text-name* is not an absolute path name, the copybook 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

For additional information about the `COPY` statement, for example, the rules for text replacement, see the related reference.

DEFINE directive

The `DEFINE` directive defines or undefines a compilation variable. The compilation variables can be used within any of the conditional compilation directives (`DEFINE`, `EVALUATE`, and `IF`). The compilation variable is treated as a symbolic reference to the literal value it currently represents.

DELETE statement

This extended source library statement removes COBOL statements from the BASIS source program.

EJECT statement

This compiler-directing statement specifies that the next source statement is to be printed at the top of the next page.

ENTER statement

The statement is treated as a comment.

EVALUATE directive

The `EVALUATE` directive provides a multi-branch method of choosing the source lines to include in a compilation group.

IF directive

The `IF` directive provides for a one-way or two-way conditional compilation.

INLINE directive

The `INLINE` directive lets you selectively prevent the compiler from considering procedures eligible for inlining.

INSERT statement

This library statement adds COBOL statements to the BASIS source program.

PROCESS (CBL) statement

This statement, which you place 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, and should be used at the resume point for a call to `CEE3SRP`. 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) should be printed at the top of each page of the source listing.

USE statement

The USE statement provides *declaratives* to specify these elements:

- Error-handling procedures: EXCEPTION/ERROR
- Debugging lines and sections: DEBUGGING

Related tasks

[“Changing the header of a source listing” on page 5](#)

[“Specifying compiler options under z/OS” on page 273](#)

[“Specifying compiler options under z/OS UNIX” on page 284](#)

[“Setting environment variables under z/OS UNIX” on page 283](#)

[“Eliminating repetitive coding” on page 772](#)

Related references

[“cob2 syntax and options” on page 287](#)

CALLINTERFACE (*Enterprise COBOL for z/OS Language Reference*)

DEFINE (*Enterprise COBOL for z/OS Language Reference*)

EVALUATE (*Enterprise COBOL for z/OS Language Reference*)

IF (*Enterprise COBOL for z/OS Language Reference*)

INLINE (*Enterprise COBOL for z/OS Language Reference*)

COPY statement (*Enterprise COBOL for z/OS Language Reference*)

Chapter 21. Debugging

You can choose between two different approaches to determine the cause of problems in the behavior of your application: source-language debugging or interactive debugging.

For source-language debugging, COBOL provides several language elements, compiler options, and listing outputs that make debugging easier.

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.

For interactive debugging, you can use z/OS Debugger. z/OS Debugger offers these productivity enhancements:

- Interactive debugging (in full-screen or line mode), or debugging in batch mode

During an interactive full-screen mode session, you can use the full-screen services and session panel windows on a 3270 device to debug your program while 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.

- Mixed-language debugging

You can debug an application that contains programs written in a different language. z/OS Debugger automatically determines the language of the program or subprogram being run.

- COBOL-CICS debugging

z/OS Debugger supports the debugging of CICS applications in both interactive and batch mode.

- Support for remote debugging

You can use the Eclipse-based debug interface and IBM Z Open Debug to debug programs that run on z/OS.

Note: IBM z/OS Debugger is a common component that is provided in the following products:

- IBM Debug for z/OS, previously known as IBM Debug Tool for z/OS
- IBM Developer for z/OS, previously known as IBM Developer for z Systems and Rational Developer for z Systems
- IBM Developer for z/OS (Enterprise Edition), which is also provided with the IBM Application Delivery Foundation for z/OS bundle
- IBM Z and Cloud Modernization Stack

The debug features provided in different products vary. For more information, see [Overview of IBM z/OS Debugger](#).

Depending on the level of Enterprise COBOL that you are using to create the COBOL application, update your debugger to a certain version:

- z/OS Debugger 15.0.5 and later supports Enterprise COBOL 6.4 and earlier versions
- z/OS Debugger 14.2 and later supports Enterprise COBOL 6.3 and earlier versions
- z/OS Debugger 14.1 supports Enterprise COBOL 6.2 and earlier versions
- z/OS Debugger 14.0 supports Enterprise COBOL 6.1 and earlier versions
- Debug Tool 13.1 supports Enterprise COBOL 5.1 and earlier versions

Related tasks

[“Debugging with source language” on page 434](#)

[“Debugging using compiler options” on page 437](#)

[“Using the debugger” on page 443](#)

[“Getting listings” on page 443](#)

[“Suppressing information in CEEDUMP processing \(IGZ1OPT\)” on page 479](#)

IBM z/OS Debugger User's Guide

Related references

IBM z/OS Debugger Reference and Messages

Language Environment Debugging Guide (Formatting and analyzing system dumps, Debugging example COBOL programs)

Debugging with source language

You can use several COBOL language features to pinpoint the cause of a failure in a program.

If a failing program is part of a large application that is already in production (precluding source updates), write a small test case to simulate the failing part of the program. Code debugging features in the test case to help detect these problems:

- Errors in program logic
- Input-output errors
- Mismatches of data types
- Uninitialized data
- Problems with procedures

Related tasks

[“Tracing program logic” on page 434](#)

[“Finding and handling input-output errors” on page 435](#)

[“Validating data” on page 435](#)

[“Moving, initializing or setting uninitialized data” on page 435](#)

[“Generating information about procedures” on page 436](#)

Related references

Source language debugging (*Enterprise COBOL for z/OS Language Reference*)

Tracing program logic

Trace the logic of your program by adding DISPLAY statements.

For example, if you determine that the problem is in an EVALUATE statement or in a set of nested IF statements, use DISPLAY statements in each path to see the logic flow. If you determine that the calculation of a numeric value is causing the problem, use DISPLAY statements to check the value of some interim results.

If you use explicit scope terminators to end statements in your program, the logic is more apparent and therefore easier to trace.

To determine whether a particular routine started and finished, you might insert code like this into your program:

```
DISPLAY "ENTER CHECK PROCEDURE"  
    : (checking procedure routine)  
DISPLAY "FINISHED CHECK PROCEDURE"
```

After you are sure that the routine works correctly, disable the DISPLAY statements in one of two ways:

- Put an asterisk in column 7 of each DISPLAY statement line to convert it to a comment line.
- Put a D in column 7 of each DISPLAY statement to convert it to a comment line. When you want to reactivate these statements, include a WITH DEBUGGING MODE clause in the ENVIRONMENT DIVISION; the D in column 7 is ignored and the DISPLAY statements are implemented.

Before you put the program into production, delete or disable the debugging aids you used and recompile the program. The program will run more efficiently and use less storage.

Related concepts

[“Scope terminators” on page 20](#)

Related references

DISPLAY statement (*Enterprise COBOL for z/OS Language Reference*)

Finding and handling input-output errors

File status keys can help you determine whether your program errors are due to input-output errors occurring on the storage media.

To use file status keys in debugging, check for a nonzero value in the status key after each input-output statement. If the value is nonzero (as reported in an error message), look at the coding of the input-output procedures in the program. You can also include procedures to correct the error based on the value of the status key.

If you determine that a problem lies in an input-output procedure, include the `USE EXCEPTION/ERROR` declarative to help debug the problem. Then, when a file fails to open, the appropriate `EXCEPTION/ERROR` declarative is performed. The appropriate declarative might be a specific one for the file or one provided for the open attributes `INPUT`, `OUTPUT`, `I-O`, or `EXTEND`.

Code each `USE AFTER STANDARD ERROR` statement in a section that follows the `DECLARATIVES` keyword in the `PROCEDURE DIVISION`.

Related tasks

[“Coding ERROR declaratives” on page 246](#)

[“Using file status keys” on page 247](#)

Related references

File status key (*Enterprise COBOL for z/OS Language Reference*)

Validating data

If you suspect that your program is trying to perform arithmetic on nonnumeric data or is receiving the wrong type of data on an input record, use the class test (the class condition) to validate the type of data.

You can use the class test to check whether the content of a data item is `ALPHABETIC`, `ALPHABETIC-LOWER`, `ALPHABETIC-UPPER`, `DBCS`, `KANJI`, or `NUMERIC`. If the data item is described implicitly or explicitly as `USAGE NATIONAL`, the class test checks the national character representation of the characters associated with the specified character class.

You can use the `UVALID` intrinsic function to check whether a national data item contains valid UTF-16 encoded data, or whether an alphanumeric or alphabetic item contains valid UTF-8 encoded data.

Related tasks

[“Coding conditional expressions” on page 95](#)

[“Testing for valid DBCS characters” on page 159](#)

Related references

Class condition (*Enterprise COBOL for z/OS Language Reference*)

`UVALID` (*Enterprise COBOL for z/OS Language Reference*)

Moving, initializing or setting uninitialized data

Use an `INITIALIZE` or `SET` statement to initialize a table or data item when you suspect that a problem might be caused by residual data in those fields.

If the problem happens intermittently and not always with the same data, it could be that a switch was not initialized but is generally set to the right value (0 or 1) by chance. By using a `SET` statement to ensure

that the switch is initialized, you can determine that the uninitialized switch is the cause of the problem or remove it as a possible cause.

Related references

INITIALIZE statement (*Enterprise COBOL for z/OS Language Reference*)

SET statement (*Enterprise COBOL for z/OS Language Reference*)

[“INITCHECK” on page 363](#)

Generating information about procedures

Generate information about your program or test case and how it is running by coding the USE FOR DEBUGGING declarative. This declarative lets you include statements in the program and indicate when they should be performed when you run your program.

For example, to determine how many times a procedure is run, you could include a debugging procedure in the USE FOR DEBUGGING declarative and use a counter to keep track of the number of times that control passes to that procedure. You can use the counter technique to check items such as these:

- How many times a PERFORM statement runs, and thus whether a particular routine is being used and whether the control structure is correct
- How many times a loop runs, and thus whether the loop is executing and whether the number for the loop is accurate

You can use debugging lines or debugging statements or both in your program.

Debugging lines are statements that are identified by a D in column 7. To make debugging lines in your program active, code the WITH DEBUGGING MODE clause on the SOURCE-COMPUTER line in the ENVIRONMENT DIVISION. Otherwise debugging lines are treated as comments.

Debugging statements are the statements that are coded in the DECLARATIVES section of the PROCEDURE DIVISION. Code each USE FOR DEBUGGING declarative in a separate section. Code the debugging statements as follows:

- Only in a DECLARATIVES section.
- Following the header USE FOR DEBUGGING.
- Only in the outermost program; they are not valid in nested programs. Debugging statements are also never triggered by procedures that are contained in nested programs.

To use debugging statements in your program, you must include the WITH DEBUGGING MODE clause and use the DEBUG runtime option.

Options restrictions:

- You cannot use the USE FOR DEBUGGING declarative in a program that you compile with the THREAD option.

[“Example: USE FOR DEBUGGING” on page 436](#)

Related references

SOURCE-COMPUTER paragraph (*Enterprise COBOL for z/OS Language Reference*)

Debugging lines (*Enterprise COBOL for z/OS Language Reference*)

Debugging sections (*Enterprise COBOL for z/OS Language Reference*)

DEBUGGING declarative (*Enterprise COBOL for z/OS Language Reference*)

Example: USE FOR DEBUGGING

This example shows the kind of statements that are needed to use a DISPLAY statement and a USE FOR DEBUGGING declarative to test a program.

The DISPLAY statement writes information to the terminal or to an output 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.

```

The DISPLAY statement in the DECLARATIVES SECTION issues this message every time the procedure Some-Routine runs:

```
Trace For Procedure-Name : Some-Routine 22
```

The number at the end of the message, 22, is the value accumulated in the data item Total; it indicates the number of times Some-Routine has run. The statements in the debugging declarative are performed before the named procedure runs.

You can also use the DISPLAY statement to trace program execution and show the flow through the program. You do this by dropping Total from the DISPLAY statement and changing the USE FOR DEBUGGING declarative in the DECLARATIVES SECTION to:

```
USE FOR DEBUGGING ON ALL PROCEDURES.
```

As a result, a message is displayed before each nondebugging procedure in the outermost program runs.

Debugging using compiler options

You can use certain compiler options to help you find errors in your program, find various elements in your program, obtain listings, and prepare your program for debugging.

You can find the following errors by using compiler options (the options are shown in parentheses):

- Uninitialized data items that are used without being initialized (INITCHECK)
- Syntax errors such as duplicate data-names (NOCOMPILE)
- Invalid data items that are used as sending data items (NUMCHECK)
- Invalid COBOL programs (PARMCHECK)
- Missing sections (SEQUENCE)
- Invalid subscript values (SSRANGE)

You can find the following elements in your program by using compiler options:

- Error messages and locations of the associated errors (FLAG)
- Program entity definitions and references; text-names and library-names from COPY or BASIS statements, and the associated data sets or files from which copybooks are obtained (XREF)

- Data items in the DATA DIVISION (MAP)
- Statement references (VBREF)

You can get a copy of your source (SOURCE) or a listing of generated code (LIST).

You prepare your program for debugging by using the TEST compiler option.

Related tasks

[“Finding coding errors” on page 438](#)

[“Finding line sequence problems” on page 438](#)

[“Checking for invalid COBOL data or invalid COBOL programs” on page 439](#)

[“Checking for valid ranges” on page 439](#)

[“Selecting the level of error to be diagnosed” on page 440](#)

[“Finding program entity definitions and references” on page 442](#)

[“Listing data items” on page 442](#)

[“Getting listings” on page 443](#)

Related references

[“COBOL compiler options” on page 329](#)

Finding coding errors

Use the NOCOMPILE option to compile conditionally or to only check syntax. When used with the SOURCE option, NOCOMPILE produces a listing that will help you find coding mistakes such as missing definitions, improperly defined data items, and duplicate data-names.

If you are compiling in the TSO foreground, you can send the messages to your screen by using the TERM compiler option and defining your data set as the SYSTERM data set.

Checking syntax only: To only check the syntax of your program, and not produce object code, use NOCOMPILE without a suboption. If you also specify the SOURCE option, the compiler produces a listing.

When you specify NOCOMPILE, several compiler options are suppressed. See the related reference below about the COMPILE option for details.

Compiling conditionally: To compile conditionally, use NOCOMPILE(x), where x is one of the severity levels of errors. Your program is compiled if all the errors are of a lower severity than x. The severity levels that you can use, from highest to lowest, are S (severe), E (error), and W (warning).

If an error of level x or higher occurs, the compilation stops and your program is only checked for syntax.

Related references

[“COMPILE” on page 345](#)

Finding line sequence problems

Use the SEQUENCE compiler option to find statements that are out of sequence. Breaks in sequence indicate that a section of a source program was moved or deleted.

When you use SEQUENCE, the compiler checks the source statement numbers to determine whether they are in ascending sequence. Two asterisks are placed beside statement numbers that are out of sequence. The total number of these statements is printed as the first line in the diagnostics after the source listing.

Related references

[“SEQUENCE” on page 395](#)

Checking for invalid COBOL data or invalid COBOL programs

Use INITCHECK and NUMCHECK to determine if your programs process invalid COBOL data at run time. Use PARMCHECK to determine if your programs have mismatched parameters at run time that causes the corruption of data beyond the end of WORKING-STORAGE.

The INITCHECK option checks for uninitialized data items and issue warning messages when they are used without being initialized.

The NUMCHECK option validates the following data items when they are used as sending data items:

- NUMCHECK(ZON) causes the compiler to generate code for an implicit numeric class test for zoned decimal (numeric USAGE DISPLAY) data items that are used as sending data items in COBOL statements.
- NUMCHECK(PAC) causes the compiler to generate code for an implicit numeric class test for packed decimal (COMP-3) data items that are used as sending data items in COBOL statements.
- NUMCHECK(BIN) causes the compiler to generate code similar to ON SIZE ERROR to test if binary data items contents are bigger than the PICTURE clause. Note that this extra code will be generated only for binary data items that are used as sending data items, and COMP-5 data items will not get this ON SIZE ERROR code generated.

The PARMCHECK option finds subprograms that write beyond the end of WORKING-STORAGE. This option tells the compiler to generate an extra data item following the last item in WORKING-STORAGE that is then used at run time to check whether a called subprogram corrupted data beyond the end of WORKING-STORAGE.

Performance consideration: PARMCHECK and NUMCHECK can somewhat degrade performance because of the extra overhead to check for invalid COBOL data. PARMCHECK will cause the compiler to generate slower code for programs with CALL statements. NUMCHECK is updated to remove redundant checks for invalid data, thus improving runtime performance. However, performance is still the best when using NONUMCHECK, and will be better at OPT(1) and OPT(2) than at OPT(0).

Related references

[“INITCHECK” on page 363](#)

[“NUMCHECK” on page 378](#)

[“PARMCHECK” on page 387](#)

[“Performance-related compiler options” on page 764](#)

Checking for valid ranges

Use the SSRANGE compiler option to check whether addresses fall within proper ranges.

SSRANGE causes the following addresses to be checked:

- Subscripted or indexed data references: Is the effective address of the specified table element within the maximum boundary of the containing group? (This checking is not done for UNBOUNDED tables and groups.)
- Variable-length data references (a reference to a data item that contains an OCCURS DEPENDING ON clause): Is the actual length greater than or equal to zero, and within the maximum defined length for the group data item? (This checking is not done for UNBOUNDED groups.)
- Reference-modified data references: Are the offset and length positive? Is the sum of the offset and length within the maximum length for the data item?

If the SSRANGE option is in effect, checking is performed at run time if the COBOL statement that contains the indexed, subscripted, variable-length, or reference-modified data item is executed.

If an effective address is outside the range of the data item that contains the referenced data, an error message is generated and the program stops. The message identifies the table or identifier that was referenced and the line number where the error occurred. Additional information is provided depending on the type of reference that caused the error.

If all subscripts, indices, and reference modifiers in a given data reference are literals and they result in a reference outside the data item, the error is diagnosed at compile time regardless of the setting of the SSRANGE option.

Performance consideration: SSRANGE can somewhat degrade performance because of the extra overhead to check each subscripted or indexed item.

Related references

[“SSRANGE” on page 401](#)

[“Performance-related compiler options” on page 764](#)

Selecting the level of error to be diagnosed

Use the FLAG compiler option to specify the level of error to be diagnosed during compilation and to indicate whether error messages are to be embedded in the listing. Use FLAG (I) or FLAG (I, I) to be notified of all errors.

Specify as the first parameter the lowest severity level of the syntax-error messages to be issued. Optionally specify the second parameter as the lowest level of the syntax-error messages to be embedded in the source listing. This severity level must be the same or higher than the level for the first parameter. If you specify both parameters, you must also specify the SOURCE compiler option.

Severity level	Resulting messages
U (unrecoverable)	U messages only
S (severe)	All S and U messages
E (error)	All E, S, and U messages
W (warning)	All W, E, S, and U messages
I (informational)	All messages

When you specify the second parameter, each syntax-error message (except a U-level message) is embedded in the source listing at the point where the compiler had enough information to detect that error. All embedded messages (except those issued by the library compiler phase) directly follow the statement to which they refer. The number of the statement that had the error is also included with the message. Embedded messages are repeated with the rest of the diagnostic messages at the end of the source listing.

Note: You can suppress some error messages and change the severity of others with the MSGEXIT suboption of the [EXIT](#) option.

When you specify the NOSOURCE compiler option, the syntax-error messages are included only at the end of the listing. Messages for unrecoverable errors are not embedded in the source listing, because an error of this severity terminates the compilation.

[“Example: embedded messages” on page 441](#)

Related tasks

[“Generating a list of compiler messages” on page 281](#)

Related references

[“Severity codes for compiler diagnostic messages” on page 282](#)

[“Messages and listings for compiler-detected errors” on page 281](#)

[“FLAG” on page 359](#)

Example: embedded messages

The following example shows the embedded messages generated by specifying a second parameter to the FLAG option. Some messages in the summary apply to more than one COBOL statement.

```

LineID  PL SL  ----+---A-1-B---+----2-----3-----4-----5-----6-----7-|---+----8  Map
and Cross Reference
...
090671**      /
090672**      *****
090673**      ***          I N I T I A L I Z E   P A R A G R A P H          **
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
...
LineID  Message code  Message text
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  "ASSIGNN" was invalid. Scanning was resumed at the next area "A" item, level-
number, or
193  IGYGR1207-S  The "ASSIGN" clause was missing or invalid in the "SELECT" entry for file
"LOCATION-FILE".
269  IGYDS1066-S  "REDEFINES" object "WS-DATE" was not the immediately preceding level-1 data
item.
90602  IGYPS2052-S  The "REDEFINES" clause was discarded.
this file  An error was found in the definition of file "LOCATION-FILE". The reference to
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		11	

```

* Statistics for COBOL program IGYTCARA:
*   Source records = 1755
*   Data Division statements = 295
*   Procedure Division statements = 479
*   Generated COBOL statements = 0
*   Program complexity factor = 486
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. Use it also to produce a cross-reference of COPY or BASIS statements to the data sets or files from which copybooks were obtained.

A sorted cross-reference includes the line number where the data-name, procedure-name, or program-name was defined and the line numbers of all references to it.

To include only the explicitly referenced data items, use the XREF (SHORT) option.

Use both the XREF (either FULL or SHORT) and the SOURCE options to print a modified cross-reference to the right of the source listing. This embedded cross-reference shows the line number where the data-name or procedure-name was defined.

For further details, see the related reference about the XREF compiler option.

[“Example: XREF output: data-name cross-references” on page 473](#)

[“Example: XREF output: program-name cross-references” on page 474](#)

[“Example: XREF output: COPY/BASIS cross-references” on page 475](#)

[“Example: XREF output: embedded cross-reference” on page 476](#)

Related tasks

[“Getting listings” on page 443](#)

Related references

[“XREF” on page 416](#)

Listing data items

Use the MAP (HEX | DEC) compiler option to create a listing of the DATA DIVISION items and all implicitly declared items. Use the MAP output to locate the contents of a data item in a system dump.

When you specify the MAP (HEX | DEC) option, an embedded MAP summary that contains condensed MAP information is generated to the right of the COBOL source data definition.

- If you specify MAP (HEX) or MAP with no suboption, data item offsets within groups will be in hexadecimal notation.
- If you specify MAP (DEC), data item offsets within groups will be in decimal notation.

When both XREF data and an embedded MAP summary are on the same line, the embedded summary is printed first.

You can select or inhibit parts of the MAP listing and embedded MAP summary by using *CONTROL MAP | NOMAP (or *CBL MAP | NOMAP) statements throughout the source. For example:

```

*CONTROL NOMAP
  01 A
  02 B
*CONTROL MAP

```

[“Example: MAP output” on page 449](#)

Related tasks

[“Getting listings” on page 443](#)

Related references

[“MAP” on page 374](#)

Using the debugger

You can use z/OS Debugger to debug your Enterprise COBOL programs. Use the TEST compiler option to prepare your COBOL program so that you can step through the executable program with the debugger.

For remote debugging, you can use the Eclipse-based debug interface and IBM Z Open Debug in conjunction with the z/OS Debugger host running under z/OS or z/OS UNIX.

You can specify the TEST suboption NOSOURCE to have smaller object programs stored on disk. The loaded size does not change, the debug information is never loaded unless requested, for example, by a debugger such as z/OS Debugger or by LE (for CEEDUMP). With the NOSOURCE suboption, you will not be able to see the source in the z/OS Debugger source window.

Specify the OPTIMIZE(0), NOSTGOPT and TEST compiler options to get the most debugging function.

Specify a non-zero OPTIMIZE level, NOSTGOPT and TEST(EJPD) compiler options to get better performance with a few restrictions on debugging function.

Specify a non-zero OPTIMIZE level, STGOPT and TEST(NOEJPD) compiler options to get the best performance but still be able to use z/OS Debugger, with some restrictions on debugging function.

For details about which compiler options to use for maximum debugging capability versus best performance, see the related reference about the TEST compiler option.

Related tasks

IBM z/OS Debugger User's Guide (Preparing your program for debugging)

Related references

[“TEST” on page 404](#)

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 a list of the options in effect for the program, statistics about the content of the program, and diagnostic messages about the compilation	Short listing	<ul style="list-style-type: none">List of options in effect for the programStatistics about the content of the programDiagnostic messages about the compilation¹	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 397

Table 60. Using compiler options to get listings (continued)

Use	Listing	Contents	Compiler option
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 items Embedded map summary (in the right margin of the listing for lines in the DATA DIVISION that contain data declarations) Nested program map (if the program contains nested programs)	“MAP” on page 374 ²
To find where a name is defined, referenced, or modified; to determine the context (such as whether a statement was used in a PERFORM block) in which a procedure is referenced; to determine the data set or file from which a copybook was obtained	Sorted cross-reference listing of names; sorted cross-reference listing of COPY/BASIS statements and copybook data sets or files	Data-names, procedure-names, and program-names; references to these names COPY/BASIS text-names and library names, and the data sets or files from which associated copybooks were obtained Embedded modified cross-reference provides line numbers where data-names and procedure-names were defined	“XREF” on page 416 ^{2,3}
To find the failing statement in a program or the address in storage of a data item that is moved while the program is running	PROCEDURE DIVISION code and assembler code produced by the compiler ³	Generated code	“LIST” on page 372 ^{2,4}
To verify you still have a valid logic path after you move or add PROCEDURE DIVISION sections	Condensed PROCEDURE DIVISION listing	Condensed statement listing, global tables, WORKING-STORAGE information, and literals	“OFFSET” on page 383
To find an instance of a certain statement	Alphabetic listing of statements	Each statement used, number of times each statement was used, line numbers where each statement was used	“VBREF” on page 412

Table 60. *Using compiler options to get listings* (continued)

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. You can also selectively suppress messages by using the MSGEXIT suboption of the EXIT compiler option.</p> <p>2. To use your line numbers in the compiled program, use the NUMBER compiler option. The compiler checks the sequence of your source statement line numbers in columns 1 through 6 as the statements are read in. When it finds a line number out of sequence, the compiler assigns to it a number with a value one higher than the line number of the preceding statement. The new value is flagged with two asterisks. A diagnostic message indicating an out-of-sequence error is included in the compilation listing.</p> <p>3. The context of the procedure reference is indicated by the characters preceding the line number.</p> <p>4. You can control the listing of generated object code by selectively placing *CONTROL LIST and *CONTROL NOLIST (or equivalently, *CBL LIST and *CBL NOLIST) statements in your source. Note that the *CONTROL statement is different than the PROCESS (or CBL) statement.</p> <p>The output is generated if:</p> <ul style="list-style-type: none"> • 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. OFFSET and LIST are mutually exclusive options with OFFSET taking precedence. 			

[“Example: short listing” on page 445](#)

[“Example: SOURCE and NUMBER output” on page 448](#)

[“Example: MAP output” on page 449](#)

[“Example: embedded map summary” on page 451](#)

[“Example: nested program map” on page 454](#)

[“Example: XREF output: data-name cross-references” on page 473](#)

[“Example: XREF output: program-name cross-references” on page 474](#)

[“Example: XREF output: COPY/BASIS cross-references” on page 475](#)

[“Example: XREF output: embedded cross-reference” on page 476](#)

[“Example: OFFSET compiler output” on page 477](#)

[“Example: VBREF compiler output” on page 478](#)

[“Example: conditional compilation output” on page 478](#)

Related tasks

[“Generating a list of compiler messages” on page 281](#)

[“Reading LIST output” on page 454](#)

Language Environment Debugging Guide (Debugging COBOL programs)

Related references

[“Messages and listings for compiler-detected errors” on page 281](#)

Example: short listing

The parenthetical numbers shown in the listing below correspond to numbered explanations that follow the listing. For illustrative purposes, some errors that cause diagnostic messages were deliberately introduced.

```

Invocation parameters:      (1)
  OPTFILE
PROCESS(CBL) statements:  (2)
  CBL NODECK
  CBL NOADV, NODYN, NONAME, NONUMBER, QUOTE, SEQ, DUMP
  CBL NOSOURCE, NOXREF, NOVBREF, NOMAP, NOOFFSET, NOLIST
Options from SYSOPTF:      (3)

```

```

C,NODU,FLAG(I),X,MAP,NOLIST,RENT,OPT(1),SSR
TEST TRUNC(OPT)
Options in effect:          (4)
NOADATA
NOADV
  AFP(VOLATILE)
  QUOTE
  ARCH(8)
  ARITH(COMPAT)
NOAWO
NOBLOCK0
  BUFSIZE(4096)
NOCICS
  CODEPAGE(1140)
  COMPILE
NOCOPYRIGHT
NOCURRENCY
  DATA(31)
  DBCS
NODECK
NODIAGTRUNC
  DISPSIGN(COMPAT)
NODLL
  DUMP
NODYNAM
NOEXIT
NOEXPORTALL
NOFASTSRT
  FLAG(I)
NOFLAGSTD
  HGPR(PRESERVE)
  INTDATE(ANSI)
  LANGUAGE(EN)
  LINECOUNT(60)
NOLIST
NOMAP
  MAXPCF(100000)
NOMDECK
NONAME
  NSYMBOL(NATIONAL)
NONUMBER
  NUMPROC(NOPFD)
  OBJECT
NOOFFSET
  OPTIMIZE(1)
  OUTDD(SYSOUT)
  PGMNAME(COMPAT)
  QUALIFY(COMPAT)
  RENT
  RMODE(AUTO)
NORULES
NOSERVICE
  SEQUENCE
NOSOURCE
  SPACE(1)
NOSQL
  SQLCCSID
NOSQLIMS
  SSRANGE(NOZLEN)
NOSTGOPT
SUPPRESS
NOTERM
  TEST(NOEJPD,SOURCE)
NOTHREAD
  TRUNC(OPT)
NOVBREF
  VLR(COMPAT)
  VSAMOPENFS(COMPAT)
NOWORD
  XMLPARSE(XMLSS)
NOXREF
NOZONECHECK
  ZONEDATA(PFD)
ZWB

```

LineID	Message code	Message text	(5)
	IGYSC3002-I	A severe error was found in the program. The "OPTIMIZE" and the "STGOPT" compiler options were 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.

192 IGYDS1050-E File "LOCATION-FILE" contained no data record descriptions. The file definition was
discarded.

192 IGYGR1207-S The "ASSIGN" clause was missing or invalid in the "SELECT" entry for file "LOCATION-
FILE".
The file definition was discarded.

888 IGYPS2052-S An error was found in the definition of file "LOCATION-FILE". The reference to this
file
was discarded.
Same message on line: 979

1000 IGYPS2121-S "WS-DATE" was not defined as a data-name. The statement was discarded.
Same message on line: 1001

1004 IGYPS2053-S An error was found in the definition of file "LOCATION-FILE". This input/output
statement
was discarded.
Same message on line: 1016

1015 IGYPS2121-S "LOC-CODE" was not defined as a data-name. The statement was discarded.

1212 IGYPS2121-S "WS-NUMERIC-DATE" was not defined as a data-name. The statement was discarded.

1655 IGYPG3113-W Truncation of high-order digit positions may occur due to precision of intermediate
results
exceeding 30 digits.
Messages Total Informational Warning Error Severe Terminating (6)
Printed: 13 1 1 1 10
* Statistics for COBOL program IGYTCARA: (7)
* Source records = 1755
* Data Division statements = 295
* Procedure Division statements = 479
* Generated COBOL statements = 0
* Program complexity factor = 486
End of compilation 1, program IGYTCARA, highest severity 12. (8)
Return code 12

```

(1)

Message about options passed to the compiler at compiler invocation. This message does not appear if no options were passed.

OPTFILE

Requests options from a SYSOPTF data set.

(2)

Options coded in the PROCESS (or CBL) statement.

NOOFFSET

Suppresses a condensed listing of the PROCEDURE DIVISION.

NOMAP

Suppresses a map report of the items defined in the DATA DIVISION.

(3)

Options obtained from the SYSOPTF data set (because the OPTFILE compiler option was specified).

NOLIST

Suppresses an assembler-language expansion of the source code.

TEST

The program was compiled for use with debugging and problem determination tools (such as Debug Tool and Fault Analyzer) and to get local variables listed in CEEDUMP.

(4)

Status of options at the start of this compilation.

(5)

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.

(6)

Count of diagnostic messages in this program, grouped by severity level.

- (7) Program statistics for the program IGYTCARA.
- (8) 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. The note numbers in the listing correspond to numbered explanations that follow the listing.

```

(1)
LineID  PL SL  ----+*A-1-B-+----2-+----3-+----4-+----5-+----6-+----7-|-+----8 Map and Cross Reference
(2)    (3) (4)
000870  /*****
000871  ***          D O M A I N L O G I C          **
000872  ***
000873  *** Initialization. Read and process update transactions until **
000874  *** EOE. Close files and stop run.          **
000875  *****/
000876  procedure division.
000877  000-do-main-logic.
000878  display "PROGRAM IGYTCARA - Beginning".
000879  perform 050-create-vsam-main-file.          930
000880  perform 100-initialize-paragraph.          982
000881  read update-transaction-file into ws-transaction-record
000882  at end                                     203 338
000883  1 IA4390      set transaction-eof to true          253
000884  end-read.
000885  IA4410      perform until transaction-eof          253
000886  1          perform 200-edit-update-transaction  1050
000887  1 IA4430      if no-errors                       372
000888  2          perform 300-update-commuter-record  1159
000889  1          else
000890  2          perform 400-print-transaction-errors  1312
000891  1          end-if
000892  1          perform 410-re-initialize-fields    1373
000893  1 IA4480      read update-transaction-file into ws-transaction-record
000894  2          at end                                     203 338
000895  2 IA4500      set transaction-eof to true          253
000896  1 IA4510      end-read
000897  IA4520      end-perform.
000898  close commuter-file update-transaction-file location-file  180 203 192
000899  print-file.                                     216
000900
000901  *-----*
000902  *   File status checked after I/O operation.   *
000903  *-----*
000904
000905  IA4600      if not i-o-okay                       241
000906  1          display "000-close"
000907  1          move 0000 to comp-code                 230
000908  1 IA4620      perform 500-vsam-error             1386
000909  1          perform 900-abnormal-termination     1432
000910  IA4630      end-if.
000911  *****/
000912  * Paragraphs 1100 and 1200 illustrates the intrinsic *
000913  * function computations.                               *
000914  *****/
000915  perform 1100-print-i-f-headings.                 1441
000916  perform 1200-print-i-f-data.                     1481
000917  display " ".
000918  display " ".
000919  display "PROGRAM IGYTCARA - Normal end".
000920  stop run.

```

- (1) Scale line, which labels Area A, Area B, and source-code column numbers
- (2) Source-code line number assigned by the compiler
- (3) Program (PL) and statement (SL) nesting level
- (4) Columns 1 through 6 of program (the sequence number area)

Example: MAP output

The following example shows output from the MAP option. The numbers used in the explanation below correspond to the numbers that annotate the output.

Example 1

The following example shows the MAP output when SOURCE (DEC) is in effect.

```

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
X = Unallocated

(2) (3) (4) (5) (6) (7) (8) (9)
Source Hierarchy and Base Displacement Asmblr Data Data Type Data Def
LineID Data Name Locator Structure Definition Definition Attributes
-----
4 PROGRAM-ID IGYTCARA-----*
58 FD COMMUTER-FILE . . . . . BLF=00001 VSAM
60 1 COMMUTER-RECORD . . . . . BLF=00001 DS 0CL80 Group F
61 2 COMMUTER-KEY . . . . . BLF=00001 000000000 DS 16C Display
62 2 FILLER . . . . . BLF=00001 000000016 DS 64C Display
64 FD COMMUTER-FILE-MST . . . . . BLF=00002 VSAM
66 1 COMMUTER-RECORD-MST . . . . . BLF=00002 DS 0CL80 Group
67 2 COMMUTER-KEY-MST . . . . . BLF=00002 DS 16C Display
68 2 FILLER . . . . . BLF=00002 000000016 DS 64C Display
140 1 STATUS-AREA . . . . . DS 0CL8 Group
141 2 COMMUTER-FILE-STATUS . . . . . 000000000 DS 2C Display
142 88 I-O-OKAY . . . . .
143 2 COMMUTER-VSAM-STATUS . . . . . 000000002 DS 0CL6 Group
144 3 VSAM-R15-RETURN-CODE . . . . . 000000002 DS 2C Binary
145 3 VSAM-FUNCTION-CODE . . . . . 000000004 DS 2C Binary
146 3 VSAM-FEEDBACK-CODE . . . . . 000000006 DS 2C Binary
148 77 UPDATE-FILE-STATUS . . . . . DS 2C Display
149 77 LOCCODE-FILE-STATUS . . . . . DS 2C Display
150 77 UPDPRINT-FILE-STATUS . . . . . DS 2C Display
152 1 FLAGS . . . . . DS 0CL3 Group
153 2 TRANSACTION-EOF-FLAG . . . . . 000000000 DS 1C Display
154 88 TRANSACTION-EOF . . . . .
155 2 LOCATION-EOF-FLAG . . . . . 000000001 DS 1C Display
156 88 LOCATION-EOF . . . . .
157 2 TRANSACTION-MATCH-FLAG . . . . . 000000002 DS 1C Display
158 88 TRANSACTION-MATCH . . . . .
159 88 TRANSACTION-MATCH-OFF . . . . .
216 1 WS-COMMUTER-RECORD . . . . . BLX=00001 DS 0CL81 Group E
217 2 WS-COMMUTER-KEY . . . . . BLX=00001 DS 0CL16 Group E
218 3 WS-COMMUTER-GENERIC-KEY . . . . . BLX=00001 DS 0CL5 Group E
219 4 COMMUTER-SHIFT . . . . . BLX=00001 DS 1C Display E
220 4 COMMUTER-HOME-CODE . . . . . BLX=00001 000000001 DS 2C Display E
221 4 COMMUTER-WORK-CODE . . . . . BLX=00001 000000003 DS 2C Display E
222 3 COMMUTER-NAME . . . . . BLX=00001 000000005 DS 9C Display E
223 3 COMMUTER-INITIALS . . . . . BLX=00001 000000014 DS 2C Display E
224 2 COMMUTER-ADDRESS . . . . . BLX=00001 000000016 DS 18C Display E
225 2 COMMUTER-CITY . . . . . BLX=00001 000000034 DS 13C Display E
226 2 COMMUTER-STATE . . . . . BLX=00001 000000047 DS 2C Display E
227 2 COMMUTER-ZIPCODE . . . . . BLX=00001 000000049 DS 3P Packed-Dec E
396 1 DETAIL1-LINE . . . . . BLL=00001 DS 0CL121 Group
397 2 FILLER . . . . . BLL=00001 000000000 DS 2C Display
398 2 PRINT-TRANSACTION-CODE . . . . . BLL=00001 000000002 DS 1C Display
399 2 FILLER . . . . . BLL=00001 000000003 DS 4C Display
400 2 PRINT-RECORD-TYPE . . . . . BLL=00001 000000007 DS 3C Display
401 2 FILLER . . . . . BLL=00001 000000010 DS 3C Display
402 2 PRINT-SHIFT . . . . . BLL=00001 000000013 DS 1C Display
403 2 FILLER . . . . . BLL=00001 000000014 DS 1C Display
404 2 PRINT-HOME-CODE . . . . . BLL=00001 000000015 DS 2C Display
405 2 FILLER . . . . . BLL=00001 000000017 DS 1C Display
406 2 PRINT-WORK-CODE . . . . . BLL=00001 000000018 DS 2C Display
407 2 FILLER . . . . . BLL=00001 000000020 DS 2C Display
408 2 PRINT-NAME . . . . . BLL=00001 000000022 DS 9C Display
454 1 DETAILX-LINE . . . . . BLL=XXXXX DS 0CL121 Group X
455 2 FILLER . . . . . BLL=XXXXX DS 36C Display X
456 2 PRINT-CITY . . . . . BLL=XXXXX DS 13C Display X
457 2 FILLER . . . . . BLL=XXXXX DS 3C Display X
458 2 PRINT-STATE . . . . . BLL=XXXXX DS 2C Display X
459 2 FILLER . . . . . BLL=XXXXX DS 1C Display X
460 2 PRINT-ZIPCODE . . . . . BLL=XXXXX DS 5C Display X
461 2 FILLER . . . . . BLL=XXXXX DS 1C Display X
462 2 PRINT-WORK-PHONE . . . . . BLL=XXXXX DS 14C Display X
463 2 FILLER . . . . . BLL=XXXXX DS 1C Display X
464 2 PRINT-WORK-JUNCTION . . . . . BLL=XXXXX DS 25C Display X
465 2 FILLER . . . . . BLL=XXXXX DS 20C Display X (10)
467 1 DETAIL2-LINE . . . . . BLL=00002 DS 0CL121 Group
468 2 FILLER . . . . . BLL=00002 000000000 DS 36C Display
469 2 PRINT-CITY . . . . . BLL=00002 000000036 DS 13C Display
470 2 FILLER . . . . . BLL=00002 000000049 DS 3C Display
471 2 PRINT-STATE . . . . . BLL=00002 000000052 DS 2C Display
472 2 FILLER . . . . . BLL=00002 000000054 DS 1C Display
473 2 PRINT-ZIPCODE . . . . . BLL=00002 000000055 DS 5C Display
474 2 FILLER . . . . . BLL=00002 000000060 DS 1C Display
475 2 PRINT-WORK-PHONE . . . . . BLL=00002 000000061 DS 14C Display
476 2 FILLER . . . . . BLL=00002 000000075 DS 1C Display
477 2 PRINT-WORK-JUNCTION . . . . . BLL=00002 000000076 DS 25C Display
478 2 FILLER . . . . . BLL=00002 000000101 DS 20C Display

```


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.

.
.
000054      Data division.
000055      File section.
000056
000058      FD  COMMUTER-FILE
000059          record 80 characters.

.
.
000060          01 commuter-record.
000061              05 commuter-key          PIC x(16).
000062              05 filler                PIC x(64).

(1)      (2)      (3)
BLF=00001      0CL80
BLF=00001,000000000 16C
BLF=00001,000000016 64C

.
.
000105      Working-storage section.
000106          01 Working-storage-for-IGYCARA  pic x.
000107
000108          77 comp-code                  pic S9999 comp.
000109          77 ws-type                   pic x(3)  value spaces.
000135          01 i-f-status-area.
000136              05 i-f-file-status     pic x(2).
000137              88 i-o-successful      value zeroes.
000138
000139
000140          01 status-area.
000141              05 commuter-file-status  pic x(2).
000142              88 i-o-okay             value zeroes.
000143              05 commuter-vsam-status.
000144                  10 vsam-r15-return-code  pic 9(2) comp.
000145                  10 vsam-function-code    pic 9(1) comp.
000146                  10 vsam-feedback-code    pic 9(3) comp.
000147
000148          77 update-file-status         pic xx.
000149          77 loccode-file-status      pic xx.
000150          77 updprint-file-status     pic xx.
000151
000216          01 ws-commuter-record EXTERNAL.
000217              05 ws-commuter-key.
000218                  10 ws-commuter-generic-key.
000219                      15 commuter-shift    pic x.
000220                      15 commuter-home-code  pic xx.
000221                      15 commuter-work-code  pic xx.
000222                  10 commuter-name         pic x(9).
000223                  10 commuter-initials     pic xx.
000224              05 commuter-address       pic x(18).
000225              05 commuter-city          pic x(13).
000226              05 commuter-state        pic xx.
000227              05 commuter-zipcode       pic 9(5) comp-3.

BLX=00001      0CL81
BLX=00001,000000000 0CL16
BLX=00001,000000000 0CL5
BLX=00001,000000000 1C
BLX=00001,000000001 2C
BLX=00001,000000003 2C
BLX=00001,000000005 9C
BLX=00001,000000014 2C
BLX=00001,000000016 18C
BLX=00001,000000034 13C
BLX=00001,000000047 2C
BLX=00001,000000049 3P

.
.
000395      Linkage Section.
000396          01 detail1-line.
000397              05 filler                pic xx.
000398              05 print-transaction-code  pic x.
000399              05 filler                pic x(4).
000400              05 print-record-type     pic x(3).
000401              05 filler                pic xxx.
000402              05 print-shift          pic x.
000403              05 filler                pic x.
000404              05 print-home-code     pic xx.
000405              05 filler                pic x.
000406              05 print-work-code     pic xx.
000407              05 filler                pic xx.
000408              05 print-name          pic x(9).
000409              05 filler                pic xx.
000410              05 print-initials      pic xx.

BLL=00001      0CL121
BLL=00001,000000000 2C
BLL=00001,000000002 1C
BLL=00001,000000003 4C
BLL=00001,000000007 3C
BLL=00001,000000010 3C
BLL=00001,000000013 1C
BLL=00001,000000014 1C
BLL=00001,000000015 2C
BLL=00001,000000017 1C
BLL=00001,000000018 2C
BLL=00001,000000020 2C
BLL=00001,000000022 9C
BLL=00001,000000031 2C
BLL=00001,000000033 2C

.
.
000487      procedure division.
000488          000-do-main-logic.
000489              display "PROGRAM IGYTCARA - Beginning".

```

(1)

Base locator used for this data item

(2)

Decimal displacement from the beginning of the containing structure. It indicates that the MAP (DEC) option is in effect. If you specified the MAP (HEX) option or MAP with no suboption, hexadecimal displacement is shown.

(3)

Pseudoassembler code showing how the data is defined

Related references

[“Symbols used in LIST and MAP output” on page 453](#)

Terms used in MAP output

The following table describes the terms used in the listings produced by the MAP compiler option.

Term	Definition	Description
ALPHABETIC	DS nC	Alphabetic data item (PICTURE A)
ALPHA-EDIT	DS nC	Alphabetic-edited data item
AN-EDIT	DS nC	Alphanumeric-edited data item
BINARY	DS 1H ² , 1F ² , 2F ² , 2C, 4C, or 8C	Binary data item (USAGE BINARY, COMPUTATIONAL, or COMPUTATIONAL - 5)
COMP-1	DS 4C	Single-precision internal floating-point data item (USAGE COMPUTATIONAL - 1)
COMP-2	DS 8C	Double-precision internal floating-point data item (USAGE COMPUTATIONAL - 2)
DBCS	DS nC	DBCS data item (USAGE DISPLAY - 1)
DBCS-EDIT	DS nC	DBCS-edited data item (USAGE DISPLAY - 1)
DISP-FLOAT	DS nC	Display floating-point data item (USAGE DISPLAY)
DISPLAY	DS nC	Alphanumeric data item (PICTURE X)
DISP-NUM	DS nC	Zoned decimal data item (USAGE DISPLAY)
DISP-NUM-EDIT	DS nC	Numeric-edited data item (USAGE DISPLAY)
FD		File definition
FUNCTION-PTR	DS nC	Function pointer (USAGE FUNCTION-POINTER)
GROUP	DS 0CLn ¹	Fixed-length alphanumeric group data item
GRP-VARLEN	DS 0CLn ¹	Variable-length alphanumeric group data item
INDEX	DS nC	Index data item (USAGE INDEX)
INDEX-NAME	DS nC	Index name
NATIONAL	DS nC	Category national data item (USAGE NATIONAL)
NAT-EDIT	DS nC	National-edited data item (USAGE NATIONAL)
NAT-FLOAT	DS nC	National floating-point data item (USAGE NATIONAL)
NAT-GROUP	DS 0CLn ¹	National group (GROUP-USAGE NATIONAL)
NAT-GRP-VARLEN	DS 0CLn ¹	National variable-length group (GROUP-USAGE NATIONAL)
NAT-NUM	DS nC	National decimal data item (USAGE NATIONAL)
NAT-NUM-EDIT	DS nC	National numeric-edited data item (USAGE NATIONAL)
OBJECT-REF	DS nC	Object-reference data item (USAGE OBJECT REFERENCE)
PACKED-DEC	DS nP	Internal decimal data item (USAGE PACKED-DECIMAL or COMPUTATIONAL - 3)

<i>Table 61. Terms used in MAP output (continued)</i>		
Term	Definition	Description
POINTER	DS <i>n</i> C	Pointer data item (USAGE POINTER)
PROCEDURE-PTR	DS <i>n</i> C	Procedure pointer (USAGE PROCEDURE-POINTER)
SD		Sort 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
<p>1. <i>n</i> is the size in bytes for fixed-length groups and the maximum size in bytes for variable-length groups. 2. If the SYNCHRONIZED clause appears, these fields are used.</p>		

Symbols used in LIST and MAP output

The following table describes the symbols used in the listings produced by the LIST or MAP option.

<i>Table 62. Symbols used in LIST and MAP output</i>	
Symbol	Definition
BLF_ <i>n</i> ¹	Base locator for files
BLL_ <i>n</i> ¹	Base locator for LINKAGE SECTION
BLO_ <i>n</i> ¹	Base locator for object instance data
BLT_ <i>n</i> ¹	Base locator for XML-TEXT and XML-NTEXT
BLV_ <i>n</i> ¹	Base locator for variably located data
BLX_ <i>n</i> ¹	Base locator for external data
ODOsv_cell	ODO save cell number
Pfm_cell	PERFORM cell number
Pfmsv_cell	Perform save cell number
TSN=N	Temporary created by the compiler
VLC_cell	Variable-length cell (ODO)
VN_cell	Variable name cell for PERFORM statement
VNGO_cell	Variable name cell for ALTER statement
VNI_cell	Variable name initialization
#Calc00000000n	Code to compute addresses of data that is present after an OCCURS DEPENDING ON clause
#WSVal00000000n	Code to initialize the WORKING-STORAGE area for a procedure
_ArgumentList	Outgoing arguments to a procedure
_ACON	Address of a symbol
_BEtempNNN	Temporary created by the optimizer

Table 62. **Symbols used in LIST and MAP output** (continued)

Symbol	Definition
_CAA	Address of the start of the Language Environment Common Anchor Area
CACHED\$STATIC	Copy of the start address of the static area (for this procedure)
_CONSTANT_AREA+n	Offset in the Constant Area
_CRENT	Address of the storage (for this module), from the CAA
_incomingArgumentList	Incoming parameters to the procedure
_parentDSA	For a nested procedure, it is the address of its parent's stack
_QCON	Offset to a symbol
_returnValue	Return value of the procedure
_VTS_n	Temporary created by the optimizer
1. n is the number of the entry. For base locators, it can also be XXXXX, indicating a data item that was deleted by STGOPT processing.	

Example: nested program map

This example shows a map of nested procedures produced by specifying the MAP compiler option. Numbers in parentheses refer to notes that follow the example.

```

Nested Program Map
Program Attribute codes (rightmost column) have the following meanings:
  C = COMMON
  I = INITIAL (1)
  U = PROCEDURE DIVISION USING... (5)
Source Nesting
LineID Level Program Name from PROGRAM-ID paragraph Program Attributes
   2      0 NESTMAIN. . . . . U
 120     1 (4) SUBPR01 . . . . . I,C,U
(2)199    2 NESTED1 . . . . . I,C,U
 253     1 SUBPR02 . . . . . 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 a program.

The LIST compiler option produces several 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:
 - Compiler options in effect
 - Types of data items present
 - Statements 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 statement that corresponds to that instruction. After you find the address of the failing instruction, go to the assembler listing and find the statement for which that instruction was generated. The line number is in the 3rd column of the assembler listing for your program. Using the line number, you can locate the STATEMENT by looking at the corresponding line in the Source Output section of the listing.

- Information about WORKING-STORAGE. This information is contained in the Data Division Map and in the Initial Heap Storage Map.
- A description of the storage is found in the Initial Heap Storage Map. The symbols in WORKING-STORAGE area of the source are mapped into the storage that is shown in the Initial Heap Storage Map.

You can use the Data Division Map along with the Initial Heap Storage Map section to find the location of data items defined in WORKING-STORAGE. These data items reside in the storage. The Initial Heap Storage Map gives the offset of each level-1 data item relative to the beginning of the storage. The Data Division Map section gives the offset of the level-*n* data items relative to their respective level-1 member. By using both pieces of information, you can determine the offset of any data member within the storage.

If you compile with the DATA24 option, data items mapped below the line will appear in the Below the Line Storage. You can follow the same process to determine their locations.

- Information about the constants and the literals used in the program. The Constant Area contains information about the constants and literals in the program, as well as those created by the compiler. This section contains the offset of each constant or literal within the Constant Area.
- Program prolog areas (PPA1, PPA2, PPA3, PPA4) contain information about the characteristics of the compiled program.
- Externals symbols dictionary contains the list of external symbols defined by or referred to, in your program.
- 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 each time a separately compiled procedure is entered.

You do not need to be able to program in assembler language to understand the LIST output. The comments that accompany most of the assembler code provide you with a conceptual understanding of the functions performed by the code.

[“Example: Program initialization code” on page 465](#)

[“Example: MD5 signature” on page 466](#)

[“Example: Timestamp and version information” on page 466](#)

[“Example: Compiler options and program information” on page 467](#)

[“Example: Assembler code generated from source code” on page 467](#)

[“Example: Program prolog areas” on page 468](#)

[“Example: initial heap storage map” on page 469](#)

[“Example: Constant area” on page 470](#)

[“Example: Base locator table” on page 471](#)

[“Example: External symbols” on page 472](#)

[“Example: DSA memory map \(stack storage map\)” on page 472](#)

Related references

[“Signature information bytes” on page 456](#)

[“Example: MAP output” on page 449](#)

Language Environment Programming Guide (Stack storage overview)

Signature information bytes

The tables in this topic show program signature information that is part of the listing of program initialization code provided when you use the LIST compiler option.

Offset in decimal	Option	Value
00	CODEPAGE	CCSID value specified for EBCDIC code page
02	ARCH	10
		11
		12
		13
		14
03	OPTIMIZE	0
		1
		2

The INFO BYTE section of the listing also provides the following values:

- The number of DATA DIVISION statements
- The number of PROCEDURE DIVISION statements

In the following table, different signature bytes represent different information:

- Signature bytes 1-5, and 26-35 refer to compiler options
- Signature bytes 6-7 refer to DATA DIVISION items
- Signature byte 8 refers to ENVIRONMENT DIVISION items
- Signature bytes 9-25 refer to PROCEDURE DIVISION statements and items

Table 64. Signature information bytes

Offset in decimal	Signature byte	Bit	Item	
			On	Off
04	28	0	SQL	NOSQL
		1	CICS	NOCICS
		2	MDECK	NOMDECK
		3	SQLCCSID	NOSQLCCSID
		4	OPTFILE	NOOPTFILE
		5	XMLPARSE (XMLSS)	XMLPARSE (COMPAT)
		6	BLOCK0	NOBLOCK0
		7	DISPSIGN (SEP)	DISPSIGN (COMPAT)
05	29	0	Program uses Java-based OO syntax	
		1	Program uses RANDOM function	
		2	Program uses NATIONAL data (Unicode)	
		3	XML PARSE with schema validation	
		4	STGOPT	NOSTGOPT
		5	AFP (VOLATILE)	AFP (NOVOLATILE)
		6	HGPR (PRESERVE)	HGPR (NOPRESERVE)
		7	NOTEST (DWARF)	Not NOTEST (DWARF)
06	30	0	QUALIFY (EXTEND)	QUALIFY (COMPAT)
		1	VLR (COMPAT)	VLR (STANDARD)
		2	COPYRIGHT string specified	COPYRIGHT string not specified
		3	SERVICE string specified	SERVICE string not specified
		4	INVDATA (FNC, x)	Not INVDATA (FNC, x) Note: When bits 4 and 5 are both off, NOINVDATA is in effect.
		5	INVDATA (NOFNC, x)	Not INVDATA (NOFNC, x) Note: When bits 4 and 5 are both off, NOINVDATA is in effect
		6	INVDATA (x, NOCS)	INVDATA(x,CS), should be zero when bits 4 and 5 are zero.
		7	NUMCHECK (ZON (LAX))	NUMCHECK (ZON (STRICT))

Table 64. *Signature information bytes (continued)*

Offset in decimal	Signature byte	Bit	Item	
			On	Off
07	31	0	NUMCHECK (ZON)	Not NUMCHECK (ZON)
		1	NUMCHECK (PAC)	Not NUMCHECK (PAC)
		2	NUMCHECK (BIN)	Not NUMCHECK (BIN)
				NONUMCHECK is in effect if bits 0, 1, and 2 are off
		3	NUMCHECK (ABD)	NUMCHECK (MSG) (if any bit of 0, 1, or 2 is on)
		4	PARMCHECK	NOPARMCHECK
		5	PARMCHECK (ABD) (if bit 4 is on)	PARMCHECK (MSG) (if bit 4 is on)
		6	NUMCHECK (ZON (NOALPHNUM))	NUMCHECK (ZON (ALPHNUM))
		7	TEST NOTEST (SEPARATE (DSNAME))	TEST NOTEST (SEPARATE (NODSNAME))
08	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	SQLIMS	NOSQLIMS
09	2	0	LIB (always on)	
		1	LIST	NOLIST
		2	MAP (HEX), MAP (DEC)	NOMAP
		3	NUM	NONUM
		4	OBJECT	NOOBJECT
		5	OFFSET	NOOFFSET
		6	OPT (1), OPT (2)	NOOPT, OPT (0)
		7	OUTDD	NOOUTDD

Table 64. *Signature information bytes (continued)*

Offset in decimal	Signature byte	Bit	Item	
			On	Off
10	3	0	NUMPROC (PFD)	NUMPROC (NOPFD)
		1	RENT	NORENT
		2	RESIDENT (always on)	
		3	SEQUENCE	NOSEQUENCE
		4	Reserved	
		5	SOURCE	NOSOURCE
		6	Not NOSSRANGE	NOSSRANGE
		7	TERM	NOTERM
11	4	0	TEST	NOTEST
		1	TRUNC (STD)	Not TRUNC (STD)
		2	WORD	NOWORD
		3	VBREF	NOVBREF
		4	XREF	NOXREF
		5	ZWB	NOZWB
		6	NAME	NONAME
		7		NOCMPR2 (always off)
12	5	0	Reserved	
		1	NUMCLS=ALT	NUMCLS=PRIM
		2	DBCS	NODBCS
		3	AWO	NOAWO
		4	TRUNC (BIN)	Not TRUNC (BIN)
		5	ADATA	NOADATA
		6	CURRENCY	NOCURRENCY
		7	Compilation unit is a class	Compilation unit is a program
13	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	

Table 64. *Signature information bytes (continued)*

Offset in decimal	Signature byte	Bit	Item	
			On	Off
14	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 or FUNCTION-POINTER item	
15	8	0	FILE STATUS clause in FILE-CONTROL paragraph	
		1	RERUN clause in I-O-CONTROL paragraph of INPUT-OUTPUT SECTION	
		2	UPSI switch defined in SPECIAL-NAMES paragraph	
		3	WSOPT: Bit to indicate the method used by the compiler to manage WORKING-STORAGE SECTION. For more information, see <i>COBOL-specific vendor interfaces</i> in the <i>z/OS Language Environment Vendor Interfaces</i> .	
		4	VSAMOPENFS	
		5	1=NUMCHECK(BIN(NOTRUNCBIN)) 0=NUMCHECK(BIN(TRUNCBIN))	
16	9	0	ACCEPT	
		1	ADD	
		2	ALTER	
		3	CALL	
		4	CANCEL	
		6	CLOSE	
17	10	0	COMPUTE	
		2	DELETE	
		4	DISPLAY	
		5	DIVIDE	

Table 64. *Signature information bytes (continued)*

Offset in decimal	Signature byte	Bit	Item	
			On	Off
18	11	1	END-PERFORM	
		2	ENTER	
		3	ENTRY	
		4	EXIT	
		5	EXEC	
		6	GO TO	
		7	IF	
19	12	0	INITIALIZE	
		1	INVOKE	
		2	INSPECT	
		3	MERGE	
		4	MOVE	
		5	MULTIPLY	
		6	OPEN	
		7	PERFORM	
20	13	0	READ	
		2	RELEASE	
		3	RETURN	
		4	REWRITE	
		5	SEARCH	
		7	SET	
21	14	0	SORT	
		1	START	
		2	STOP	
		3	STRING	
		4	SUBTRACT	
		7	UNSTRING	

Table 64. *Signature information bytes (continued)*

Offset in decimal	Signature byte	Bit	Item	
			On	Off
22	15	0	USE	
		1	WRITE	
		2	CONTINUE	
		3	END-ADD	
		4	END-CALL	
		5	END-COMPUTE	
		6	END-DELETE	
		7	END-DIVIDE	
23	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	
24	17	0	END-STRING	
		1	END-SUBTRACT	
		2	END-UNSTRING	
		3	END-WRITE	
		4	GOBACK	
		5	EVALUATE	
		7	SERVICE	
		25	18	0
1	END-EXEC			
2	XML			
3	END-XML			
4	ALLOCATE			
5	FREE			
6	JSON			
7	END-JSON			
26	19	0-7	Reserved	

Table 64. *Signature information bytes (continued)*

Offset in decimal	Signature byte	Bit	Item	
			On	Off
27	20	0-7	Reserved	
28	21	0	Hexadecimal literal	
		1	Altered GO TO	
		2	I-O ERROR declarative	
		3	DECIMAL-POINT IS COMMA clause	
		4	DEBUGGING declarative	
		5	Program segmentation	
		6	OPEN . . . EXTEND	
		7	EXIT PROGRAM	
29	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	
30	23	0	COPY	
		1	BASIS	
		2	DBCS name in program	
		3	Shift-out and Shift-in in program	
		4	SUPPRESS NOSUPPRESS	
		5	SSRANGE (ZLEN) (if bit 6 in byte 3 is on)	SSRANGE (NOZLEN) (if bit 6 in byte 3 is on)
		6	SSRANGE (ABD) (if bit 6 in byte 3 is on)	SSRANGE (MSG) (if bit 6 in byte 3 is on)
		7	INLINE NOINLINE	

Table 64. *Signature information bytes (continued)*

Offset in decimal	Signature byte	Bit	Item	
			On	Off
40	24	0	DBCS literal	
		1	REPLACE	
		2	Reference modification was used.	
		3	Nested program	
		4	INITIAL (either IS INITIAL or compiler option INITIAL)	
		5	COMMON	
		6	SELECT . . . OPTIONAL	
		7	EXTERNAL	
41	25	0	GLOBAL	
		1	RECORD IS VARYING	
		2	VOLATILE	
		3	Program uses UTF-8 data	
		5	Intrinsic function was used	
		6	Z-literal found	
		7	RECURSIVE	
42	26	0	RMODE (ANY)	Not RMODE (ANY)
		1-3	Reserved	
		4	Reserved	
		5	INTDATE (LILIAN)	INTDATE (ANSI)
		6	TEST NOTEST (SEPARATE)	TEST NOTEST (NOSEPARATE)
		7	Reserved	
		43	27	0
1	PGMNAME (LONGMIXED)			Not PGMNAME (LONGMIXED)
2	DLL			NODLL
3	EXPORTALL			NOEXPORTALL
4	TEST NOTEST (SOURCE)			TEST NOTEST (NOSOURCE)
5	ARITH (EXTEND)			ARITH (COMPAT)
6	THREAD			NOTHREAD
7	TEST (EJPD)			TEST (NOEJPD)
44	28	0-7	Build level info	

Table 64. Signature information bytes (continued)

Offset in decimal	Signature byte	Bit	Item	
			On	Off
52	32	0	LP (64)	LP (32)
		1	Reserved	
		2	SOURCE (HEX)	SOURCE (DEC)
		3	SMARTBIN	NOSMARTBIN
		4	Compilation unit is a function.	Compilation unit is not a function.
		5	Reserved	
		6	Program contains at least one JAVA-SHAREABLE directive.	Program does not contain a JAVA-SHAREABLE directive.
		7	Program contains one or more calls to Java using a CALL statement.	Program does not contain a call to Java using a CALL statement.
53	33	0	JAVAIOP (JAVA64)	JAVAIOP (NOJAVA64)
		1-7	Reserved	
54	34	0-7	Reserved	
55	35	0-7	TUNE architecture level	

Check return code: A return code greater than 4 from the compiler could mean that some of the statements shown in the information bytes might have been discarded from the program.

Related references

“LIST” on page 372

z/OS Language Environment Vendor Interfaces (COBOL-specific vendor interfaces)

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.

The information such as PROGRAM-ID, COMPILED TIME, and COMPILED DATE included in the initialization code of Enterprise COBOL 4 or earlier is not included in the initialization code of Enterprise COBOL 5 or later, so the program it depends on might behave differently with Enterprise COBOL 5 and 6. For more details, see Error behavior changes for incorrect programs (*Enterprise COBOL Migration Guide*)

```

(1)      (2)      (3)      (4)      (5)
000000          000003      PROC      IGYTCARA
000000      47F0 F014      000003      BC      R15,20(,R15)      # Skip over constant area
000004      01C3 C5C5      000003      DC      X'01C3C5C5'      # Eyecatcher: CEE
000008      0000 0978      000003      DC      X'00000978'      # Stack Size
00000C      0000 8910      000003      DC      X'00008910'      # Offset to PPA1
000010      47F0 F001      000003      BC      R15,1(,R15)      # Wrong Entry Point: cause exception
000014          000003      EQU
000014      90EC D00C      000003      L3282:  STM      R14,R12,12(,R13)      # Save GPRs Used
000018      4110 F024      000003      LA      R1,36(,R15)      # Args for boot strap routine
00001C      98EF F034      000003      LM      R14,R15,52(,R15)      #
000020      07FF          000003      BR      R15      # Branch to boot strap routine
000022      0000          000003      DC      X'0000'      # Available half-word
000024          000003      L3284:  EQU      *      # Boot Strap Info Block
000024      0000 0000      000003      DC      X'00000000'      # address of entry label
000028      0000 0000      000003      DC      X'00000000'      # WSA24 allocation size
00002C      0000 8A0C      000003      DC      X'00008A0C'      # address of Saved Option String
000030      0000 8948      000003      DC      X'00008948'      # address of entry point name
000034      0000 0054      000003      DC      X'00000054'      # A(Label L3283)
000038      0000 0000      000003      DC      X'00000000'      # address of boot strap routine(IGZXBST)
00003C          000003      L3285:  EQU      *      # CEE Parameter Block
00003C      0000 0024      000003      DC      X'00000024'      # address of infoBlockLabel

```

```

000040 0000 8A1C          000003          DC      X'00008A1C'      # A(PARMCEE-CEEPPARMBlock)
000044          000003 L3280:    EQU      *            # Handle growing stack
000044 58F0 C31C          000003          L       R15,796(,R12) # Load CEECAAOGETS
000048 184E          000003          LR      R4,R14        # Required NAB
00004A 05EF          000003          BALR   R14,R15       # Extend Stack
00004C 0000 0000 0000    000003          DC      X'00000000'   # Argument list size = 0
000050 A7F4 0009          000003          J       L3281        # Branch back
000054          000003 @MAINENT DS        OH # PRIMARY ENTRY POINT ADDRESS
000054          000003 L3283:    EQU      *            # User Code Entry Point
000054 18EF          000003          LR      R14,R15     # Load NAB
000056 4100 E978          000003          LA      R0,2424(,R14) # New NAB Address
00005A 5500 C314          000003          CL      R0,788(,R12) # Exceed current storage segment?
00005E A724 FFF3          000003          JH      L3280        # Yes: branch to recovery code
000062          000003 L3281:    EQU      *            # Stack now has sufficient room
000062 5000 E04C          000003          ST      R0,76(,R14)  # Update NAB
000066 A708 0010          000003          LHI    R0,16        # COBOL Language Word upper half
00006A 8900 0010          000003          SLL    R0,16        # shift to upper half of register
00006E A70A 0301          000003          AHI    R0,769       # add COBOL Language Word lower half
000072 5000 E000          000003          ST      R0,0(,R14)  # Save Language Word
000076 50D0 E004          000003          ST      R13,4(,R14) # Save Back Chain
00007A 18DE          000003          LR      R13,R14     # Set new DSA
00007C 4100 D6D0          000003          LA      R0,1744(,R13) # Address of COBDSACB
000080 5000 D074          000003          ST      R0,116(,R13) # Saved in member slot1
000084 4100 0000          000003          LA      R0,0(,R0)   #
000088 5000 D070          000003          ST      R0,112(,R13) # zero member slot0

```

- (1) Offset from the start of the COBOL program
- (2) Hexadecimal representation of assembler instructions
- (3) Source line number
- (4) Pseudo-assembler representation of the code generated for the COBOL program
- (5) Comments that explain the pseudo-assembler code

Related references

[“Signature information bytes” on page 456](#)

Example: MD5 signature

The following example shows LIST output about the MD5 signature. This information is also included in the DWARF debugging data in the application module. MD5 signature is located at 16 bytes before the Timestamp and Version Information section.

```

000608 AAEE 60C2 DAA3          =X'AAEE60C2DAA3'    md5 signature
00060E 776D AEB5 E753          =X'776DAEB5E753'    md5 signature
000614 E767 C4E1          =X'E767C4E1'        md5 signature

```

Note: The MD5 signature is shown only if the TEST option is specified or if -g is specified with **cob2** under z/OS UNIX.

The presence or absence of the MD5 signature is indicated by a compilation flag bit of the PPA2. If the bit is set to 1, the MD5 signature is present; if the bit is set to 0, the MD5 signature is absent. For details about PPA2, see *z/OS Language Environment Vendor Interfaces*.

Related references

[“Example: Program prolog areas” on page 468](#)

Example: Timestamp and version information

The following example shows LIST output about the version of the compiler and the date and time of compilation.

```

                                Timestamp and Version Information
0029C8 F2F0 F2F0          =C'2022'            Compiled Year
0029CC F0F7 F1F7          =C'0531'            Compiled Date MMDD
0029D0 F1F2 F3F1 F2F2    =C'123122'          Compiled Time HHMMSS
0029D6 F0F6 F0F3 F0F0    =C'060400'          VERSION

```

Example: Compiler options and program information

The following example shows LIST output for the compiler options and program information.

```

DATA VALIDATION AND UPDATE PROGRAM      IGYTCARA Date 05/31/2022 Time 10:48:16

                                Compiler Options and Program Information Section

(1)  (2)  (3)                                (4)                                (5)
0029DC      0030                                =X'0030'                                Size of Compiler Options and Prog Info Section
0029DE      (+00) 0474                            =X'0474'                                UNSIGNED BINARY CODE PAGE CCSID VALUE
0029E0      (+02) 06                              =X'06'                                  ARCHITECTURE LEVEL
0029E1      (+03) 00                              =X'00'                                  OPTIMIZATION LEVEL
0029E2      (+04) 1406                            =X'1406'                                INFO. BYTES 28-29
0029E4      (+06) 0000                            =X'0000'                                RESERVED
0029E6      (+08) A04875CC2001                    =X'A04875CC2001'                       INFO. BYTES 1-6
0029EC      (+14) 100010884909                    =X'100010884909'                       INFO. BYTES 7-12
0029F2      (+20) 002008800C00                    =X'002008800C00'                       INFO. BYTES 13-18
0029F8      (+26) 000001A000                      =X'000001A000'                         INFO. BYTES 19-23
0029FD      (+31) 00                              =X'00'                                  COBOL SIGNATURE LEVEL
0029FE      (+32) 0000002F                        =X'0000002F'                            # DATA DIVISION STATEMENTS
002A02      (+36) 0000005B                        =X'0000005B'                            # PROCEDURE DIVISION STATEMENTS
002A06      (+40) 18808008                        =X'18808008'                            INFO. BYTES 24-27
002A0A      (+44) E2F1F6F0F1F1F540                =C'PXXXXXX'                            BUILD LEVEL INFO

                                Compiler Options and Program Information Section End

```

- (1) Offset in the program object
- (2) Offset in decimal
- (3) Contents of the bytes in hexadecimal format
- (4) Assembler representation of the bytes
- (5) Explanation of the bytes in the section

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 statement that corresponds to the instruction that failed.

```

000964:          display "PROGRAM IGYTCARA - Beginning".                                (1)
(2)  (3)  (4)                                (5)                                (6)
0001EA      E320 3394 0171      000964      LAY      R2,5012(,R3)      #
0001F0      D203 05E8 2000      000964      MVC      1512(4,R13),0(R2)      #
0001F6      E320 3398 0171      000964      LAY      R2,5016(,R3)      #
0001FC      D203 05EC 2000      000964      MVC      1516(4,R13),0(R2)      #
000202      4120 39C8      000964      LA      R2,2504(,R3)      #
000206      5020 05F0      000964      ST      R2,1520(,R13)      #
00020A      E320 338C 0171      000964      LAY      R2,5004(,R3)      #
000210      D203 05F4 2000      000964      MVC      1524(4,R13),0(R2)      #
000216      E320 339C 0171      000964      LAY      R2,5020(,R3)      #
00021C      D203 05F8 2000      000964      MVC      1528(4,R13),0(R2)      #
000222      D703 05FC 05FC      000964      XC      1532(4,R13),1532(R13)      #
000228      4110 05E8      000964      LA      R1,1512(,R13)      # _ArgumentList
00022C      E3F0 31D4 0158      000964      LY      R15,4564(,R3)      # _ACON
000232      58C0 0080      000964      L      R12,128(,R13)      # _CAA
000236      0DEF      000964      BASR   R14,R15      # Call "IGZXDSP"
000965:          perform 050-create-vsam-main-file.
000238      5820 0670      000965      L      R2,1648(,R13)      # VN_cell
00023C      5020 0544      000965      ST      R2,1348(,R13)      # PfmSv_Cell
000240      C020 0000 0007      000965      LARL   R2      #
000246      5020 0670      000965      ST      R2,1648(,R13)      # VN_cell
00024A      A7F4 02F4      000965      J      050-CREATE-VSAM-MAIN-FILE      #
00024E      5820 0544      000965      L      R2,1348(,R13)      # PfmSv_Cell
000252      5020 0670      000965      ST      R2,1648(,R13)      # VN_cell

```

- (1) Source code interspersed with the pseudo-assembler instructions
- (2) Relative location of the object code instruction in the module, in hexadecimal notation

(3)

Object code instructions, 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)

Source line number associated with this assembler code

(5)

Object code instructions, in compiler-generated pseudo assembler

(6)

Explanation of the instruction and the operands used by the instructions

Related references

[“Symbols used in LIST and MAP output” on page 453](#)

Example: Program prolog areas

The following example shows LIST output for the program prolog area. The Program Prologue Area (PPA) is comprised of several sections that contain information about the compiled program.

There is a PPA1 for every procedure in your program, including procedures generated by the compiler. The offset to its corresponding PPA1 is recorded at offset 12 (X'C') from the start of each procedure. The PPA1 contains information about the procedure as well as offsets to the PPA2 and PPA3 sections.

For details on how to use the program prolog areas to locate information in the listing file, see *z/OS Language Environment Vendor Interfaces*.

```

DATA VALIDATION AND UPDATE PROGRAM      IGYTCARA Date 05/31/2022 Time 10:48:16

      1      2      PPA1:  Entry Point Constants      3      4
0081E0  1CCEA506      =F'483304710'      Flags
0081E4  00008310      =A(PPA2-IGYTCARA)
0081E8  00008378      =A(PPA3-IGYTCARA)
0081EC  00000000      =F'0'              No EPD
0081F0  FFFE0000      =F'-131072'        Register Save Mask
0081F4  40000000      =F'1073741824'     Member Flags
0081F8  90              =AL1(144)           Flags
0081F9  0000978       =AL3(2424)          Callee's DSA use/8
0081FC  0000           =AL1(0)             Flags
0081FE  0012           =H'18'              Offset/2 to CDL
008200  D00006D0       =F'-805304624'     State variable location
008204  00000000       =F'0'               CDL function length/2
008208  00000000       =F'0'               CDL function EP offset
00820C  00000000       =F'0'               CDL prolog
008210  00000000       =F'0'               CDL epilog
008214  00000000       =F'0'               CDL end
008218  0008 ****     AL2(8),C'IGYTCARA'
      PPA1  End

```

There is one PPA2 for each program. The offset to the PPA2 is recorded in each PPA1. The PPA2 contains offsets to the Timestamp and Version Information section of the listing as well as to the PPA4 section.

If the TEST option is not in effect, the PPA2 section looks like this:

```

      PPA2:  Entry Point Constants
000800  04002203      =F'67117571'      Flags
000804  FFFFF800      =A(CEESTART-PPA2)
000808  00000058      =F'88'             A(PPA4-PPA2)
00080C  FFFFFF80      =A(TIMESTAMP-PPA2)
000810  FFFFF800      =A(PrimaryEntryPoint-PPA2)
000814  02200000      =F'35651584'      Flags
      PPA2  End

```

If the TEST option is in effect, the PPA2 section looks like this:

```

      PPA2:  Entry Point Constants
000830  04002203      =F'67117571'      Flags
000834  FFFFF7D0      =A(CEESTART-PPA2)
000838  00000058      =F'88'             A(PPA4-PPA2)
00083C  FFFFFF80      =A(TIMESTAMP-PPA2)
000840  FFFFF7D0      =A(PrimaryEntryPoint-PPA2)
000844  02600000      =F'39845888'      Flags
      PPA2  End

```


There is one PPA3 for each program (including each nested program) in a COBOL source file. Each entry contains offsets, relative to the PPA3 itself, to the base locator table and to the special register table. The PPA3 also contains an offset from the start of the program to the first COBOL statement.

```

PPA3:  Entry Point Constants
0014D8  00000000      =F'0'           Flags
0014DC  000000C0      =F'192'        A(Base_Locator_Table-PPA3)
0014E0  000000D8      =F'216'        A(Special_Register_Table-PPA3)
0014E4  00000184      =X'184'        A(User_Entry-CUEntry)
PPA3  End

```

There is one PPA4 for each program. It has offsets to various compiler generated tables, such as the storage (the Initial Heap Storage Map and Below The Line Storage Map sections). The offset to the PPA4 is recorded in a field of the PPA2.

```

PPA4:  Entry Point Constants
000710  22000000      =F'570425344'  Flags 1
000714  00020100      =F'131328'     Flags 2
000718  00000000      =F'0'          A(NORENTstatic)
00071C  00000000      =F'0'          Q(RENTstatic)
000720  0000006C      =F'108'        A(DATA31_address_cell-RENTstatic)
000724  FFFFF8F0      =F'-1808'     A(Code-PPA4)
000728  00000760      =F'1888'     Code Length
00072C  00000000      =F'0'         Length NORENTstatic
000730  00000070      =F'112'     Length RENTstatic
000734  00000094      =F'148'     Length DATA31
000738  003F         =X'3F'       A(CUName-PPA4)
00073A  0000         =X'0'       PPA4 Minor Ver
000744  7FFFFFFF     =X'7FFFFFFF' Offset UsrWrkStig
000748  00000000     =X'0'       Length UsrWrkStig
00074C  00          =X'0'       Has Externals
00074D  0000         =X'0'       A(SYSDEBUGName-PPA4)
PPA4  End

```

- 1 Relative location, in hexadecimal format, of the PPA field in the object module
- 2 The contents of the field, in hexadecimal
- 3 An assembler-like syntax defining the field
- 4 A description of the contents of the field.

Related references

WORKING-STORAGE SECTION changes
(Enterprise COBOL for z/OS Migration Guide)
z/OS Language Environment Vendor Interfaces

Example: initial heap storage map

The three map sections in the listing, INITIAL HEAP STORAGE MAP, WORKING-STORAGE MAP, and BELOW THE LINE STORAGE MAP are collectively called the initial heap storage maps of a program. The storage for these areas is allocated at the start of the program and persists until the end of the run unit or until the program is canceled.

The layouts of these three map sections are similar:

- The first column shows the offset of the item from a block of storage allocated by the compiler.
- The second column is the size of the symbol, including all of its sublevel members.
- The third column is the name of the area being described.

See the following INITIAL HEAP STORAGE MAP as an example of the layout. If the NORENT compiler option is in effect, the WORKING-STORAGE data items are mapped in the INITIAL HEAP STORAGE MAP section. For COBOL data items, the offset is the offset to the start of the level-01 data item from the block of storage allocated by the compiler. The starting address of this block resides in the Constant Area.

```

* * * * * I N I T I A L   H E A P   S T O R A G E   M A P   * * * * *

```

OFFSET (HEX)	LENGTH (HEX)	NAME
0	4	BLL_Ptrs
4	C	BLT_Ptrs
10	60	GPCB
70	4	WS-BASE-ADDRESS
74	8	TS2=6

If the RENT and DATA (31) compiler options are in effect, the WORKING-STORAGE data items are shown under the WORKING-STORAGE MAP. If the RENT and DATA (24) options are in effect, the WORKING-STORAGE data items are shown under the BELOW THE LINE STORAGE MAP. An INITIAL HEAP STORAGE MAP section is also shown where compiler generated internal data items and locators are mapped.

```

* * * * * B E L O W   T H E   L I N E   S T O R A G E   M A P   * * * * *

```

OFFSET (HEX)	LENGTH (HEX)	NAME
0	4	JNIENVPTR
8	2	RETURN-CODE
10	2	SORT-RETURN
18	8	SORT-CONTROL
20	4	SORT-CORE-SIZE
28	4	SORT-FILE-SIZE
30	4	SORT-MODE-SIZE
38	8	SORT-MESSAGE
40	4	TALLY
48	1	SHIFT-OUT
50	1	SHIFT-IN
58	4	XML-CODE
60	1E	XML-EVENT
80	4	XML-INFORMATION
88	50	COMMUTER-FILE
D8	50	COMMUTER-FILE-MST
128	7A	PRINT-FILE
1A8	1	WORKING-STORAGE-FOR-IGYCARA
1B0	2	COMP-CODE
1B8	3	WS-TYPE
1C0	2	I-F-STATUS-AREA
1C8	8	STATUS-AREA
1D0	2	UPDATE-FILE-STATUS

Example: Constant area

The following example shows LIST output about strings and other literals from the COBOL source as well as those generated by the compiler.

The compiler generates loads from (and stores to) the Constant Area by loading the starting address of Constant Area and adding the fixed offsets to the respective constants or literals.

(1)	(2)	CONSTANT AREA:				(3)	(4)
006A98 (+0)	00CCDDFF	00000000	C9C7E8E3	C3C1D9C1	00000000	00000000	C9C7E9E2 D9E3C3C4
.....IGYTCARAIGZSR	TCDD					
006AB8 (+32)	40000A00	40000000	00000008	00000000	E2E8E2D6	E4E34040	00100000 00000000
.....SYSOUT							
006AD8 (+64)	0E000000	00000001	0F000000	0000001E	00000000	40000000	00000003 0064003C
.....							
006AF8 (+96)	000FE800	9F0F0000	00000011	00000000	E3D9C1D5	E2C1C3E3	4B40C3D6 C4C50000
CODE..							..Y.....TRANSACTION
006B18 (+128)	0000000E	00000000	E2C8C9C6	E340C3D6	C4C54040	40400000	C8D6D4C5 40D3D6C3
LOC						SHIFT CODE ..HOME
006B38 (+160)	4B40C3D6	C4C50000	E6D6D9D2	40D3D6C3	4B40C3D6	C4C50000	D3C1E2E3 40D5C1D4
NAM							. CODE..WORK LOC. CODE..LAST
006B58 (+192)	C5404040	40400000	C9D5C9E3	C9C1D3E2	40404040	40400000	C4E4D7D3 C9C3C1E3
E ..INITIALS	..DUPLICAT						
006B78 (+224)	C540D9C5	C34B0000	D9C5C34B	40D5D6E3	40C6D6E4	D5C40000	C1C4C4D9 C5E2E240
FOUND..ADDRESS							E REC...REC. NOT
006B98 (+256)	40404040	40400000	C3C9E3E8	40404040	40404040	40400000	E2E3C1E3 C540C3D6
CO							..CITY ..STATE
006BB8 (+288)	C4C54040	40400000	E9C9D7C3	D6C4C540	40404040	40400000	C8D6D4C5 40D7C8D6
PHO							DE ..ZIPCODE ..HOME
006BD8 (+320)	D5C54040	40400000	E6D6D9D2	40D7C8D6	D5C54040	40400000	C8D6D4C5 40D1E4D5
JUN							NE ..WORK PHONE ..HOME
006BF8 (+352)	C3E3C9D6	D5400000	E6D6D9D2	40D1E4D5	C3E3C9D6	D5400000	C4D9C9E5 C9D5C740
JUNCTION ..DRIVING							CTION ..WORK
006C18 (+384)	E2E3C1E3	E4E20000	40D9C5D7	D6D9E340	407B7A40	C9C7E8E3	C3C1D9C1 40404040
IGYTCARA							STATUS.. REPORT #:
006C38 (+416)	40404040	40404040	40404040	40404040	40404040	40404040	40404000 00000033

```

| 006C58 (+448) C3D6D4D4 E4E3C5D9 40C6C9D3 C540E4D7 C4C1E3C5 40D3C9E2 E3404040 40404040 | COMMUTER FILE UPDATE
LIST |
006C78 (+480) 40404040 40404040 40404040 40404040 40400000 00000032 40404040 40404040
| 006C98 (+512) D7C1C7C5 407B7A40 00000000 00000010 40D7D9D6 C7D9C1D4 407B7A40 C9C7E8E3 | PAGE #: ..... PROGRAM #:
IGYT|
006CB8 (+544) C3C1D9C1 40404040 404040D9 E4D540E3 C9D4C57A 40000000 00000025 7A000000 | CARA RUN
TIME: .....|
006CD8 (+576) 00000030 00000000 D9E4D540 C4C1E3C5 7A400000 0000000A 61000000 0000000B | .....RUN
DATE: .....|

```

- (1) Offset in csect.
- (2) Offset in base 10.
- (3) 8 columns containing the bytes in the Constant Area
- (4) Character representation. A dot (.) is used for non-printable characters.

Example: Base locator table

The following example shows LIST output for the base locator table.

```

                                Base Locator Table
008AB0 01                      =X'1'          Table Version
008AB1 00                      =X'0'          Reserved
008AB2 0008                    =H'8'          Header length
008AB4 00000010                =F'16'         Array byte length
008AB8 2A00                    =X'2A00'       Flags & info (element 1)
008ABA 00000014                =X'14'         Offset to cells
008ABE 03                      =X'3'          Cell count
008ABF 0A00                    =X'A00'        Flags & info (element 2)
008AC1 00000000                =X'0'          Offset to cells
008AC5 05                      =X'5'          Cell count
008AC6 0000                    =X'0'          Flags & info (end of
array)
                                Base Locator Table End

```

For more information about the base locator table, see *z/OS Language Environment Vendor Interfaces*.

Related references

z/OS Language Environment Vendor Interfaces (Base locator table)

Example: Special register table

The following example shows LIST output for the special register table. The special register table has a similar format to the base locator table.

```

                                Special Register Table
0015B0 01                      =X'1'          Table Version
0015B1 00                      =X'0'          Reserved
0015B2 0008                    =H'8'          Header length
0015B4 00000006                =F'6'          Array byte length
0015B8 12                      =X'12'         Flags & info (element 1)
0015B9 00000018                =X'18'         Offset to cells
0015BD 00                      =X'0'          Flags & info (end of
array)
                                Special Register Table End

```

Each entry in the special register table consists of the following items:

- A byte which represents the following information:
 - Special register ID number (bits 0 - 4). ID = 1 represents the RETURN - CODE register
 - Access mode (bits 5 - 8)
 - MODE = 0; Base Address = Top of Stack
 - MODE = 1; Base Addr = NORENT Static

- MODE = 2; Base Addr = 32-bit RENT static
- MODE = 3; 24-bit NORENT static
- An offset to the special register

The end of the special register table is indicated by a null byte.

Example: External symbols

The following example shows LIST output for external symbols defined by, or referred to in your program. The external symbol dictionary contains one entry per external symbol defined by or referred to in the program.

Each entry contains the address, length and symbol type. Symbol types are:

- ED**
External Definition
- SD**
Section Definition
- LD**
Label Definition
- ER**
External Reference
- PR**
Pseudo Register

EXTERNAL SYMBOL DICTIONARY					
	TYPE	ID	ADDR	LENGTH	NAME
	SD	1	000000	000000	IGYTCARA
	ED	2	000000	000000	C_CEESG003
	ED	3	000000	008AC8	C_CODE
	LD	4	000000	000000	IGYTCARA#C
	ER	5	000000	000000	CEESTART
	ER	6	000000	000000	CEEBETBL
	ED	7	000000	000000	C_WSA
	PR	8	000000	002204	IGYTCARA#S
	ED	9	000000	000022	B_IDRL
	ER	10	000000	000000	IGZXBST
	ER	11	000000	000000	IGYTCARA
	ER	12	000000	000000	IGZXPRS
	ER	13	000000	000000	IGZXCMSG
	ER	14	000000	000000	IGZXDSP
	ER	15	000000	000000	IGZXVCLS

Example: DSA memory map (stack storage map)

The following example shows LIST output for the dynamic save area (DSA). The DSA contains information about the contents of the storage acquired when a separately compiled procedure is entered.

***** STACK STORAGE MAP *****			
	1	2	3
	OFFSET (HEX)	LENGTH (HEX)	NAME
Block name: IGYTCARA			
	80	4	_@CAA
	C8	3	_BEtemp200
	CC	3	_BEtemp204
	D0	3	_BEtemp208
	D4	3	_BEtemp212

D8	3	_BEtemp216
DC	3	_BEtemp220
E0	3	_BEtemp224
E4	3	_BEtemp228
E8	10	_BEtemp232
F8	20	_BEtemp248
118	20	_BEtemp280
138	4	_BEtemp312
13C	4	_BEtemp316
140	4	_BEtemp320
144	4	_BEtemp324
148	4	_BEtemp328
14C	4	_BEtemp332
150	4	_BEtemp336
154	4	_BEtemp340
158	4	_BEtemp344
15C	4	_BEtemp348
160	4	_BEtemp352
164	4	_BEtemp356
168	4	_BEtemp360
16C	4	_BEtemp364
170	4	_BEtemp368
174	4	_BEtemp372
178	4	_BEtemp376

- (1) Hexadecimal offset of the DSA field from the start of the DSA
- (2) Length (in hexadecimal) of the DSA field
- (3) Symbol name

Example: XREF output: data-name cross-references

The following example shows a sorted cross-reference of data-names that is produced by the XREF compiler option. Numbers in parentheses refer to notes after the example.

Example 1

The following example shows the XREF output when SOURCE(DEC) is in effect.

An "M" preceding a data-name reference indicates that the data-name is modified by this reference.

(1A) Defined	(2) Cross-reference of data-names	(3) References
265	ABEND-ITEM1	
266	ABEND-ITEM2	
347	ADD-CODE	1102 1162
381	ADDRESS-ERROR.	M1126
280	AREA-CODE.	1236 1261 1324 1345
382	CITY-ERROR	M1129
1C	A.	1D
39	A.	3A
38	AG	

(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) Defined	(6) Cross-reference of procedures	(7) References
877	000-DO-MAIN-LOGIC	
930	050-CREATE-STL-MAIN-FILE . .	P879
982	100-INITIALIZE-PARAGRAPH . . .	P880

```

1441 1100-PRINT-I-F-HEADINGS. . . . P915
1481 1200-PRINT-I-F-DATA. . . . . P916
1543 1210-GET-MILES-TIME. . . . . P1510
1636 1220-STORE-MILES-TIME. . . . . P1511
1652 1230-PRINT-SUB-I-F-DATA. . . . P1532
1676 1240-COMPUTE-SUMMARY . . . . . P1533
1050 200-EDIT-UPDATE-TRANSACTION. . P886
1124 210-EDIT-THE-REST. . . . . P1116
1159 300-UPDATE-COMMUTER-RECORD . . P888
1207 310-FORMAT-COMMUTER-RECORD . . P1164 P1179
1258 320-PRINT-COMMUTER-RECORD. . . P1165 P1176 P1182 P1192
1288 330-PRINT-REPORT . . . . . P1178 P1202 P1256 P1280 P1340 P1365 P1369
1312 400-PRINT-TRANSACTION-ERRORS . P890

```

Example 2

The following example shows the XREF output when SOURCE (HEX) is in effect.

An "M" preceding a data-name reference indicates that the data-name is modified by this reference.

(1B) Defined	(2) Cross-reference of data-names	(3) References
1C	A.	1D
39	A.	3A
38	AG	

Cross-reference of data-names:

(1)

Line number where the name was defined.

(2)

Data-name.

(3)

Line numbers where the name was used. If M precedes the line number, the data item was explicitly modified at the location.

Cross-reference of procedure references:

(4)

Explanations of the context usage codes for procedure references.

(5)

Line number where the procedure-name is defined.

(6)

Procedure-name.

(7)

Line numbers where the procedure is referenced, and the context usage code for the procedure.

[“Example: XREF output: program-name cross-references” on page 474](#)

[“Example: XREF output: COPY/BASIS cross-references” on page 475](#)

[“Example: XREF output: embedded cross-reference” on page 476](#)

Example: XREF output: program-name cross-references

The following example shows a sorted cross-reference of program-names produced by the XREF compiler option. Numbers in parentheses refer to notes that follow the example.

(1)	(2)	(3)
-----	-----	-----

Defined	Cross-reference of programs	References
EXTERNAL	EXTERNAL1.	25
2	X.	41
12	X1.	33 7
20	X11.	25 16
27	X12.	32 17
35	X2.	40 8

- (1) Line number where the program-name was defined. If the program is external, the word EXTERNAL is displayed instead of a definition line number.
- (2) Program-name.
- (3) Line numbers where the program is referenced.

Example: XREF output: COPY/BASIS cross-references

The following example shows a sorted cross-reference of copybooks to the library-names and data-set names of the associated copybooks, produced by the XREF compiler option under z/OS. Numbers in parentheses refer to notes after the example.

COPY/BASIS cross-reference of text-names, library names

(1) Text-name (Member)	(1) Library (DDNAME)	(2) File name (Data set name)	(3) Concat Level	(4) ISPF Created
ACTIONS	OTHERLIB	USERID.COBOL.COPY	0	1992/07/11
ACTIONS	SYSLIB	USERID.COBOL.COPY	0	1992/07/11
CUSTOMER	ALTDDXXY	USERID.COBOL.LIB3	0	2007/06/01
CUSTOMER	SYSLIB	USERID.COBOL.LIB2PDSE	1	2007/06/07
HOUSE	ALTDDXXY	USERID.COBOL.LIB2	1	2007/06/07
HOUSE	SYSLIB	USERID.COBOL.LIB2PDSE	1	
IMOTOR	SYSLIB	USERID.COBOL.LIB4X	3	2007/06/07
ISOVERFY	SYSLIB	USERID.COBOL.COPY	0	
NSMAP	SYSLIB	USERID.COBOL.LIB3	2	

- (1) Text-name and library (an abbreviation for library-name) are from the statement COPY *text-name* OF *library-name* in the source, for example, Copy ACTIONS OF OTHERLIB.
- (2) The name of the data set from which the COPY member was copied.
- (3) Abbreviation for concatenation level. Indicates how many levels deep a given data set is from the first data set in the concatenation for a given ddname.

For example, four data sets in the example above are concatenated to ddname SYSLIB:

DDNAME	DSNAME	(concatenation level)
SYSLIB DD	DSN=USERID.COBOL.COPY,	0
DD	DSN=USERID.COBOL.LIB2PDSE,	1
DD	DSN=USERID.COBOL.LIB3,	2
DD	DSN=USERID.COBOL.LIB4X	3

Thus for example member NSMAP shown in the listing above was found in data set USERID.COBOL.LIB3, which is two levels down from the first data set in the SYSLIB concatenation.

- (4) Creation date is shown if the PDSE was edited with STATS ON in ISPF.

Tip: Under z/OS, if there is more than one data set in your SYSLIB concatenation, the COPY/BASIS cross-reference might in some cases be incomplete or missing. For details, see the related reference about the XREF compiler option.

If you compile in the z/OS UNIX shell, the cross-reference looks like the excerpt shown below.

```

COPY/BASIS cross-reference of text-names, library names, and file names

(5)          (5)          (6)
Text-name    Library-name  File name

'/copydir/copyM.cbl'  SYSLIB          /u/JSMITH/cobol//copydir/copyM.cbl
'/copyA.cpy'         SYSLIB          /u/JSMITH/cobol//copyA.cpy
'cobol/copyA.cpy'    ALTDD2         /u/JSMITH/cobol/copyA.cpy
'copy/stuff.cpy'     ALTDD2         /u/JSMITH/copy/stuff.cpy
'copydir/copyM.cbl'  SYSLIB          /u/JSMITH/cobol/copydir/copyM.cbl
'copydir/copyM.cbl'  SYSLIB (default) /u/JSMITH/cobol/copydir/copyM.cbl
'stuff.cpy'         ALTDD          /u/JSMITH/copy/stuff.cpy
'copyA.cpy'         (7) SYSLIB (default) /u/JSMITH/cobol/copyA.cpy
"reallyXXVeryLongLon> SYSLIB (default) (8) <JSMITH/cobol/
reallyXXVeryLongLongName.cpy
OTHERDD          ALTDD2         /u/JSMITH//copy/other.cob
. . .

Note: Some names were truncated.  > = truncated on right  < = truncated on left

```

(5) From the COPY statement in the source; for example the COPY statement corresponding to the third item in the cross-reference above would be:

```
COPY 'cobol/copyA.cpy' OF ALTDD2
```

(6) The fully qualified path of the file from which the COPY member was copied

(7) Truncation of a long text-name or library-name on the right is marked by a greater-than sign (>).

(8) Truncation of a long file name on the left is marked by a less-than sign (<).

Related references
[“XREF” on page 416](#)

Example: XREF output: embedded cross-reference

The following example shows a modified cross-reference that is embedded in the source listing. The cross-reference is produced by the XREF compiler option.

```

LineID  PL  SL  -----*A-1-B-----2-----3-----4-----5-----6-----7-|-----8  Map and Cross Reference
. . .
000878          procedure division.
000879          000-do-main-logic.
000880          display "PROGRAM IGYTCARA - Beginning".
000881          perform 050-create-vsam-main-file.                                932 (1)
000882          perform 100-initialize-paragraph.                                984
000883          read update-transaction-file into ws-transaction-record          204 340
000884          at end
000885          1          set transaction-eof to true                            254
000886          end-read.

000984          100-initialize-paragraph.
000985          move spaces to ws-transaction-record                            IMP 340 (2)
000986          move spaces to ws-commuter-record                              IMP 316
000987          move zeroes to commuter-zipcode                               IMP 327
000988          move zeroes to commuter-home-phone                            IMP 328
000989          move zeroes to commuter-work-phone                            IMP 329
000990          move zeroes to commuter-update-date                          IMP 333
000991          open input update-transaction-file                            204
000992          location-file                                                193

```



```

000993          i-o commuter-file                181
000994          output print-file                217
.
.
.
001442          1100-print-i-f-headings.
001443
001444          open output print-file.          217
001445
001446          move function when-compiled to when-comp.      IFN 698 (2)
001447          move when-comp (5:2) to compile-month.          698 640
001448          move when-comp (7:2) to compile-day.            698 642
001449          move when-comp (3:2) to compile-year.            698 644
001450
001451          move function current-date (5:2) to current-month.  IFN 649
001452          move function current-date (7:2) to current-day.  IFN 651
001453          move function current-date (3:2) to current-year.  IFN 653
001454
001455          write print-record from i-f-header-line-1          222 635
001456          after new-page.                                  138
.
.
.

```

(1) Line number of the definition of the data-name or procedure-name in the program

(2) Special definition symbols:

UND

The user name is undefined.

DUP

The user name is defined more than once.

IMP

Implicitly defined name, such as special registers and figurative constants.

IFN

Intrinsic function reference.

EXT

External reference.

The program-name is unresolved because the NOCOMPILE option is in effect.

Example: OFFSET compiler output

The following example shows a compiler listing that has a condensed statement listing, global tables, WORKING-STORAGE information, and literals. The listing is output from the OFFSET compiler option.

```

DATA VALIDATION AND UPDATE PROGRAM  IGYTCARA Date 05/31/2022  Time 10:48:16
.
.
.
(1)  (2)  (3)
LINEID  HEXLOC  VERBCODE      LINEID  HEXLOC  VERBCODE      LINEID  HEXLOC  VERBCODE
000011  0001B8  DISPLAY      000012  0001CC  PERFORM      000021  0001D0  IF
000022  0001E2  MOVE         000024  0001EA  MOVE         000012  000204  PERFORM
000013  000214  DISPLAY      000014  000228  PERFORM      000021  00022C  IF
000022  00023E  MOVE         000024  000246  MOVE         000014  00024A  PERFORM
000015  00025A  DISPLAY      000021  000272  IF           000022  000284  MOVE
000024  00028C  MOVE         000016  000290  PERFORM      000017  0002A0  DISPLAY
000018  0002B8  STOP

```

(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 are listed every time an instruction is generated for a different line/statement pair than the previous instruction.

(3) Statement used.

Note: Due to the out of line code used for error message formatting, Language Environment generated offsets, as indicated in "From compile unit {name} at entry point {name} at compile unit offset {offset}...", might be outside the offset range of the program. In these cases, refer to the statement number in the COBOL message (IGZnnnns) to locate the problem.

Related references
 "OFFSET" on page 383

Example: VBREF compiler output

The following example shows an alphabetic listing of all the statements in a program, and shows where each is referenced. The listing is produced by the VBREF compiler option.

(1)	(2)	(3)
2	ACCEPT	1010 1012
2	ADD	1290 1306
1	CALL	1406
5	CLOSE	898 945 970 1526 1535
20	COMPUTE	1506 1640 1644 1657 1660 1663 1664 1665 1678 1682 1686 1691 1696 1701 1709 1713
		1718 1723 1728 1733
2	CONTINUE	1062 1069
2	DELETE	964 1193
48	DISPLAY	878 906 917 918 919 933 940 942 947 953 960 966 972 996 997 998 999 1003 1006 1037
		1090 1168 1171 1185 1195 1387 1388 1389 1390 1391 1392 1393 1401 1402 1403 1404
		1405 1433 1485 1486 1492 1497 1498 1520 1521 1528 1529 1624
2	EVALUATE	1161 1557
47	IF	887 905 932 939 946 952 959 965 971 993 1002 1036 1051 1054 1071 1074 1077 1089
		1102 1111 1115 1125 1128 1131 1134 1137 1141 1145 1148 1151 1167 1184 1194 1240
		1247 1265 1272 1289 1321 1330 1339 1351 1361 1484 1496 1519 1527
183	MOVE	907 937 957 983 984 985 986 987 988 1004 1011 1013 1025 1038 1052 1055 1060 1067
		1072 1075 1078 1079 1080 1081 1082 1083 1091 1103 1112 1126 1129 1132 1135 1139
		1143 1146 1149 1152 1160 1163 1169 1175 1177 1180 1181 1186 1191 1196 1201 1208
		1209 1210 1211 1212 1213 1214 1215 1216 1217 1218 1219 1220 1221 1222 1229 1230
		1231 1232 1233 1234 1235 1239 1241 1244 1248 1250 1251 1253 1254 1255 1257 1258
		1259 1260 1264 1266 1269 1273 1275 1276 1278 1279 1291 1294 1299 1301 1303 1307
		1313 1314 1315 1316 1317 1318 1319 1320 1322 1323 1327 1328 1331 1333 1334 1336
		1338 1341 1342 1343 1344 1348 1349 1352 1354 1355 1357 1362 1364 1368 1374 1375
		1376 1377 1378 1379 1380 1381 1414 1417 1422 1425 1445 1446 1447 1448 1450 1451
		1452 1457 1464 1489 1502 1507 1508 1509 1517 1551 1561 1566 1571 1576 1581 1586
		1591 1596 1601 1606 1611 1616 1621 1626 1627 1679 1683 1688 1693 1698 1703 1710
		1715 1720 1725 1730 1735
5	OPEN	931 951 989 1443 1483
62	PERFORM	879 880 885 886 888 890 892 908 909 915 916 934 935 941 943 948 949 954 955 961
		962 967 968 973 974 1000 1005 1008 1023 1039 1092 1093 1116 1164 1165 1170 1172
		1176 1178 1179 1182 1187 1188 1192 1197 1198 1202 1246 1256 1271 1280 1329 1340
		1350 1359 1365 1369 1504 1510 1511 1532 1533
8	READ	881 893 958 1014 1026 1085 1490 1514
1	REWRITE	1183
4	SEARCH	1058 1065 1413 1421
46	SET	883 895 1016 1028 1041 1057 1064 1084 1087 1363 1412 1420 1493 1499 1516 1522 1548
		1550 1559 1560 1564 1565 1569 1570 1574 1575 1579 1580 1584 1585 1589 1590 1594
		1595 1599 1600 1604 1605 1609 1610 1614 1615 1619 1620 1639 1643
2	STOP	920 1434
4	STRING	1236 1261 1324 1345
33	WRITE	938 1166 1292 1293 1295 1296 1297 1298 1300 1302 1305 1454 1459 1462 1465 1467 1469
		1471 1512 1654 1655 1667 1668 1669 1740 1742 1744 1745 1746 1747 1748 1749 1750

- (1) Number of times the statement is used in the program
- (2) statement
- (3) Line numbers where the statement is used

Example: conditional compilation output

The following examples show the listings of a program that contains conditional compilation statements.

Example 1

This example shows the default conditional compilation output. The note numbers in the listing correspond to numbered explanations that follow the listing.

LineID	PL	SL	-----*A-1-B--*-----2-----3-----4-----5-----6-----7- -----8 Map and Cross Reference
000001			identification division.
000002			program-id. prog.
000003			data division.
000004			working-storage section.
000005			01 x pic 9(9) binary.
000006			procedure division.

```

000007      MainProgram.
000008          >>define var as 12
000009          >>evaluate var
000010          >>when 10
000011      *      display 'var is 10'                (1)
000012          >>when 11 thru 13
000013          display 'var is 11, 12 or 13'      (2)
000014          >>when other
000015      *      display 'invalid value'          (1)
000016          >>end-evaluate
000017          goback.
000018      end program prog.

```

(1)

Those branches of the EVALUATE directive were false at compile time, so the code in those branches was omitted from the resultant program.

(2)

That branch of the EVALUATE directive evaluated to true at compile time, so the code in that branch was included in the resultant program.

Example 2

This example shows the conditional compilation output when CONDCOMP (SKIPSRC) is in effect. Notice that the false branches labeled with (1) in the previous example are omitted.

```

LineID PL SL -----A-1-B-+-----2-----3-----4-----5-----6-----7-|-----8
Map and Cross Reference

000001      identification division.
000002      program-id. prog.
000003      data division.
000004      working-storage section.
000005      01 x pic 9(9) binary.
000006      procedure division.
000007      MainProgram.
000008          >>define var as 12
000009          >>evaluate var
000010          >>when 10
000011      *      display 'var is 10'                (1)
000012          >>when 11 thru 13
000013          display 'var is 11, 12 or 13'      (2)
000014          >>when other
000015      *      display 'invalid value'          (1)
000016          >>end-evaluate
000017          goback.
000018      end program prog.

```

Related references

EVALUATE directive (*Enterprise COBOL for z/OS Language Reference*)

Conditional compilation (*Enterprise COBOL for z/OS Language Reference*)

Suppressing information in CEEDUMP processing (IGZ1OPT)

If TEST (DWARF) is in effect, CEEDUMP processing might include a large amount of information in the dump, depending on the size of the WORKING-STORAGE SECTION. You can suppress this information at execution time by setting up the JCL of a JOB.

1. In the Language Environment sample data set, .SCEESAMP, use the sample JCL IGZ1OPT to create a load module called IGZUOPT. Change the JOB card and load library name, and run this JCL to generate IGZUOPT.
2. Put this module in a data set in the STEPLIB concatenation, DWARF information will be suppressed during the CEEDUMP processing. This can reduce the volume of output for a COBOL program in the CEEDUMP.

JOB STEP1 of the JCL assembles an assembler program that invokes a MACRO called IGZXOPT. This macro is used to specify special COBOL runtime options. Currently, only the SKIPDWARF option is supported with the following syntax:

```
IGZXOPT SKIPDWARF=ON | OFF
```

The setting of SKIPDWARF can be ON or OFF, and the default value is OFF:

- If ON is specified, DWARF processing in CEEDUMP is suppressed.
- If OFF is specified, DWARF processing proceeds normally. Specifying OFF is equivalent to omitting IGZUOPT from the STEPLIB.

Related references

[“TEST” on page 404](#)

Generating a report of dynamically called COBOL programs

Use ddname IGZPROUT at the run step of your JCL to generate a report of all dynamically called programs that are compiled with Enterprise COBOL 5 or later.

The report details include the Program Name, Compiler Version, and Time and Date of compilation of the programs. Only programs that are compiled with COBOL 5 or later will be listed. Details of cancelled programs are printed at the end.

Note: If a program is cancelled multiple times, then the report prints it only once.

Recommended attributes for the IGZPROUT ddname are as follows:

```
DSORG = PS   RECFM = FB   78 ≤ LRECL ≤ 80
```

Here are the examples of setting the IGZPROUT ddname:

```
//IGZPROUT DD SYSOUT=*
```

or

```
//IGZPROUT DD DSN=dynamic.call.report.dsname,DISP=SHR
```

or

```
//IGZPROUT DD DSN=dynamic.call.report.dsname,  
// UNIT=SYSDA,SPACE=(TRK,(5,10),RLSE),  
// DCB=(DSORG=PS,LRECL=80,RECFM=FB,BLKSIZE=0),  
// DISP=(NEW,CATLG,DELETE)
```

You can get an output report as follows:

```
***** DYNAMICALLY CALLED PROGRAM REPORT *****  
***** FOR COBOL 5 AND LATER RELEASES *****  
  
NAME          COMPILER VER.      DATE/TIME COMPILED  
-----  
PROG1         V06 R02 M00        2021/01/01 15:30:02  
PROG2         V06 R02 M00        2021/01/01 15:30:12  
  
The following programs are marked as cancelled at termination:  
  
PROG3         V06 R03 M00        2022/02/13 17:36:29  
  
***** END OF PROGRAM REPORT *****
```

Note: For dynamic calls, the program-name in the PROGRAM- ID paragraph or ENTRY statement must be identical to the corresponding program object name or program object alias of the program object that contains the program. If the PROGRAM- ID paragraph or the ENTRY statement is not identical to the corresponding program object (PDSE member) name or program object alias of the program object that contains the program, then the report will list the program-name in the PROGRAM- ID paragraph or the ENTRY statement, as opposed to listing the program object (PDSE member) name or the program object alias name.

Related references

[“Making dynamic calls” on page 536](#)

Part 3. Targeting COBOL programs for certain environments

Chapter 22. Developing COBOL programs for CICS

COBOL programs that are written for CICS can run under CICS Transaction Server. CICS COBOL application programs that use CICS services must use the CICS command-level interface.

When you use the CICS compiler option, the Enterprise COBOL compiler handles both native COBOL statements and embedded CICS statements in the source program. You can still use the separate CICS translator to translate CICS statements to COBOL code, but use of the integrated CICS translator is recommended instead.

After you compile and bind 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 scope of COBOL information. For further information, see the Related tasks.

You can determine how runtime errors are handled by setting the CBLPSHPOP runtime option. For information about CICS HANDLE and CBLPSHPOP, see the Related tasks.

Related concepts

[“Integrated CICS translator” on page 488](#)

Related tasks

[“Coding COBOL programs to run under CICS” on page 483](#)

[“Compiling with the CICS option” on page 487](#)

[“Using the separate CICS translator” on page 489](#)

[“Handling errors by using CICS HANDLE” on page 491](#)

Language Environment Programming Guide (Condition handling under CICS:
using the CBLPSHPOP runtime option)

[Developing CICS Applications](#)

Related references

[“CICS” on page 342](#)

Coding COBOL programs to run under CICS

To code a program to run under CICS, code CICS commands in the PROCEDURE DIVISION by using the EXEC CICS command format.

```
EXEC CICS command-name command-options  
END-EXEC
```

CICS commands have the basic format shown above. Within EXEC commands, use the space as a word separator; do not use a comma or a semicolon. Do not code COBOL statements within EXEC CICS commands.

Restriction: You cannot run COBOL programs that have object-oriented syntax for Java interoperability in CICS. In addition, if you write programs to run under CICS, do not use the following code:

- FILE-CONTROL entry in the ENVIRONMENT DIVISION, unless the FILE-CONTROL entry is used for a SORT statement
- FILE SECTION of the DATA DIVISION, unless the FILE SECTION is used for a SORT statement
- User-specified parameters to the main program
- USE declaratives (except USE FOR DEBUGGING)
- 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 CONSOLE
- DISPLAY UPON SYSPUNCH
- MERGE
- OPEN
- READ
- RERUN
- REWRITE
- START
- STOP *literal*
- WRITE

If you plan to use the separate CICS translator, you must put any REPLACE statements that contain EXEC commands after the PROCEDURE DIVISION header for the program, otherwise the commands will not be translated.

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.

Coding a COBOL program to run above the 16 MB line: Under Enterprise COBOL, the following restrictions apply when you code a COBOL program to run above the 16 MB line:

- If you use IMS/ESA® without DBCTL, DL/I CALL statements are supported only if all the data passed in the call resides below the 16 MB line. Therefore, you must specify the DATA(24) compiler option. However, if you use IMS/ESA 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 level of the IMS product.

- If the receiving program is link-edited with AMODE 31, addresses that are 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 that are passed must be 24 bits long.

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 not allowed. You can specify the DISPSIGN option to control output formatting for DISPLAY of signed numeric items.

Related concepts

[“Integrated CICS translator” on page 488](#)

Related tasks

[“Sorting under CICS” on page 238](#)

[“Getting the system date under CICS” on page 485](#)

[“Calling to or from COBOL programs” on page 485](#)

[“Determining the success of ECI calls” on page 486](#)

[“Using the separate CICS translator” on page 489](#)

Related references

[“CICS SORT application restrictions” on page 238](#)

[“DISPSIGN” on page 352](#)

Getting the system date under CICS

To retrieve the system date in a CICS program, use a format-2 ACCEPT statement or the CURRENT-DATE intrinsic function.

You can use any of these format-2 ACCEPT statements in CICS to get the system date:

- ACCEPT *identifier-2* FROM DATE (two-digit year)
- ACCEPT *identifier-2* FROM DATE YYYYMMDD
- ACCEPT *identifier-2* FROM DAY (two-digit year)
- ACCEPT *identifier-2* FROM DAY YYYYDDD
- ACCEPT *identifier-2* FROM DAY-OF-WEEK (one-digit integer, where 1 represents Monday)

You can use this format-2 ACCEPT statement in CICS to get the system time:

- ACCEPT *identifier-2* FROM TIME

Alternatively, you can use the CURRENT-DATE intrinsic function, which can also provide the time.

These methods work in both CICS and non-CICS environments.

Do not use a format-1 ACCEPT statement in a CICS program.

Related tasks

[“Assigning input from a screen or file \(ACCEPT\)” on page 35](#)

Related references

CURRENT-DATE (*Enterprise COBOL for z/OS Language Reference*)

Calling to or from COBOL programs

You can make calls to or from VS COBOL II, COBOL for MVS & VM, COBOL for OS/390 & VM, and Enterprise COBOL programs by using the CALL statement.

If you are calling a separately compiled COBOL program that was processed with either the separate CICS translator or the integrated CICS translator, you must pass DFHEIBLK and DFHCOMMAREA as the first two parameters in the CALL statement.

Called programs that are processed by the separate CICS translator or the integrated CICS translator can contain any function that is supported by CICS for the language.

Dynamic calls:

You can use COBOL dynamic calls when running under CICS. If a COBOL program contains EXEC CICS statements or contains EXEC SQL statements, the NODYNAM compiler option is required. To dynamically call a program in this case, you can use CALL *identifier* with the NODYNAM compiler option.

If a COBOL program contains no EXEC CICS statements and contains no EXEC SQL statements, there is no requirement to compile with NODYNAM. To dynamically call a program in this case, you can use either CALL *literal* with the DYNAM compiler option, or CALL *identifier*.

Note: END-EXEC cannot be followed by a period when it is associated with EXEC CICS statements even though it is required for EXEC SQL statements.

You must define dynamically called programs in the CICS program processing table (PPT) if you are not using CICS autoinstall. Under CICS, COBOL programs do not support dynamic calls to subprograms that have the RELOAD=YES option coded in their CICS PROGRAM definition. Dynamic calls to programs that are defined with RELOAD=YES can cause a storage shortage. Use the RELOAD=NO option for programs that are to be dynamically called by COBOL.

Interlanguage communication (ILC):

Support for ILC with other high-level languages is available. Where ILC is not supported, you can use CICS LINK, XCTL, and RETURN instead.

The following table shows the calling relationship between COBOL and assembler programs. In the table, assembler programs that conform to the interface that is 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.

<i>Table 65. Calls between COBOL and assembler under CICS</i>		
Calls between COBOL and assembler programs	Language Environment-conforming assembler program	Non-Language Environment-conforming assembler program
From an Enterprise COBOL program to the assembler program?	Yes	Yes
From the assembler program to an Enterprise COBOL program?	Yes	No

Nested programs:

When you compile with the integrated CICS translator, the translator generates the DFHEIBLK and DFHCOMMAREA control blocks with the GLOBAL clause in the outermost program. Therefore if you code nested programs, you do not have to pass these control blocks as arguments on calls to the nested programs.

If you code nested programs and you plan to use the separate CICS translator, pass DFHEIBLK and DFHCOMMAREA as parameters to the nested programs that contain EXEC commands or references to the EXEC interface block (EIB). You must pass the same parameters also to any program that forms part of the control hierarchy between such a program and its top-level program.

For detailed information and examples on calling COBOL programs under CICS, see [Calling subprograms from COBOL programs](#) in *Developing CICS Applications*.

Related concepts

[“Integrated CICS translator”](#) on page 488

Related tasks

[“Using the separate CICS translator”](#) on page 489

[“Choosing the DYNAM or NODYNAM compiler option”](#) on page 502

[“Handling errors when calling programs”](#) on page 251

ILC under CICS (*Language Environment Writing Interlanguage Communication Applications*)

The external CICS interface (*CICS Transaction Server for z/OS External Interfaces Guide*)

Related references

[“DYNAM”](#) on page 356

Determining the success of ECI calls

After calls to the external CICS interface (ECI), the content of the RETURN-CODE special register is set to an unpredictable value. 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 occurs at termination, set the RETURN-CODE special register before you terminate your program. To make the job return code reflect the status of the last call to CICS, set the RETURN-CODE special register based on the response codes from the last call to the external CICS interface.

Related tasks

[Using EXCI with CICS](#)

Compiling with the CICS option

Use the CICS compiler option to enable the integrated CICS translator and to specify CICS suboptions.

If you specify the NOCICS option, the compiler diagnoses and discards any CICS statements that it finds in your source program. If you have already used the separate CICS translator, you must use NOCICS.

You can specify the CICS option in any of the compiler option sources: compiler invocation, PROCESS or CBL statements, or installation default. If the CICS option is the COBOL installation default, you cannot specify CICS suboptions. However, making the CICS option the installation default is not recommended, because the changes that are made by the integrated CICS translator are not appropriate for non-CICS applications.

All CBL or PROCESS statements must precede any comment lines, in accordance with the rules for Enterprise COBOL.

The COBOL compiler passes to the integrated CICS translator the CICS suboption string that you provide in the CICS compiler option. The compiler does not analyze the suboption string.

When you use the integrated CICS translator, you must compile with the following options:

Compiler option	Comment
CICS	If you specify DYNAM or NORENT, the compiler forces NODYNAM and RENT on.
NODYNAM	Must be in effect with CICS
RENT	Must be in effect with CICS

In addition, IBM recommends that you use the compiler option WORD(CICS) to cause the compiler to flag language elements that are not supported under CICS.

To compile your program with the integrated CICS translator, you can use the standard JCL procedural statements that are supplied with COBOL. In addition to specifying the above compiler options, you must change your JCL in two ways:

- Specify the STEPLIB override for the COBOL step.
- Add the data set that contains the integrated CICS translator services, unless these services are in the linklist.

The default name of the data set for CICS Transaction Server 6.1 is CICSTS61.CICS.SDFHLOAD, but your installation might have changed the name. For example, you might have the following line in your JCL:

```
//STEPLIB DD DSN=CICSTS61.CICS.SDFHLOAD,DISP=SHR
```

The COBOL compiler listing includes the error diagnostics (such as syntax errors in the CICS statements) that the integrated CICS translator generates. The listing reflects the input source; it does not include the COBOL statements that the integrated CICS translator generates.

Compiling a sequence of programs: When you use the CICS option to compile a source file that contains a sequence of COBOL programs, the order of precedence of the options from highest to lowest is:

- Options that are specified in the CBL or PROCESS card that initiates the unit of compilation
- Options that are specified when the compiler is started
- CICS default options

Related concepts

[“Integrated CICS translator” on page 488](#)

Related tasks

[“Coding COBOL programs to run under CICS” on page 483](#)

[“Separating CICS suboptions” on page 488](#)

[Developing CICS Applications](#)

Related references

[“CICS” on page 342](#)

[“Conflicting compiler options” on page 334](#)

Separating CICS suboptions

You can partition the specification of CICS suboptions into multiple CBL statements. CICS suboptions are cumulative. The compiler concatenates them from multiple sources in the order that they are specified.

For example, suppose that a JCL file has the following code:

```
//STEP1 EXEC IGYWC, . . .
//PARM.COBOL="CICS("FLAG(I) )"
//COBOL.SYSIN DD *
  CBL CICS("DEBUG")
  CBL CICS("LINKAGE")
  IDENTIFICATION DIVISION.
  PROGRAM-ID. COBOL1.
```

During compilation, the compiler passes the following CICS suboption string to the integrated CICS translator:

```
"FLAG(I) DEBUG LINKAGE"
```

The concatenated strings are delimited with single spaces and with a pair of quotation marks (" ") or a pair of apostrophes (' ') around the group. When the compiler finds multiple instances of the same CICS suboption, the last specification of the suboption in the concatenated string takes effect. The compiler limits the length of the concatenated CICS suboption string to 4 KB.

Related references

[“CICS” on page 342](#)

Integrated CICS translator

When you compile a COBOL program using the CICS compiler option, the COBOL compiler works with the integrated CICS translator to handle both native COBOL and embedded CICS statements in the source program.

When the compiler encounters CICS statements, and at other significant points in the source program, the compiler interfaces with the integrated CICS translator. All text between EXEC CICS and END-EXEC statements is passed to the translator. The translator takes appropriate actions and then returns to the compiler, typically indicating which native language statements to generate.

Although you can still translate embedded CICS statements separately, it is recommended that you use the integrated CICS translator instead. Certain restrictions that apply when you use the separate translator do not apply when you use the integrated translator, and using the integrated translator provides several advantages:

- You can use Debug Tool to debug the original source instead of the expanded source that the separate CICS translator generates.
- You do not need to separately translate the EXEC CICS or EXEC DLI statements that are in copybooks.
- There is no intermediate data set for a translated but not compiled version of the source program.
- Only one output listing instead of two is produced.
- Using nested programs that contain EXEC CICS statements is simpler. DFHCOMMAREA and DFHEIBLK are generated with the GLOBAL attribute in the outermost program. You do not need to pass them

as arguments on calls to nested programs or specify them in the USING phrase of the PROCEDURE DIVISION header of nested programs.

- You can keep nested programs that contain EXEC CICS statements in separate files, and include those nested programs by using COPY statements.
- REPLACE statements can affect EXEC CICS statements.
- You can compile programs that contain CICS statements in a batch compilation (compilation of a sequence of programs).
- Because the compiler generates binary fields in CICS control blocks with format COMP-5 instead of BINARY, there is no dependency on the setting of the TRUNC compiler option. You can use any setting of the TRUNC option in CICS programs, subject only to the requirements of the application logic and use of user-defined binary fields.

Note: The CICS documentation states that the EXCI translator option is not supported for programs compiled with the integrated CICS translator, but CICS has reversed this position. You can now compile with the EXCI translator option and ignore the warning message DFH7006I.

Related concepts

[Developing CICS Applications](#) (The integrated CICS translator)

Related tasks

[“Coding COBOL programs to run under CICS” on page 483](#)

[“Compiling with the CICS option” on page 487](#)

Related references

[“CICS” on page 342](#)

[“TRUNC” on page 409](#)

Using the separate CICS translator

To run a COBOL program under CICS, you can use the separate CICS translator to convert the CICS commands to COBOL statements, and then compile and link the program to create the executable module. However, using the CICS translator that is integrated with Enterprise COBOL is recommended. The separate CICS translator has not been updated for newer COBOL language such as floating comment delimiters, JSON GENERATE and JSON PARSE, and compiler directives. To use the latest features of the COBOL compiler, use the integrated CICS translator.

To translate CICS statements separately, use the COBOL3 translator option. This option causes the following line to be inserted:

```
CBL RENT,NODYNAM,
```

You can suppress the insertion of a CBL statement by using the CICS translator option NOCBLCARD.

After you use the separate CICS translator, use the following compiler options when you compile the program:

Required compiler option	Condition
RENT	
NODYNAM	The program is translated by the CICS translator.

In addition, IBM recommends that you use the compiler option WORD(CICS) to cause the compiler to flag language elements that are not supported under CICS.

The following TRUNC compiler option recommendations are based on expected values for binary data items:

Table 68. **TRUNC compiler options recommended for the separate CICS translator**

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.

For example, if you use the separate CICS translator and have a data item defined as PIC S9(8) BINARY that might receive a value greater than eight digits, use the TRUNC (BIN) compiler option, change the item to USAGE COMP-5, or change the PICTURE clause.

You might also want to avoid using these options, which have no effect:

- ADV
- FASTSRT
- OUTDD

The input data set for the compiler is the data set that you received as a result of translation, which is SYSPUNCH by default.

Related concepts

[“Integrated CICS translator” on page 488](#)

Related tasks

[“Compiling with the CICS option” on page 487](#)

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 that are not supported under CICS are flagged 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
- DELETE³
- FACTORY
- FD
- FILE¹
- FILE-CONTROL¹
- INPUT-OUTPUT¹
- INVOKE
- I-O-CONTROL
- MERGE
- METHOD
- OBJECT
- OPEN
- READ
- RERUN
- REWRITE

- ~~SD~~^{1, 2}
- SELF
- START
- SUPER
- WRITE

Notes:

1. 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 to remove the words from the list of words marked as restricted.
2. The SORT keyword is not restricted, but the SD keyword is. This allows you to use the format 2 (table) sort statement but not the format 1 (file) sort statement.
3. If you restrict the DELETE keyword, you may still use the DELETE function of BASIS processing.

Related tasks

[“Compiling with the CICS option” on page 487](#)

[“Sorting under CICS” on page 238](#)

Related references

[“WORD” on page 415](#)

Handling errors by using CICS HANDLE

The setting of the CBLPSHPOP runtime option affects the state of the HANDLE specifications when a program calls COBOL subprograms using a CALL statement.

When CBLPSHPOP is ON and a COBOL subprogram (not a nested program) is called with a CALL statement, the following actions occur:

1. As part of program initialization, the run time suspends the HANDLE specifications of the calling program (using EXEC CICS PUSH HANDLE).
2. The default actions for HANDLE apply until the called program issues its own HANDLE commands.
3. As part of program termination, the run time reinstates the HANDLE specifications of the calling program (using EXEC CICS POP HANDLE).

If you use the CICS HANDLE CONDITION or CICS 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, or abends that were caused by another program invoked with 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 occur.

Performance considerations: When CBLPSHPOP is OFF, the run time does not perform CICS PUSH or POP on a CALL to any COBOL subprogram. If the subprograms do not use any of the EXEC CICS condition-handling commands, you can run with CBLPSHPOP(OFF), thus eliminating the overhead of the PUSH HANDLE and POP HANDLE commands. As a result, performance can be improved compared to running with CBLPSHPOP(ON).

If you are migrating an application from the VS COBOL II run time to the Language Environment run time, see the related reference for information about the CBLPSHPOP option for additional considerations.

[“Example: handling errors by using CICS HANDLE” on page 492](#)

Related tasks

[“Running efficiently with CICS, IMS, or VSAM” on page 769](#)

Related references

Enterprise COBOL for z/OS Migration Guide

Enterprise COBOL for z/OS Performance Tuning Guide

Example: handling errors by using CICS HANDLE

The following example shows 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. A condition is handled in one of three scenarios.

(1)

CBPLSHPOP(ON): If the CICS READ command in program B causes a condition, the condition is not handled by program A (the HANDLE specifications are suspended because the run time performs a CICS PUSH HANDLE). The condition turns into a transaction abend.

(2)

CBPLSHPOP(OFF): If the CICS READ command in program B causes a condition, the condition is not handled by program A (the run time diagnoses the attempt to perform cross-program branching by using a CICS HANDLE command with the LABEL option). The condition turns 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 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'.

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('MAINFILE')
        INTO(RECORD)
        RIDFLD(ACCTNO)
        END-EXEC.

END PROGRAM B.
```

Chapter 23. Programming for a Db2 environment

In general, the coding for a COBOL program will be the same if you want the program to access a Db2 database. However, to retrieve, update, insert, and delete Db2 data and use other Db2 services, you must use SQL statements.

To communicate with Db2, do these steps:

- Code any SQL statements that you need, delimiting them with EXEC SQL and END-EXEC statements.
- Either use the Db2 stand-alone precompiler, or compile with the SQL compiler option and use the Db2 coprocessor.

Related concepts

[“Db2 coprocessor” on page 493](#)

[“COBOL and Db2 CCSID determination” on page 498](#)

Related tasks

[“Using the separate Db2 precompiler” on page 494](#)

[“Coding SQL statements” on page 494](#)

[“Compiling with the SQL option” on page 497](#)

[“Choosing the DYNAM or NODYNAM compiler option” on page 502](#)

Related references

[“Differences in how the Db2 precompiler and coprocessor behave” on page 500](#)

Db2 coprocessor

When you use the Db2 coprocessor (called *SQL statement coprocessor* by Db2), the compiler handles your source programs that contain embedded SQL statements without your having to use a separate precompile step.

To use the Db2 coprocessor, specify the SQL compiler option.

When the compiler encounters SQL statements in the source program, it interfaces with the Db2 coprocessor. All text between EXEC SQL and END-EXEC statements is passed to the coprocessor. The coprocessor takes appropriate actions for the SQL statements and indicates to the compiler which native COBOL statements to generate for them.

Although the use of a separate precompile step continues to be supported, it is recommended that you use the coprocessor instead:

- Interactive debugging with Debug Tool is enhanced when you use the coprocessor because you see the SQL statements (not the generated COBOL source) in the listing.
- 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. With the 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 copybooks.
 - REPLACE statements work in SQL statements.

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 in the DBRMLIB 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.

Related concepts

[“COBOL and Db2 CCSID determination” on page 498](#)

Related tasks

[“Using the separate Db2 precompiler” on page 494](#)

[“Compiling with the SQL option” on page 497](#)

Related references

[“Differences in how the Db2 precompiler and coprocessor behave” on page 500](#)

[“SQL” on page 398](#)

Using the separate Db2 precompiler

To run a COBOL program that has SQL statements, you can use the separate Db2 precompiler to convert the SQL statements to COBOL statements, and then compile and link the program to create the executable module.

However, using the Db2 precompiler that is integrated with Enterprise COBOL is recommended, since the separate Db2 precompiler is no longer being enhanced by IBM. In particular, the separate Db2 precompiler has not been updated for newer COBOL language such as floating comment delimiters, JSON GENERATE and JSON PARSE, and compiler directives. To use the latest features of the COBOL compiler, use the integrated Db2 coprocessor.

Note: The Db2 interface used by the precompiler is 31-bit. The precompiler output cannot be compiled with LP(64). You must use LP(32) for Db2 precompiled programs.

Related concepts

[“Db2 coprocessor” on page 493](#)

Related references

[“Differences in how the Db2 precompiler and coprocessor behave” on page 500](#)

Coding SQL statements

Delimit SQL statements with EXEC SQL and END-EXEC. The EXEC SQL and END-EXEC delimiters must each be complete on one line. You cannot continue them across multiple lines. Do not code COBOL statements within EXEC SQL statements.

You also need to do these special steps:

- Code an EXEC SQL INCLUDE statement to include an SQL communication area (SQLCA) in the WORKING-STORAGE SECTION or LOCAL-STORAGE SECTION of the outermost program. LOCAL-STORAGE is recommended for recursive programs and programs that use the THREAD compiler option.
- Define all host variables that you use in SQL statements in the WORKING-STORAGE SECTION, LOCAL-STORAGE SECTION, or LINKAGE SECTION. However, you do not need to identify them with EXEC SQL BEGIN DECLARE SECTION and EXEC SQL END DECLARE SECTION.

Restriction: You cannot use SQL statements in object-oriented classes or methods.

Related tasks

[“Using SQL INCLUDE with the Db2 coprocessor” on page 495](#)

[“Using character data in SQL statements” on page 495](#)

[“Using national decimal data in SQL statements” on page 496](#)

[“Using national group items in SQL statements” on page 496](#)

[“Using binary items in SQL statements” on page 497](#)

[“Determining the success of SQL statements” on page 497](#)

DB2 Application Programming and SQL Guide (Coding SQL statements in a COBOL application)

Related references

[“Code-page determination for string host variables in SQL statements” on page 499](#)
DB2 SQL Reference

Using SQL INCLUDE with the Db2 coprocessor

An SQL INCLUDE statement is treated identically to a native COBOL COPY statement when you use the SQL compiler option.

The following two lines are therefore treated the same way. (The period that ends the EXEC SQL INCLUDE statement is required.)

```
EXEC SQL INCLUDE name END-EXEC.  
COPY "name" .
```

The processing of the *name* in an SQL INCLUDE statement follows the same rules as those of the literal in a COPY *literal-1* statement that does not have a REPLACING phrase.

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.

Related references

Chapter 20, “Compiler-directing statements,” on page 429
[“Differences in how the Db2 precompiler and coprocessor behave” on page 500](#)
COPY statement (*Enterprise COBOL for z/OS Language Reference*)

Using character data in SQL statements

You can code any of the following USAGE clauses to describe host variables for character data that you use in EXEC SQL statements: USAGE DISPLAY for single-byte or UTF-8 data, USAGE DISPLAY-1 for DBCS data, USAGE NATIONAL for UTF-16 data, or USAGE UTF-8 for UTF-8 data.

When you use the stand-alone Db2 precompiler, you must specify the code page (CCSID) in EXEC SQL DECLARE statements for host variables that are declared with USAGE NATIONAL. You must specify the code page for host variables that are declared with USAGE DISPLAY or DISPLAY-1 only if the CCSID that is in effect for the COBOL CODEPAGE compiler option does not match the CCSIDs that are used by Db2 for character and graphic data.

Consider the following code. The two highlighted statements are unnecessary when you use the integrated Db2 coprocessor (with the SQLCCSID compiler option, as detailed in the related concept below), because the code-page information is handled implicitly.

```
CBL CODEPAGE(1140) NSYMBOL(NATIONAL)  
.  
WORKING-STORAGE SECTION.  
EXEC SQL INCLUDE SQLCA END-EXEC.  
01 INT1 PIC S9(4) USAGE COMP.  
01 C1140.  
49 C1140-LEN PIC S9(4) USAGE COMP.  
49 C1140-TEXT PIC X(50).  
EXEC SQL DECLARE :C1140 VARIABLE CCSID 1140 END-EXEC.  
01 G1200.  
49 G1200-LEN PIC S9(4) USAGE COMP.  
49 G1200-TEXT PIC N(50) USAGE NATIONAL.  
EXEC SQL DECLARE :G1200 VARIABLE CCSID 1200 END-EXEC.  
.  
EXEC SQL FETCH C1 INTO :INT1, :C1140, :G1200 END-EXEC.
```

If you specify EXEC SQL DECLARE *variable-name* VARIABLE CCSID *nnnn* END-EXEC, that specification overrides the implied CCSID. For example, the following code would cause Db2 to treat C1208-TEXT as

encoded in UTF-8 (CCSID 1208) rather than as encoded in the CCSID in effect for the COBOL CODEPAGE compiler option:

```
01 C1208.  
   49 C1208-LEN PIC S9(4) USAGE COMP.  
   49 C1208-TEXT PIC X(50).  
EXEC SQL DECLARE :C1208 VARIABLE CCSID 1208 END-EXEC.
```

The NSYMBOL compiler option has no effect on a character literal inside an EXEC SQL statement. Character literals in an EXEC SQL statement follow the SQL rules for character constants.

Related concepts

[“COBOL and Db2 CCSID determination” on page 498](#)

Related tasks

[“Using the separate Db2 precompiler” on page 494](#)
DB2 Application Programming and SQL Guide (Coding SQL statements in a COBOL application)

Related references

[“Differences in how the Db2 precompiler and coprocessor behave” on page 500](#)
[“CODEPAGE” on page 343](#)
DB2 SQL Reference

Using national decimal data in SQL statements

You can use national decimal host variables in EXEC SQL statements when you use either the integrated Db2 coprocessor or the Db2 precompiler. You do not need to specify the CCSID in EXEC SQL DECLARE statements in either case. CCSID 1200 is used automatically.

Any national decimal host variable that you specify in an EXEC SQL statement must have the following characteristics:

- It must be signed.
- It must be specified with the SIGN LEADING SEPARATE clause.
- USAGE NATIONAL must be in effect implicitly or explicitly.

Related concepts

[“Formats for numeric data” on page 47](#)

Related tasks

[“Defining national numeric data items” on page 132](#)

Related references

[“Differences in how the Db2 precompiler and coprocessor behave” on page 500](#)

Using national group items in SQL statements

You can use a national group item as a host variable in an EXEC SQL statement. The national group item is treated with group semantics (that is, as shorthand for the set of host variables that are subordinate to the group item) rather than as an elementary item.

Because all subordinate items in a national group must have USAGE NATIONAL, a national group item cannot describe a variable-length string.

Related tasks

[“Using national groups” on page 137](#)

Using binary items in SQL statements

For binary data items that you specify in an EXEC SQL statement, you can define the data items as either USAGE COMP-5 or as USAGE BINARY, COMP, or COMP-4.

If you define the binary data items as USAGE BINARY, COMP, or COMP-4, use the TRUNC (BIN) option. (This technique might have a larger effect on performance than using USAGE COMP-5 on individual data items.) If instead TRUNC (OPT) or TRUNC (STD) is in effect, the compiler accepts the items but the data might not be valid because of the decimal truncation rules. You need to ensure that truncation does not affect the validity of the data.

Related concepts

[“Formats for numeric data” on page 47](#)

Related references

[“TRUNC” on page 409](#)

Determining the success of SQL statements

When Db2 finishes executing an SQL statement, Db2 sends a return code in the SQLCA structure, with one exception, to indicate whether the operation succeeded or failed. In your program, test the return code and take any necessary action.

The exception 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.

After execution of SQL statements, the content of the RETURN-CODE special register might not be valid. Therefore, even if your COBOL program terminates normally after successfully using SQL statements, the job step could end with an undefined return code. To ensure that a meaningful return code is given at termination, set the RETURN-CODE special register before terminating your program.

Related tasks

DB2 Application Programming and SQL Guide (Coding SQL statements in a COBOL application)

Compiling with the SQL option

You use the SQL compiler option to enable the Db2 coprocessor and to specify Db2 suboptions.

You can specify the SQL option in any of the compiler option sources: compiler invocation, PROCESS or CBL statements, OPTFILE, 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 in the SQL compiler option is made available to the Db2 coprocessor. Only the Db2 coprocessor views the contents of the string.

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 LNKLIST. Typically, this data set is called xxxxxx.SDSNLOAD. For example, for Db2 11 it might be *DSNB10.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(MAINFILE),DISP=SHR
//STEPLIB DD DSN=DSN910.SDSNLOAD,DISP=SHR
```

Compiling a batch of programs: If you use the SQL option when compiling a source file that contains a sequence of COBOL programs (a batch compile sequence), SQL must be in effect for only the first program of the sequence. Although you can specify SQL upon compiler invocation, the option will be in effect for only the first program. If you specify SQL in a CBL or PROCESS statement for a program other than the first program in the batch, you will receive a compiler diagnostic message.

Related concepts

[“Db2 coprocessor” on page 493](#)

[“COBOL and Db2 CCSID](#)

[determination” on page 498](#)

Related tasks

[“Separating Db2 suboptions” on page 498](#)

[“Choosing the DYNAM or NODYNAM compiler option” on page 502](#)

Related references

[“DYNAM” on page 356](#)

[“SQL” on page 398](#)

DB2 Command Reference

Separating Db2 suboptions

Because of the concatenation of multiple SQL option specifications, you can separate Db2 suboptions (which might not fit in one CBL statement) into multiple CBL statements.

The options that you include in the suboption string are cumulative. The compiler concatenates these suboptions from multiple sources in the order that they are specified. For example, suppose that your source file has the following code:

```
//STEP1 EXEC IGYWC, . . .
// PARM.COBOL='SQL("string1")'
//COBOL.SYSIN DD *
CBL SQL("string2")
CBL SQL("string3")
IDENTIFICATION DIVISION.
PROGRAM-ID. DRIVER1.
```

During compilation, the compiler passes the following suboption string to the Db2 coprocessor:

```
"string1 string2 string3"
```

The concatenated strings are delimited with single spaces. If the compiler finds multiple instances of the same SQL suboption, the last specification of that suboption in the concatenated string takes effect. The compiler limits the length of the concatenated Db2 suboption string to 4 KB.

COBOL and Db2 CCSID determination

All Db2 string data other than BLOB, BINARY, and VARBINARY data has an associated encoding scheme and a coded character set ID (CCSID). This is true for fixed-length and variable-length character strings, fixed-length and variable-length graphic character strings, CLOB host variables, and DBCLOB host variables.

When you use the integrated Db2 coprocessor, the determination of the code page CCSID that will be associated with the string host variables used in SQL statement processing depends on the setting of the COBOL SQLCCSID option, on the programming techniques used, and on various Db2 configuration options.

IBM recommends that you set the COBOL CCSID value the same as the Db2 DSNHDECP value and/or the value for precompiler CCSID option.

When you use the SQL and SQLCCSID COBOL compiler options, the CCSID value *nnnnn* that is specified in the CODEPAGE compiler option, or that is determined from the COBOL data type of a host variable, is communicated automatically from COBOL to Db2. Db2 associates the COBOL CCSID with host variables, overriding the CCSID that would otherwise be implied by Db2 external mechanisms and defaults. This associated CCSID is used for the processing of the SQL statements that reference host variables.

When you use the SQL and NOSQLCCSID compiler options, the CCSID value *nnnnn* that is specified in the CODEPAGE compiler option is used only for processing COBOL statements within the COBOL program; that CCSID is not used for the processing of SQL statements. Instead, Db2 assumes in processing SQL statements that host variable data values are encoded according to the CCSID or CCSIDs that are specified through Db2 external mechanisms and defaults.

Related concepts

[“Db2 coprocessor” on page 493](#)

Related tasks

[“Programming with the SQLCCSID or NOSQLCCSID option” on page 500](#)

Related references

[“Code-page determination for string host variables in SQL statements” on page 499](#)

[“CODEPAGE” on page 343](#)

[“SQL” on page 398](#)

[“SQLCCSID” on page 399](#)

Code-page determination for string host variables in SQL statements

When you use the integrated Db2 coprocessor (SQL compiler option), the code page for processing string host variables in SQL statements is determined as shown below, in descending order of precedence.

- A host variable that has USAGE NATIONAL is always processed by Db2 using CCSID 1200 (Unicode UTF-16). For example:

```
01 hostvariable pic n(10) usage national.
```

- A host variable that has USAGE UTF-8 is always processed by Db2 using CCSID 1208 (Unicode UTF-8). For example:

```
01 hostvariable pic u byte-length 10 usage utf-8.
```

- An alphanumeric host variable that has an explicit FOR BIT DATA declaration is set by Db2 to CCSID 66535, which indicates that the variable does not represent encoded characters. For example:

```
EXEC SQL DECLARE hostvariable VARIABLE FOR BIT DATA END-EXEC
```

- A BLOB, BINARY, or VARBINARY host variable has no CCSID association. These string types do not represent encoded characters.
- A host variable for which you specify an explicit CCSID override in the SQLDA is processed with that CCSID.
- A host variable that you specify in a declaration with an explicit CCSID is processed with that CCSID. For example:

```
EXEC SQL DECLARE hostvariable VARIABLE CCSID nnnnn END-EXEC
```

- An alphanumeric host variable, if the SQLCCSID compiler option is in effect, is processed with the CCSID *nnnnn* from the CODEPAGE compiler option.
- A DBCS host variable, if the SQLCCSID option is in effect, is processed with the mapped value *mmmmm*, which is the pure DBCS CCSID component of the mixed (MBCS) CCSID *nnnnn* from the CODEPAGE (*nnnnn*) compiler option.

- An alphanumeric or DBCS host variable, if the NOSQLCCSID option is in effect, is processed with the CCSID from the Db2 ENCODING bind option, if specified, or from the APPLICATION ENCODING set in DSNHDECP through the Db2 installation panel DSNTIPF.

Related references

[“CODEPAGE” on page 343](#)

[“SQLCCSID” on page 399](#)

Programming with the SQLCCSID or NOSQLCCSID option

In general, the SQLCCSID option is recommended for new applications that use the integrated Db2 coprocessor, and as a long-term direction for existing applications. The NOSQLCCSID option is recommended as a mechanism for migrating existing precompiler-based applications to use the integrated Db2 coprocessor.

The SQLCCSID option is recommended for COBOL-Db2 applications that have any of these characteristics:

- Use COBOL Unicode support
- Use other COBOL syntax that is indirectly sensitive to CCSID encoding, such as XML support or object-oriented syntax for Java interoperability
- Process character data that is encoded in a CCSID that is different from the default CCSID assumed by Db2

The NOSQLCCSID option is recommended for applications that require the highest compatibility with the behavior of the Db2 precompiler.

For applications that use COBOL alphanumeric data items as host variables interacting with Db2 string data that is defined with the FOR BIT DATA subtype, you must either:

- Use the NOSQLCCSID compiler option
- Specify explicit FOR BIT DATA declarations for those host variables, for example:

```
EXEC SQL DECLARE hostvariable VARIABLE FOR BIT DATA END-EXEC
```

Usage notes

- If you use the Db2 DCLGEN command to generate COBOL declarations for a table, you can optionally create FOR BIT DATA declarations automatically. To do so, specify the DCLBIT(YES) option of the DCLGEN command.
- **Performance consideration:** Using the SQLCCSID compiler option could result in some performance overhead in SQL processing, because with SQLCCSID in effect the default Db2 CCSID association mechanism is overridden with a mechanism that works on a per-host-variable basis.

Related concepts

[“Db2 coprocessor” on page 493](#)

Related tasks

[“Using the separate Db2 precompiler” on page 494](#)

Related references

[“SQLCCSID” on page 399](#)

Differences in how the Db2 precompiler and coprocessor behave

The sections that follow enumerate the differences in behavior between the stand-alone COBOL Db2 precompiler and the integrated COBOL Db2 coprocessor.

For details about the CCSID determination under the Db2 precompiler and coprocessor, see [“COBOL and Db2 CCSID determination” on page 498](#).

Period at the end of EXEC SQL INCLUDE statements

Precompiler: The Db2 precompiler does not require that a period end each EXEC SQL INCLUDE statement. If a period is specified, the precompiler processes it as part of the statement. If a period is not specified, the precompiler accepts the statement as if a period had been specified.

Coprocessor: The Db2 coprocessor treats each EXEC SQL INCLUDE statement like a COPY statement, and requires that a period end the statement. For example:

```
IF A = B THEN
  EXEC SQL INCLUDE some_code_here END-EXEC.
ELSE
  .
END-IF .
```

Note that the period does not terminate the IF statement.

EXEC SQL and REPLACE or COPY REPLACING

Precompiler: With the Db2 precompiler, COBOL REPLACE statements and the REPLACING phrase of the COPY statement act on the expanded source created from the EXEC SQL statement. COBOL rules for REPLACE and REPLACING are used.

Coprocessor: With the Db2 coprocessor, REPLACE and COPY . . . REPLACING statements act on the original source program, including EXEC SQL statements.

Different behavior can result, as in the following example:

```
REPLACE == ABC == By == XYZ ==.
01  G.
   02  ABC PIC X(10).
   EXEC SQL SELECT * INTO :G.ABC FROM TABLE1 END-EXEC
```

With the precompiler, the reference to G . ABC will appear as ABC of G in the expanded source and will be replaced with XYZ of G. With the coprocessor, replacement will not occur, because ABC is not delimited by separators in the original source string G . ABC.

Source code after an END-EXEC statement

Precompiler: The Db2 precompiler ignores any code that follows END-EXEC statements on the same line.

Coprocessor: The Db2 coprocessor processes code that follows END-EXEC statements on the same line.

Multiple definitions of host variables

Precompiler: The Db2 precompiler does not require that host variable references be unique. The first definition that maps to a valid Db2 data type is used.

Coprocessor: The Db2 coprocessor requires that each host variable reference be unique. The coprocessor diagnoses nonunique references to host variables. You must fully qualify host variable references to make them unique or use the [“QUALIFY” on page 390](#) compiler option.

EXEC SQL statement continuation lines

Precompiler: The Db2 precompiler requires that EXEC SQL statements start in columns 12 through 72. Continuation lines of the statements can start anywhere in columns 8 through 72.

Coprocessor: The Db2 coprocessor requires that all lines of an EXEC SQL statement, including continuation lines, be coded in columns 12 through 72.

Bit-data host variables

Precompiler: With the Db2 precompiler, a COBOL alphanumeric data item can be used as a host variable to hold Db2 character data that has subtype FOR BIT DATA. An explicit EXEC SQL DECLARE VARIABLE statement that declares that host variable as FOR BIT DATA is not required.

Coprocessor: With the Db2 coprocessor, a COBOL alphanumeric data item can be used as a host variable to hold Db2 character data that has subtype FOR BIT DATA if an explicit EXEC SQL DECLARE VARIABLE statement for that host variable is specified in the COBOL program. For example:

```
EXEC SQL DECLARE :HV1 VARIABLE FOR BIT DATA END-EXEC.
```

As an alternative to adding EXEC SQL DECLARE . . . FOR BIT DATA statements, you can use the NOSQLCCSID compiler option. For details, see the related reference about code-page determination below.

SQL-INIT-FLAG

Precompiler: With the Db2 precompiler, if you pass host variables that might be located at different addresses when the program is called more than once, the called program must reset SQL-INIT-FLAG. Resetting this flag indicates to Db2 that storage must be initialized when the next SQL statement runs. To reset the flag, insert the statement MOVE ZERO TO SQL-INIT-FLAG in the PROCEDURE DIVISION of the called program ahead of any executable SQL statements that use those host variables.

Coprocessor: With the Db2 coprocessor, the called program does not need to reset SQL-INIT-FLAG. An SQL-INIT-FLAG is automatically defined in the program to aid program portability. However, statements that modify SQL-INIT-FLAG, such as MOVE ZERO TO SQL-INIT-FLAG, have no effect on the SQL processing in the program.

Related concepts

[“Db2 coprocessor” on page 493](#)

[“COBOL and Db2 CCSID determination” on page 498](#)

Related tasks

[“Using the separate Db2 precompiler” on page 494](#)

Related references

[“Code-page determination for string host variables in SQL statements” on page 499](#)

[“SQLCCSID” on page 399](#)

Choosing the DYNAM or NODYNAM compiler option

For COBOL programs that have EXEC SQL statements, your choice of the compiler option DYNAM or NODYNAM depends on the operating environment.

When you run under:

- TSO or IMS: You can use either the DYNAM or NODYNAM compiler option.

Note that IMS and Db2 share a common alias name, DSNHLI, for the language interface module. You must concatenate your libraries as follows:

- If you use IMS with the DYNAM option, concatenate the IMS library first.
- If you run your application only under Db2, concatenate the Db2 library first.

- CICS or the Db2 call attach facility (CAF): You must use the NODYNAM compiler option.

Because stored procedures use CAF, you must also compile COBOL stored procedures with the NODYNAM option.

Related tasks

[“Compiling with the SQL option” on page 497](#)

DB2 Application Programming and SQL Guide (Programming for the call attachment facility)

Related references

[“DYNAM” on page 356](#)

Chapter 24. Developing COBOL programs for 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 Enterprise COBOL, you can invoke IMS facilities using the following interfaces:

- CBLTDLI call
- Language Environment callable service CEETDLI
- EXEC SQLIMS statements

CEETDLI behaves essentially the same way as CBLTDLI, except that CEETDLI enables LE condition handling to be used. There are some instances when you cannot use Language Environment condition handling when using CBLTDLI under IMS.

You can also run object-oriented COBOL programs in a Java dependent region. You can mix the object-oriented COBOL and Java languages in a single application.

Related concepts

[“IMS SQL coprocessor” on page 505](#)

Related tasks

[“Coding SQLIMS statements” on page 506](#)

[“Compiling with the SQLIMS option” on page 507](#)

[“Compiling and linking COBOL programs for running under IMS” on page 508](#)

[“Using object-oriented COBOL and Java under IMS” on page 509](#)

[“Calling a COBOL method from a Java application under IMS” on page 509](#)

[“Building a mixed COBOL-Java application that starts with COBOL” on page 510](#)

[“Writing mixed-language IMS applications” on page 510](#)

IMS SQL coprocessor

When you use the IMS SQL coprocessor (called *SQL statement coprocessor* by IMS), the compiler handles your source programs that contain embedded SQL statements.

When the compiler encounters SQLIMS statements in the source program, it interfaces with the IMS SQL coprocessor. All text between EXEC SQLIMS and END-EXEC statements is passed to the coprocessor. The coprocessor takes appropriate actions for the SQLIMS statements and indicates to the compiler what native COBOL statements to generate for them.

Notes:

- The IMS SQL coprocessor processes embedded SQLIMS statements, not embedded SQL statements.
- IMS program might contain EXEC SQL statements for accessing a Db2 SQL database, EXEC SQLIMS statements for accessing an IMS DLI databases, or both. The SQL option enables EXEC SQL statements while the SQLIMS option enables EXEC SQLIMS statements.

With the IMS SQL coprocessor, you can use statements in the following ways:

- Use EXEC SQLIMS statements in any nested program.
- Use EXEC SQLIMS statements in COPYBOOKS.
- REPLACE statements work in SQLIMS statements.

Related tasks

[“Coding SQLIMS statements” on page 506](#)

[“Compiling with the SQLIMS option” on page 507](#)

[“Compiling and linking COBOL programs for running under IMS” on page 508](#)

Related references

[“SQLIMS” on page 400](#)

Coding SQLIMS statements

Delimit SQLIMS statements with EXEC SQLIMS and END-EXEC. The EXEC SQLIMS and END-EXEC delimiters must each be complete on one line. Do not code COBOL statements within EXEC SQLIMS statements.

Code an EXEC SQLIMS INCLUDE statement to include an SQLIMS communication area (SQLCA) in the WORKING-STORAGE SECTION or LOCAL-STORAGE SECTION of the outermost program. The LOCAL-STORAGE SECTION is recommended for recursive programs and programs that use the THREAD compiler option.

Restriction: You cannot use SQLIMS statements in object-oriented classes or methods.

Related tasks

[“Using SQLIMS INCLUDE with the IMS SQL coprocessor” on page 506](#)

[“Using character data in SQLIMS statements” on page 506](#)

[“Using binary items in SQLIMS statements” on page 507](#)

[“Determining the success of SQLIMS statements” on page 507](#)

Using SQLIMS INCLUDE with the IMS SQL coprocessor

An SQLIMS INCLUDE statement is treated identically to a native COBOL COPY statement when you use the SQLIMS compiler option.

The following two lines are therefore treated the same way. The period that ends the EXEC SQLIMS INCLUDE statement is required.

```
EXEC SQLIMS INCLUDE name END-EXEC.  
COPY "name".
```

The processing of the *name* in an SQLIMS INCLUDE statement follows the same rules as the literal in a COPY *literal-1* statement that does not have a REPLACING phrase.

The library search order for SQLIMS INCLUDE statements is the same SYSLIB concatenation as the compiler uses to resolve COBOL COPY statements that do not specify a library-name.

Related references

Chapter 20, “Compiler-directing statements,” on page 429

COPY statement (*Enterprise COBOL for z/OS Language Reference*)

Using character data in SQLIMS statements

Alphanumeric host data items for use in EXEC SQLIMS statements (host variables) must be defined as USAGE DISPLAY.

Note: Do not use character data items that are defined with USAGE DISPLAY-1 or USAGE NATIONAL as SQLIMS host variables.

Related concepts

[“IMS SQL coprocessor” on page 505](#)

Related references

[“CODEPAGE” on page 343](#)

Using binary items in SQLIMS statements

For binary data items that you specify in an EXEC SQLIMS statement, you can define the data items as either USAGE COMP-5 or as USAGE BINARY, COMP, or COMP-4.

If you define the binary data items as USAGE BINARY, COMP, or COMP-4, use the TRUNC (BIN) compiler option. Using this option might have a larger effect on performance than using USAGE COMP-5 on individual data items. If instead you use the TRUNC (OPT) or TRUNC (STD) compiler options, the compiler accepts the items but the data might not be valid because of the decimal truncation rules. You must ensure that truncation does not affect the validity of the data.

Related concepts

[“Formats for numeric data” on page 47](#)

Related references

[“TRUNC” on page 409](#)

Determining the success of SQLIMS statements

When IMS finishes running an SQLIMS statement, IMS sends a return code in the SQLIMSCA structure to indicate whether the operation succeeded or failed. In your program, test the return code and take any necessary action.

After execution of SQLIMS statements, the content of the RETURN-CODE special register might not be valid. Therefore, even if a program terminates normally after successfully using SQLIMS statements, the job step might end with an undefined return code. To ensure that a meaningful return code is given at termination, set the RETURN-CODE special register before you end the program.

Related tasks

IMS Application Programming Guide

Compiling with the SQLIMS option

Use the SQLIMS compiler option to enable the IMS SQL coprocessor and to specify IMS suboptions.

You can specify the SQLIMS option in any of the compiler option sources: compiler invocation, PROCESS or CBL statements, or installation default. However, you cannot specify IMS suboptions when the SQLIMS option is the COBOL installation default. The IMS suboption string in the SQLIMS compiler option is only available to the IMS SQL coprocessor.

To use the IMS SQL coprocessor, you must compile with the SQLIMS option and IMS must be available on the system on which you compile.

You can use standard JCL procedural statements to compile your program with the IMS SQL coprocessor. In addition to specifying the above compiler options, specify the following item in your JCL:

STEPLIB override for the COBOL step, adding the data set that contains the Db2 coprocessor services, unless these services are in the LNKLIST. Typically, this data set is called xxxxxx.SDSNLOAD. For example, for Db2 11 it might be DSNB10.SDSNLOAD, but your installation might have changed the name.

For example, you might have the following lines in your JCL:

```
//STEPLIB DD DSN=IMS.SDFSRESL,DISP=SHR
```

Compiling a batch of programs:

If you use the SQLIMS option when you compile a source file that contains a sequence of COBOL programs (a batch compile sequence), SQLIMS is in effect for only the first program of the sequence. Although you can specify SQLIMS upon compiler invocation, the option is in effect for only the first program. If you specify SQLIMS in a CBL or PROCESS statement for a program other than the first program in the batch, a compiler diagnostic message is issued.

Related concepts

[“IMS SQL coprocessor” on page 505](#)

Related tasks

[“Separating IMS suboptions” on page 508](#)

Related references

[“SQL” on page 398](#)

Separating IMS suboptions

Because of the concatenation of multiple SQLIMS option specifications, you can separate IMS suboptions (which might not fit in one CBL statement) into multiple CBL statements.

The options that you include in the suboption string are cumulative. The compiler concatenates these suboptions from multiple sources in the order that they are specified. For example, suppose that your source file contains the following code:

```
//STEP1 EXEC IGYWC, . . .  
// PARM.COBOL='SQLIMS("string1")'  
//COBOL.SYSIN DD *  
CBL SQLIMS("string2")  
CBL SQLIMS("string3")  
IDENTIFICATION DIVISION.  
PROGRAM-ID. DRIVER1.
```

During compilation, the compiler passes the following suboption string to the IMS SQL coprocessor:

```
"string1 string2 string3"
```

The concatenated strings are delimited with single spaces. If the compiler finds multiple instances of the same SQLIMS suboption, the last specification of that suboption in the concatenated string takes effect. The compiler limits the length of the concatenated IMS suboption string to 4 KB.

Related concepts

[“IMS SQL coprocessor” on page 505](#)

Related tasks

[“Compiling with the SQLIMS option” on page 507](#)

Compiling and linking COBOL programs for running under IMS

For best performance in the IMS environment, use the RENT compiler option. RENT causes COBOL to generate reentrant code. You can then run your application programs in either *preloaded* mode (the programs are always resident in storage) or *nonpreload* mode without having to recompile using different options.

Preloading can boost performance because subsequent requests for a program can be handled faster when the program is already in storage (rather than being fetched from a library each time it is needed).

For IMS programs, using the RENT compiler option is recommended. You must use the RENT compiler option for a program that is to be run preloaded or both preloaded and nonpreloaded. When you preload a program object that contains COBOL programs, all of the COBOL programs in that program object must be compiled using the RENT option.

You can place programs compiled with the RENT option in the z/OS link pack area. There they can be shared among the IMS dependent regions.

To run above the 16 MB line, an application program must be compiled with RENT. The data for IMS application programs can reside above the 16 MB line, and you can use DATA(31) RENT for programs that use IMS services.

For proper execution of COBOL programs under IMS, observe the following guidelines for the link-edit attributes:

- To link program objects that contain only COBOL programs compiled with the RENT compiler option, link as RENT.
- To link program objects that contain a mixture of COBOL RENT programs and other programs, use the link-edit attributes recommended for the other programs.

Related concepts

[“Storage and its addressability” on page 39](#)

Related tasks

[“Choosing the DYNAM or NODYNAM compiler option” on page 502](#)
Language Environment Programming Guide (Condition handling under IMS)

Related references

[“DATA” on page 349](#)

[“RENT” on page 391](#)

Enterprise COBOL for z/OS Migration Guide (IMS considerations)

Using object-oriented COBOL and Java under IMS

You can mix object-oriented COBOL and Java in an application that runs in a Java dependent region.

For example, you can:

- Call a COBOL method from a Java application. You can build the messaging portion of your application in Java and call COBOL methods to access IMS databases.
- Build a mixed COBOL and Java application that starts with the main method of a COBOL class and that invokes Java routines.

You must run these applications in either a Java message processing (JMP) dependent region or a Java batch processing (JBP) dependent region. A program that reads from the message queue (regardless of the language) must run in a JMP dependent region.

Related tasks

[“Defining a factory section” on page 695](#)

[Chapter 39, “Writing object-oriented programs,” on page 667](#)

[Chapter 40, “Communicating with Java methods,” on page 707](#)

[Chapter 16, “Compiling, linking, and running OO applications,” on page 291](#)

IMS Application Programming Guide

Calling a COBOL method from a Java application under IMS

You can use the object-oriented language support in Enterprise COBOL to write COBOL methods that a Java program can call under IMS.

When you define a COBOL class and compile it using Enterprise COBOL, the compiler generates a Java class definition with native methods and the object code that implements those native methods. You can then create an instance and invoke the methods of this class from a Java program that runs in a Java dependent region, just as you would use any other class.

For example, you can define a COBOL class that uses the appropriate DL/I calls to access an IMS database. To make the implementation of this class available to a Java program, do the following steps:

1. Compile the COBOL class using Enterprise COBOL.

The compiler generates a Java source file (.java) that contains the class definition, and an object module (.o) that contains the implementation of the native methods.

2. Compile the generated Java source file using the Java compiler.
The Java compiler creates a class file (.class).
3. Link the object code into a dynamic link library (DLL) in the z/OS UNIX file system (.so). The directory that contains the COBOL DLLs must be listed in the LIBPATH, as specified in the IMS.PROCLIB member that is indicated by the ENVIRON= parameter of the IMS region procedure.
4. Update the sharable application class path in the main JVM options member (ibm.jvm.sharable.application.class.path in the IMS.PROCLIB member that is specified by the JVMOPMAS= parameter of the IMS region procedure) to enable the JVM to access the Java class file.

A Java program cannot call procedural COBOL programs directly. To reuse existing COBOL IMS code, use one of the following techniques:

- Restructure the COBOL code as a method in a COBOL class.
- Write a COBOL class definition and method that serves as a wrapper for the existing procedural code. The wrapper code can use COBOL CALL statements to access procedural COBOL programs.

Related tasks

[Chapter 16, “Compiling, linking, and running OO applications,” on page 291](#)

[“Structuring OO applications” on page 703](#)

[“Wrapping procedure-oriented COBOL programs” on page 703](#)

[IMS Application Programming Guide](#)

Building a mixed COBOL-Java application that starts with COBOL

An application that runs in a Java dependent region must start with the main method of a class.

A COBOL class definition that has a main factory method meets this requirement; therefore, you can use a main factory method as the first routine of a mixed COBOL and Java application under IMS.

Enterprise COBOL generates a Java class with a main method, which the Java dependent region can find, instantiate, and invoke. Although you can code the entire application in COBOL, you would probably build this type of application to call a Java routine. When the COBOL run time runs within the JVM of a Java dependent region, it automatically finds and uses this JVM to invoke methods on Java classes.

The COBOL application should use DL/I calls for processing messages (GU and GN) and synchronizing transactions (CHKP).

Related tasks

[“Structuring OO applications” on page 703](#)

[IMS Application Programming Guide](#)

[IBM SDK for Java - Tools Documentation](#)

Writing mixed-language IMS applications

When you write mixed-language IMS applications, you need to be aware of the effects of the STOP RUN statement. You also need to understand how to process messages and synchronize transactions, access databases, and use the application interface block (AIB).

Related tasks

[“Using the STOP RUN statement” on page 510](#)

[“Processing messages and synchronizing transactions” on page 511](#)

[“Accessing databases” on page 511](#)

[“Using the application interface block” on page 511](#)

Using the STOP RUN statement

If you use the STOP RUN statement in the COBOL portion of your application, the statement terminates all COBOL and Java routines (including the JVM).

Control is returned immediately to IMS. The program and the transaction are left in a stopped state.

Processing messages and synchronizing transactions

IMS message-processing applications must do all message processing and transaction synchronization either in COBOL or Java, rather than distributing this logic between application components written in both languages.

COBOL components use CALL statements to DL/I services to process messages (GU and GN) and synchronize transactions (CHKP). Java components use Java classes for IMS to do these functions. You can use object instances of classes derived from `IMSFieldMessage` to communicate entire IMS messages between the COBOL and Java components of the application.

Related tasks

IMS Application Programming Guide

Related references

IMS Application Programming API Reference

Accessing databases

You can use either Java, COBOL, or a mixture of the two languages to access IMS databases.

Limitation: EXEC SQL statements for Db2 database access are not supported in COBOL routines that run in a Java dependent region.

Recommendation: Do not access the same database program communication block (PCB) from both Java and COBOL. The Java and COBOL parts of the application share the same database position. Changes in database position from calls in one part of the application affect the database position in another part of the application. (This problem occurs whether the affected parts of an application are written in the same language or in different languages.)

Suppose that a Java component of a mixed application builds an SQL SELECT clause and uses Java Database Connectivity (JDBC) to query and retrieve results from an IMS database. The Java class libraries for IMS construct the appropriate request to IMS to establish the correct position in the database. If you then invoke a COBOL method that builds a segment search argument (SSA) and issues a GU (Get Unique) request to IMS against the same database PCB, the request probably altered the position in the database for that PCB. If so, subsequent JDBC requests to retrieve more records by using the initial SQL SELECT clause are incorrect because the database position changed. If you must access the same PCB from multiple languages, reestablish the database position after an interlanguage call before you access more records in the database.

Related tasks

IMS Application Programming Guide

Using the application interface block

COBOL applications that run in a Java dependent region normally must use the AIB interface because the Java dependent region does not provide PCB addresses to its application.

To use the AIB interface, specify the PCB requested for the call by placing the PCB name (which must be defined as part of the PSBGEN) in the resource name field of the AIB. (The AIB requires that all PCBs in a program specification block (PSB) definition have a name.) You do not specify the PCB address directly, and your application does not need to know the relative PCB position in the PCB list. Upon the completion of the call, the AIB returns the PCB address that corresponds to the PCB name that the application passed.

Alternatively, you can obtain PCB addresses by making an IMS INQY call using subfunction FIND, and the PCB name as the resource name. The call returns the address of the PCB, which you can then pass to a COBOL program. (This approach still requires that the PCB name be defined as part of the PSBGEN, but the application does not have to use the AIB interface.)

[“Example: using the application interface block” on page 512](#)

Related tasks

IMS Application Programming Guide

Example: using the application interface block

The following example shows how you can use the AIB interface in a COBOL application.

```
Local-storage section.  
  copy AIB.  
  .  
  .  
Linkage section.  
01 IOPCB.  
  05 logtterm      pic x(08).  
  05              pic x(02).  
  05 tpstat       pic x(02).  
  05 iodate       pic s9(7)   comp-3.  
  05 iotime       pic s9(7)   comp-3.  
  05              pic x(02).  
  05 seqnum       pic x(02).  
  05 mod          pic x(08).  
Procedure division.  
  Move spaces to input-area  
  Move spaces to AIB  
  Move "DFSAIB" to AIBRID  
  Move length of AIB to AIBRLEN  
  Move "IOPCB" to AIBRSNM1  
  Move length of input-area to AIBOALEN  
  Call "CEETDLI" using GU, AIB, input-area  
  Set address of IOPCB to AIBRESA1  
  If tpstat = spaces  
* . . process input message
```

Chapter 25. Running COBOL programs under z/OS UNIX

To run COBOL programs in the z/OS UNIX environment, compile them using Enterprise COBOL or COBOL for OS/390 & VM. The programs must be reentrant, so use the compiler and linker option RENT.

If you are going to run the programs from the z/OS UNIX file system, use the linker option AMODE 31. Any AMODE 24 program that you call from within a z/OS UNIX application must reside in an MVS PDSE.

Restrictions: The following restrictions apply to running under z/OS UNIX:

- SORT and MERGE statements are not supported.
- You cannot use the old COBOL interfaces for preinitialization (runtime option RTEREUS) to establish a reusable environment.
- You cannot run a COBOL program compiled with the NOTHREAD option 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 NOTHREAD 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 run a COBOL program in more than one thread if you compile all the COBOL programs in the application with the THREAD option.

You can use Debug Tool to debug z/OS UNIX programs in remote debug mode, for example, by using the Debug Perspective of IBM Developer for z/OS, or in full-screen mode (MFI) using a VTAM® terminal.

Related tasks

Chapter 15, “Compiling under z/OS UNIX,” on page 283

“Running OO applications under z/OS UNIX” on page 293

“Running in z/OS UNIX environments” on page 513

“Setting and accessing environment variables” on page 514

“Calling UNIX/POSIX APIs” on page 516

“Accessing main program parameters under z/OS UNIX” on page 518

Language Environment Programming Guide

Related references

“RENT” on page 391

Running in z/OS UNIX environments

You can run COBOL programs in any of the z/OS UNIX execution environments, either from within a z/OS UNIX shell or from outside a shell.

- You can run programs in either the OMVS shell (OMVS) or the ISPF shell (ISHELL).

Enter the program-name at the shell prompt. The program must be in the current directory or in your search path.

You can specify runtime options only by setting the environment variable _CEE_RUNOPTS before starting the program.

You can run programs that reside 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 a z/OS UNIX file, not to your terminal.

- From outside a shell, you can run programs either under TSO/E or in batch.

To call a COBOL program that resides in a z/OS UNIX file from the TSO/E prompt, use the BPXBATCH utility or a spawn() syscall in a REXX exec.

To call a COBOL program that resides in a z/OS UNIX file with the EXEC JCL statement, use the BPXBATCH utility.

Related tasks

[“Running OO applications under z/OS UNIX” on page 293](#)

[“Setting and accessing environment variables” on page 514](#)

[“Calling UNIX/POSIX APIs” on page 516](#)

[“Accessing main program parameters under z/OS UNIX” on page 518](#)

[“Defining and allocating QSAM files” on page 180](#)

[“Allocating line-sequential files” on page 216](#)

[“Allocating VSAM files” on page 208](#)

[“Displaying values on a screen or in a file \(DISPLAY\)” on page 35](#)

Language Environment Programming Guide (Running POSIX-enabled programs)

Related references

[“TEST” on page 404](#)

UNIX System Services User's Guide (The BPXBATCH utility)

Language Environment Programming Reference

Setting and accessing environment variables

You can set environment variables for z/OS UNIX COBOL programs either from the shell with commands `export` and `set`, or from the program.

Although setting and resetting environment variables from the shell before you begin to run a program is a typical procedure, you can set, reset, and access environment variables from the program while it is running.

If you are running a program with BPXBATCH, you can set environment variables by using an `STDENV DD` statement.

To reset an environment variable as if it had not been set, use the z/OS UNIX shell command `unset`. To reset an environment variable from a COBOL program, call the `setenv()` function.

To see the values of all environment variables, use the `export` command with no parameters. To access the value of an environment variable from a COBOL program, call the `getenv()` function.

[“Example: setting and accessing environment variables” on page 516](#)

Related tasks

[“Running in z/OS UNIX environments” on page 513](#)

[“Setting environment variables that affect execution” on page 515](#)

[“Accessing main program parameters under z/OS UNIX” on page 518](#)

[“Running OO applications under z/OS UNIX” on page 293](#)

[“Setting environment variables under z/OS UNIX” on page 283](#)

Related references

[“Runtime environment variables” on page 515](#)

Language Environment Programming Reference

MVS Program Management: User's Guide and Reference

Setting environment variables that affect execution

To set environment variables for z/OS UNIX COBOL programs from a shell, use the `export` or `set` command. To set environment variables from within the program, call POSIX functions `setenv()` or `putenv()`.

For example, to set the environment variable `MYFILE`:

```
export MYFILE=/usr/mystuff/notes.txt
```

[“Example: setting and accessing environment variables” on page 516](#)

Related tasks

[“Calling UNIX/POSIX APIs” on page 516](#)

[“Setting environment variables under z/OS UNIX” on page 283](#)

Related references

[“Runtime environment variables” on page 515](#)

Runtime environment variables

Several runtime variables are of interest for COBOL programs.

These are the runtime environment variables:

`_CEE_ENVFILE`

Specifies a file from which to read environment variables.

`_CEE_RUNOPTS`

Specifies runtime options.

`CLASSPATH`

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

`COBJVMINIOPTIONS`

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

`_IGZ_SYSOUT`

Specifies where to direct `DISPLAY` output. `stdout` and `stderr` are the only allowable values.

`LIBPATH`

Specifies directory paths of dynamic link libraries.

`PATH`

Specifies directory paths of executable programs.

`STEPLIB`

Specifies location of programs that are not in the `LNKLST`.

Related tasks

[“Displaying data on the system logical output device” on page 36](#)

Related references

XL C/C++ Programming Guide (`_CEE_ENVFILE`)

Language Environment Programming Reference

Example: setting and accessing environment variables

The following example shows how you can access and set environment variables from a COBOL program by calling the standard POSIX functions `getenv()` and `putenv()`.

Because `getenv()` and `putenv()` are C functions, you must pass arguments BY VALUE. Pass character strings as BY VALUE pointers that point to null-terminated strings. Compile programs that call these functions with the `NODYNAM` and `PGMNAME(LONGMIXED)` options.

```
CBL pgmname(longmixed),nodynam
Identification division.
Program-id. "envdemo".
Data division.
Working-storage section.
01 P pointer.
01 PATH pic x(5) value Z"PATH".
01 var-ptr pointer.
01 var-len pic 9(4) binary.
01 putenv-arg pic x(14) value Z"MYVAR=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 ABCDEFGH
  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 z/OS UNIX COBOL programs and from traditional z/OS COBOL programs by using the `CALL literal` statement. 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.

Restriction: You cannot use the `>>CALLINTERFACE DYNAM` directive with these APIs.

You can call the `fork()`, `exec()`, and `spawn()` functions from a COBOL program or from a non-COBOL program in the same process as COBOL programs. However, 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`. File status code 92 is returned if you attempt such 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 conditions are 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 a VS COBOL II program.
- With one exception, 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()` function is a main program, and all COBOL programs in the process start in initial state with all files closed.

Sample code for calling some of the POSIX routines is provided in the SIGYSAMP data set.

Purpose	Sample	Functions used
Shows how to use some of the file and directory routines	IGYTFL1	<ul style="list-style-type: none"> • <code>getcwd()</code> • <code>mkdir()</code> • <code>rmdir()</code> • <code>access()</code>
Shows how to use the <code>iconv</code> routines to convert data	IGYTENV	<ul style="list-style-type: none"> • <code>iconv_open()</code> • <code>iconv()</code> • <code>iconv_close()</code>
Shows the use of the <code>exec()</code> routine to run a new program along with other process-related routines	IGYTEXC, IGYTEXC1	<ul style="list-style-type: none"> • <code>fork()</code> • <code>getpid()</code> • <code>getppid()</code> • <code>execl()</code> • <code>perror()</code> • <code>wait()</code>
Shows how to get the <code>errno</code> value	IGYTERNO, IGYTGETE	<ul style="list-style-type: none"> • <code>perror()</code> • <code>fopen()</code>
Shows the use of the interprocess communication message routines	IGYTMSQ, IGYTMSQ2	<ul style="list-style-type: none"> • <code>ftok()</code> • <code>msgget()</code> • <code>msgsnd()</code> • <code>perror()</code> • <code>fopen()</code> • <code>fclose()</code> • <code>msgrcv()</code> • <code>msgctl()</code> • <code>perror()</code>

Related tasks

[“Running in z/OS UNIX environments” on page 513](#)

[“Setting and accessing environment variables” on page 514](#)

[“Accessing main program parameters under z/OS UNIX” on page 518](#)

Language Environment Programming Guide

Related references

XL C/C++ Run-Time Library Reference

UNIX System Services Programming: Assembler Callable Services Reference

Accessing main program parameters under z/OS UNIX

When you run a COBOL program from the z/OS UNIX shell command line or with an `exec()` or `spawn()` function, the parameter list consists of three parameters passed by reference. You can access these parameters with standard COBOL coding.

argument count

A binary fullword integer that contains the number of elements in each of the arrays that are passed in 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 that contains 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 in the `spawn()` or `exec()` function or in 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.

[“Example: accessing main program parameters under z/OS UNIX” on page 518](#)

Related tasks

[“Running in z/OS UNIX environments” on page 513](#)

[“Setting and accessing environment variables” on page 514](#)

[“Calling UNIX/POSIX APIs” on page 516](#)

[“Accessing main program parameters under z/OS” on page 564](#)

Example: accessing main program parameters under z/OS UNIX

The following example shows the three parameters that are passed by reference, and shows the coding that you can use to access them.

```
Identification division.
Program-id. "EXECED".
*****
* This sample program displays arguments received via exec() *
* function of z/OS UNIX *
*****
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 - 1)
End-Perform
Display "Display of arguments complete."
Goback.

```

- (1)** This count contains the number of elements in the arrays that are passed in the second and third parameters.
- (2)** This array contains a pointer to the length of the n th entry in the argument list.
- (3)** This array contains a pointer to the n th character string passed as an argument in the `spawn()` or `exec()` function or in the command invocation.

Chapter 26. Developing AMODE 64 programs

In general, the design considerations for AMODE 64 applications are the same as AMODE 31. With a few exceptions, the syntax and semantics of COBOL language features are the same between LP (64) and LP (32). See related tasks for the key differences and considerations.

Related tasks

[“Pointer data items with AMODE 64 programs” on page 521](#)

[“Compiler limits with AMODE 64 support” on page 522](#)

[“CALL statement with AMODE 64 support” on page 522](#)

[“Using compiler options to compile AMODE 64 programs” on page 522](#)

[“Other AMODE 64 considerations” on page 523](#)

Pointer data items with AMODE 64 programs

One fundamental difference between LP (64) and LP (32) is the size of address fields in the underlying generated program code. The storage allocation for some data items depends on the setting of the LP compiler option.

To access the larger addressing space, data items with USAGE POINTER, USAGE FUNCTION-POINTER, and USAGE INDEX are 8 bytes when LP (64) is in effect, and 4 bytes when LP (32) is in effect; data items with USAGE PROCEDURE-POINTER are 8-byte elementary items regardless of the LP compiler option setting.

LENGTH OF special register

The LENGTH OF special register has the following implicit definition with LP (64):

```
PICTURE 9(18) USAGE IS BINARY.
```

The corresponding implicit definition with LP (32) is:

```
PICTURE 9(9) USAGE IS BINARY.
```

This can affect the size of group items that contain these data items. If the group item is defined in a COBOL copybook, and if the copybook is used in both LP (64) and LP (32) compilations, you should be aware that the difference in size might affect the program logic.

USAGE POINTER-32 data item

A new data item with USAGE POINTER-32 is added to the compiler when introducing AMODE 64 support. This can be used with the LP (64) or LP (32) compiler option, and can be used to define the pointer data item that contains below the bar address regardless of whether the data item is used in an AMODE 64 or AMODE 31 program. The same copybook can then be shared between LP (64) and LP (32) programs.

Related tasks

[“LP” on page 373](#)

[FUNCTION-POINTER phrase \(*Enterprise COBOL for z/OS Language Reference*\)](#)

[INDEX phrase \(*Enterprise COBOL for z/OS Language Reference*\)](#)

[LENGTH OF \(*Enterprise COBOL for z/OS Language Reference*\)](#)

[POINTER phrase \(*Enterprise COBOL for z/OS Language Reference*\)](#)

[POINTER-32 phrase \(*Enterprise COBOL for z/OS Language Reference*\)](#)

[PROCEDURE-POINTER phrase \(*Enterprise COBOL for z/OS Language Reference*\)](#)

Compiler limits with AMODE 64 support

In general, the maximum size of tables and elementary alphanumeric data items in LP (64) is 2,147,483,646 bytes. See Compiler limits (*Enterprise COBOL for z/OS Language Reference*) for more details.

The LOCAL-STORAGE SECTION is allocated on the Language Environment stack. Its total size is limited by the settings in the Language Environment as well as the requirements of internal variables used by the compiler. The actual limit available for COBOL programs is less than 2,147,483,646 bytes.

The Language Environment STACK64 runtime option controls the allocation of stack storage for AMODE 64 applications. The default value is STACK64 (1M, 1M, 128M). Use this option to specify the maximum size of the stack required by your program.

The WORKING-STORAGE SECTION is allocated on the Language Environment heap. Its total size is limited by the 64-bit storage capacity of the machine.

Related tasks

[“LP” on page 373](#)

CALL statement with AMODE 64 support

The CALL statement transfers control from one object program to another within the run unit.

In addition to existing syntax rules, the following additional rules apply in LP (64):

1. Static and dynamic call supports calling other AMODE 64 Language Environment conforming programs.
2. AMODE 64 COBOL programs cannot be called by non-Language Environment conforming programs.

Assembler programs that LOAD and then branch to the entry point of the subprogram will not work. Instead, use the LE macro CEEFETCH to fetch and call AMODE 64 COBOL programs.

3. Parameter passing convention is XPLINK.

The only calling convention supported by Language Environment in AMODE 64 is XPLINK. Explicit setting of the XPLINK runtime option is not required.

Related tasks

CALL statement (*Enterprise COBOL for z/OS Language Reference*)

[“LP” on page 373](#)

Using compiler options to compile AMODE 64 programs

To compile a program to be AMODE 64, you must use the LP (64) compiler option. You should be aware that with LP (64) some compiler options or suboptions are discarded during compilation.

Note the following compiler options when you compile with LP (64):

DLL and RENT

The compiler generated code for LP (64) is reentrant and is enabled for DLL support. The LP (64) option implies RENT. If NORENT is specified, the compiler discards this option with a warning message. When LP (64) is in effect, the DLL and NODLL options are ignored, without warning, and NODLL will always appear in the listing. Object files produced by the compiler can be linked in DLL or non-DLL applications. When linking in DLL application, specify the DYNAM(DLL) and RENT Binder options. When linking in non-DLL application, specify the DYNAM(NO) and RENT options.

“RMODE” on page 392

The LP (64) option implies RMODE (ANY). If the RMODE (24) option is specified, the compiler discards the option with a warning message. AMODE 64 COBOL programs are loaded below the bar.

DATA

The DATA (24) option is discarded when LP (64) is in effect. Data items in WORKING-STORAGE SECTION are allocated above the bar. Data items in LOCAL section are allocated on the Language Environment stack, which is also above the bar.

AFP and HGPR

These two options control the compiler's register usage and are discarded by the compiler when LP (64) is in effect. The generated code is guaranteed to work in the Language Environment, and can call or be called by other Language Environment AMODE 64 high-level languages. User application cannot control register usage of the COBOL compiler.

"CICS" on page 342, "SQL" on page 398, SQLIMS, "SQLCCSID" on page 399, "THREAD" on page 408

These options are discarded in LP (64). In addition, the output of the SQL precompiler is not compatible with AMODE 64.

Runtime options

The following COBOL only runtime options are supported in LP (64):

- AIXBLD
- CBLQDA
- DEBUG
- SIMVRD
- UPSI

The following runtime options are not supported:

- ALL31

This option is meaningful only in AMODE 31.

- CBLOPTS
- CBLPSHPOP

This option applies only for OS VS program under CICS.

- MSGFILE

This option is not supported by the Language Environment in AMODE 64.

- RTEREUS

This option supports COBOL runtime reuse environment using a legacy design, and is not supported in LP (64). To establish a reusable runtime environment, use the LE preinitialization environment feature.

Related references

["LP" on page 373](#)

Other AMODE 64 considerations

When you compile the programs to be AMODE 64, you should be aware that some compiler features are not supported.

Note the following features when you compile programs with LP (64):

ALTER statement

The ALTER statement encourages the usage of unstructured programming practice. The EVALUATE statement provides the same functions and should be used instead.

GO TO. statement

Not supported in LP (64).

Object-oriented COBOL statements

Not supported in LP (64). Interoperability with JAVA programs can be done directly by using JAVA JNI interface.

XML PARSE and XML GENERATE statements

Not supported in LP (64).

JSON PARSE and JSON GENERATE statements

Not supported in LP (64).

POSIX runtime option

The LE runtime option POSIX(ON) and POSIX(OFF) are supported. Pure AMODE 64 COBOL by itself has no requirement on the POSIX option.

Tools and API libraries designed to operate in the contemporary open environment supporting LE tend to require POSIX(ON). It is strongly recommended that the POSIX(ON) option be used when running AMODE 64 COBOL applications.

This is especially true if the application contains programs of other programming languages, requires signal processing, or uses other API libraries. There might be additional requirements and restrictions on the POSIX option and options in general. It is the application's responsibility to set the options accordingly.

Subsystems

The COBOL compiler with LP (64) does not support the integrated CICS translator or the separate CICS translator, the IMS SQL coprocessor, and the Db2 precompiler.

Required APARs

The following APARs are required to compile or run AMODE 64 COBOL programs. Ensure that the appropriated level of LE PTF(s) for the following APARs has been installed on your system.

- PH10606, PH10740 and PH13196 (z/OS 2.2 Language Environment)
- PH07107, PH09543 add PH13761 (z/OS 2.3 Language Environment)

Note: Users can compile AMODE 64 COBOL programs on z/OS 2.2 or above, but must be on z/OS 2.3 or above to run AMODE 64 COBOL compilers.

Related references

[“LP” on page 373](#)

Summary of changes (*Enterprise COBOL for z/OS Language Reference*)

Chapter 27. Developing COBOL programs in the mixed AMODE Language Environment

This chapter describes developing programs in the mixed AMODE Language Environment.

COBOL programs in AMODE 31 interacting with Java programs in AMODE 64

This topic describes writing COBOL programs running in AMODE 31 to interact with Java programs in AMODE 64.

Note: This topic does not apply to COBOL programs that contain OO features. Examples are programs that contain INVOKE statements, OBJECT-REFERENCE data items, and class definitions. An exception is JNIEnvPtr, a COBOL special register that is used in OO programs but can be used by non-OO COBOL programs to make Java Native Interface (JNI) calls.

Application calls between COBOL programs in AMODE 31 and Java programs in AMODE 64

This section describes how to configure and enable COBOL/Java program calls in a mixed 31/64-bit environment.

Applications initiated from COBOL in AMODE 31 calling Java methods in AMODE 64

For applications that start from COBOL and need to make JNI calls, it is necessary to get the JNI environment pointer through the COBOL special register JNIEnvPtr. This special register can be used to establish addressability to the JNIEnv and JNINativeInterface structures defined in copybook JNI.cpy, which is also provided as member JNI in copybook data set SIGYLIB.

Since the COBOL program will start the Java Virtual Machine (JVM), you can set the environment variable *COBJVMINITOPTIONS* to the desired JVM initialization options. To leverage the mixed AMODE JVM support, you must specify the JVM option `-XX:+Enable3164Interoperability`.

For example:

```
export COBJVMINITOPTIONS="-XX:+Enable3164Interoperability -Djava.library.path=."
```

Also set the *CLASSPATH* environment variable appropriately.

Applications initiated from Java in AMODE 64 calling COBOL native methods in AMODE 31

For applications that start from Java, the JNI environment pointer is passed as a parameter to COBOL native programs, so there is no need to use the JNIEnvPtr special register. For coding convenience, the JNIEnvPtr special register is still available. The JVM option `XX:+Enable3164Interoperability` is required to invoke native functions (COBOL programs) in AMODE 31.

Programming considerations:

In the mixed 31/64-bit environment, references to Java types must be declared as PIC 9(18) COMP-5 in COBOL, that is, as an 8-byte data item. The OBJECT-REFERENCE type cannot be used as this data item maps to an incorrect 4-byte size in AMODE 31 COBOL programs. This applies to COBOL data items that are used to interact with JNI types such as jobject, jclass, jfieldID, jmethodID, jarray, jarray<type>, etc., when using them as parameters in JNI calls.

Note: Accesses to EXTERNAL data items and EXTERNAL files are not serialized.

Building the mixed AMODE COBOL/Java interoperability application

This section describes how to build the mixed AMODE COBOL/Java interoperability application.

It is recommended that you build and run COBOL JNI applications in z/OS UNIX.

Compiling

To compile the COBOL program, use the `cob2` command in a z/OS UNIX shell. Specify the `RENT`, `DLL` and `PGMNAME (LONGMIXED)` compiler options. The `THREAD` option is not mandatory, provided that there is only one COBOL program active in the run-unit at any time.

If the COBOL program includes the file `JNI.cpy` by using a `COPY` statement, specify the include subdirectory of the COBOL install directory, typically `/usr/lpp/cobol/include`, in the search order for copybooks. You can specify the include subdirectory by using the `-I` option of the `cob2` command or by setting the `SYSLIB` environment variable.

Linking

To prepare the COBOL program for execution, link the object file with the following two DLL side files to create an executable module:

- `libjvm31.x`, which is provided with your 64-bit IBM Java Software Development Kit: `$JAVA_HOME/lib/s390x/j9vm`.
- `igzjni2.x`, which is provided in the `lib` subdirectory of the COBOL install directory in the z/OS UNIX file system, typically `/usr/lpp/cobol/lib`. This DLL side file is also available as the member `IGZXJNI2` in the `SCEELIB` PDS.

Do not link with the sidedeck `igzcjava.x`. This is used only by OO features, which is not supported in the mixed AMODE environment. Linking both `igzjni2.x` and `igzcjava.x` in the same program can lead to ABENDs.

Note: `$JAVA_HOME` is the home directory of the 64-bit IBM Java Software Development Kit.

Running the mixed AMODE COBOL/Java interoperability program

This section lists the steps to run the program under z/OS UNIX.

To run the program under z/OS UNIX, set up your environment variables as follows:

- **STEPLIB:** Include the Language Environment datasets `SCEERUN` and `SCEERUN2` in the `STEPLIB` concatenation.

For example:

```
export STEPLIB=CEE.SCEERUN2:CEE.SCEERUN
```

or use the default Language Environment installed on your system.

- **Java environment variables:** set the `LIBPATH` and `CLASSPATH` environment variables. `$JAVAHOME` is the home directory of the installed Java JDK.

For example:

```
export LIBPATH=${$JAVAHOME}/lib/s390x/j9vm:./
export CLASSPATH=.
```

- **COBJVMINITOPTIONS:** Set this environment variable to Java JDK initialization options. The JVM option `-XX:+Enable3164Interoperability` is needed to initialize the mixed AMODE environment.

For example:

```
export COBJVMINITOPTIONS="-Djava.library.path=. -XX:+Enable3164Interoperability"
```

Examples for mixed AMODE COBOL/Java interoperability application

This section provides examples for mixed AMODE COBOL/Java interoperability application.

Below is an example of a COBOL program starting in AMODE 31 and calling the main method in Java in AMODE 64.

COBOL program example:

Compile and link the program below with cob2, where the program text is in source file prog.cbl.

```
cob2 -o prog.cbl -I/usr/lpp/cobol/include -q"dll" -q"pgmname(longmixed)" prog.cbl
/usr/lpp/cobol/lib/igzxjni2.x $JAVA_HOME/lib/s390x/j9vm/libjvm31.x
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. "PROG" recursive.
*
ENVIRONMENT DIVISION.
*
DATA DIVISION.
WORKING-STORAGE SECTION.

01 WS-CLASS-NAME      PIC X(50) VALUE LOW-VALUES.
01 WS-METHOD-NAME   PIC X(50).
01 WS-METHOD-SX     PIC X(50).

01 WS-MAIN-METHOD-ID PIC S9(18) COMP-5.
01 WS-CLASS-ID       PIC S9(18) COMP-5.
01 WS-NULL           PIC S9(18) COMP-5 VALUE ZERO.

01 WS-JNI-ENV        USAGE POINTER.
*
LINKAGE SECTION.

COPY JNI.

PROCEDURE DIVISION.
0010-MAIN.

    SET ADDRESS OF JNIEnv TO JNIEnvPtr.
    SET ADDRESS OF JNINativeInterface TO JNIEnv.

    DISPLAY "set class name".
    MOVE z"Greetings" TO WS-CLASS-NAME.
    CALL "__etoa" USING WS-CLASS-NAME.

* *****
* FIND CLASS
* *****
CALL FindClass USING BY VALUE
                JNIEnvPtr
                ADDRESS OF WS-CLASS-NAME
                RETURNING WS-CLASS-ID .

    IF WS-CLASS-ID = ZERO
        DISPLAY "CANT LOAD CLASS " WS-CLASS-NAME
        GOBACK
    END-IF.

* *****
* FIND METHOD (main)
* *****

    MOVE z"main" TO WS-METHOD-NAME.
    CALL "__etoa" USING WS-METHOD-NAME.

    MOVE z"([Ljava/lang/String;)V" TO WS-METHOD-SX.
    CALL "__etoa" USING WS-METHOD-SX.

    CALL GetStaticMethodId USING BY VALUE
                            JNIEnvPtr
                            WS-CLASS-ID
                            ADDRESS OF WS-METHOD-NAME
                            ADDRESS OF WS-METHOD-SX
                            RETURNING
                                WS-MAIN-METHOD-ID.

* *****
```

```

* CALL METHOD (main)
* *****
CALL CallStaticVoidMethod USING BY VALUE
                             JNIEnvPTR
                             WS-CLASS-ID
                             WS-MAIN-METHOD-ID.

GOBACK.
0100-INIT.
EXIT.

```

Java program example:

Create a Java Class Hello World instance, with static main method.

```

public class Greetings {
    public static void main(String[] args){
        System.out.println("Greetings from AMODE 64");
    }
}

```

Other considerations and restrictions

This topic lists considerations and restrictions when running the mixed AMODE COBOL/Java interoperability program.

The considerations for COBOL applications with mixed AMODE 31 and AMODE 64 programs also applies to interactions with Java programs. For interactions between AMODE 31 and AMODE 64 considerations and restrictions, see [“Other considerations and restrictions” on page 530](#).

For more information about Java 9 virtual machine reference, see [Using 31-bit native C or C++ code with the 64-bit Java VM \(z/OS only\)](#).

COBOL applications with mixed AMODE 31 and AMODE 64 programs

This topic describes COBOL applications with mixed AMODE 31 and AMODE 64 programs.

Dynamic call between AMODE 31 and AMODE 64 programs

Calls between AMODE 64 programs and AMODE 31 programs (when a subprogram has a different AMODE from the caller) can only be done via dynamic CALL. The caller can be AMODE 31 or 64. No indication in the CALL statement is needed. The dynamic call handler in the COBOL runtime determines the AMODE of the subprogram and processes accordingly.

The programs must be compiled by COBOL 6.3 or above. If the RETURNING phrase in the CALL statement is not used, the AMODE 31 program can be compiled with COBOL 5.1 and later releases. Non-COBOL programs are not supported by mixed AMODE 31/64 COBOL dynamic call.

The user program needs to make sure data items passed in parameters, directly or indirectly, are accessible by the subprogram. The DATA directive can be used to specify the storage location as required.

Parameter Passing

Both CALL ... USING BY REFERENCE and CALL ... USING BY CONTENT are supported.

CALL ... USING BY VALUE parameter is not supported.

If the CALL ... RETURNING phrase is used, the returning data item must not be larger than 24 bytes.

The RETURNING phrase supports for the following data items:

- alphanumeric

- zoned decimal
- COMP-3 (packed-decimal)
- COMP-4
- COMP-5
- COMP (BINARY)

Note: There is an extra copying required to return a data item across two AMODE settings. You can avoid the requirement by passing the data item with CALL ... USING BY REFERENCE. This method has better performance, especially for large data items.

Subprogram in dynamic call

When compiling the caller program, the DLL option is not needed. This is to facilitate mixed AMODE usage when the subprogram of an existing pure AMODE 31 application is recompiled with LP(64) while the caller program can remain in AMODE 31. The subprogram can be called by the caller program without recompiling the caller program with the DLL compiler option.

The subprogram must be a DLL.

To build DLL subprograms:

- When building a DLL, specify the compiler options EXPORTALL and DLL, and the binder options RENT and DYNAM=DLL.
- The program-id in the DLL program must be the same as the DLL module name. Otherwise the DLL cannot be called using dynamic call.

Note: As an effort to simplify DLL usage with LP(64), the DLL compiler option is not needed when building AMODE 64 programs. You can still specify it, there is no need to remove it in existing LP(64) compilations. The compiler would accept and ignore it. The EXPORTALL compiler option is still needed. The Binder option settings for DLL remain unchanged.

External file

The same external file can not be accessed by programs with different AMODE settings.

Related tasks

[“Making dynamic calls” on page 536](#)

Related references

DATA (*Enterprise COBOL for z/OS Language Reference*)

CALL statement (*Enterprise COBOL for z/OS Language Reference*)

COBOL runtime option AMODE3164

Specify the COBOL runtime option AMODE3164 when running mixed AMODE COBOL programs. The option can be specified in the JCL using the IGZOPTS DD statement.

For example:

```
//IGZOPTS DD *
        AMODE3164
/*
```

When running under z/OS UNIX, specify the option using the `_IGZ_RUNOPTS` environment variable. For example:

```
export _IGZ_RUNOPTS="AMODE3164"
```

For more information about how to specify COBOL runtime options, see [“Specifying COBOL runtime options” on page 422](#).

Related tasks

[“Dynamic call between AMODE 31 and AMODE 64 programs” on page 528](#)

Other considerations and restrictions

This topic lists AMODE 31 and AMODE 64 considerations and restrictions.

Performance

There is an extra overhead to transition between AMODE 31 and 64. Avoid crossing AMODE in performance sensitive areas and group programs that need to call each other frequently into the same AMODE.

In the CALL statement, use CALL . . . BY REFERENCE whenever possible. This requires less processing when the parameters are accessed in the other AMODE. Instead of using the RETURNING phrase, return the data item as a BY REFERENCE parameter. This avoids additional copying when the call is returned. CALL . . . BY VALUE is not supported.

Multi-threading

It is an existing limitation that the THREAD compiler option is not supported for COBOL programs compiled with LP(64) (running in AMODE 64). This limitation is extended to COBOL programs compiled with LP(32) (running in AMODE 31) when the mixed AMODE 31/64 feature is used. This means that when the LE runtime environment contains both AMODE 31 and AMODE 64 programs within any LE enclave, only one thread can contain COBOL programs.

Note: AMODE 31 and AMODE 64 programs are running in different LE enclaves in the mixed LE environment.

Condition handling

The condition handling support does not extend across AMODE 31/64 boundary. When processing a condition, Language Environment does not traverse stack frames to find a condition handler until an AMODE 31 and AMODE 64 boundary is reached. If a handler is not found at this point, the condition is treated as unhandled. Applications that rely on condition handling might need to be modified.

Additional information

For more information about AMODE 31/64 inter operation, see [Introduction to AMODE 31 and AMODE 64 programs interoperability](#) on Language Environment Vendor Interface for AMODE 31 and AMODE 64 Interoperability.

Related tasks

[“Dynamic call between AMODE 31 and AMODE 64 programs” on page 528](#)

Part 4. Structuring complex applications

Chapter 28. Using subprograms

Many applications consist of several separately compiled programs linked together. A *run unit* (the COBOL term that is synonymous with the Language Environment term *enclave*) includes one or more object programs and can include object programs written in other Language Environment member languages.

Language Environment provides interlanguage support that lets your Enterprise COBOL programs call and be called by programs that meet the requirements of Language Environment.

Name prefix alert: Do not use program-names that start with prefixes used by IBM products. If you use programs whose names start with such prefixes, CALL statements might resolve to IBM library or compiler routines rather than to the intended program. For a list of prefixes to avoid, see the related task about identifying a program.

Related concepts

[“Main programs, subprograms, and calls” on page 533](#)

Related tasks

[“Identifying a program” on page 3](#)

[“Ending and reentering main programs or subprograms” on page 534](#)

[“Transferring control to another program” on page 535](#)

[“Making recursive calls” on page 545](#)

[“Calling to and from object-oriented programs” on page 545](#)

[“Using procedure and function pointers” on page 545](#)

[“Making programs reentrant” on page 548](#)

[“Handling COBOL limitations with multithreading” on page 582](#)

Language Environment Writing ILC Communication Applications

Related references

Language Environment Programming Guide (Register conventions)

Main programs, subprograms, and calls

If a COBOL program is the first program in a run unit, that COBOL program is the *main program*. Otherwise, it and all other COBOL programs in the run unit are *subprograms*. No specific source-code statements or options identify a COBOL program as a main program or subprogram.

Whether a COBOL program is a main program or subprogram can be significant for either of two reasons:

- Effect of program termination statements
- State of the program when it is reentered after returning

In the PROCEDURE DIVISION, a program can call another program (generally called a *subprogram*), and this called program can itself call other programs. The program that calls another program is referred to as the *calling* program, and the program it calls is referred to as the *called* program. When the processing of the called program is completed, the called program can either transfer control back to the calling program or end the run unit.

The called COBOL program starts running at the top of the PROCEDURE DIVISION.

Related tasks

[“Ending and reentering main programs or subprograms” on page 534](#)

[“Transferring control to another program” on page 535](#)

[“Making recursive calls” on page 545](#)

Related references

Language Environment Programming Guide

Ending and reentering main programs or subprograms

Whether a program is left in its last-used state or its initial state, and to which caller it returns, can depend on the termination statements that you use.

You can use any of three termination statements in a program, but they have different effects as shown in the following table.

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. In a threaded environment, the thread is not terminated unless the program is the first (oldest) one in the thread.
STOP RUN	Return to calling program. ¹ (Might be the operating system, and application will end.) STOP RUN terminates the run unit (also called an enclave in Language Environment terms), and deletes all dynamically called programs in the run unit and all programs link-edited with them. (It does not delete the main program.) In a threaded environment, the entire Language Environment enclave is terminated, including all threads running within the enclave. In a CICS environment, the entire transaction is terminated, including all programs running within the transaction.	Return directly to the program that called the main program. ¹ (Might be the operating system, and application 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.) In a threaded environment, the entire Language Environment enclave is terminated, including all threads running within the enclave. In a CICS environment, the entire transaction is terminated, including all programs running within the transaction.
GOBACK	Return to calling program. ¹ (Might be the operating system, and application will end.) GOBACK 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.) In a threaded environment, the thread is terminated. ²	Return to calling program. In a threaded environment, if the program is the first program in a thread, the thread is terminated. ²
<p>1. If the main program is called by a program written in another language that does not follow Language Environment linkage conventions, return is to this calling program.</p> <p>2. If the thread is the initial thread of execution in an enclave, the enclave is terminated.</p>		

A subprogram is usually left in its *last-used state* when it terminates with EXIT PROGRAM or GOBACK. The next time the subprogram is called in the run unit, its internal values are as they were left, except that return points for PERFORMed paragraphs and sections are reset to their initial values. (In contrast, a main program is initialized each time it is called.)

There are some cases in which programs will be in their initial state:

- A subprogram that is dynamically called and then canceled will be in the initial state the next time it is called.
- A program that has the INITIAL clause in the PROGRAM-ID paragraph, or is compiled with the INITIAL compiler option, will be in the initial state each time it is called.
- Data items defined in the LOCAL-STORAGE SECTION will be reset to the initial state specified by their VALUE clauses each time the program is called.

Related concepts

[“Comparison of WORKING-STORAGE and LOCAL-STORAGE” on page 14](#)

Language Environment Programming Guide (What happens during termination: thread termination)

Related tasks

[“Calling nested COBOL programs” on page 542](#)

[“Making recursive calls” on page 545](#)

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, and calls to dynamic link libraries (DLLs).

In addition to making calls between Enterprise COBOL programs, you can also make static and dynamic calls between Enterprise COBOL and programs compiled with older compilers in all environments including CICS.

For restrictions about making calls with older levels of programs, see *Interoperability with older levels of IBM COBOL programs* in the *Enterprise COBOL for z/OS Migration Guide*.

Calling nested programs lets you 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, z/OS UNIX programs, and applications that interoperate with C/C++.

Under z/OS, linking two program objects 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 binder (linkage-editor) control statements as either a NAME or an ALIAS of the program object that contains the subprogram.

Related concepts

[“AMODE switching” on page 538](#)

[“Performance considerations of static and dynamic calls” on page 540](#)

[“Nested programs” on page 543](#)

Related tasks

[“Making static calls” on page 536](#)

[“Making dynamic calls” on page 536](#)

[“Making both static and dynamic calls” on page 540](#)

[“Calling nested COBOL programs” on page 542](#)

Related references

Enterprise COBOL for z/OS Migration Guide

(Interoperability with older levels of IBM COBOL programs)

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 CALL *literal* calls are handled as static calls.

With static calls statement, the COBOL program and all called programs are part of the same program object. 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 their initial state each 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.

[“Examples: static and dynamic CALL statements” on page 541](#)

Related concepts

[“Performance considerations of static and dynamic calls” on page 540](#)

Related tasks

[“Making dynamic calls” on page 536](#)

[“Making both static and dynamic calls” on page 540](#)

[“Calling to and from object-oriented programs” on page 545](#)

Related references

[“DLL” on page 353](#)

[“DYNAM” on page 356](#)

CALL statement (*Enterprise COBOL for z/OS Language Reference*)

Making dynamic calls

When you use a 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.

In these forms of the CALL statement, the called COBOL subprogram is not link-edited with the main program. Instead, it is link-edited into a separate program object, and is loaded at run time only when it is required (that is, when called). The program-name in the PROGRAM-ID paragraph or ENTRY statement must be identical to the corresponding program object name or program object alias of the program object that contains the program.

Each subprogram that you call with a dynamic CALL statement can be part of a different program object that is a member of either the system link library or a private library that you supply. In either case it must be in an MVS load library; it cannot reside in the z/OS UNIX 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 that contains 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 you call the same COBOL program in different run units, a separate copy of WORKING-STORAGE is allocated for each run unit.

Restrictions: You cannot make dynamic calls 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 was executed)

[“Examples: static and dynamic CALL statements” on page 541](#)

Related concepts

[“When to use a dynamic call with subprograms” on page 537](#)

[“Performance considerations of static and dynamic calls” on page 540](#)

Related tasks

[“Canceling a subprogram” on page 537](#)

[“Making static calls” on page 536](#)

[“Making both static and dynamic calls” on page 540](#)

Related references

[“DLL” on page 353](#)

[“DYNAM” on page 356](#)

ENTRY statement (*Enterprise COBOL for z/OS Language Reference*)

CALL statement (*Enterprise COBOL for z/OS Language Reference*)

Language Environment Programming Reference

Canceling a subprogram

When you issue a CANCEL statement for a subprogram, the storage that is occupied by the subprogram is freed. 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 that an intervening CANCEL statement is executed before you specify different entry points in a dynamic CALL statement to that subprogram.

After a CANCEL statement is processed for a dynamically called contained program, the program will be in its first-used state. However, the program is not loaded with the initial call, and storage is not freed after the program is canceled.

[“Examples: static and dynamic CALL statements” on page 541](#)

Related concepts

[“Performance considerations of static and dynamic calls” on page 540](#)

When to use a dynamic call with subprograms

Your decision to use dynamic calls with subprograms depends on factors such as location of the program object, frequency of calls to the subprograms, size of the subprograms, ease of maintenance, the need to call subprograms in their unused state, the need for AMODE switching, and when the program-names are known.

The program object that you want to dynamically call must be in an MVS load library rather than in the z/OS UNIX file system.

If subprograms are called in only a few conditions, you can use dynamic calls to bring in the subprograms 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.

If you are concerned about ease of maintenance, dynamic calls can help. Applications do not have to be link-edited again when dynamically called subprograms are changed.

When you cannot use the INITIAL attribute to ensure that a subprogram is placed in its unused state each time that it is called, you can set the unused state by using a combination of dynamic CALL and CANCEL statements. When you cancel a subprogram that was first called by a COBOL program, the next call causes 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.

Suppose you have an AMODE 24 program in the same run unit with Enterprise COBOL programs that you want to run in 31-bit addressing mode. COBOL dynamic call processing includes AMODE switching for AMODE 24 programs that call AMODE 31 programs, and vice versa. To have this implicit AMODE switching done, the Language Environment runtime options ALL31(OFF) and STACK(, , BELOW) must be in effect.

When dynamic call 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.

If you do not know the program-name to be called until run time, use the format CALL *identifier*, where *identifier* is a data item that will contain the name of the called program at run time. For example, you could use CALL *identifier* when the program to be called varies depending on conditional processing in your program. CALL *identifier* is always dynamic, even if you use the NODYNAM compiler option.

[“Examples: static and dynamic CALL statements” on page 541](#)

Related concepts

[“AMODE switching” on page 538](#)

[“Performance considerations of static and dynamic calls” on page 540](#)

Related tasks

[“Making dynamic calls” on page 536](#)

Related references

[“DYNAM” on page 356](#)

CALL statement (*Enterprise COBOL for z/OS Language Reference*)
Language Environment Programming Reference

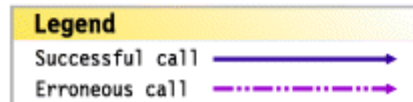
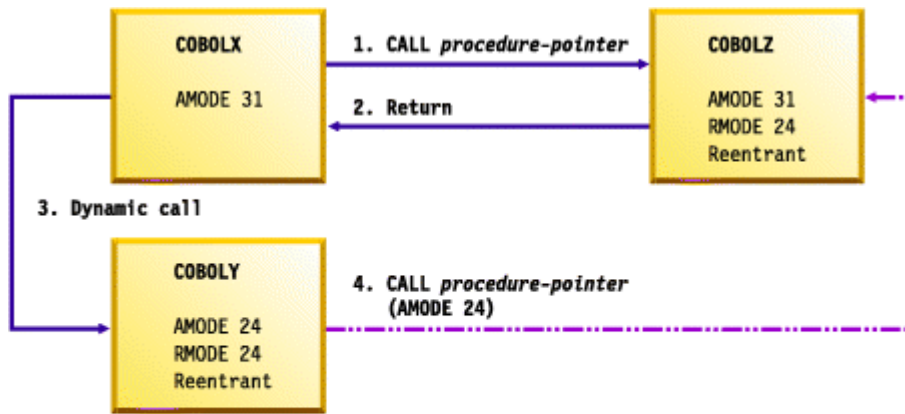
AMODE switching

When you have an application that has COBOL subprograms, some of the COBOL subprograms can be AMODE 31 and some can be AMODE 24. To have this mixed AMODE support, the calls must be dynamic and the Language Environment runtime options ALL31(OFF) and STACK(, , BELOW) must be in effect.

If your application consists of only COBOL programs, and you are using dynamic calls, each COBOL subprogram will always be entered in the proper AMODE. For example, if you are using a dynamic call from an AMODE 31 COBOL program to an AMODE 24 COBOL program, the AMODE is automatically switched.

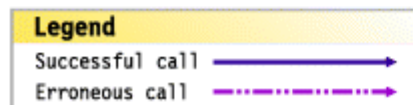
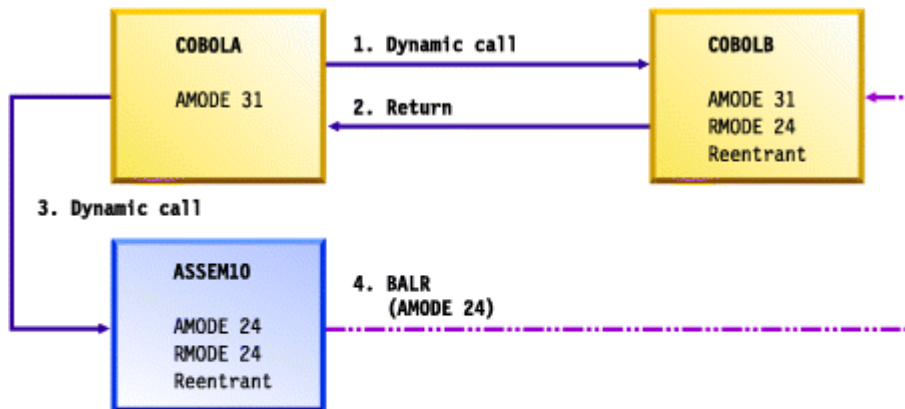
However, if you are using procedure pointers, function pointers, or other languages that call COBOL subprograms, you must ensure that when a COBOL program is called more than once in an enclave, it is entered in the same AMODE each time that it is called. The AMODE is not automatically switched in this case.

The following scenario shows that AMODE problems can arise when procedure pointers are used to call COBOL subprograms. This scenario is not supported because the COBOL program COBOLY is not entered in the same AMODE each time that it is called.



1. COBOLX is AMODE 31. It uses the SET statement to set a procedure pointer to COBOLZ. COBOLZ is a reentrant program object and is AMODE 31 and RMODE 24. COBOLX calls COBOLZ using the procedure pointer. COBOLZ is entered in AMODE 31.
2. COBOLZ returns to COBOLX.
3. COBOLX dynamically calls COBOLY, passing the procedure pointer for COBOLZ. COBOLY is a reentrant program object, and is AMODE 24 and RMODE 24. COBOLY is entered in AMODE 24.
4. COBOLY calls COBOLZ using the procedure pointer. This call causes COBOLZ to be entered in AMODE 24, which is not the same AMODE in which COBOLZ was entered when it was called the first time.

The following scenario uses a mix of COBOL and assembler language. This scenario is not supported because the COBOL program COBOLB is not entered in the same AMODE each time that it is called.



1. COBOLA is AMODE 31. COBOLA dynamically calls COBOLB. COBOLB is a reentrant program object and is AMODE 31 and RMODE 24. COBOLB is entered in AMODE 31.
2. COBOLB returns to COBOLA.
3. COBOLA dynamically calls ASSEM10, which is in assembler language. ASSEM10 is a reentrant program object, and is AMODE 24 and RMODE 24. ASSEM10 is entered in AMODE 24.
4. ASSEM10 loads COBOLB. ASSEM10 does a BALR instruction to COBOLB. COBOLB is entered in AMODE 24, which is not the same AMODE in which COBOLB was entered when it was called the first time.

Related concepts

[“Storage and its addressability” on page 39](#)

[“When to use a dynamic call with subprograms” on page 537](#)

Related tasks

[“Making dynamic calls” on page 536](#)

Related references

Language Environment Programming Reference (ALL31)

Performance considerations of static and dynamic calls

Because a statically called program is link-edited into the same program object 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.

Note: Static calls between programs compiled with the same or similar COBOL versions perform better than those compiled with mixed COBOL versions. For example, static calls between COBOL 6 programs are faster than static calls between COBOL 6 and 4 programs. In mixed cases such as COBOL 6 and 4 programs, dynamic calls perform better than static calls because of the extra processing needed to maintain backward compatibility.

For the purpose of this discussion, the following COBOL versions are considered similar:

- COBOL 4.2 and prior releases
- COBOL 5.1 and later releases

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 concepts

[“When to use a dynamic call with subprograms” on page 537](#)

Related tasks

[“Making static calls” on page 536](#)

[“Making dynamic calls” on page 536](#)

Making both static and dynamic calls

You can use 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 program object. The CALL *identifier* statement results in the dynamic invocation of a separate program object.

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

[“DYNAM” on page 356](#)

Examples: static and dynamic CALL statements

This example shows how you can code static and dynamic calls.

The 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 static calls:

```
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 dynamic calls:

```
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 executed.

(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 executed 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 after 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, you must save them.

Calling nested COBOL programs

By calling nested programs, you can create applications that use structured programming techniques. You can also call nested programs instead of PERFORM procedures to prevent unintentional modification of data items.

Use either CALL *literal* or CALL *identifier* statements to make calls to nested programs.

You can call a contained program only from its directly containing program unless you identify the contained program as COMMON in its PROGRAM- ID paragraph. 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:

- Code an IDENTIFICATION DIVISION in each program. All other divisions are optional.
- Optionally make the name of each contained program unique. Although the names of contained programs are not required to be unique (as described in the related reference about scope of names), making the names unique could help make your application more maintainable. You can use any valid user-defined word or an alphanumeric literal as the name of a contained program.
- In the outermost program, code any CONFIGURATION SECTION entries that might be required. Contained programs cannot have a CONFIGURATION SECTION.

- Include each contained program in the containing program immediately before the END PROGRAM marker of the containing program.
- Use an END PROGRAM marker to terminate contained and containing programs.

You cannot use the THREAD option when compiling programs that contain nested programs.

Related concepts

[“Nested programs” on page 543](#)

Related references

[“Scope of names” on page 544](#)

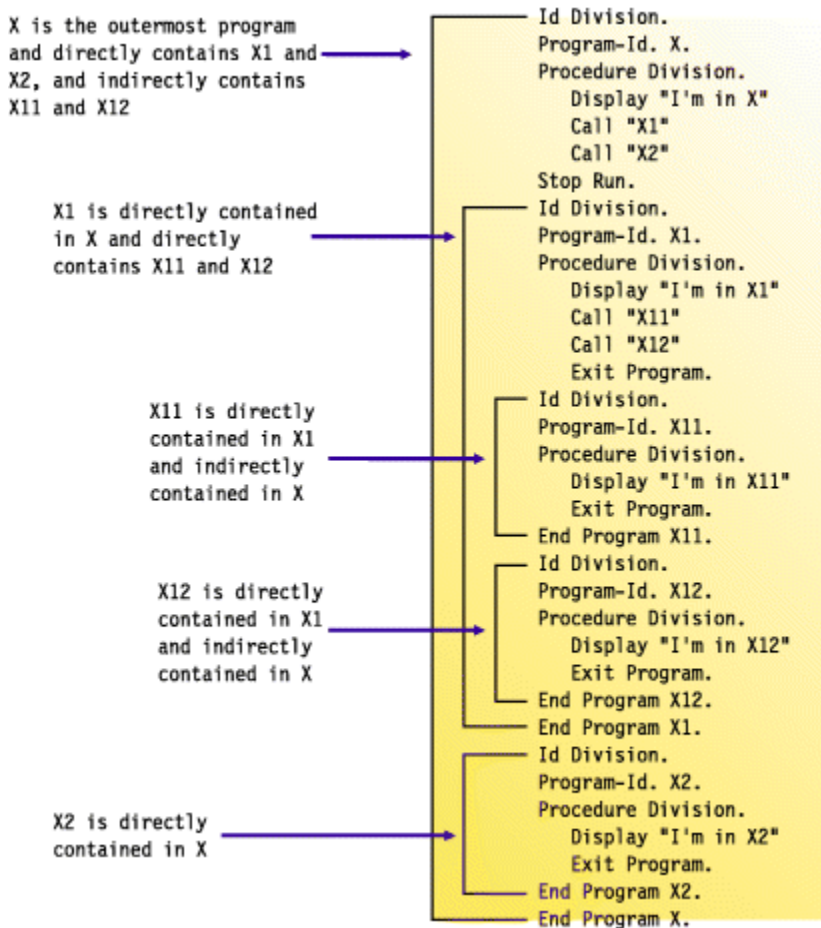
Nested programs

A COBOL program can *nest*, or contain, other COBOL programs. The nested programs can themselves contain other programs. A nested program can be directly or indirectly contained in a program.

There are four main advantages to nesting called programs:

- Nested programs provide a method for creating modular functions and maintaining structured programming techniques. They can be used analogously to perform procedures (using the PERFORM statement), but with more structured control flow and with the ability to protect local data items.
- Nested programs let you debug a program before including it in an application.
- Nested programs enable you to compile an application with a single invocation of the compiler.
- Calls to nested programs have the best performance of all the forms of COBOL CALL statements.

The following example describes a nested structure that has directly and indirectly contained programs:



[“Example: structure of nested programs” on page 544](#)

Related tasks

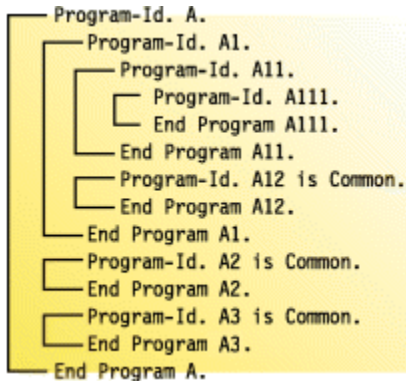
[“Calling nested COBOL programs” on page 542](#)

Related references

[“Scope of names” on page 544](#)

Example: structure of nested programs

The following example shows a nested structure with some contained programs that are identified as COMMON.



The following table describes the calling hierarchy for the structure that is shown in the example above. Programs A12, A2, and A3 are identified as COMMON, and the calls associated with them differ.

This program	Can call these programs	And can be called by these programs
A	A1, A2, A3	None
A1	A11, A12, A2, A3	A
A11	A111, A12, A2, A3	A1
A111	A12, A2, A3	A11
A12	A2, A3	A1, A11, A111
A2	A3	A, A1, A11, A111, A12, A3
A3	A2	A, A1, A11, A111, A12, A2

In this example, note that:

- A2 cannot call A1 because A1 is not common and is not contained in A2.
- A1 can call A2 because A2 is common.

Scope of names

Names in nested structures are divided into two classes: local and global. The class determines whether a name is known beyond the scope of the program that declares it. A specific search sequence locates the declaration of a name after it is referenced in a program.

Local names

Names (except the program-name) are local unless declared to be otherwise. Local names are visible or accessible only within the program in which they are declared. They are not visible or accessible to contained and containing programs.

Global names

A name that is global (indicated by using the GLOBAL clause) is visible and accessible to the program in which it is declared and to all the programs that are directly and indirectly contained in that program. Therefore, the contained programs can share common data and files from the containing program simply by referencing the names of the items.

Any item that is subordinate to a global item (including condition-names and indexes) is automatically global.

You can declare the same name with the GLOBAL clause more than one time, provided that each declaration occurs in a different program. Be aware that you can mask, or hide, a name in a nested structure by having the same name occur in different programs in the same containing structure. However, such masking could cause problems during a search for a name declaration.

Searches for name declarations

When a name is referenced in a program, a search is made to locate the declaration for that name. The search begins in the program that contains the reference and continues outward to the containing programs until a match is found. The search follows this process:

1. Declarations in the program are searched.
2. If no match is found, only global declarations are searched in successive outer containing programs.
3. The search ends when the first matching name is found. If no match is found, an error exists.

The search is for a global name, not for a particular type of object associated with the name such as a data item or file connector. The search stops when any match is found, regardless of the type of object. If the object declared is of a different type than that expected, an error condition exists.

Making recursive calls

A called program can directly or indirectly execute its caller. For example, program X calls program Y, program Y calls program Z, and program Z then calls program X. This type of call is *recursive*.

To make a recursive call, you must code the RECURSIVE clause in the PROGRAM-ID paragraph of the recursively called program. If you try to recursively call a COBOL program that does not have the RECURSIVE clause in the PROGRAM-ID paragraph, a condition is signaled. If the condition remains unhandled, the run unit will end.

Related tasks

[“Identifying a program as recursive” on page 4](#)

Related references

PROGRAM-ID paragraph (*Enterprise COBOL for z/OS Language Reference*)

Calling to and from object-oriented programs

When you create applications that contain object-oriented (OO) programs, the OO COBOL programs are DLL programs and can be in one or more dynamic link libraries (DLLs). Each class definition must be in a separate DLL, however.

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 means to ensure that the DLL linkage mechanism is followed are available.

Using procedure and function pointers

You can set procedure-pointer and function-pointer data items only by using format 6 of the SET statement.

Procedure pointers are data items defined with the USAGE IS PROCEDURE-POINTER clause. *Function pointers* are data items defined with the USAGE IS FUNCTION-POINTER clause. In this information, "pointer" refers to either a procedure-pointer data item or a function-pointer data item. You can set either of these data items to contain entry addresses of, or pointers to, these entry points:

- Another COBOL program that is not nested. 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, so that the routine is 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 statement. It will return a pointer that you can either use as a function pointer or 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).

The SET statement sets the pointer to refer either to an entry point in the same program object as your program, to a separate program object, or to an entry point that is exported from a DLL, depending on the DYNAM | NODYNAM and DLL | NODLL compiler options. Therefore, consider these factors when using these pointer data items:

- If you compile a program with the NODYNAM and NODLL options and set a 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 program object. Otherwise the reference cannot be resolved.
- If you compile a program with the NODLL option and either set a pointer item to an identifier that will contain the name of the entry point at run time or set the pointer item to a literal and compile with the DYNAM option, then the pointer item, whether a literal or variable, must point to an entry point in a separate program objectlink. The entry point can be either the primary entry point or an alternate entry point named in an ALIAS binder (linkage-editor) statement.
- If you compile with the NODYNAM and DLL options and set a pointer item to a literal value (the actual name of an entry point), the value must refer to an entry point in the same program object or to an entry-point name that is exported from a DLL module. In the latter case you must include the DLL side file for the target DLL module in the link-edit of your program object.
- If you compile with the NODYNAM and DLL options and set a 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 a pointer item to an entry address in a dynamically called program object, and your program subsequently cancels that dynamically called module, then that pointer item becomes undefined. Reference to it thereafter is not reliable.

Procedure pointer and function pointer calls are supported for AMODE 24 applications. However, the addressing mode cannot be switched for these calls, so the called and calling programs must have the same addressing mode at execution time.

COBOL entry points with the AMODE ANY attribute can be entered in either AMODE 31 or AMODE 24. However, the AMODE value that is in effect when the program is entered for the first time must also be in effect for all subsequent reentries of the program during the current Language Environment enclave.

For COBOL 5 and later, procedure and function pointers point to a function descriptor rather than directly to the entry point. If you have a data-only module, a table for example, you can no longer use the technique:

```
77 DATA-FUNCTION-PTR USAGE FUNCTION-POINTER.  
77 DATA-PTR REDEFINES DATA-FUNCTION-PTR USAGE POINTER.  
  
SET DATA-FUNCTION-PTR TO ENTRY "DATAONLY"  
SET ADDRESS OF DATA TO DATA_PTR
```

However, you need to do the following change:

```
77 DATA-PTR USAGE POINTER.
```

```
CALL "DATAONLY" RETURNING DATA-PTR  
SET ADDRESS OF DATA TO DATA-PTR
```

with the data-only module revised to return the address of the data, and below is the LP(32) example:

```
DATAONLY CSECT  
        USING *,15  
        LA 15,DATA  
        BR 14  
*  
DATA DC ...  
...  
END
```

The revised method also works with earlier COBOLs.

Calls to procedure and function pointers must be from a module with a Language Environment stack frame, as will be the case for any high-level programming language. If such a call is to be made from an assembler module, an LE stack frame must be provided by using the CEEENTRY and CEETERM macros, along with the associated register content requirements.

Related tasks

[“Deciding which type of pointer to use” on page 547](#)

[“Calling alternate entry points” on page 548](#)

[“Using procedure or function pointers with DLLs” on page 572](#)

Related references

[“DLL” on page 353](#)

[“DYNAM” on page 356](#)

CANCEL statement (*Enterprise COBOL for z/OS Language Reference*)

Format 6: SET for procedure-pointer and function-pointer data items
(*Enterprise COBOL for z/OS Language Reference*)

ENTRY statement (*Enterprise COBOL for z/OS Language Reference*)

[MVS Program Management: User's Guide and Reference](#)

Deciding which type of pointer to use

Use procedure pointers to call other COBOL programs and to call Language Environment callable services. Use function pointers to communicate with C/C++ programs or with services provided by the Java Native Interface.

Procedure pointers are more efficient than function pointers for COBOL-to-COBOL calls, and are required for calls to Language Environment condition-handling services.

Many callable services written in C return function pointers. You can call such a C function pointer from your COBOL program by using COBOL function pointers as shown below.

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. DEMO.  
ENVIRONMENT DIVISION.  
DATA DIVISION.  
*  
WORKING-STORAGE SECTION.  
01 FP USAGE FUNCTION-POINTER.  
*  
PROCEDURE DIVISION.  
CALL "c-function" RETURNING FP.  
CALL FP.
```

Related tasks

[“Using procedure or function pointers with DLLs” on page 572](#)

[“Accessing JNI services” on page 707](#)

Calling alternate entry points

You can specify another entry point at which a program will begin running by using the ENTRY label in the called program.

Static calls to alternate entry points work without restriction.

Dynamic calls to alternate entry points require either explicitly specified NAME and/or ALIAS binder (linkage-editor) control statements, or use of the NAME compiler option, which generates them automatically.

Dynamically calling a compile unit at more than one entry point, including the primary entry point will produce a unique instance of WORKING-STORAGE for each such entry, as if separate compile units were being called.

If you share one WORKING-STORAGE instance among several entry points without statically linking the compile unit with its caller, you must make it a DLL and use EXPORTALL to expose the secondary entry points, and call the compile unit using a DLL call rather than a dynamic call.

Dynamic call and procedure-pointer call to alternate entry points must be run under AMODE 31. These calls are not supported for programs compiled with LP(64).

[“Examples: static and dynamic CALL statements” on page 541](#)

Related references

[“NAME” on page 377](#)

CANCEL statement (*Enterprise COBOL for z/OS Language Reference*)

ENTRY statement (*Enterprise COBOL for z/OS Language Reference*)

MVS Program Management: User's Guide and 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 a program that resides in the link pack area), you must make the program *reentrant* by compiling with the RENT option.

You do not need to worry about multiple copies of variables. The compiler creates the necessary reentrancy controls in the object module.

The following Enterprise COBOL 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 z/OS UNIX environment
- Programs that are enabled for DLL support
- Programs that use object-oriented syntax

For reentrant programs, use the DATA compiler option and the HEAP and ALL31 runtime options to control whether dynamic data areas, such as WORKING-STORAGE, are obtained from storage below or above the 16 MB line.

Related concepts

[“Storage and its addressability” on page 39](#)

Related tasks

[“Compiling programs to create](#)

[DLLs” on page 568](#)

[Chapter 16, “Compiling, linking, and running OO applications,” on page 291](#)

Related references

[“RENT” on page 391](#)

[“DATA” on page 349](#)

Language Environment Programming Reference (ALL31, HEAP)

Chapter 29. Sharing data

If a run unit consists of several separately compiled programs that call each other, the programs must be able to communicate with each other. They also usually need access to common data.

This information describes how you can write programs that share data with other programs. In this information, a *subprogram* is any program that is called by another program.

Related tasks

[“Using data from another program” on page 16](#)

[“Sharing data with Java” on page 711](#)

[“Passing data” on page 551](#)

[“Coding the LINKAGE SECTION” on page 555](#)

[“Coding the PROCEDURE DIVISION for passing arguments” on page 556](#)

[“Passing return-code information” on page 559](#)

[“Sharing data by using the EXTERNAL clause” on page 560](#)

[“Sharing files between programs \(external files\)” on page 561](#)

[“Accessing main program parameters under z/OS” on page 564](#)

Passing data

You can choose among three ways of passing data between programs: BY REFERENCE, BY CONTENT, or BY VALUE.

BY REFERENCE

The subprogram refers to and processes the data items in the storage of the calling program rather than working on a copy of the data. BY REFERENCE is the assumed passing mechanism for a parameter if none of the three ways is specified or implied for the parameter.

BY CONTENT

The calling program passes only the contents of the *literal* or *identifier*. The called program cannot change the value of the *literal* or *identifier* in the calling program, even if it modifies the data item in which it received the *literal* or *identifier*.

BY VALUE

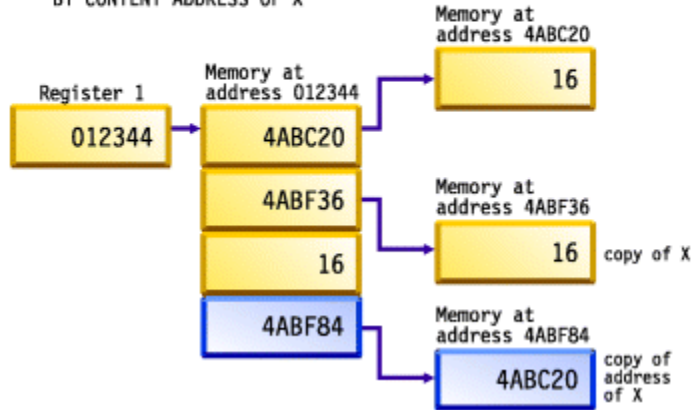
The calling program or method passes the value of the *literal* or *identifier*, not a reference to the sending data item. The called program or invoked method can change the parameter. However, because the subprogram or method has access only to a temporary copy of the sending data item, any change does not affect the argument in the calling program.

The following figure shows the differences in values passed BY REFERENCE, BY CONTENT, and BY VALUE:

```

MOVE 16 TO X.
CALL ABC USING
  BY REFERENCE X
  BY CONTENT X
  BY VALUE X
  BY CONTENT ADDRESS OF X

```



Determine which of these 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.
CALL . . . BY REFERENCE ADDRESS OF <i>identifier</i>	To pass the address of <i>identifier</i> to a called program, where <i>identifier</i> is an item in the LINKAGE SECTION	Any changes made by the subprogram to the address affect the address in the calling program.
CALL . . . BY REFERENCE <i>file-name</i>	To pass a data control block (DCB) to assembler programs	The file-name must reference a QSAM sequential file. ¹
CALL . . . BY CONTENT ADDRESS OF <i>identifier</i>	To pass a copy of the address of <i>identifier</i> to a called program	Any changes to the copy of the address will not affect the address of <i>identifier</i> , but changes to <i>identifier</i> using the copy of the address will cause changes to <i>identifier</i> .
CALL . . . BY CONTENT <i>identifier</i>	To pass a copy of the identifier to the subprogram	Changes to the parameter by the subprogram will not affect the caller's identifier.
CALL . . . BY CONTENT <i>literal</i>	To pass a copy of a literal value to a called program	
CALL . . . BY CONTENT LENGTH OF <i>identifier</i>	To pass a copy of the length of a data item	The calling program passes the length of the <i>identifier</i> from its LENGTH special register.

Table 71. **Methods for passing data in the CALL statement** (continued)

Code	Purpose	Comments
A combination of BY REFERENCE and BY CONTENT such as: CALL 'ERRPROC' USING BY REFERENCE A BY CONTENT LENGTH OF A.	To pass both a data item and a copy of its length to a subprogram	
CALL . . . BY VALUE <i>identifier</i>	To pass data to a program, such as a C/C++ program, that uses BY VALUE parameter linkage conventions	A copy of the identifier is passed directly in the parameter list.
CALL . . . BY VALUE <i>literal</i>	To pass data to a program, such as a C/C++ program, that uses BY VALUE parameter linkage conventions	A copy of the literal is passed directly in the parameter list.
CALL . . . BY VALUE ADDRESS OF <i>identifier</i>	To pass the address of <i>identifier</i> to a called program. This is the recommended way to pass data to a C/C++ program that expects a pointer to the data.	Any changes to the copy of the address will not affect the address of <i>identifier</i> , but changes to <i>identifier</i> using the copy of the address will cause changes to <i>identifier</i> .
CALL . . . RETURNING	To call a C/C++ function with a function return value	
<p>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 are not supported by IBM.</p>		

Related concepts

[“Storage and its addressability” on page 39](#)

Related tasks

[“Describing arguments in the calling program” on page 553](#)

[“Describing parameters in the called program” on page 554](#)

[“Testing for OMITTED arguments” on page 555](#)

[“Specifying CALL . . . RETURNING” on page 560](#)

[“Sharing data by using the EXTERNAL clause” on page 560](#)

[“Sharing files between programs \(external files\)” on page 561](#)

[“Sharing data with Java” on page 711](#)

Related references

CALL statement (*Enterprise COBOL for z/OS Language Reference*)

The USING phrase (*Enterprise COBOL for z/OS Language Reference*)

INVOKE statement (*Enterprise COBOL for z/OS Language Reference*)

Describing arguments in the calling program

In the calling program, describe arguments in the DATA DIVISION in the same manner as other data items in the DATA DIVISION.

Storage for arguments is allocated only in the outermost program. For example, program A calls program B, which calls program C. Data items are allocated in program A. They are described in the LINKAGE SECTION of programs B and C, making the one set of data available to all three programs.

If you reference data in a file, the file must be open when the data is referenced.

Code the USING phrase of the CALL statement to pass the arguments. If you pass a data item BY VALUE, it must be an elementary item.

To pass CALL arguments from AMODE 31 programs to AMODE 24 programs, you must ensure that the arguments are in storage below the 16 MB line to be addressed by the AMODE 24 subprogram.

- For reentrant AMODE 31 programs, compile the program with the DATA(24) option, or specify the Language Environment runtime option HEAP(, , BELOW) if WORKING-STORAGE is allocated from HEAP storage. For more information about when WORKING-STORAGE is allocated from HEAP storage, see [“Storage and its addressability” on page 39](#).
- For nonreentrant programs that are compiled with the NORENT option, compile with the RMODE(24) or RMODE(AUTO) option. Consequently, the following items are allocated below the 16 MB line, and can be passed as arguments to AMODE 24 programs:
 - WORKING-STORAGE data items without the EXTERNAL clause
 - FD record areas
 - QSAM buffers
- For mixed AMODE applications, the Language Environment runtime options ALL31(OFF) and STACK(, , BELOW) are required. Consequently, the LOCAL-STORAGE SECTION data items and data items with the EXTERNAL attributes will be allocated below the 16 MB line, and can be passed as arguments to AMODE 24 programs.

Related concepts

[“Storage and its addressability” on page 39](#)

Related tasks

[“Coding the LINKAGE SECTION” on page 555](#)

[“Coding the PROCEDURE DIVISION for passing arguments” on page 556](#)

Related references

The USING phrase (*Enterprise COBOL for z/OS Language Reference*)

Describing parameters in the called program

You must know what data is being passed from the calling program and describe it in the LINKAGE SECTION of each program that is called directly or indirectly by the calling program.

Code the USING phrase after the PROCEDURE DIVISION header to name the parameters that receive the data that is passed from the calling program.

When arguments are passed to the subprogram BY REFERENCE, it is invalid for the subprogram to specify any relationship between its parameters and any fields other than those that are passed and defined in the main program. The subprogram must not:

- Define a parameter to be larger in total number of bytes than the corresponding argument.
- Use subscript references to refer to elements beyond the limits of tables that are passed as arguments by the calling program.
- Use reference modification to access data beyond the length of defined parameters.
- Manipulate the address of a parameter in order to access other data items that are defined in the calling program.

If any of the rules above are violated, unexpected results might occur.

Related tasks

[“Coding the LINKAGE SECTION” on page 555](#)

Related references

The USING phrase (*Enterprise COBOL for z/OS Language Reference*)

Testing for OMITTED arguments

You can specify that one or more BY REFERENCE arguments are not to be passed to a called program by coding the OMITTED keyword in place of those arguments in the CALL statement.

For example, to omit the second argument when calling program sub1, code this statement:

```
Call 'sub1' Using PARM1, OMITTED, PARM3
```

The arguments in the USING phrase of the CALL statement must match the parameters of the called program in number and position.

In a called program, you can test whether an argument was passed as OMITTED by comparing the address of the corresponding parameter to NULL. For example:

```
Program-ID. sub1.  
.  
.  
Procedure Division Using RPARAM1, RPARAM2, RPARAM3.  
  If Address Of RPARAM2 = Null Then  
    Display 'No 2nd argument was passed this time'  
  Else  
    Perform Process-Param-2  
  End-If
```

Related references

CALL statement (*Enterprise COBOL for z/OS Language Reference*)

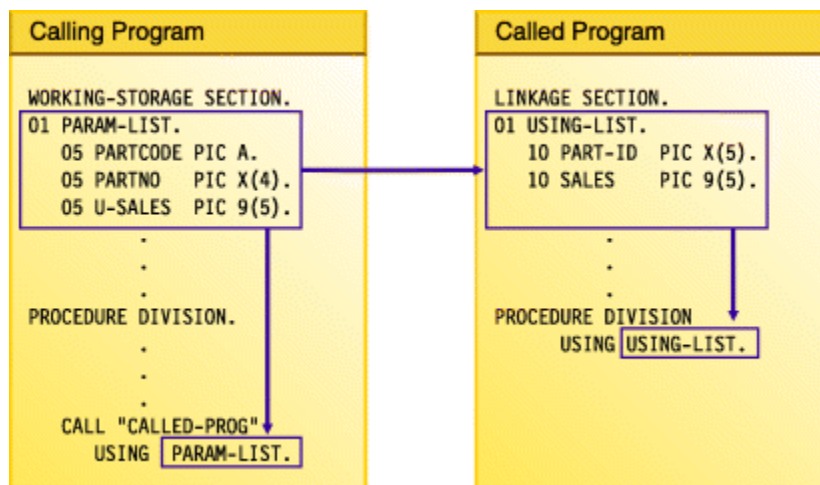
The USING phrase (*Enterprise COBOL for z/OS Language Reference*)

Coding the LINKAGE SECTION

Code the same number of data-names in the identifier list of the called program as the number of arguments in the calling program. Synchronize by position, because the compiler passes the first argument from the calling program to the first identifier of the called program, and so on.

You will introduce errors if the number of data-names in the identifier list of a called program is greater than the number of arguments passed from the calling program. The compiler does not try to match arguments and parameters.

The following figure shows a data item being passed from one program to another (implicitly BY REFERENCE):



In the calling program, the code for parts (PARTCODE) and the part number (PARTNO) are distinct data items. In the called program, by contrast, the code for parts and the part number are combined into one data item (PART-ID). In the called program, a reference to PART-ID is the only valid reference to these items.

Related tasks

[“Accessing main program parameters under z/OS” on page 564](#)

Coding the PROCEDURE DIVISION for passing arguments

If you pass an argument BY VALUE, code the USING BY VALUE clause in the PROCEDURE DIVISION header of the subprogram. If you pass an argument BY REFERENCE or BY CONTENT, you do not need to indicate in the header how the argument was passed.

```
PROCEDURE DIVISION USING BY VALUE. . .
PROCEDURE DIVISION USING. . .
PROCEDURE DIVISION USING BY REFERENCE. . .
```

The first header above indicates that the data items are passed BY VALUE; the second or third headers indicate that the items are passed BY REFERENCE or BY CONTENT.

Related references

The procedure division header (*Enterprise COBOL for z/OS Language Reference*)

The USING phrase (*Enterprise COBOL for z/OS Language Reference*)

CALL statement (*Enterprise COBOL for z/OS Language Reference*)

Grouping data to be passed

Consider grouping all the data items that you need to pass between programs and putting them under one level-01 item. If you do so, you can pass a single level-01 record.

Note that if you pass a data item BY VALUE, it must be an elementary item.

To lessen the possibility of mismatched records, put the level-01 record into a copy library and copy it into both programs. That is, copy it in the WORKING-STORAGE SECTION of the calling program and in the LINKAGE SECTION of the called program.

Related tasks

[“Coding the LINKAGE SECTION” on page 555](#)

Related references

CALL statement (*Enterprise COBOL for z/OS Language Reference*)

Handling null-terminated strings

COBOL supports null-terminated strings when you use string-handling statements together with null-terminated literals and the hexadecimal literal X'00'.

You can manipulate null-terminated strings (passed from a C program, for example) by using string-handling mechanisms such as those in the following code:

```
01 L          pic X(20) value z'ab'.
01 M          pic X(20) value z'cd'.
01 N          pic X(20).
01 N-Length  pic 99      value zero.
01 Y          pic X(13) value 'Hello, World!'.
```

To determine the length of a null-terminated string, and display the value of the string and its length, code:

```
Inspect N tallying N-length for characters before initial X'00'
Display 'N: ' N(1:N-length) ' Length: ' N-length
```


To move a null-terminated string to an alphanumeric string, but delete the null, code:

```
Unstring N delimited by X'00' into X
```

To create a null-terminated string, code:

```
String Y delimited by size  
X'00' delimited by size  
into N.
```

To concatenate two null-terminated strings, code:

```
String L delimited by x'00'  
M delimited by x'00'  
X'00' delimited by size  
into N.
```

Related tasks

[“Manipulating null-terminated strings” on page 108](#)

Related references

Null-terminated alphanumeric literals

(*Enterprise COBOL for z/OS Language Reference*)

Using pointers to process a chained list

When you need to pass and receive addresses of record areas, you can use pointer data items, which are either data items that are defined with the `USAGE IS POINTER` clause or are `ADDRESS OF` special registers.

A typical application for using pointer data items is in processing a *chained list*, a series of records in which each record points to the next.

When you pass addresses between programs in a chained list, you can use `NULL` to assign the value of an address that is not valid (nonnumeric 0) to a pointer item in either of two ways:

- Use a `VALUE IS NULL` clause in its data definition.
- Use `NULL` as the sending field in a `SET` statement.

In the case of a chained list in which the pointer data item in the last record contains a null value, you can use this code to check for the end of the list:

```
IF PTR-NEXT-REC = NULL  
  (logic for end of chain)
```

If the program has not reached the end of the list, the program can process the record and move on to the next record.

The data passed from a calling program might contain header information that you want to ignore. Because pointer data items are not numeric, you cannot directly perform arithmetic on them. However, to bypass header information, you can use the `SET` statement to increment the passed address.

[“Example: using pointers to process a chained list” on page 558](#)

Related tasks

[“Coding the LINKAGE SECTION” on page 555](#)

[“Coding the PROCEDURE DIVISION for passing arguments” on page 556](#)

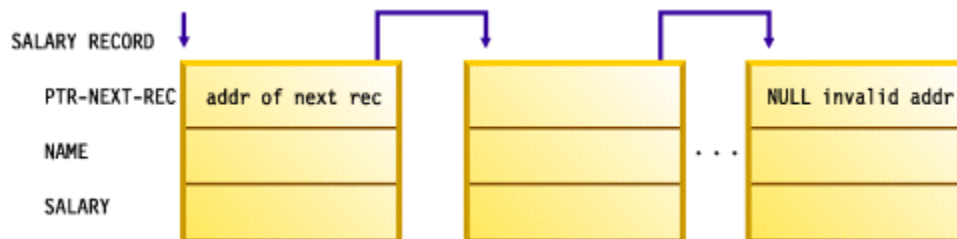
Related references

`SET` statement (*Enterprise COBOL for z/OS Language Reference*)

Example: using pointers to process a chained list

The following example shows how you might process a linked list, that is, a chained list of data items.

For this example, picture a chained list of data that consists of individual salary records. The following figure shows one way to visualize how the records are linked in storage. The first item in each record except the last points to the next record. The first item in the last record contains a null value (instead of a valid address) to indicate that it is the last record.



The high-level pseudocode for an application that processes these records might be:

```
Obtain address of first record in chained list from routine
Check for end of the list
Do until end of the list
    Process record
    Traverse to the next record
End
```

The following code contains an outline of the calling program, `LISTS`, used in this example of processing a chained list.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. LISTS.
ENVIRONMENT DIVISION.
DATA DIVISION.
*****
WORKING-STORAGE SECTION.
77 PTR-FIRST          POINTER VALUE IS NULL.           (1)
77 DEPT-TOTAL        PIC 9(4) VALUE IS 0.
*****
LINKAGE SECTION.
01 SALARY-REC.                                           (2)
   02 PTR-NEXT-REC   POINTER.
   02 NAME           PIC X(20).
   02 DEPT           PIC 9(4).
   02 SALARY         PIC 9(6).
01 DEPT-X           PIC 9(4).
*****
PROCEDURE DIVISION USING DEPT-X.
*****
* FOR EVERYONE IN THE DEPARTMENT RECEIVED AS DEPT-X,
* GO THROUGH ALL THE RECORDS IN THE CHAINED LIST BASED ON THE
* ADDRESS OBTAINED FROM THE PROGRAM CHAIN-ANCH
* AND ACCUMULATE THE SALARIES.
* IN EACH RECORD, PTR-NEXT-REC IS A POINTER TO THE NEXT RECORD
* IN THE LIST; IN THE LAST RECORD, PTR-NEXT-REC IS NULL.
* DISPLAY THE TOTAL.
*****
   CALL "CHAIN-ANCH" USING PTR-FIRST                       (3)
   SET ADDRESS OF SALARY-REC TO PTR-FIRST                   (4)
*****
   PERFORM WITH TEST BEFORE UNTIL ADDRESS OF SALARY-REC = NULL (5)

   IF DEPT = DEPT-X
      THEN ADD SALARY TO DEPT-TOTAL
      ELSE CONTINUE
   END-IF
   SET ADDRESS OF SALARY-REC TO PTR-NEXT-REC               (6)

END-PERFORM
*****
DISPLAY DEPT-TOTAL
GOBACK.
```

- (1) PTR-FIRST is defined as a pointer data item with an initial value of NULL. On a successful return from the call to CHAIN-ANCH, PTR-FIRST contains the address of the first record in the chained list. If something goes wrong with the call, and PTR-FIRST never receives the value of the address of the first record in the chain, a null value remains in PTR-FIRST and, according to the logic of the program, the records will not be processed.
- (2) The LINKAGE SECTION of the calling program contains the description of the records in the chained list. It also contains the description of the department code that is passed in the USING clause of the CALL statement.
- (3) To obtain the address of the first SALARY-REC record area, the LISTS program calls the program CHAIN-ANCH.
- (4) The SET statement bases the record description SALARY-REC on the address contained in PTR-FIRST.
- (5) The chained list in this example is set up so that the last record contains an address that is not valid. This check for the end of the chained list is accomplished with a do-while structure where the value NULL is assigned to the pointer data item in the last record.
- (6) The address of the record in the LINKAGE-SECTION is set equal to the address of the next record by means of the pointer data item sent as the first field in SALARY-REC. The record-processing routine repeats, processing the next record in the chained list.

To increment addresses received from another program, you could set up the LINKAGE SECTION and PROCEDURE DIVISION like this:

```
LINKAGE SECTION.
01 RECORD-A.
   02 HEADER          PIC X(12).
   02 REAL-SALARY-REC PIC X(30).
.
.
.
01 SALARY-REC.
   02 PTR-NEXT-REC    POINTER.
   02 NAME            PIC X(20).
   02 DEPT            PIC 9(4).
   02 SALARY          PIC 9(6).
.
.
.
PROCEDURE DIVISION USING DEPT-X.
.
.
.
SET ADDRESS OF SALARY-REC TO ADDRESS OF REAL-SALARY-REC
```

The address of SALARY-REC is now based on the address of REAL-SALARY-REC, or RECORD-A + 12.

Related tasks

[“Using pointers to process a chained list” on page 557](#)

Passing return-code information

Use the RETURN-CODE special register to pass return codes between programs. (Methods do not return information in the RETURN-CODE special register, but they can check the register after a call to a program.)

You can also use the RETURNING phrase in the PROCEDURE DIVISION header of a method to return information to an invoking program or method. If you use PROCEDURE DIVISION . . . RETURNING with CALL . . . RETURNING, the RETURN-CODE register will not be set.

Using 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 special register 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, the RETURN-CODE special register of the COBOL program might be updated with an invalid value. Unless you set this special register to a meaningful value before your Enterprise COBOL program returns to the operating system, a return code that is invalid will be passed to the system.

For equivalent function between COBOL and C programs, have your COBOL program call the C program with the RETURNING phrase. If the C program (function) correctly declares a function value, the RETURNING value of the calling COBOL program will be set.

You cannot set the RETURN-CODE special register by using the INVOKE statement.

Using PROCEDURE DIVISION RETURNING . . .

Use the RETURNING phrase in the PROCEDURE DIVISION header of a program to return information to the calling program.

```
PROCEDURE DIVISION RETURNING dataname2
```

When the called program in the example above successfully returns to its caller, the value in *dataname2* is stored into the identifier that was specified in the RETURNING phrase of the CALL statement:

```
CALL . . . RETURNING dataname2
```

CEEPIPI: The results of specifying PROCEDURE DIVISION RETURNING in programs that are called with the Language Environment preinitialization service (CEEPIPI) are undefined.

Specifying CALL . . . RETURNING

You can specify the RETURNING phrase of the CALL statement for calls to C/C++ functions or to COBOL subroutines.

The RETURNING phrase has the following format.

```
CALL . . . RETURNING dataname2
```

The return value of the called program is stored into *dataname2*. You must define *dataname2* in the DATA DIVISION of the calling program. The data type of the return value that is declared in the target function must be identical to the data type of *dataname2*.

Sharing data by using the EXTERNAL clause

Use the EXTERNAL clause to enable separately compiled programs and methods (including programs in a batch sequence) to share data items. Code EXTERNAL in the level-01 data description in the WORKING-STORAGE SECTION.

The following rules apply:

- Items that are subordinate to an EXTERNAL group item are themselves EXTERNAL.

- You cannot use the name of an EXTERNAL data item as the name for another EXTERNAL item in the same program.
- You cannot code the VALUE clause for any group item or subordinate item that is EXTERNAL.

In the run unit, any COBOL program or method that has the same data description for the item as the program that contains the item can access and process that item. For example, suppose program A has the following data description:

```
01 EXT-ITEM1      EXTERNAL      PIC 99.
```

Program B can access that data item if B has the identical data description in its WORKING-STORAGE SECTION.

Any program that has access to an EXTERNAL data item can change the value of that item. Therefore do not use this clause for data items that you need to protect.

Sharing files between programs (external files)

To enable separately compiled programs or methods in a run unit to access a file as a common file, use the EXTERNAL clause for the file.

It is recommended that you follow these guidelines:

- Use the same data-name in the FILE STATUS clause of all the programs that check the file status code.
- For each program that checks the same file status field, code the EXTERNAL clause in the level-01 data definition for the file status field.

Using an external file has these benefits:

- Even if the main program does not contain any input or output statements, it can reference the record area of the file.
- Each subprogram can control a single input or output function, such as OPEN or READ.
- Each program has access to the file.

[“Example: using external files” on page 561](#)

Related tasks

[“Using data in input and output operations” on page 11](#)

Related references

EXTERNAL clause (*Enterprise COBOL for z/OS Language Reference*)

Example: using external files

The following example shows the use of an external file in several programs. COPY statements ensure that each subprogram contains an identical description of the file.

The following table describes the main program and subprograms.

Name	Function
ef1	The main program, which calls all the subprograms and then verifies the contents of a record area
ef1openo	Opens the external file for output and checks the file status code
ef1write	Writes a record to the external file and checks the file status code
ef1openi	Opens the external file for input and checks the file status code
ef1read	Reads a record from the external file and checks the file status code

Name	Function
ef1close	Closes the external file and checks the file status code

Each program uses three copybooks:

- efselect is placed in the FILE-CONTROL paragraph:

```
Select ef1
Assign To ef1
File Status Is efs1
Organization Is Sequential.
```

- effile is placed in the FILE SECTION:

```
Fd ef1 Is External
      Record Contains 80 Characters
      Recording Mode F.
01  ef-record-1.
   02 ef-item-1 Pic X(80).
```

- efwrkstg is placed in the WORKING-STORAGE SECTION:

```
01  efs1          Pic 99 External.
```

Input/output using external files

```
IDENTIFICATION DIVISION.
Program-Id.
  ef1.
*
* This main program controls external file processing.
*
ENVIRONMENT DIVISION.
Input-Output Section.
File-Control.
  Copy efselect.
DATA DIVISION.
FILE SECTION.
  Copy effile.
WORKING-STORAGE SECTION.
  Copy efwrkstg.
PROCEDURE DIVISION.
  Call "eflopeno"
  Call "ef1write"
  Call "ef1close"
  Call "eflopeni"
  Call "ef1read"
  If ef-record-1 = "First record" Then
    Display "First record correct"
  Else
    Display "First record incorrect"
    Display "Expected: " "First record"
    Display "Found   : " ef-record-1
  End-If
  Call "ef1close"
  Goback.
End Program ef1.
IDENTIFICATION DIVISION.
Program-Id.
  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 ef1
  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 ef1
  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 efile.
WORKING-STORAGE SECTION.
    Copy efwrkstg.
PROCEDURE DIVISION.
    Close ef1
    If efs1 Not = 0
        Display "file status " efs1 " on close"
        Stop Run
    End-If
    Goback.
End Program eflclose.

```

Accessing main program parameters under z/OS

When you run an Enterprise COBOL program under z/OS and pass the program a parameter string, for example, by using JCL or a TSO command, the parameter list consists of a character string that has a halfword prefix that contains the string length.

You can access the parameter string by using a LINKAGE SECTION and standard COBOL coding as shown in the example referenced below:

[“Example: accessing main program parameters under z/OS” on page 564](#)

Alternatively, you can obtain the parameter string by calling either of the following Language Environment callable services, which are described in the Related references below:

- CEE3PRM (query parameter string): obtain the parameter string (if not longer than 80 characters)
- CEE3PR2 (query parameter string long): obtain the parameter string and its length

In either case, the parameter string might contain program arguments, runtime options, or both. The setting of the CBLOPTS runtime option determines the relative order in which program arguments and runtime options are expected. If CBLOPTS(ON) (the default) is in effect, and program arguments and runtime options are both passed in the parameter string, they must appear in the following order, separated by a forward slash:

```
program_arguments/runtime_options
```

For further details, see the related information referenced below.

Related tasks

[“Coding the LINKAGE SECTION” on page 555](#)

[“Accessing main program parameters under z/OS UNIX” on page 518](#)

Appendix I, “Accessing JCL parameters,” on page 955

Language Environment Programming Guide (Specifying runtime options and program arguments, Preparing your main routine to receive parameters)

Related references

Language Environment Customization (CBLOPTS (COBOL only))

Language Environment Programming Reference (CEE3PRM, CEE3PR2)

Example: accessing main program parameters under z/OS

The following example shows how to receive a parameter string that is passed to a COBOL program that runs under z/OS, and shows the coding that you can use to access the parameter string.

```
IDENTIFICATION DIVISION.
```



```

PROGRAM-ID. "testarg".
*
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
*
DATA DIVISION.
WORKING-STORAGE SECTION.
*
linkage section.
01  os-parm.
    05 parm-len          pic s999 comp.
    05 parm-string.
        10 parm-char     pic x occurs 0 to 100 times
                        depending on parm-len.
*
PROCEDURE DIVISION using os-parm.
    display "parm-len=" parm-len
    display "parm-string=" parm-string " "
    evaluate parm-string
        when "01" display "case one"
        when "02" display "case two"
        when "95" display "case ninety-five"
        when other display "case unknown"
    end-evaluate
GOBACK.

```

Suppose that the CBLOPTS (ON) runtime option is in effect, and that you pass the following argument in the JCL or TSO command that you use to run the program:

```
'95/'
```

Then the resulting output is:

```

parm-len=002
parm-string='95'
case ninety-five

```

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 program object or the application relate to each other or to other DLLs
- Deciding what linking or calling mechanisms to use

Depending on whether you want to create a DLL program object or a program object that references a separate DLL, you need to use slightly different compiler and binder (linkage-editor) options.

Related concepts

[“Dynamic link libraries \(DLLs\)” on page 567](#)

Related tasks

[“Creating a DLL under z/OS UNIX” on page 286](#)

[“Compiling programs to create](#)

[DLLs” on page 568](#)

[“Linking DLLs” on page 569](#)

[“Using CALL identifier with DLLs” on page 570](#)

[“Using DLL linkage and dynamic calls together” on page 571](#)

[“Using COBOL DLLs with C/C++ programs” on page 574](#)

[“Using DLLs in OO COBOL](#)

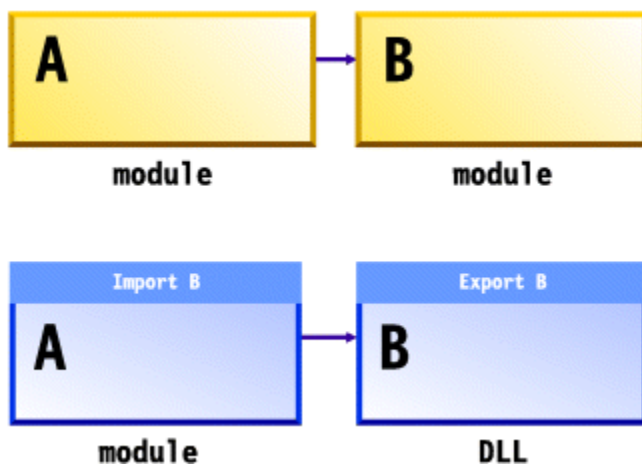
[applications” on page 575](#)

[“Using procedure or function pointers with DLLs” on page 572](#)

Dynamic link libraries (DLLs)

A DLL is a program object that can be accessed from other separate program objects.

A DLL differs from a traditional program object 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 program object 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 program object with a dynamic CALL.



A DLL application is an application that references imported definitions of programs, functions, or variables.

64-bit COBOL supports both DLL and non-DLL applications. The DLL compilation option is not needed as the object file generated by the compiler is compatible with both DLL and non-DLL usage. The RENT

option is also not needed as all 64-bit COBOL programs are reentrant. The NORENT compilation option is not supported. The EXPORTALL compilation option is needed in order to export symbols from a DLL.

Although some functions of z/OS DLLs overlap the functions provided by COBOL dynamic CALL statements, DLLs have several advantages over regular z/OS program objects and dynamic calls:

- DLLs are common across COBOL and C/C++, thus providing better interoperability 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 program objects with names of up to 160 characters.

DLLs are supported by IBM z/OS Language Environment, based on function provided by the z/OS program management binder. DLL support is available for applications running under z/OS in batch or in TSO, CICS, z/OS UNIX, or IMS environments.

Related references

[“PGMNAME” on page 388](#)

[MVS Program Management: User's Guide and Reference](#) (Binder support for DLLs)

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.

Consideration for AMODE 64

A COBOL program compiled with the LP (64) compiler option is already enabled for DLL support and is reentrant. The LP (64) option implies DLL and RENT. If NODLL or NORENT are specified, the compiler discards these options with warning messages. A program compiled with LP (64) can be linked with a DLL or non-DLL application.

The LP (64) option implies RMODE (ANY). If RMODE (24) is specified, the compiler discards the option with a warning message. AMODE 64 COBOL programs are loaded below the bar.

In an application with DLL support, use the following compiler options depending on where the programs or classes are:

Programs or classes in:	In LP (32) Compile with:	In LP (64) Compile with:
Root program object	DLL, RENT, NOEXPORTALL	NOEXPORTALL
DLL program objects used by other program objects	DLL, RENT, EXPORTALL	EXPORTALL

If a DLL program object includes some programs that are used only from within the DLL module, you can hide these routines by compiling them with NOEXPORTALL.

[“Example: sample JCL for a procedural DLL application” on page 569](#)

Related tasks

[“Creating a DLL under z/OS UNIX” on page 286](#)

[“Linking DLLs” on page 569](#)

Chapter 30, [“Creating a DLL or a DLL application,” on page 567](#)

Related references

[“DLL” on page 353](#)

[“EXPORTALL” on page 359](#)

[“LP” on page 373](#)

[“RENT” on page 391](#)

Linking DLLs

You can link DLL-enabled object modules into separate DLL program objects, 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.

The DLL support in the z/OS binder is recommended for linking DLL applications. The binder can directly receive the output of COBOL compilers.

A binder-based DLL must reside in a PDSE or in a z/OS UNIX file.

When using the binder to link a DLL application, use the following options:

Type of code	Link using binder parameters:
DLL applications	DYNAM(DLL), RENT
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 in which 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 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.

[“Example: sample JCL for a procedural DLL application” on page 569](#)

Related tasks

[“Creating a DLL under z/OS UNIX” on page 286](#)

Chapter 30, [“Creating a DLL or a DLL application,” on page 567](#)

[“Compiling programs to create DLLs” on page 568](#)

Related references

MVS Program Management: User's Guide and Reference (Binder support for DLLs)

Example: sample JCL for a procedural DLL application

The following example shows how to create an application that consists of a main program that calls a DLL subprogram.

The first step creates the DLL program object that contains the subprogram DemoDLLSubprogram. The second step creates the main program object that contains 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
//      DD DSN=&LEPFX..SCEERUN2,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` and `CALL literal` statements to make calls to DLLs. However, there are a few additional considerations for the `CALL identifier` case.

For the content of the *identifier* or for the *literal*, use the name of either of the following programs:

- A nested program in the same compilation unit that is eligible to be called from the program that contains 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 runtime environment interprets the program-name in the *identifier* according to the setting of the `PGMNAME` compiler option of the program that contains 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 z/OS UNIX file system is case sensitive. If the target DLL is a PDSE member, the DLL member name must be eight characters or less. For the purpose of the search for the DLL as a PDSE member, the run time automatically converts the name to uppercase.

If the runtime 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” on page 571](#)

[“Compiling programs to create](#)

[DLLs” on page 568](#)

[“Linking DLLs” on page 569](#)

Related references

[“DLL” on page 353](#)

[“PGMNAME” on page 388](#)

[CALL statement \(*Enterprise COBOL for z/OS Language Reference*\)](#)

[“Search order for DLLs in the z/OS UNIX file system” on page 571](#)

Search order for DLLs in the z/OS UNIX file system

When you use the z/OS UNIX file system, the search order for resolving a DLL reference in a CALL statement depends on the setting of the Language Environment POSIX runtime option.

If the POSIX runtime option is ON, the search order is as follows:

1. The runtime environment looks for the DLL in the z/OS UNIX file system. 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 z/OS UNIX file system is case sensitive.
2. If the runtime environment does not find the DLL in the z/OS UNIX file system, 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 runtime option is set to OFF, the search order is reversed:

1. The runtime environment tries to load the DLL from the search order for the load library of the caller.
2. If the runtime environment cannot load the DLL from this load library, it tries to load the DLL from the z/OS UNIX file system.

Related tasks

[“Using CALL identifier with DLLs” on page 570](#)

Related references

[Language Environment Programming Reference \(POSIX\)](#)

Using DLL linkage and dynamic calls together

For applications (that is, Language Environment enclaves) that are structured as multiple separately bound modules, each module can be invoked by using dynamic call linkage or DLL linkage. For a certain module, use exclusively one form of linkage to enter it. However, the caller can contain CALL statements with both linkage types, calling out to different modules.

DLL linkage refers to a call in a program that is compiled with the DLL and NODYNAM options, or a call in a program that is compiled with the LP(64) and NODYNAM options, or a call with the CALLINTERFACE compiler directive that specifies DLL. In such calls, the called subprogram is resolved 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.

Within a compilation unit you can call a specific program with only one of the calling conventions: Dynamic, DLL or Static. If a program is called by using different calling conventions, the compiler diagnoses this case and force all the calls to have the same convention as the first call statement that is encountered for that program.

A program can contain CALL statements with both dynamic call linkage and DLL linkage. It can do so by using the CALLINTERFACE compiler directive to specify the linkage type of a particular call. 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. You cannot cancel programs that are called by 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.

Related concepts

[“Dynamic link libraries \(DLLs\)” on page 567](#)

Related tasks

[“Compiling programs to create DLLs” on page 568](#)

[“Linking DLLs” on page 569](#)

[“Using procedure or function pointers with DLLs” on page 572](#)

[“Calling DLLs from non-DLLs” on page 572](#)

Related references

[“DLL” on page 353](#)

[“EXPORTALL” on page 359](#)

CALLINTERFACE (*Enterprise COBOL for z/OS Language Reference*)

Using procedure or function pointers with DLLs

In run units that contain both DLLs and non-DLLs, use procedure- and function-pointer data items with care.

The SET *procedure-pointer-1* TO ENTRY *entry-name* statement, SET *function-pointer-1* TO ENTRY *entry-name* statement, and the CALL statement have a call linkage type that associates with them. The call linkage type is determined by the compiler options and the CALLINTERFACE directive that are in effect on that statement. In a program that is compiled with the DLL option, the default call linkage type is DLL. Otherwise, the linkage type is non-DLL. This default can be overridden by the CALLINTERFACE directive.

For a procedure-pointer or function-pointer data item that is set by a SET statement with linkage type non-DLL, it must not be used by a CALL statement with linkage type DLL. For a SET statement with linkage type DLL and the *entry-name* is an identifier, and if the NODYNAM option is in effect, the *entry-name* 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. In this case, note also that:

- 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 that contains 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 z/OS UNIX file system is case sensitive.
- If the target DLL is a PDSE member, the DLL member name must have eight characters or less. For the purpose of the search for the DLL as a PDSE member, the name is automatically converted to uppercase.

Related tasks

[“Using CALL identifier with DLLs” on page 570](#)

[“Using procedure and function pointers” on page 545](#)

[“Compiling programs to create DLLs” on page 568](#)

[“Linking DLLs” on page 569](#)

Related references

[“DLL” on page 353](#)

[“EXPORTALL” on page 359](#)

CALLINTERFACE (*Enterprise COBOL for z/OS Language Reference*)

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 program object 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 program object 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 function-pointer`, where *function-pointer* is set to a function descriptor of the target program. 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” on page 573](#)

Related tasks

[“Using procedure and function pointers” on page 545](#)

Example: calling DLLs from non-DLLs

The following example shows how a COBOL program that is not in a DLL (COBOL1) can call a COBOL program that is in a DLL (program ooc05R in DLL OOC05R).

```

CBL NODYNAM
  IDENTIFICATION DIVISION.
  PROGRAM-ID. 'COBOL1'.
  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 FUNCTION-POINTER.
  77 DLL-RC PIC S9(9) BINARY.
  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'OOC05R' 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

  *      Use the function pointer on the call statement to call the
  *      program in the DLL.
  *      Call the program in the DLL.
  CALL DLL-PROGRAM-HANDLE

  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 z/OS C/C++ products, except for COBOL EXTERNAL data. In particular, COBOL applications can call functions that are exported from C/C++ DLLs, and C/C++ applications can call COBOL programs that are exported from COBOL DLLs.

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.

The COBOL options DLL, RENT, and EXPORTALL work much the same way as the C/C++ DLL, RENT, and EXPORTALL options. (The DLL option applies only to C.) However, the C/C++ compiler produces DLL-enabled code by default.

You can pass a C/C++ DLL function pointer to COBOL and use it within COBOL, receiving the C/C++ function pointer as a function-pointer data item. 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 function-pointer.
Procedure Division.
    Call "c-function" returning fp.
    Call fp.

```

Related tasks

[“Compiling programs to create DLLs” on page 568](#)

[“Linking DLLs” on page 569](#)

Related references

[“DLL” on page 353](#)

[“EXPORTALL” on page 359](#)

[“RENT” on page 391](#)

EXTERNAL clause (*Enterprise COBOL for z/OS Language Reference*)

CALLINTERFACE (*Enterprise COBOL for z/OS Language Reference*)

Using DLLs in OO COBOL applications

You must compile each COBOL class definition using the DLL, THREAD, RENT, and DBCS compiler options, and link-edit it into a separate DLL module using the RENT binder option.

Related tasks

[Chapter 16, “Compiling, linking, and running OO applications,” on page 291](#)

[“Compiling programs to create](#)

[DLLs” on page 568](#)

[“Linking DLLs” on page 569](#)

Related references

[“DLL” on page 353](#)

[“THREAD” on page 408](#)

[“RENT” on page 391](#)

[“DBCS” on page 350](#)

Chapter 31. Preparing COBOL programs for multithreading

You can run COBOL programs in multiple threads within a process under batch, TSO, IMS, or z/OS UNIX.

There is no explicit COBOL language to use for multithreaded execution; rather, you compile with the `THREAD` compiler option.

COBOL does not directly support managing program threads. However, you can run COBOL programs that you compile with the `THREAD` compiler option in multithreaded application servers, in applications that use a C/C++ driver program to create the threads, in programs that interoperate with Java and use Java threads, and in applications that use PL/I tasking. In other words, other programs can call COBOL programs in such a way that the COBOL programs run in multiple threads within a process or as multiple program invocation instances within a thread. Your threaded application must run within a single Language Environment enclave.

Choosing LOCAL-STORAGE or WORKING-STORAGE: Because you must code your multithreaded programs as recursive, the persistence of data is that of any recursive program:

- Data items in the `LOCAL-STORAGE SECTION` are automatically allocated for each instance of a program invocation. When a program runs in multiple threads simultaneously, each invocation has a separate copy of `LOCAL-STORAGE` data.
- Data items in the `WORKING-STORAGE SECTION` are allocated once for each program and are thus available in their last-used state to all invocations of the program.

For the data that you want to isolate to an individual program invocation instance, define the data in the `LOCAL-STORAGE SECTION`. In general, this choice is appropriate for working data in threaded programs. If you define data in `WORKING-STORAGE` and your program changes the contents of the data, you must take one of the following actions:

- Structure your application so that you do not access data in `WORKING-STORAGE` simultaneously from multiple threads.
- If you do access data simultaneously from separate threads, write appropriate serialization code.

Related concepts

[“Multithreading” on page 577](#)

Related tasks

[“Choosing `THREAD` to support multithreading” on page 578](#)

[“Transferring control to multithreaded programs” on page 579](#)

[“Ending multithreaded programs” on page 579](#)

[“Processing files with multithreading” on page 580](#)

[“Handling COBOL limitations with multithreading” on page 582](#)

Related references

[“`THREAD`” on page 408](#)

`PROGRAM-ID` paragraph (*Enterprise COBOL for z/OS Language Reference*)

Multithreading

To use COBOL support for multithreading, you need to understand how processes, threads, run units, and program invocation instances relate to each other.

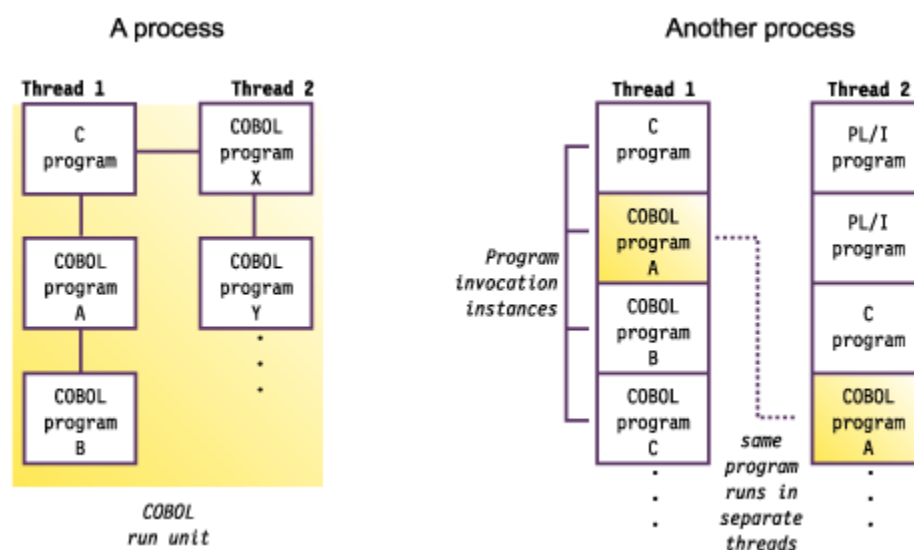
The operating system and multithreaded applications can handle execution flow within a *process*, which is the course of events when all or part of a program runs. Programs that run within a process can share resources. Processes can be manipulated. For example, they can have a high or low priority in terms of the amount of time that the system devotes to running the process.

Within a process, an application can initiate one or more *threads*, each of which is a stream of computer instructions that controls that thread. A multithreaded process begins with one stream of instructions (one thread) and can later create other instruction streams to perform tasks. These multiple threads can run concurrently. Within a thread, control is transferred between executing programs.

In a multithreaded environment, a COBOL *run unit* is the portion of the process that includes threads that have actively executing COBOL programs. The COBOL run unit continues until no COBOL program is active in the execution stack for any of the threads. For example, a called COBOL program contains a GOBACK statement and returns control to a C program. Within the run unit, COBOL programs can call non-COBOL programs, and vice versa.

Within a thread, control is transferred between separate COBOL and non-COBOL programs. For example, a COBOL program can call another COBOL program or a C program. Each separately called program is a *program invocation instance*. Program invocation instances of a particular program can exist in multiple threads within a given process.

The following illustration shows these relationships between processes, threads, run units, and program invocation instances.



Related concepts

Language Environment Programming Guide (Program management model, Understanding the basics: threads)

Related tasks

[“Choosing THREAD to support multithreading” on page 578](#)

[“Transferring control to multithreaded programs” on page 579](#)

[“Ending multithreaded programs” on page 579](#)

[“Processing files with multithreading” on page 580](#)

[“Handling COBOL limitations with multithreading” on page 582](#)

Related references

[“THREAD” on page 408](#)

Choosing THREAD to support multithreading

Use the THREAD compiler option for multithreading support. Use THREAD if your program will be called in more than one thread in a single process by an application. However, THREAD might adversely affect performance because of the serialization logic that is automatically generated.

In order to run COBOL programs in more than one thread, you must compile all of the COBOL programs in the application using the THREAD compiler option. You must also compile them with the RENT compiler option and link them with the RENT option of the binder (linkage-editor).

Use the `THREAD` option when you compile object-oriented (OO) clients and classes.

Language restrictions: When you use the `THREAD` option, you cannot use certain language elements. For details, see the related reference below.

Recursion: Before you compile a program using the `THREAD` compiler option, you must specify the `RECURSIVE` phrase in the `PROGRAM-ID` paragraph. If you do not do so, an error will occur.

Related tasks

[“Sharing data in recursive or multithreaded programs” on page 16](#)
[“Compiling OO applications under z/OS UNIX” on page 291](#)

Related references

[“THREAD” on page 408](#)

Transferring control to multithreaded programs

When you write COBOL programs for a multithreaded environment, choose appropriate program linkage statements.

As in single-threaded environments, a called program is in its initial state when it is first called within a run unit and when it is first called after a `CANCEL` to the called program. Ensure that the program that you name on a `CANCEL` statement is not active on any thread. If you try to cancel an active program, a severity-3 Language Environment condition occurs.

If your threaded application requires preinitialization, use the Language Environment services (CEEPIPI interface). You cannot use the COBOL-specific interfaces for preinitialization (runtime option `RTEREUS`) to establish a reusable environment from any program that has been compiled with the `THREAD` option.

Related concepts

Language Environment Programming Guide (What happens during termination: enclave termination)

Related tasks

[“Ending multithreaded programs” on page 579](#)
[“Ending and reentering main programs or subprograms” on page 534](#)

Ending multithreaded programs

You can end a multithreaded program by using `GOBACK`, `EXIT PROGRAM`, or `STOP RUN`.

Use `GOBACK` to return to the caller of the program. When you use `GOBACK` from the first program in a thread, the thread is terminated. If that thread is the initial thread in an enclave, the entire enclave is terminated.

Use `EXIT PROGRAM` as you would `GOBACK`, except from a main program where it has no effect.

Use `STOP RUN` to terminate the entire Language Environment enclave and to return control to the caller of the main program (which might be the operating system). All threads that are executing within the enclave are terminated.

Related concepts

Language Environment Programming Guide (What happens during termination: enclave termination)

Related tasks

[“Ending and reentering main programs or subprograms” on page 534](#)

Processing files with multithreading

In threaded applications, you can code COBOL statements for input and output in QSAM, VSAM, and line-sequential files.

Each file definition (FD) has an implicit serialization lock. This lock is used with automatic serialization logic during the input or output operation that is associated with the execution of the following statements:

- OPEN
- CLOSE
- READ
- WRITE
- REWRITE
- START
- DELETE

Automatic serialization also occurs for the implicit MOVE that is associated with the following statements:

```
WRITE record-name FROM identifier
READ file-name INTO identifier
```

Automatic serialization is not applied to any statements specified within the following conditional phrases:

- AT END
- NOT AT END
- INVALID KEY
- NOT INVALID KEY
- AT END-OF-PAGE
- NOT AT END-OF-PAGE

Related concepts

[“File-definition \(FD\) storage” on page 580](#)

Related tasks

[“Closing QSAM files” on page 179](#)

[“Closing VSAM files” on page 203](#)

[“Coding ERROR declaratives” on page 246](#)

[“Serializing file access with multithreading” on page 581](#)

File-definition (FD) storage

On all program invocations, the storage that is associated with a file definition (such as FD records and the record area that is associated with the SAME RECORD AREA clause) is allocated and available in its last-used state.

All threads of execution share this storage. You can depend on automatic serialization for this storage during the execution of the OPEN, CLOSE, READ, WRITE, REWRITE, START, and DELETE statements, but not between uses of these statements.

Related tasks

[“Serializing file access with multithreading” on page 581](#)

Serializing file access with multithreading

To take full advantage of automatic serialization and to avoid explicitly writing your own serialization logic, use one of the recommended file organizations and usage patterns when you access files in threaded programs.

Use one of the following file organizations:

- Sequential organization
- Line-sequential organization
- Relative organization with sequential access
- Indexed organization with sequential access

Use the following pattern for input:

```
OPEN INPUT fn
READ fn INTO local-storage-item
* Process the record from the local-storage item
CLOSE fn
```

Use the following pattern for output:

```
OPEN OUTPUT fn
* Construct output record in local-storage item
WRITE rec FROM local-storage-item
CLOSE fn
```

With other usage patterns, you must take one of the following actions:

- Verify the safety of your application logic. Ensure that two instances of the program are never simultaneously active on different threads.
- Code explicit serialization logic by using calls to POSIX services.

To avoid serialization problems when you access a file from multiple threads, define the data items that are associated with the file (such as file-status data items and key arguments) in the LOCAL - STORAGE SECTION.

[“Example: usage patterns of file input and output with multithreading” on page 581](#)

Related tasks

[“Calling UNIX/POSIX APIs” on page 516](#)

Example: usage patterns of file input and output with multithreading

The following examples show the need for explicit serialization logic when you deviate from the recommended usage pattern for file input and output in your multithreaded applications. These examples also explain the unexpected behavior that might result if you fail to handle serialization properly.

In each example, two instances of a program that contains the sample operations are running within one run unit on two different threads.

```
READ F1
REWRITE R1
```

In the example above, the second thread might execute the READ statement after the READ statement is executed on the first thread but before the REWRITE statement is executed on the first thread. The

REWRITE statement might not update the record that you intended. To ensure the results that you want, write explicit serialization logic.

```
READ F1
. . .
* Process the data in the FD record description entry for F1
. . .
```

In the example above, the second thread might execute the READ statement while the first thread is still processing a record in the FD record description entry. The second READ statement would overlay the record that the first thread is processing. To avoid this problem, use the recommended technique:

```
READ F1 INTO LOCAL-STORAGE-item
```

Other cases: You must give similar consideration to other usage patterns that involve a sequence of related input and output operations, such as START followed by READ NEXT, or READ followed by DELETE. Take appropriate steps to ensure the correct processing of file input and output.

Handling COBOL limitations with multithreading

Some COBOL applications depend on subsystems or other applications. In a multithreaded environment, these dependencies and others result in some limitations on COBOL programs.

In general, you must synchronize access to resources that are visible to the application within a run unit. Exceptions to this requirement are DISPLAY and ACCEPT, which you can use from multiple threads, and supported COBOL file I/O statements that have the recommended usage pattern; all synchronization is provided for these by the runtime environment.

CICS: You cannot run multithreaded applications in CICS. In CICS, you can run a COBOL program that has been compiled with the THREAD option and that is part of an application that has no multiple threads or PL/I tasks.

Recursive: Because you must code the programs in a multithreaded application as recursive, you must adhere to all the restrictions and programming considerations that apply to recursive programs, such as not coding nested programs.

Reentrancy: You must compile your multithreading programs with the RENT compiler option and link them with the RENT option of the binder (linkage-editor).

POSIX and PL/I: If you use POSIX threads in your multithreaded application, you must specify the Language Environment runtime option POSIX(ON). If the application uses PL/I tasking, you must specify POSIX(OFF). You cannot mix POSIX threads and PL/I tasks in the same application.

PL/I tasking: To include COBOL programs in applications that contain multiple PL/I tasks, follow these guidelines:

- Compile all COBOL programs that you run in multiple PL/I tasks with the THREAD option. If you compile any COBOL program with the NOTHREAD option, all of the COBOL programs must run in one PL/I task.
- You can call COBOL programs compiled with the THREAD option from one or more PL/I tasks. However, calls from PL/I programs to COBOL programs cannot include the TASK or EVENT option. The PL/I tasking call must first call a PL/I program or function that in turn calls the COBOL program. This indirection is required because you cannot specify the COBOL program directly as the target of a PL/I CALL statement that includes the TASK or EVENT option.
- Be aware that issuing a STOP RUN statement from a COBOL program or a STOP statement from a PL/I program terminates the entire Language Environment enclave, including all the tasks of execution.
- Do not code explicit POSIX threading (calls to pthread_create()) in any run unit that includes PL/I tasking.

C and Language Environment conforming assembler: You can combine your multithreaded COBOL programs with C programs and Language Environment conforming assembler programs in the same run unit when those programs are also appropriately coded for multithreaded execution.

AMODE: You must run your multithreaded applications with AMODE 31. You can run a COBOL program that has been compiled with the THREAD option with AMODE 24 as part of an application that does not have multiple threads or PL/I tasks.

Asynchronous signals: In a threaded application your COBOL program might be interrupted by an asynchronous signal or interrupt. If your program contains logic that cannot tolerate such an interrupt, you must disable the interrupts for the duration of that logic. Call a C/C++ function to set the signal mask appropriately.

Older COBOL programs: To run your COBOL programs on multiple threads of a multithreaded application, you must compile them with Enterprise COBOL and use the THREAD option. Run applications that contain programs compiled by older compilers only on one thread.

IGZETUN and IGZEOPT: Do not use the modules IGZETUN (for storage tuning) or IGZEOPT (for runtime options) for applications in which the main program has been compiled with the THREAD option; these CSECTs are ignored.

UPSI switches: All programs and all threads in an application share a single copy of UPSI switches. If you modify switches in a threaded application, you must code appropriate serialization logic.

Related tasks

[“Making recursive calls” on page 545](#)

[“Serializing file access with multithreading” on page 581](#)

XL C/C++ Programming Guide (Using threads in z/OS UNIX System Services applications)

Language Environment Writing ILC Communication Applications

Chapter 32. Using user-defined functions

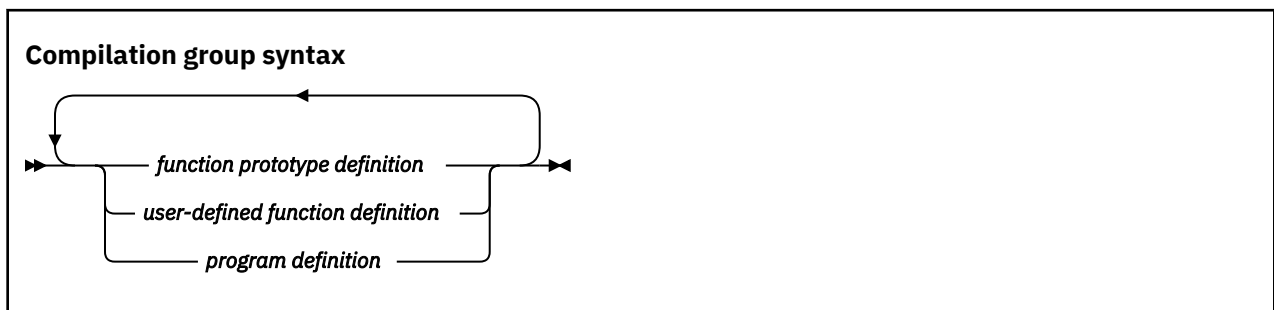
COBOL user-defined functions allow you to write your own function definitions that may be invoked by using the `FUNCTION` keyword. User-defined functions are always recursive and may invoke themselves.

User-defined functions can be invoked statically, dynamically, or via dynamic-link library (DLL). The `ENTRY-INTERFACE` phrase of the `FUNCTION-ID` paragraph determines the kind of invocation associated with the function. The `CALLINTERFACE` directive has no effect on the interface convention for user-defined functions.

Function pointers to user-defined functions are not supported. If a user-defined function is invoked via the `CALL` statement the behavior is unpredictable.

Structuring user-defined functions

The user-defined function definition is a stand-alone compilation unit and cannot be nested within other compilation units such as programs, methods, or other user-defined functions. It is recommended that you code your user-defined function in a separate file, that is, a separate batch compilation, and use a function prototype to invoke the function from the calling program. If you choose to code your user-defined function without a prototype in the same batch compilation, the user-defined function definition must appear first within the compilation group, before the programs or functions that call it. Doing this requires special link-editing steps if the entry point of the program is the default entry point. If you choose to code your user-defined function with a prototype in the same batch compilation, the user-defined function can appear in any order amongst the various programs or functions that call it, as long as the prototype appears before the first program or function that invokes it or is included from a copybook if the functions are called from more than one program. The first program or user-defined function in the batch program is chosen as the default entry point, so special link-editing steps might be required if the order of programs is not chosen carefully.



Note: Currently, the SQL, CICS, and JAVAIOP facilities cannot be used with user-defined functions due to the need to include the user-defined function definition ahead of the program using them.

Coding user-defined function definitions

User-defined functions must contain the `FUNCTION-ID` paragraph within the `IDENTIFICATION DIVISION`. The `END FUNCTION` keyword indicates the end of a user-defined function definition. User-defined functions must contain the `RETURNING` phrase of the `PROCEDURE DIVISION` header to indicate the returning item of the function.

Zero or more parameters for user-defined functions may appear in the `USING` phrase of the `PROCEDURE DIVISION` header. Parameters may be passed `BY REFERENCE` or `BY VALUE`.

The following example shows a user-defined function definition called "docalc":

```
Identification division.  
Function-id. docalc.  
Data division.  
Linkage section.  
1 kind pic x(3).
```

```

1 argA pic 999.
1 argB pic v999.
1 res pic 999v999.
Procedure division
  using by reference kind argA argB
  returning res.
  if kind equal "add" then
    compute res = argA + argB
  end-if
  goback.
End function docalc.

```

Related references

FUNCTION-ID paragraph (*Enterprise COBOL for z/OS Language Reference*)

Compiling user-defined function definitions

When the compiler encounters a user-defined function definition, it automatically extracts characteristics from the definition. These characteristics together form a function prototype. The function name, returning item data description entry, and parameter data description entries are some of the characteristics gathered.

When a function invocation is encountered by the compiler in a program or user-defined function definition, the compiler will determine if the arguments and returning item are appropriate. The arguments will be checked against the corresponding formal parameters in a process known as *conformance checking*. See *Conformance of parameters for user-defined functions* in the *Enterprise COBOL for z/OS Language Reference*.

Invoking user-defined functions

User-defined functions may be invoked in most places that intrinsic functions may be invoked.

To invoke user-defined functions without the FUNCTION keyword, specify the function name in the REPOSITORY paragraph. For details, see *REPOSITORY paragraph* in the *Enterprise COBOL for z/OS Language Reference*.

The following example shows a compilation group containing a user-defined function definition called "docalc" and a program containing the function invocation:

```

Identification division.
  Function-id. docalc.
Data division.
Linkage section.
  1 kind pic x(3).
  1 argA pic 999.
  1 argB pic v999.
  1 res pic 999v999.
Procedure division
  using by reference kind argA argB
  returning res.
  if kind equal "add" then
    compute res = argA + argB
  end-if
  goback.
End function docalc.
Identification division.
  Program-id. 'mainprog'.
Environment division.
  Configuration section.
  Repository.
    function docalc.
Data division.
  Working-storage section.
  1 result pic 999v999 usage display.
Procedure division.
  compute result = docalc("add" 10 0.23)
  display "hello from mainprog, result=" result
  goback.
End program 'mainprog'.

```

After running this program, the output is as follows:

```
hello from mainprog, result=010230
```

Specifying external names for user-defined functions

User-defined functions may be optionally specified with an external name in the FUNCTION-ID paragraph of the IDENTIFICATION DIVISION. You can provide an external name that is exposed to the operating environment that is different from the name used to invoke the function. If an external name is provided, the PGMNAME option applies to the external name (not to the internal name).

For details on the external name specification, see *FUNCTION-ID paragraph* in the *Enterprise COBOL for z/OS Language Reference*. For details on the PGMNAME option, see “PGMNAME” on page 388.

The external name will be the name exposed to the operating environment. It cannot be used to invoke the function within a COBOL program.

The following example shows a user-defined function of "docalc" with an external name of "XYZCALC":

```
Identification division.
  Function-id. docalc as 'XYZCALC'.
Data division.
  Linkage section.
    1 kind pic x(3).
    1 argA pic 999.
    1 argB pic v999.
    1 res pic 999v999.
Procedure division
  using by reference kind argA argB
  returning res.
  if kind equal "add" then
    compute res = argA + argB
  end-if
  goback.
End function docalc.
```

Link-editing user-defined functions

For AMODE 31 (31-bit) compilations using the LP(32) option, when a compilation group contains user-defined functions and programs, and one of the programs is intended to be the main program, you must provide a binder ENTRY statement for the main program to avoid defaulting to the first user-defined function. For AMODE 64 (64-bit) compilations using the LP(64) option, programs will generate the CELQSTRT entry (but not user-defined functions), so use the usual ENTRY CELQSTRT statement.

See the following example of the JCL step:

```
//LKED1 EXEC PGM=IEWL,
// PARM='LIST,MAP '
//SYSLIB DD DSN=CEE.SCEELKED,DISP=SHR
//OBJECT DD DSN=&&OBJECT,DISP=(SHR,PASS)
//SYSLIN DD *
INCLUDE OBJECT(OBJ1)
ENTRY MAINPROG
/*
```

Passing arguments BY CONTENT to user-defined functions

Though the formal parameters of a user-defined function definition may only be specified as BY REFERENCE or BY VALUE, you can still effectively pass arguments to a user-defined function BY CONTENT using the CONTENT-OF intrinsic function. To achieve this, you must specify the formal parameters as BY REFERENCE in the function definition. On the user-defined function invocation, the CONTENT-OF intrinsic function would then be used to wrap each argument that you would like to be passed effectively BY CONTENT.

See the following example of a function definition and a program containing an invocation of the function:

```
Identification division.
  Function-id. docalc.
```

```

Data division.
Linkage section.
  1 kind pic x(3).
  1 argA pic 999.
  1 argB pic v999.
  1 res pic 999v999.
Procedure division
  using by reference kind argA argB
  returning res.
  if kind equal "add" then
    compute res = argA + argB
  end-if
  goback.
End function docalc.
Identification division.
  Program-id. 'mainprog'.
Environment division.
Configuration section.
  Repository.
    function content-of intrinsic
    function docalc.
Data division.
Working-storage section.
  1 result pic 999v999 usage display.
  1 arg-1 pic x(3).
  1 arg-2 pic 999.
  1 arg-3 pic v999.
Procedure division.
  move "add" to arg-1
  move 10 to arg-2
  move 0.23 to arg-3
* arg-1 is effectively BY CONTENT
* arg-2 is BY REFERENCE
* arg-3 is BY REFERENCE
  compute result = docalc(content-of(arg-1) arg-2 arg-3)
  display "hello from mainprog, result=" result
  goback.
End program 'mainprog'.

```

The CONTENT-OF intrinsic function produces a temporary copy of the input argument. The temporary copy is then provided BY REFERENCE to the user-defined function and any modifications to the formal parameter within the user-defined function definition are effectively made to the temporary copy.

Related references

CONTENT-OF intrinsic function (*Enterprise COBOL for z/OS Language Reference*)

Chapter 33. Using function prototypes

Function prototypes enable you to invoke COBOL user-defined functions in the following ways.

- Enable you to invoke COBOL user-defined functions or non-COBOL external functions such as C functions using the FUNCTION keyword.
- Enable you to invoke user-defined functions whose definitions are written in other files such as compilation groups.
- Enable you to invoke user-defined functions with a linkage interface (such as STATIC, DYNAMIC, or DLL) that was specified on the prototype using the ENTRY-INTERFACE phrase of the FUNCTION-ID paragraph.

The function prototype definition is a stand-alone compilation unit that, when compiled, produces no object code but rather informs the compiler about a function prototype. When the compiler subsequently encounters an invocation to the function, the compiler is able to perform conformance checking as well as generate the correct calling code for the specified linkage interface. Function prototype definitions must appear within the compilation group before any invocations to the function. Therefore it is recommended that all function prototypes appear first, before program and user-defined function definitions.

Function prototype example

The following example contains a function prototype definition of function "myfunc", a program definition that invokes "myfunc", and a user-defined function definition for function "myfunc". Note that the IS PROTOTYPE phrase of the FUNCTION-ID paragraph indicates this compilation unit is a function prototype.

```
Identification division.
  Function-id. myfunc is prototype.
Data division.
  Linkage section.
    1 retval pic 9(9).
Procedure division returning retval.
  End function myfunc.
*****
Identification division.
  Program-id. myprog.
Data division.
  Working-storage section.
    1 a pic 9(9).
Procedure division.
  compute a = function myfunc
  goback.
End program myprog.
*****
Identification division.
  Function-id. myfunc.
Data division.
  Linkage section.
    1 parm1 pic x(10).
    1 retval pic 9(9).
Procedure division returning retval.
  compute retval = 0
  goback.
End function myfunc.
```

Function prototype with external name

Occasionally, it might be desirable to have the compiler generate code for invocations to function prototypes whose name either cannot be expressed as a COBOL word, or whose name might have restrictions or requirements in the external operating environment. In such cases you can code the AS *external-name* phrase on the FUNCTION-ID paragraph of the function prototype definition to provide the externalized name.

For example, you can name your function prototype "GetRecord", but have invocations of "GetRecord" actually invoke a function whose external name in your operating environment is "QQGETREC". To do this, you can code as follows:

```
Identification division.
  Function-id. GetRecord as 'QQGETREC' is prototype.
Data division.
  Linkage section.
    1 retval pic x(100).
  Procedure division returning retval.
  End function GetRecord.
*****
Identification division.
  Program-id. myprog.
Data division.
  Working-storage section.
    1 a pic x(100).
  Procedure division.
    move function GetRecord to a
      goback.
  End program myprog.
```

"GetRecord" might then be defined in another batch compilation as the following user-defined function:

```
Identification division.
  Function-id. GetRecord as 'QQGETREC'.
Data division.
  Linkage section.
    1 retval pic x(100).
  Procedure division returning retval.
    move "data" to retval
    goback.
  End function GetRecord.
```

Suppose instead of "QQGETREC", you choose "abc123Get_Customer_Record". To use this external name, you need to use either the PGMNAME compiler option or the ENTRY-NAME phrase of the FUNCTION-ID paragraph as in the following example:

```
Identification division.
  Function-id. GetRecord as 'abc123Get_Customer_Record' is prototype
    entry-name is longmixed.
Data division.
  Linkage section.
    1 retval pic x(100).
  Procedure division returning retval.
  End function GetRecord.
*****
Identification division.
  Program-id. myprog.
Data division.
  Working-storage section.
    1 a pic x(100).
  Procedure division.
    move function GetRecord to a
      goback.
  End program myprog.
```

Function prototype best practices

It is recommended that function prototypes be defined in copybooks and included into the compilation groups that reference them using the COPY statement. This practice ensures that prototype definitions across the application are the same. Furthermore, it is also recommended that the function prototype definition be included via COPY in the same compilation group as the user-defined function definition to allow the compiler to check for consistency.

For example, this function prototype is placed in a copybook called GETREC.

```
Identification division.
  Function-id. GetRecord is prototype.
Data division.
  Linkage section.
    1 retval pic x(100).
```

```
Procedure division returning retval.  
End function GetRecord.
```

This copybook is included along with the user-defined function definition in another compilation group.

```
Copy GETREC.  
Identification division.  
  Function-id. GetRecord.  
Data division.  
  Linkage section.  
  1 retval pic x(100).  
Procedure division returning retval.  
  move "data" to retval  
  goback.  
End function GetRecord.
```

The copybook is also included in any compilation group where "GetRecord" is invoked.

```
Copy GETREC.  
Identification division.  
  Program-id. myprog.  
Data division.  
  Working-storage section.  
  1 a pic x(100).  
Procedure division.  
  move function GetRecord to a  
  goback.  
End program myprog.
```

Language standard considerations

The function prototypes feature in Enterprise COBOL is based on the ISO/IEC 2014 COBOL standard. Some parts of this feature in Enterprise COBOL, however, have been modified and are specific to Enterprise COBOL.

The modified parts are as follows:

- The optional ENTRY-INTERFACE paragraph: this is based loosely upon the ENTRY-CONVENTION clause of the OPTIONAL paragraph of the IDENTIFICATION DIVISION in the standard. The implementation of the IDENTIFICATION DIVISION syntax is different than the COBOL standard, so to follow the conventions, ENTRY-INTERFACE was made a paragraph rather than a clause.
- The optional ENTRY-NAME paragraph: the reason is the same as above.

Part 5. Using COBOL for Web Services

Chapter 34. Web Services interface

COBOL can provide or request Web Services, using architectures based on SOAP (Simple Object Access Protocol) or REST (REpresentational State Transfer). Data in the interface to such services is typically represented in JSON (JavaScript Object Notation) or XML (eXtensible Markup Language). The following information describes COBOL and other facilities that can be used to generate and consume JSON and XML.

Processing JSON input

You can process JSON text input in a COBOL program by using the JSON PARSE statement. The statement identifies the source data item containing the JSON text, and the receiving data item that is populated by the parser.

In addition, the z/OS Client Web Enablement Toolkit enables applications to participate in the client/server space by providing a built-in z/OS JSON parser (provided with z/OS 2.2, or z/OS 2.1 with the PTF for APAR OA46575 installed) to consume JSON text from any source. Be aware that this parser supports JSON texts in EBCDIC codepage 1047 only, thus JSON text received by the application in another encoding format must first be converted to EBCDIC 1047 before it can be input into the parser. For details, see the description of the z/OS JSON parser in *z/OS MVS Programming: Callable Services for High-Level Languages*.

Related tasks

[Chapter 35, “Processing JSON input,” on page 597](#)

[Chapter 36, “Producing JSON output,” on page 607](#)

[Chapter 37, “Processing XML input,” on page 611](#)

[Chapter 38, “Producing XML output,” on page 651](#)

Chapter 35. Processing JSON input

You can process JSON text input in a COBOL program by using the JSON PARSE statement. The statement identifies the source data item containing the JSON text, and the receiving data item that is populated by the parser.

You can optionally also specify the following phrases:

- WITH DETAIL to indicate that messages should be generated for any nonexception and exception conditions
- NAME OF to provide alternative names for the populated data items
- SUPPRESS for data items to be excluded from assignment by the JSON parser
- CONVERTING for data items to be converted from JSON BOOLEAN name/value pairs
- ON EXCEPTION to receive control if an exception occurs
- NOT ON EXCEPTION to receive control if an exception does not occur

The JSON text input is assumed to be encoded in UTF-8 (CCSID 1208) and must be contained within an alphanumeric, national, or UTF-8 group item, or elementary data item of category alphanumeric, national, or UTF-8.

Specifying the JSON PARSE statement will pass control to the JSON parser and will read the input JSON text and populate the receiving data item using the same semantics as the equivalent COBOL MOVE statements.

Following the execution of a JSON PARSE statement, you can use these special registers to receive information from the parser:

- A non-zero JSON-CODE will indicate the kind of exception conditions that have occurred
- A non-zero JSON-STATUS will indicate the kind of nonexception conditions that have occurred

Parsing JSON documents

Consider a COBOL source program containing the following statements.

```
Identification division.
  Program-id. jparse1.
Data division.
  Working-storage section.
    1 msg.
      4 ver usage comp-1.
      4 uid pic 9999 usage display.
      4 txt pic x(32).
  Linkage section.
    1 json-text pic x(128).
Procedure division using json-text.
  Json parse json-text into msg
  end-json.
  If ver equal to 5 then
    display "Message ID is " uid
    display "Message text is '" txt "'".
  Goback.
End program jparse1.
```

The JSON PARSE statement above identifies data item *json-text* as the UTF-8 encoded source of JSON text, and data item *msg-data* as the receiver of the JSON values.

Assuming that data item *json-text* contains:

```
{ "msg": { "ver": 5, "uid": 1000, "txt": "Hello World!" } }
```

then the output of executing the program is:

```
Message ID is 1000
Message text is 'Hello World!
```

How to match JSON names that are not valid COBOL data names to data items

JSON allows many more characters and types of characters to appear in JSON names than COBOL allows in data names. To facilitate the match of JSON names with COBOL data names, you can use the NAME phrase on the JSON PARSE statement. Consider the following JSON text.

```
{"abc+":100}
```

The JSON name *abc+* is not a valid COBOL data name but you can use the NAME phrase to match it to a valid COBOL data name. The following COBOL program illustrates how to parse that JSON text into a COBOL data item.

```
Identification division.
  Program-id. name1.
Data division.
  Working-storage section.
    1 mydata pic 999.
  Linkage section.
    1 json-text pic x(128).
Procedure division using json-text.
  Json parse json-text into mydata
    name of mydata is "abc+"
  end-json.
  Display "mydata is " mydata.
  Goback.
End program name1.
```

Notice the use of the NAME phrase. Executing the program produces the following output:

```
mydata is 100
```

There are several important details to consider from the above example:

- Characters appearing in literal-1 on the NAME phrase are assumed to be encoded using the CCSID of the CODEPAGE compiler option in effect.
- Characters appearing in literal-1 will be matched to the JSON names in a case-sensitive manner, unlike COBOL data names which are matched in a case-insensitive manner.
- The NAME phrase, in aggregate, must not result in an ambiguous name specification¹.

1. For more details about ambiguous name specifications, see the "NAME phrase" of the JSON PARSE statement in the *Enterprise COBOL Language Reference*.

Preventing data items from being populated by the JSON PARSE statement

It is possible you may not want specific data items subordinate to the receiver to be populated by the JSON PARSE statement. To prevent specific data items from being populated you can use the SUPPRESS phrase of the JSON PARSE statement to tell the JSON parser to ignore data items. Consider the following COBOL program:

```
Identification division.
  Program-id. supp1.
Data division.
  Working-storage section.
    1 msg.
    4 ver usage comp-1.
    4 uid pic 9999 usage display.
    4 txt pic x(32).
  Linkage section.
    1 json-text          pic x(128).
```

```

Procedure division using json-text.
  Move 2 to uid.
  Json parse json-text INTO msg
    SUPPRESS uid
  end-json.
  If ver equal to 5 then
    display "Message ID is " uid
    display "Message text is '" txt "'".
  Goback.
End program supp1.

```

Notice that the data item *uid* has been set in the program to the value 2 and we wish to suppress its assignment in the JSON PARSE statement using the SUPPRESS phrase. Assuming the incoming JSON text in data item *json-text* contains:

```
{ "msg": { "ver": 5, "uid": 10, "txt": "Hello" } }
```

then the execution of the program results in this output:

```

Message ID is 0002
Message text is 'Hello

```

The data item *uid* retained the value 2 instead of being populated with the value 10.

Handling JSON arrays

JSON arrays can be parsed into COBOL data items whose data description entries contain the OCCURS clause or the OCCURS DEPENDING ON clause. Consider the following example where JSON array named *msg* is parsed into the similarly named COBOL data item.

Assume the JSON text contained in data item *json-text* is:

```
{ "some-data": { "msg": [ { "ver": 5, "uid": 10, "txt": "Hello" },
  { "ver": 5, "uid": 11, "txt": "World" }, { "ver": 5, "uid": 12, "txt": "!" } ] } }
```

Following is a COBOL program that parses this JSON text using a fixed occurrence table with the OCCURS clause.

```

Identification division.
  Program-id. occ1.
Data division.
  Working-storage section.
    1 some-data.
      2 msg occurs 3.
        4 ver usage comp-1.
        4 uid pic 9999 usage display.
        4 txt pic x(32).
  Linkage section.
    1 json-text pic x(128).
Procedure division using json-text.
  Json parse json-text into some-data
  end-json.
  If ver(1) equal to 5 then
    Display "Message ID is " uid(1)
    Display "Message text is '" txt(1) "'".
  If ver(2) equal to 5 then
    Display "Message ID is " uid(2)
    Display "Message text is '" txt(2) "'".
  If ver(3) equal to 5 then
    Display "Message ID is " uid(3)
    Display "Message text is '" txt(3) "'".
  Goback.
End program occ1.

```

Executing the program results in this output:

```

Message ID is 0010
Message text is 'Hello
Message ID is 0011
Message text is 'World
Message ID is 0012
Message text is '!'

```

Parsing into a variable occurrence table with the OCCURS DEPENDING ON clause can be done similarly:

```
Identification division.
  Program-id. odo1.
Data division.
Working-storage section.
  1 i pic 9.
  1 n pic 9.
  1 t pic x(128).
  1 msg_count pic 9.
  1 some-data.
    2 msg occurs 0 to 5 depending on n.
    4 ver usage comp-1.
    4 uid pic 9999 usage display.
    4 txt pic x(32).
Linkage section.
  1 json-text pic x(128).
Procedure division using json-text.
Main section.
  Move 4 to n.
  Move 0 to ver(1).
  Move 0 to ver(2).
  Move 0 to ver(3).
  Move 0 to ver(4).
  Json parse json-text into some-data
  end-json.
  Perform disp_msg varying i from 1 by 1 until i > n.
  Display "Message count: " msg_count.
  Goback.
Disp_msg section.
  If ver(i) equal to 5 then
    display "Message ID is " uid(I)
    display "Message text is '" txt(I) '"
    add 1 to msg_count
  else
    display "Invalid Message Version, ID is " uid(I).
End program odo1.
```

Executing the program results in this output:

```
Message ID is 0010
Message text is 'Hello
Message ID is 0011
Message text is 'World
Message ID is 0012
Message text is '!
Invalid Message Version, ID is 0001
Message count: 3
```

Note that subordinate data items of table element *msg(4)* are not assigned by the JSON PARSE statement because the JSON text does not contain a fourth table entry for the *msg* table. Also the OCCURS DEPENDING ON *object*, defined in this example as *n* must not be subordinate to data item *some-data* and needs to be given a value before the JSON PARSE statement receives program control. The value of the OCCURS DEPENDING ON *object* is the maximum number of table elements that the JSON PARSE statement may populate. If, in the JSON text, there are more table elements than the value of the OCCURS DEPENDING ON *object*, then those table elements are ignored and the condition is indicated in the JSON-STATUS special register. The OCCURS DEPENDING ON *object* is not set or updated by the JSON PARSE statement.

Parsing JSON anonymous arrays

You can parse JSON anonymous arrays by specifying the data name of the table on the JSON PARSE statement while using the NAME IS OMITTED phrase.

Consider the following example:

```
Identification division.
  Program-id. myprog.
Data division.
Working-storage section.
  1 ACT.
    2 B1 occurs 2.
      3 C1.
```

```

        4 M1 pic 9.
        4 D1 occurs 2.
        5 N1 pic 9.
1 json-text pic u dynamic.
Procedure division.
  move spaces to ACT
  move
-   ' [{"C1":{"M1":1,"D1":[{"N1":3},{"N1":4}]}} ,
    '{"C1":{"M1":2,"D1":[{"N1":5},{"N1":6}]}}]'
    to json-text
  json parse json-text into b1
  name b1 is omitted
end-json
display M1(1)
display M1(2)
display N1(1 1)
display N1(1 2)
display N1(2 1)
display N1(2 2)
goback
.
End program myprog.

```

Running the program will produce the following output:

```

1
2
3
4
5
6

```

Note that the JSON PARSE receiver b1 was not subscripted in order to refer to all occurrences of b1.

Consider the next example where the table b1 has two dimensions and is nested within another table like below:

```

Identification division.
  Program-id. myprog.
Data division.
Working-storage section.
  1 ACT.
  2 TOPTABLE occurs 3.
  3 B1 occurs 2.
  4 C1.
  5 M1 pic 9.
  5 D1 occurs 2.
  6 N1 pic 9.
1 json-text pic u dynamic.
Procedure division.
  move spaces to ACT
  move
-   ' [{"C1":{"M1":1,"D1":[{"N1":3},{"N1":4}]}} ,
    '{"C1":{"M1":2,"D1":[{"N1":5},{"N1":6}]}}]'
    to json-text
  json parse json-text into b1(2)
  name b1 is omitted
end-json
display M1(2 1)
display M1(2 2)
display N1(2 1 1)
display N1(2 1 2)
display N1(2 2 1)
display N1(2 2 2)
goback
.
End program myprog.

```

Running the program will produce the following output:

```

1
2
3
4
5
6

```

Note that b1(2) was specified with a single index subscript on the JSON PARSE statement. The value of the subscript 2 indicates that the second occurrence of table TOPTABLE shall be populated. The omission

of a second index indicates to the JSON PARSE statement that the statement shall assume the input JSON text contains the entire table b1 (rather than a single occurrence of b1, which is what would have been assumed if a second index subscript was specified). This pattern applies generally to multiple nested tables, that is, an anonymous array can be parsed by the combination of the NAME IS OMITTED phrase and by specifying the receiving item with one less subscript than its dimension.

Handling JSON null values

This section describes the ways to parse JSON null values using the JSON PARSE statement.

You can parse JSON null values in one of the following ways:

- Using the IGNORING phrase to ignore JSON null values.
- Using the CONVERTING phrase to convert JSON null values to specific values.
- Using the INDICATING phrase to assign a satellite data item as a null indicator.

Details of each way are as follows.

Ignoring JSON null values

If you want JSON null values to simply be ignored by the JSON PARSE statement, use the IGNORING phrase. You can specify individual items on the IGNORING phrase when you know in advance what items can be null, or you can use the ALL keyword to ignore all null values.

Converting JSON null values to specific values

Use the CONVERTING phrase when you want a JSON null to be converted into a specific value in the COBOL data item where the null occurred. For example, consider the following COBOL source:

```
`Identification division.`
Program-id. myprog.
Data division.
Working-storage section.
1 json-text pic u dynamic.
1 A.
2 B-data PIC X(10).
Procedure division.
move all 'z' to B-data
move u'{"A":{"B-data":null}}' to json-text
json parse json-text into A
converting B-data from json null using space
display "B-data: '" B-data '"'
goback.
End program myprog.
```

When compiled and run, the JSON PARSE statement encounters "B-data":null so SPACE is effectively moved into B-data, and indeed the output of this program is as follows:

```
B-data: ' '
```

Note that not all items can be used on the CONVERTING phrase due to the MOVE compatibility rules.

Assigning a satellite data item as a null indicator

Use the INDICATING phrase in the case where you can or want to designate satellite data items as null indicators. For example, consider the following COBOL source:

```
`Identification division.`
Program-id. myprog.
Data division.
Working-storage section.
1 json-text pic u dynamic.
1 A.
2 B-data pic x(10).
2 B-indicator pic x.
Procedure division.
```

```

move all 'z' to A.
move u'{"A":{"B-data":null}}' to json-text
json parse json-text into A
indicating B-data is json null
using 'Y' and 'N' in B-indicator
display "B-data: " B-data ""
display "B-indicator: " B-indicator
goback.
End program myprog.

```

When compiled and run, the JSON PARSE statement encounters "B-data":null and hence populates B-indicator with 'Y', as can be seen in the program output:

```

B-data: 'zzzzzzzzzz'
B-indicator: Y

```

Note that B-data was not populated by JSON PARSE.

JSON PARSE example

This example shows the processing of JSON text by the JSON PARSE statement into various types of COBOL data items. The JSON text is included directly in the program source for the purpose of this example. The output of the program is shown after.

```

Identification division.
  Program-id. jp_ex.
Data division.
  Working-storage section.
    1 jtxt-1047-client-data.
      3 pic x(16) value '{"client-data":{''.
      3 pic x(28) value '"account-num":123456789012,'.
      3 pic x(19) value '"balance":-125.53,'.
      3 pic x(17) value '"billing-info":{''.
      3 pic x(22) value '"name-first":"John",''.
      3 pic x(22) value '"name-last":"Smith",''.
      3 pic x(37) value '"addr-street":"12345 First Avenue",''.
      3 pic x(25) value '"addr-city":"New York",''.
      3 pic x(27) value '"addr-region":"New York",''.
      3 pic x(21) value '"addr-code":"10203"''.
      3 pic x(3) value '}''.
      3 pic x(2) value '}''.
      3 pic x(1) value '}''.
    1 jtxt-1047-transactions.
      3 pic x(16) value '{"transactions":['.
      3 pic x(14) value ' {"tx-record":['.
      3 pic x(3) value '['.
      3 pic x(4) value '{'.
      3 pic x(19) value '"tx-uid":107,'.
      3 pic x(34) value '"tx-item-desc":"prod a ver 1",''.
      3 pic x(30) value '"tx-item-uid":"ab142424",''.
      3 pic x(26) value '"tx-priceinUS$":12.34,'.
      3 pic x(35) value '"tx-comment":"express shipping"''.
      3 pic x(5) value '},''.
      3 pic x(4) value '{'.
      3 pic x(19) value '"tx-uid":1904,'.
      3 pic x(35) value '"tx-item-desc":"prod g ver 2",''.
      3 pic x(30) value '"tx-item-uid":"gb051533",''.
      3 pic x(27) value '"tx-priceinUS$":833.22,'.
      3 pic x(35) value '"tx-comment":"digital download"''.
      3 pic x(5) value '},''.
      3 pic x(3) value ']''.
      3 pic x(2) value '}''.
      3 pic x(1) value '}''.
    1 jtxt-1208 pic x(1000) value is all x'20'.
    77 txnum pic 999999 usage display value zero.
    1 client-data.
      3 account-num pic 999,999,999,999.
      3 balance pic $$$9.99CR.
      3 billing-info.
        5 name-first pic n(20).
        5 name-last pic n(20).
        5 addr-street pic n(20).
        5 addr-city pic n(20).
        5 addr-region pic n(20).
        5 addr-code pic n(10).
    3 transactions.

```

```

5 tx-record occurs 0 to 100 depending txnum.
7 tx-uid          pic 99999 usage display.
7 tx-item-desc   pic x(50).
7 tx-item-uid    pic AA/9999B99.
7 tx-price       pic $$$9.99.
7 tx-comment     pic n(20).
Procedure division.
  Initialize jtxt-1208 all value.
*****
* In this example, input to JSON PARSE starts in EBCDIC codepage *
* 1047 and then be converted to UTF-8 (codepage 1208).          *
* Convert to specific codepages using the display-of function.  *
* The first argument to display-of should be type 'national',   *
* which the COBOL compiler represents in UTF-16.                *
*****
  Move function display-of(
    function national-of(
      jtxt-1047-client-data) 1208)
    to jtxt-1208(1:function length(jtxt-1047-client-data)).

  Json parse jtxt-1208 into client-data
  with detail
  suppress transactions
  not on exception
  display "Successful JSON Parse"
end-json.

  Display "Account Number:"
  Display " " account-num
  Display "Balance:"
  Display " " balance
  Display "Client Information: "
  Display " Name:"
  Display " " function display-of(name-last)
  Display " " function display-of(name-first)
  Display " Address:"
  Display " " function display-of(addr-street)
  Display " " function display-of(addr-city)
  Display " " function display-of(addr-region)
  Display " " function display-of(addr-code).

  Move 2 to txnum.
  Initialize jtxt-1208 all value.
  Move function display-of(
    function national-of(
      jtxt-1047-transactions) 1208)
    to jtxt-1208(1:function length(jtxt-1047-transactions)).

  Json parse jtxt-1208 into transactions
  with detail
  name tx-price is 'tx-priceinUS$'
  not on exception
  display "Successful JSON Parse"
end-json.

  Display "Transactions:"
  Display " Record 1:"
  Display " TXID: " tx-uid(1)
  Display " Description: " tx-item-desc(1)
  Display " Item ID: " tx-item-uid(1)
  Display " Price: " tx-price(1)
  Display " Comment: "
    function display-of(tx-comment(1))
  Display " Record 2:"
  Display " TXID: " tx-uid(2)
  Display " Description: " tx-item-desc(2)
  Display " Item ID: " tx-item-uid(2)
  Display " Price: " tx-price(2)
  Display " Comment: "
    function display-of(tx-comment(2))

  Goback.
End program jp_ex.

```

The output of the program is:

```

Successful JSON Parse
Account Number:
  123,456,789,012
Balance:

```



```
$125.53CR
Client Information:
  Name:
    Smith
    John
  Address:
    12345 First Avenue
    New York
    New York
    10203
Successful JSON Parse
Transactions:
  Record 1:
    TXID:      00107
    Description: prod a ver 1
    Item ID:   ab/1424 24
    Price:     $12.34
    Comment:   express shipping
  Record 2:
    TXID:      01904
    Description: prod g ver 2
    Item ID:   gb/0515 33
    Price:     $833.22
    Comment:   digital download
```

Related references

JSON PARSE statement (*Enterprise COBOL for z/OS Language Reference*)

Chapter 36. Producing JSON output

You can express COBOL data items as JSON text by using the JSON GENERATE statement, which identifies the source and output data items.

You can optionally also specify:

- A data item to receive the count of characters generated.
- Alternative names for the input data items
- Data items to be excluded from the output JSON text
- Data items to be converted into JSON BOOLEAN name/value pairs
- A statement to receive control if an exception occurs

The JSON text can be used to represent a resource for the interface to a Web service, and is encoded in UTF-8 if the output data item is alphanumeric or UTF-8, or UTF-16 if the output data item is national.

Using the JSON GENERATE statement

Consider the following example:

```
01 Greeting.
02 Msg pic x(80) value 'Hello, World!'.
01 Jtext national pic n(80).
01 i binary pic 99.
...
JSON generate Jtext from Greeting count in i
on exception
display 'JSON generation error: ' json-code
not on exception
display function display-of(Jtext(1:i))
End-JSON
```

The above code sequence produces the following output:

```
{"Greeting": {"msg": "Hello, World!"}}
```

The following example is more complex which illustrates optional phrases that:

- Provide alternative JSON names for the included data items (NAME)
- Allow you to exclude sensitive or unwanted information from the output (SUPPRESS)

```
01 GRP.
05 Ac-No PIC AA9999 value 'SX1234'.
05 More.
10 Stuff PIC S99V9 OCCURS 2.
05 SSN PIC 999/99/9999 value '987-65-4321'.
01 d pic x(80).
01 i binary pic 99.
...
move 7.8 to stuff(1), move -9 to stuff(2)
JSON generate d from grp count i
NAME of stuff is 'Value' SUPPRESS ssn
display function display-of(function national-of(
d(1:i) 1208))
```

The example produces the following output:

```
{"GRP": {"Ac-No": "SX1234", "More": {"Value": [7.8, -9.0]}}}
```

Generating JSON anonymous arrays

You can generate JSON anonymous arrays by specifying the data name of the table on the JSON GENERATE statement while using the NAME IS OMITTED phrase.

Consider the following example:

```

Identification division.
Program-id. myprog.
Data division.
Working-storage section.
  1 ACT.
    2 B1 occurs 2.
      3 C1.
        4 M1 pic 9.
        4 D1 occurs 2.
          5 N1 pic 9.
    1 json-text pic u dynamic.
Procedure division.
  move 1 to M1(1)
  move 2 to M1(2)
  move 3 to N1(1 1)
  move 4 to N1(1 2)
  move 5 to N1(2 1)
  move 6 to N1(2 2)
  move spaces to json-text
  json generate json-text from b1
    name b1 is omitted
  end-json
  goback
  .
End program myprog.

```

Running the program will produce the following JSON text in the item `json-text`:

```
[{"C1": {"M1": 1, "D1": [{"N1": 3}, {"N1": 4}]}, {"C1": {"M1": 2, "D1": [{"N1": 5}, {"N1": 6}]}}
```

Note that the JSON GENERATE sender `b1` was not subscripted in order to refer to all occurrences of `b1`.

Consider the next example where the table `b1` has two dimensions and is nested within another table like below:

```

Identification division.
Program-id. myprog.
Data division.
Working-storage section.
  1 ACT.
    2 TOPTABLE occurs 3.
      3 B1 occurs 2.
        4 C1.
          5 M1 pic 9.
          5 D1 occurs 2.
            6 N1 pic 9.
    1 json-text pic u dynamic.
Procedure division.
  move 1 to M1(1 1)
  move 2 to M1(1 2)
  move 3 to N1(1 1 1)
  move 4 to N1(1 1 2)
  move 5 to N1(1 2 1)
  move 6 to N1(1 2 2)
  move spaces to json-text
  json generate json-text from b1(1)
    name b1 is omitted
  end-json
  goback
  .
End program myprog.

```

Running the program will produce the following JSON text in the item `json-text`:

```
[{"C1": {"M1": 1, "D1": [{"N1": 3}, {"N1": 4}]}, {"C1": {"M1": 2, "D1": [{"N1": 5}, {"N1": 6}]}}
```

Note that `b1(1)` was specified with a single index subscript on the JSON GENERATE statement. The value of the subscript `1` indicates that the first occurrence of table `TOPTABLE` shall be generated. The omission of a second index indicates to the JSON GENERATE statement that the statement shall operate on the entire table `b1` (rather than operating on a single occurrence of `b1`, which is what would have occurred if a second index subscript was specified). This pattern applies generally to multiple nested tables, that is, an anonymous array can be generated by the combination of the NAME IS OMITTED phrase and specifying the sending item with one less subscript than its dimension.

Generating JSON null values

This section describes the ways to generate JSON null values using the JSON GENERATE statement.

You can use either of the following ways to generate JSON null values:

- Using the CONVERTING phrase to convert certain values of COBOL data items to JSON null.
- Using the INDICATING phrase to assign a satellite data item as a null indicator.

Details of each way are as follows.

Converting certain values of COBOL data items to JSON null

Use the CONVERTING phrase when you want certain values of COBOL data items to represent JSON null. For example, consider the following COBOL source:

```
CBL CODEPAGE(1047)
Identification division.
  Program-id. myprog.
Data division.
  Working-storage section.
    1 json-text pic u dynamic.
    1 A.
    2 B-data PIC X(10).
  Procedure division.
    move space to B-data
    set length of json-text to 0
    json generate json-text from A
      converting B-data to json null using space
    display function display-of(json-text 1047)
    goback.
  End program myprog.
```

When compiled and run, the JSON GENERATE statement converts B-data to a JSON null when B-data contains spaces. The output of the program is as follows:

```
{"A":{"B-data":null}}
```

Note that not all items can be specified on the CONVERTING phrase due to the MOVE compatibility rules.

Assigning a satellite data item as a null indicator

Use the INDICATING phrase in the case where you can or want to designate a satellite data item as a null indicator. For example, consider the following COBOL source:

```
`CBL CODEPAGE(1047)`
Identification division.
  Program-id. myprog.
Data division.
  Working-storage section.
    1 json-text pic u dynamic.
    1 A.
    2 B-data pic x(10).
    2 B-indicator pic x.
  Procedure division.
    move "Y" to B-indicator
    json generate json-text from A
      indicating B-data is json null
      using 'Y' in B-indicator
    display function display-of(json-text 1047)
    goback.
  End program myprog.
```

When compiled and run, the JSON GENERATE statement generates a JSON null if B-indicator contains 'Y', as specified on the INDICATING phrase. The output of the program is as follows:

```
{"A":{"B-data":null}}
```

Chapter 37. Processing XML input

You can process XML input in a COBOL program by using the XML PARSE statement.

The XML PARSE statement is the COBOL language interface to either of two high-speed XML parsers. You use the XMLPARSE compiler option to select the appropriate parser for your application:

- XMLPARSE (XMLSS) selects the z/OS XML System Services parser.

This option provides enhanced features such as namespace processing, validation of XML documents with respect to an XML schema, and conversion of text fragments to national character representation (Unicode UTF-16).

- XMLPARSE (COMPAT) selects the XML parser that is built into the COBOL library.

This option provides compatibility with XML parsing in Enterprise COBOL 3 and 4.

Processing XML input involves passing control between the XML parser and a processing procedure in which you handle parser events.

Use the following COBOL facilities to process XML input:

- The XML PARSE statement to begin XML parsing and to identify the source XML document and the processing procedure.

You can also use the following optional phrases of the XML PARSE statement:

- ENCODING to specify the encoding of the XML document
- VALIDATING to identify an XML schema against which the XML document is to be validated
- The processing procedure to control the parsing, that is, receive and process XML events and associated document fragments, and return to the parser for continued processing
- Special registers to exchange information between the parser and the processing procedure:
 - XML - CODE to receive the status of XML parsing and, in some cases, to return information to the parser
 - XML - EVENT to receive the name of each XML event from the parser
 - XML - INFORMATION provides a mechanism to easily determine whether an XML event is complete
 - XML - NTEXT to receive XML document fragments that are returned as national character data
 - XML - TEXT to receive document fragments that are returned as alphanumeric data
 - XML - NAMESPACE or XML - NNAMESPACE to receive a namespace identifier for a NAMESPACE-DECLARATION XML event, or for an element name or attribute name that is in a namespace
 - XML - NAMESPACE - PREFIX or XML - NNAMESPACE - PREFIX to receive a namespace prefix for a NAMESPACE-DECLARATION XML event, or for an element name or attribute name that is prefixed
- The optional RETURNING NATIONAL phrase of the XML PARSE statement to indicate that the fragments of an XML document in an alphanumeric data item are to be converted to UTF-16 and returned to the processing procedure in the national special registers XML - NTEXT, XML - NNAMESPACE, and XML - NNAMESPACE - PREFIX

You can use the ENCODING, VALIDATING, and RETURNING NATIONAL phrases of the XML PARSE statement only if XMLPARSE (XMLSS) is in effect.

Link-edit consideration: COBOL programs that contain the XML PARSE statement must be link-edited with AMODE 31.

Related concepts

[“XML parser in COBOL” on page 612](#)

Related tasks

[“Accessing XML documents” on page 613](#)

[“Parsing XML documents” on page 613](#)
[“Handling XML PARSE exceptions” on page 633](#)
[“Terminating XML parsing” on page 637](#)

Related references

[“XMLPARSE” on page 415 \(compiler option\)](#)

[“The encoding of XML documents” on page 628](#)
[Appendix C, “XML reference material,” on page 795](#)
[Extensible Markup Language \(XML\)](#)

XML parser in COBOL

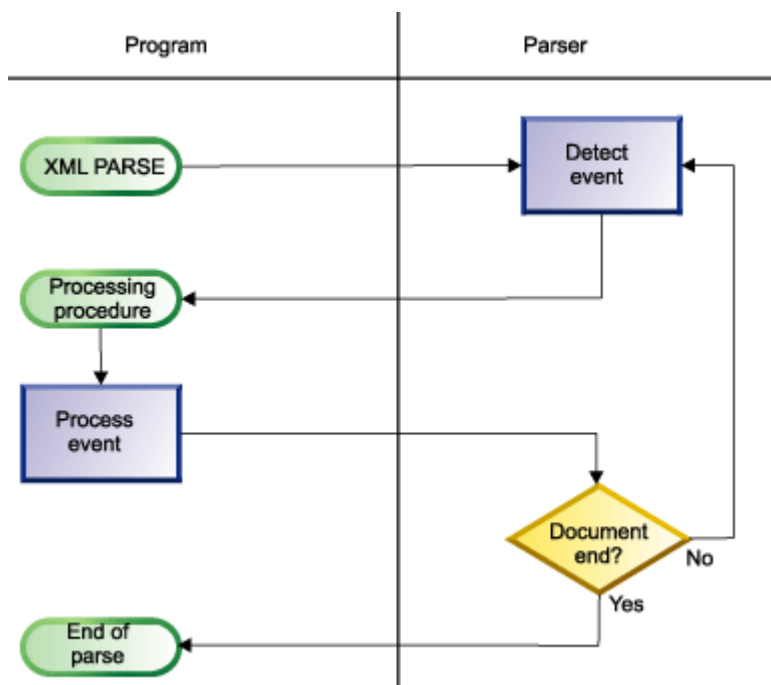
Enterprise COBOL provides an event-based interface that lets you parse XML documents and transform them to COBOL data structures.

The XML parser finds fragments within the source XML document, and your processing procedure acts on those fragments. The fragments are associated with XML events; you code the processing procedure to handle each XML event.

Execution of the XML PARSE statement begins the parsing and establishes the processing procedure with the parser. The parser transfers control to the processing procedure for each XML event that it detects while processing the document. After processing the event, the processing procedure automatically returns control to the parser. Each normal return from the processing procedure causes the parser to continue analyzing the XML document to report the next event. Throughout this operation, control passes back and forth between the parser and the processing procedure.

In the XML PARSE statement, you can also specify two imperative statements to which you want control to be passed at the end of the parsing: one if a normal end occurs, and the other if an exception condition exists.

The following figure shows a high-level overview of the basic exchange of control between the parser and your COBOL program:



Normally, parsing continues until the entire XML document has been parsed.

The XML parser checks XML documents for most aspects of well formedness. A document is *well formed* if it adheres to the XML syntax in the *XML specification* and follows some additional rules such as proper use of end tags and uniqueness of attribute names.

When you parse an XML document with validation against an XML schema, the z/OS XML System Services parser additionally verifies that the XML document adheres to the content and structure prescribed in the schema. For example, the parser checks that there are no unexpected elements or attributes, that no required elements or attributes are missing, and that any values of elements or attributes are legal.

Related concepts

[“XML schemas” on page 624](#)

[“XML input document encoding” on page 629](#)

Related tasks

[“Parsing XML documents” on page 613](#)

[“Parsing XML documents with validation” on page 622](#)

[“Handling XML PARSE exceptions” on page 633](#)

[“Terminating XML parsing” on page 637](#)

Related references

[“The encoding of XML documents” on page 628](#)

[XML specification](#)

Accessing XML documents

Before you can parse an XML document using an XML PARSE statement, you must make the document available to your program. Common methods of acquiring an XML document are by retrieval from a WebSphere MQ message, a CICS transient queue or communication area, or an IMS message processing queue; or by reading the document from a file.

If the XML document that you want to parse is held in a file, use ordinary COBOL facilities to place the document into a data item in your program:

- A FILE-CONTROL entry to define the file to your program.
- An OPEN statement to open the file.
- READ statements to read all the records from the file into a data item (either an elementary item of category alphanumeric or national, or an alphanumeric or national group). You can define the data item in the WORKING-STORAGE SECTION or the LOCAL-STORAGE SECTION.
- Optionally, the STRING statement to string all of the separate records together into one continuous stream, to remove extraneous blanks, and to handle variable-length records.

If the XMLPARSE (XMLSS) option is in effect, you can parse an XML document that is in a file by passing the parser one record (or *segment*) of text from the file at a time. This capability is useful for parsing very large XML documents.

Related tasks

[“Coding COBOL programs to run under CICS” on page 483](#)

[Chapter 24, “Developing COBOL programs for IMS,” on page 505](#)

[“Parsing XML documents one segment at a time” on page 625](#)

Related references

[“XMLPARSE” on page 415](#) (compiler option)

Parsing XML documents

To parse XML documents, use the XML PARSE statement, specifying the XML document that is to be parsed and the processing procedure for handling XML events that occur during parsing, as shown in the following code fragment.

```
XML PARSE xml-document
```

```

PROCESSING PROCEDURE xml-event-handler
ON EXCEPTION
  DISPLAY 'XML document error ' XML-CODE
  STOP RUN
NOT ON EXCEPTION
  DISPLAY 'XML document was successfully parsed.'
END-XML

```

In the XML PARSE statement, you first identify the *parse data item* (xml-document in the example above) that contains the XML document character stream. In the DATA DIVISION, define the parse data item as an elementary data item of category national or as a national group item if the encoding of the document is Unicode UTF-16; otherwise, define the parse data item as an elementary alphanumeric data item or an alphanumeric group item:

- If the parse data item is national, the XML document must be encoded in UTF-16, CCSID 1200.
- If the parse data item is alphanumeric, its content must be encoded in one of the supported code pages described in the related reference about the encoding of XML documents.

Next, specify the name of the processing procedure (xml-event-handler in the example above) that is to handle the XML events that occur during parsing of the document.

If the XMLPARSE (XMLSS) compiler option is in effect, you can also use any of these optional phrases of the XML PARSE statement:

- ENCODING, to specify the CCSID of the document
- RETURNING NATIONAL, to cause the parser to automatically convert UTF-8 or single-byte characters to national characters for return to the processing procedure
- VALIDATING, to cause the parser to validate the document against an XML schema

In addition, you can specify either or both of the following optional phrases (as shown in the fragment above) to indicate the action to be taken after parsing finishes:

- ON EXCEPTION, to receive control if an unhandled exception occurs during parsing
- NOT ON EXCEPTION, to receive control otherwise

You can end the XML PARSE statement with the explicit scope terminator END-XML. Use END-XML to nest an XML PARSE statement that uses the ON EXCEPTION or NOT ON EXCEPTION phrase in a conditional statement.

The parser passes control to the processing procedure for each XML event. Control returns to the parser at the end of the processing procedure. This exchange of control between the XML parser and the processing procedure continues until one of the following events occurs:

- The entire XML document was parsed, as indicated by the END-OF-DOCUMENT event.
- If XMLPARSE (XMLSS) is in effect, either:
 - The parser detects an error in the document and signals an EXCEPTION event (regardless of the kind of exception).
 - The parser signals an END-OF-INPUT event, and the processing procedure returns to the parser with special register XML-CODE still set to zero, which indicates that no further XML data will be provided to the parser.
- If XMLPARSE (COMPAT) is in effect, either:
 - The parser signals an encoding conflict EXCEPTION event, and the processing procedure does not reset special register XML-CODE to zero or to the correct CCSID before returning to the parser.
 - The parser detects an error in the document and signals an EXCEPTION event (other than an encoding conflict), and the processing procedure does not reset special register XML-CODE to zero before returning to the parser.
- The parsing process is terminated deliberately by the user's code in the processing procedure that sets the XML-CODE special register to -1 before it returns to the parser.

Related concepts

[“XML events” on page 617](#)

[“XML-CODE” on page 618](#)

[“XML schemas” on page 624](#)

[“XML-INFORMATION” on page 620](#)

Related tasks

[“Writing procedures to process XML” on page 615](#)

[“Parsing XML documents with validation” on page 622](#)

[“Parsing XML documents one segment at a time” on page 625](#)

[“Parsing XML documents encoded in UTF-8” on page 632](#)

Related references

[“XMLPARSE” on page 415 \(compiler option\)](#)

[“The encoding of XML documents” on page 628](#)

[“XML PARSE exceptions with XMLPARSE\(XMLSS\) in effect” on page 795](#)

[“XML PARSE exceptions with XMLPARSE\(COMPAT\) in effect” on page 797](#)

XML PARSE statement (*Enterprise COBOL for z/OS Language Reference*)

Writing procedures to process XML

In your processing procedure, code statements to handle XML events.

For each event that the parser encounters, the parser passes information to the processing procedure in several special registers. Use the content of those special registers to populate COBOL data structures and to control the processing.

Examine the XML - EVENT special register to determine which event the parser passed to the processing procedure. XML - EVENT contains an event name, such as 'START-OF-ELEMENT'. Obtain the text associated with the event from the XML - TEXT or XML - NTEXT special register.

If the XMLPARSE (XMLSS) option is in effect, you can use special register XML - NAMESPACE or XML - NAMESPACE to determine the namespace identifier, if any, that is associated with the XML event, and examine the XML - NAMESPACE - PREFIX or XML - NAMESPACE - PREFIX special register to determine the associated prefix, if any.

When used in nested programs, the XML special registers are implicitly defined as GLOBAL in the outermost program.

For additional details about the XML special registers, see the following table.

Special register	Implicit definition and usage	Content
XML - EVENT ¹	PICTURE X(30) USAGE DISPLAY VALUE SPACE	The name of the XML event
XML - CODE ²	PICTURE S9(9) USAGE BINARY VALUE ZERO	An exception code or zero for each XML event
XML - INFORMATION ¹	PICTURE S9(9) USAGE BINARY VALUE 0	A mechanism to easily determine whether an XML EVENT is complete
XML - TEXT ¹	Variable-length elementary category alphanumeric item	Text (corresponding to the event that the parser encountered) from the XML document if you specify an alphanumeric item for the XML PARSE identifier ³
XML - NTEXT ¹	Variable-length elementary category national item	Text (corresponding to the event that the parser encountered) from the XML document if you specify a national item for the XML PARSE identifier ³

Table 74. *Special registers used by the XML parser (continued)*

Special register	Implicit definition and usage	Content
XML -NAMESPACE ^{1, 4}	Variable-length elementary category alphanumeric item	The namespace identifier for a NAMESPACE - DECLARATION XML event or for an element or attribute name that is in a namespace, if the XML document is in an alphanumeric data item ³
XML -NNAMESPACE ^{1, 4}	Variable-length elementary category national item	The namespace identifier for a NAMESPACE - DECLARATION XML event or for an element or attribute name that is in a namespace, if the XML document is in a national data item or the RETURNING NATIONAL phrase is specified in the XML PARSE statement
XML -NAMESPACE -PREFIX ^{1, 4}	Variable-length elementary category national item	The prefix, if any, for a NAMESPACE - DECLARATION XML event or for an element or attribute name that is in a nondefault namespace, if the XML document is in an alphanumeric data item ³
XML -NNAMESPACE -PREFIX ^{1, 4}	Variable-length elementary category national item	The prefix, if any, for a NAMESPACE - DECLARATION XML event or for an element or attribute name that is in a nondefault namespace, if the XML document is in a national data item or the RETURNING NATIONAL phrase is specified in the XML PARSE statement

1. You cannot use this special register as a receiving data item.

2. The XML GENERATE statement also uses XML -CODE. Therefore, if you have an XML GENERATE statement in the processing procedure, save the value of XML -CODE before the XML GENERATE statement, and restore the saved value after the XML GENERATE statement.

3. If you specify the RETURNING NATIONAL phrase in the XML PARSE statement for an alphanumeric data item, text is returned in the corresponding national special register. You can specify the RETURNING NATIONAL phrase only if the XMLPARSE (XMLSS) option is in effect.

4. The parser sets the namespace special registers only if the XMLPARSE (XMLSS) option is in effect.

Restrictions:

- A processing procedure must not directly execute an XML PARSE statement. However, if a processing procedure passes control to a method or outermost program by using an INVOKE or CALL statement, the target method or program can execute the same or a different XML PARSE statement. You can also execute the same XML statement or different XML statements simultaneously from a program that is running on multiple threads.
- The range of the processing procedure must not cause the execution of any GOBACK or EXIT PROGRAM statement, except to return control from a method or program to which control was passed by an INVOKE or CALL statement, respectively, that is executed in the range of the processing procedure.

You can code a STOP RUN statement in a processing procedure to end the run unit.

The compiler inserts a return mechanism after the last statement in each processing procedure.

[“Example: program for processing XML” on page 638](#)

Related concepts

[“XML events” on page 617](#)

[“XML-CODE” on page 618](#)

[“XML-TEXT and XML-NTEXT” on page 620](#)

[“XML-NAMESPACE and XML-NNAMESPACE” on page 621](#)

[“XML-NAMESPACE-PREFIX and XML-NNAMESPACE-PREFIX” on page 621](#)

Related tasks

[“Parsing XML documents one segment at a time” on page 625](#)

[“Parsing XML documents with validation” on page 622](#)

[“Terminating XML parsing” on page 637](#)

Related references

[“XMLPARSE” on page 415](#) (compiler option)
[XML-EVENT](#) (*Enterprise COBOL for z/OS Language Reference*)

XML events

An *XML event* results when the XML parser detects various conditions (such as END-OF-INPUT or EXCEPTION) or encounters document fragments (such as CONTENT-CHARACTERS or START-OF-CDATA-SECTION) while processing an XML document.

For each event that occurs during XML parsing, the parser sets the associated event name in the XML-EVENT special register, and passes the XML-EVENT special register to the processing procedure. Depending on the event, the parser sets other special registers to contain additional information about the event.

In most cases, the parser sets the XML-TEXT or XML-NTEXT special register to the XML fragment that caused the event:

- If the XMLPARSE (COMPAT) compiler option is in effect, the parser sets XML-NTEXT if the XML document is in a national data item, or if the parser finds a character reference; otherwise, the parser sets XML-TEXT.
- If XMLPARSE (XMLSS) is in effect, the parser sets XML-NTEXT if the RETURNING NATIONAL phrase is specified in the XML PARSE statement, or if the XML document is in a national data item; otherwise, the parser sets XML-TEXT.

If XMLPARSE (XMLSS) is in effect, the parser sets the namespace special registers for a NAMESPACE-DECLARATION event, or if it encounters a name that is in a namespace.

When the parser detects an encoding conflict or a well-formedness or validation error in the document, it sets XML-EVENT to 'EXCEPTION' and provides additional information about the exception in the XML-CODE special register. You can parse with validation only if XMLPARSE (XMLSS) is in effect. For further details, see the related task about parsing with validation.

For a detailed description of the set of XML events, see the related reference about XML-EVENT.

[“Example: parsing a simple document” on page 637](#)

Related concepts

[“XML parser in COBOL” on page 612](#)
[“XML-CODE” on page 618](#)
[“XML-INFORMATION” on page 620](#)
[“XML-TEXT and XML-NTEXT” on page 620](#)
[“XML-NAMESPACE and XML-NNAMESPACE” on page 621](#)
[“XML-NAMESPACE-PREFIX and XML-NNAMESPACE-PREFIX” on page 621](#)

Related tasks

[“Writing procedures to process XML” on page 615](#)
[“Parsing XML documents with validation” on page 622](#)

Related references

[“XMLPARSE” on page 415](#) (compiler option)

[“XML PARSE exceptions with XMLPARSE\(XMLSS\) in effect” on page 795](#)
[“XML PARSE exceptions with XMLPARSE\(COMPAT\) in effect” on page 797](#)
[XML-EVENT](#) (*Enterprise COBOL for z/OS Language Reference*)

XML-CODE

For each XML event except an EXCEPTION event, the parser sets the value of the XML - CODE special register to zero. For an EXCEPTION event, the parser sets XML - CODE to a value that identifies the specific exception.

For information about the possible exception codes, see the related references.

When the parser returns control to the XML PARSE statement from your processing procedure, XML - CODE generally contains the most recent value that was set by the parser. However, for any event other than EXCEPTION, if you set XML - CODE to -1 in your processing procedure, parsing terminates with a user-initiated exception condition when control returns to the parser, and XML - CODE retains the value -1.

For an EXCEPTION XML event when XMLPARSE (COMPAT) is in effect, your processing procedure can, in some cases, set XML - CODE to a meaningful value before control returns to the parser. (For details, see the related tasks about handling XML PARSE exceptions and handling encoding conflicts.) If you set XML - CODE to any other nonzero value or set it for any other exception, the parser resets XML - CODE to the original exception code.

For a START - OF - DOCUMENT XML event when compiler option XMLPARSE (COMPAT) is in effect, your processing procedure can set XML - CODE to 1 before control returns to the parser. This action instructs the parser to release (at the end of parsing) any Language Environment resources acquired during parsing.

The following table shows the results of setting XML - CODE to various values. The leftmost column shows the type of XML event passed to the processing procedure; the other column headings show the XML - CODE value set by the processing procedure. The cell at the intersection of each row and column shows the action that the parser takes upon return from the processing procedure for a given combination of XML event and XML - CODE value.

XML event type	XML-CODE set to -1	XML-CODE set to 0	XML-CODE set to 1	XML-CODE set to other nonzero values
Fatal EXCEPTION	Ignores setting; keeps original XML - CODE value	Ignores setting; keeps original XML - CODE value	Ignores setting; keeps original XML - CODE value	Ignores setting; keeps original XML - CODE value
Warning EXCEPTION (Reason code 800 or 801)	Ignores setting; keeps original XML - CODE value	Next event is ATTRIBUTE - NAME or START - OF - ELEMENT	Ignores setting; keeps original XML - CODE value	Ignores setting; keeps original XML - CODE value
END - OF - INPUT	Ends immediately; XML - CODE = -1 ¹	Next event is END - OF - DOCUMENT ²	Next event depends on input ²	Fatal runtime error (message 230S)
Normal event	Ends immediately; XML - CODE = -1 ¹	XML - CODE already 0, no change	Fatal runtime error (message 230S)	Fatal runtime error (message 230S)
1. See the related task about terminating XML parsing. 2. See the related task about parsing documents one segment at a time.				

Table 76. Results of processing-procedure changes to XML - CODE with XMLPARSE (COMPAT) in effect

XML event type	-1	0	XML - CODE-100,000	Other nonzero value
Encoding-conflict exception (exception codes 50 - 99)	Ignores setting; keeps original XML - CODE value	Chooses encoding depending on the specific exception code ¹	Ignores setting; keeps original XML - CODE value	Ignores setting; keeps original XML - CODE value
Encoding-choice exception (exception codes > 100,000)	Ignores setting; keeps original XML - CODE value	Parses using the CODEPAGE value ²	Parses using the difference (shown above) as the encoding value ²	Ignores setting; keeps original XML - CODE value
Other exception	Ignores setting; keeps original XML - CODE value	Limited continuation only for exception codes 1 - 49 ³	Ignores setting; keeps original XML - CODE value	Ignores setting; keeps original XML - CODE value
Normal event (except START-OF-DOCUMENT)	Ends immediately; XML - CODE = -1 ⁴	[No apparent change to XML - CODE]	Ends immediately; XML - CODE = -1	Ends immediately; XML - CODE = -1
START-OF-DOCUMENT	Ends immediately; XML - CODE = -1 ⁴	[No apparent change to XML - CODE]	Ends immediately; XML - CODE = -1	<ul style="list-style-type: none"> • XML - CODE = 1 • Else ends immediately; XML - CODE = -1

1. See the exception codes in the related reference about XML PARSE exceptions with XMLPARSE (COMPAT) in effect.
2. See the related task about handling encoding conflicts.
3. See the related task about handling XML PARSE exceptions.
4. See the related task about terminating XML parsing.

XML generation also uses the XML - CODE special register. For details, see the related task about handling XML GENERATE exceptions.

Related concepts

[“How the XML parser handles errors” on page 635](#)

Related tasks

[“Writing procedures to process XML” on page 615](#)

[“Parsing XML documents one segment at a time” on page 625](#)

[“Handling XML PARSE exceptions” on page 633](#)

[“Terminating XML parsing” on page 637](#)

[“Handling XML GENERATE exceptions” on page 656](#)

Related references

[“XML PARSE exceptions with XMLPARSE\(XMLSS\) in effect” on page 795](#)

[“XML PARSE exceptions with XMLPARSE\(COMPAT\) in effect” on page 797](#)

[“XML GENERATE exceptions” on page 803](#)

XML-CODE (*Enterprise COBOL for z/OS Language Reference*)

XML-EVENT (*Enterprise COBOL for z/OS Language Reference*)

XML-INFORMATION

For most XML events, the parser sets XML - INFORMATION to indicate whether an XML EVENT is complete or whether the XML content spans multiple events.

The application program logic can use the XML - INFORMATION special register to concatenate pieces of parsed XML content together.

Related concepts

[“XML events” on page 617](#)

[“XML-CODE” on page 618](#)

Related tasks

[“Writing procedures to process XML” on page 615](#)

Related references

XML-TEXT (*Enterprise COBOL for z/OS Language Reference*)

XML-NTEXT (*Enterprise COBOL for z/OS Language Reference*)

XML-TEXT and XML-NTEXT

For most XML events, the parser sets XML - TEXT or XML - NTEXT to an associated document fragment.

Typically, the parser sets XML - TEXT if the XML document is in an alphanumeric data item. The parser sets XML - NTEXT if:

- The XML document is in a national data item.
- The XMLPARSE (XMLSS) option is in effect and the RETURNING NATIONAL phrase is specified in the XML PARSE statement.
- The ATTRIBUTE - NATIONAL - CHARACTER or CONTENT - NATIONAL - CHARACTER event occurs.

The special registers XML - TEXT and XML - NTEXT are mutually exclusive. When the parser sets XML - TEXT, XML - NTEXT is empty with length zero. When the parser sets XML - NTEXT, XML - TEXT is empty with length zero.

To determine the number of character encoding units in XML - NTEXT, use the LENGTH intrinsic function; for example FUNCTION LENGTH (XML - NTEXT). To determine the number of bytes in XML - NTEXT, use special register LENGTH OF XML - NTEXT. The number of character encoding units differs from the number of bytes.

To determine the number of bytes in XML - TEXT, use either special register LENGTH OF XML - TEXT or the LENGTH intrinsic function; each returns the number of bytes.

The XML - TEXT and XML - NTEXT special registers are undefined outside the processing procedure.

Related concepts

[“XML events” on page 617](#)

[“XML-CODE” on page 618](#)

Related tasks

[“Writing procedures to process XML” on page 615](#)

Related references

[“XMLPARSE” on page 415 \(compiler option\)](#)

XML-TEXT (*Enterprise COBOL for z/OS Language Reference*)

XML-NTEXT (*Enterprise COBOL for z/OS Language Reference*)

XML-NAMESPACE and XML-NNAMESPACE

If the XMLPARSE (XMLSS) option is in effect, the XML parser sets the XML - NAMESPACE or XML - NNAMESPACE special register to the namespace identifier for a NAMESPACE - DECLARATION XML event, or if it encounters an element name or attribute name that is in a namespace.

The parser sets XML - NNAMESPACE if the XML document is in a national data item, or if the RETURNING NATIONAL phrase is specified in the XML PARSE statement. Otherwise, the parser sets XML - NAMESPACE.

The special registers XML - NAMESPACE and XML - NNAMESPACE are mutually exclusive: If the parser sets XML - NAMESPACE, XML - NNAMESPACE is empty with length zero. If the parser sets XML - NNAMESPACE, XML - NAMESPACE is empty with length zero.

To determine the number of character encoding units in XML - NNAMESPACE, use the LENGTH intrinsic function; for example: FUNCTION LENGTH (XML - NNAMESPACE). To determine the number of bytes in XML - NNAMESPACE, use special register LENGTH OF XML - NNAMESPACE. The number of character encoding units differs from the number of bytes.

To determine the number of bytes in XML - NAMESPACE, use either special register LENGTH OF XML - NAMESPACE or the LENGTH intrinsic function; each returns the number of bytes.

The XML namespace special registers are undefined outside the processing procedure.

Related concepts

[“XML events” on page 617](#)

[“XML-CODE” on page 618](#)

[“XML-NAMESPACE-PREFIX and XML-NNAMESPACE-PREFIX” on page 621](#)

[“XML-TEXT and XML-NTEXT” on page 620](#)

Related tasks

[“Writing procedures to process XML” on page 615](#)

Related references

[“XMLPARSE” on page 415 \(compiler option\)](#)

XML-NAMESPACE-PREFIX and XML-NNAMESPACE-PREFIX

If the XMLPARSE (XMLSS) option is in effect, the XML parser sets the XML - NAMESPACE - PREFIX special register or the XML - NNAMESPACE - PREFIX special register for a NAMESPACE - DECLARATION XML event that also defines a namespace prefix, or if an element name or attribute name in a namespace is prefixed.

The parser sets XML - NNAMESPACE - PREFIX if the XML document is in a national data item, or the RETURNING NATIONAL phrase is specified in the XML PARSE statement. Otherwise, the parser sets XML - NAMESPACE - PREFIX.

The special registers XML - NAMESPACE - PREFIX and XML - NNAMESPACE - PREFIX are mutually exclusive: If the parser sets XML - NAMESPACE - PREFIX, XML - NNAMESPACE - PREFIX is empty with length zero. If the parser sets XML - NNAMESPACE - PREFIX, XML - NAMESPACE - PREFIX is empty with length zero.

To determine the number of character encoding units in XML - NNAMESPACE - PREFIX, use the LENGTH intrinsic function; for example: FUNCTION LENGTH (XML - NNAMESPACE - PREFIX). To determine the number of bytes in XML - NNAMESPACE - PREFIX, use special register LENGTH OF XML - NNAMESPACE - PREFIX. The number of character encoding units differs from the number of bytes.

To determine the number of bytes in XML - NAMESPACE - PREFIX, use either special register LENGTH OF XML - NAMESPACE - PREFIX or the LENGTH intrinsic function; each returns the number of bytes.

The XML namespace-prefix special registers are undefined outside the processing procedure.

Related concepts

[“XML events” on page 617](#)

[“XML-NAMESPACE and XML-NNAMESPACE” on page 621](#)

Related tasks

[“Writing procedures to process XML” on page 615](#)

Related references

[“XMLPARSE” on page 415](#) (compiler option)

Transforming XML text to COBOL data items

Because XML data is neither fixed length nor fixed format, you need to use special techniques when you move XML data to a COBOL data item.

For alphanumeric items, decide whether the XML data should go at the left (default) end, or at the right end, of the COBOL data item. If the data should go at the right end, specify the `JUSTIFIED RIGHT` clause in the definition of the item.

Give special consideration to numeric XML values, particularly "decorated" monetary values such as '\$1,234.00' or '\$1234'. These two strings might mean the same thing in XML, but need quite different definitions if used as COBOL sending fields.

Use one of the following techniques when you move XML data to COBOL data items:

- If the format is reasonably regular, code a `MOVE` to an alphanumeric item that you redefine appropriately as a numeric-edited item. Then do the final move to a numeric (operational) item by moving from, and thus de-editing, the numeric-edited item. (A regular format would have the same number of digits after the decimal point, a comma separator for values greater than 999, and so on.)
- For simplicity and vastly increased flexibility, use the following intrinsic functions for alphanumeric XML data:
 - `NUMVAL` to extract and decode simple numeric values from XML data that represents plain numbers
 - `NUMVAL-C` to extract and decode numeric values from XML data that represents monetary quantities

However, using these functions is at the expense of performance.

Related tasks

[“Converting to numbers \(NUMVAL, NUMVAL-C, NUMVAL-F\)” on page 115](#)

[“Using national data \(Unicode\) in COBOL” on page 125](#)

[“Writing procedures to process XML” on page 615](#)

Parsing XML documents with validation

Validating an XML document determines whether the structure and content of the document conform to a set of rules. In Enterprise COBOL, the rules are expressed in an *XML schema*, which is essentially a blueprint for a class of documents.

To validate XML documents while parsing, use the `VALIDATING` phrase of the `XML PARSE` statement. To do so, you must compile your program using the `XMLPARSE (XMLSS)` compiler option.

You can validate XML documents only against an XML schema.

In Enterprise COBOL, a schema used for XML validation must be in a preprocessed format known as *Optimized Schema Representation*, or *OSR*. To generate a schema in OSR format from a text-form schema, use the `z/OS UNIX` command `xsdosrg`, which invokes the OSR generator provided by `z/OS System Services`. (Alternatively, you can call the OSR generator programmatically. For details, see the related reference about `z/OS XML System Services`.)

For example, to convert the text-form schema in file `item.xsd` to a schema in preprocessed format in file `item.osr`, you can use the following `z/OS UNIX` command:

```
xsdosrg -v -o /u/HLQ/xml/item.osr /u/HLQ/xml/item.xsd
```

Use one of two forms of the `VALIDATING` phrase, depending on the location of the preprocessed schema:

- In one form, you use the `FILE` keyword and specify an XML schema name. In this case, the schema must be in an MVS data set or a `z/OS UNIX` file.
- In the other form, you specify the identifier of a data item that contains the schema.

If you use the FILE keyword and specify an XML schema name, the COBOL runtime library automatically retrieves the schema during execution of the XML PARSE statement. The following code fragment shows this method of specifying validation:

```
XML PARSE document-item
  VALIDATING WITH FILE schema-name
  PROCESSING PROCEDURE xml-event-handler
  ON EXCEPTION
    DISPLAY 'Document has an error.'
    GOBACK
  NOT ON EXCEPTION
    DISPLAY 'Document is valid.'
END-XML
```

To associate an XML schema name with the external file that contains the schema, code the XML - SCHEMA clause in the SPECIAL - NAMES paragraph, specifying either a literal or a user-defined word to identify the file.

For example, you can associate the XML schema name schema - name shown in the fragment above with the ddname DDSHEMA by coding the ddname as a literal in the XML - SCHEMA clause as follows:

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
  XML-SCHEMA schema-name IS 'DDSHEMA'.
```

For running the program, you can associate ddname DDSHEMA with the z/OS UNIX file item.osr by coding a DD statement as follows:

```
//GO.DDSHEMA DD PATH='/u/HLQ/xml/item.osr'
```

Or you can use an analogous TSO ALLOCATE command.

Alternatively, DDSHEMA in the example above could be the name of an environment variable that identifies the external file by means of a DSN option that specifies an MVS data set or a PATH option that specifies a z/OS UNIX file.

If your schema is in an MVS data set, the data set can be any sequential data set (for example, QSAM fixed blocked or variable blocked, or VSAM ESDS).

For further details about how to associate an XML schema name with the external file that contains the schema, see the related reference about the XML - SCHEMA clause.

Restriction: XML validation using the FILE keyword is not supported under CICS.

The automatic retrieval that occurs when you use the FILE keyword is convenient. But if you have several XML documents of the same type to validate, reading the schema into memory once and then reusing the schema for each of the documents provides better performance than automatic retrieval. In this case, you use the other form of the VALIDATING phrase, in which you specify an identifier that references an alphanumeric data item that contains the XML schema. For example:

```
XML PARSE document-item
  VALIDATING WITH xmlschema
  PROCESSING PROCEDURE xml-event-handler
  ON EXCEPTION
    DISPLAY 'Document has an error.'
    GOBACK
  NOT ON EXCEPTION
    DISPLAY 'Document is valid.'
END-XML
```

Read the preprocessed schema into the data item, for example by using normal COBOL statements.

For more information about this form of the VALIDATING phrase, see the related reference about the XML PARSE statement.

During parsing with validation, normal XML events are returned until an exception occurs due to a validation error or well-formedness error. If an XML document is not valid, the parser signals an XML exception and passes control to the processing procedure with special register XML - EVENT containing 'EXCEPTION' and special register XML - CODE containing return code 24 in the high-order halfword and a specific validation reason code in the low-order halfword.

For information about the return code and reason code for exceptions that might occur when parsing XML documents with validation, see the related reference about exceptions with XMLPARSE (XMLSS) in effect.

[“Example: parsing XML documents with validation” on page 648](#)

Related concepts

[“XML-CODE” on page 618](#)

[“XML schemas” on page 624](#)

Related tasks

[“Handling XML PARSE exceptions” on page 633](#)

Related references

[“XMLPARSE” on page 415 \(compiler option\)](#)

[“XML PARSE exceptions with XMLPARSE\(XMLSS\) in effect” on page 795](#)

XML PARSE statement (*Enterprise COBOL for z/OS Language Reference*)

XML-SCHEMA clause (*Enterprise COBOL for z/OS Language Reference*)

z/OS XML System Services User's Guide and Reference

XML schemas

An *XML schema* is a mechanism, defined by the W3C, for describing and constraining the structure and content of XML documents. An XML schema, which is itself expressed in XML, effectively defines a class of XML documents of a given type, for example, purchase orders.

For Enterprise COBOL, XML schemas used for validating XML documents must be in a preprocessed format known as *Optimized Schema Representation (OSR)*. For information about this format, see the related reference about z/OS XML System Services.

Consider an XML document that describes an item for stock-keeping purposes:

```
<stockItem itemNumber="453-SR">
  <itemName>Stainless steel rope thimbles</itemName>
  <quantityOnHand>23</quantityOnHand>
</stockItem>
```

The example document above is both well formed and valid according to the following schema. (The numbers that precede each line are not part of the schema, but are used in the explanations after the schema.)

```
1. <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
2.
3. <xsd:element name="stockItem" type="stockItemType"/>
4.
5. <xsd:complexType name="stockItemType">
6.   <xsd:sequence>
7.     <xsd:element name="itemName" type="xsd:string" minOccurs="0"/>
8.     <xsd:element name="quantityOnHand">
9.       <xsd:simpleType>
10.        <xsd:restriction base="xsd:nonNegativeInteger">
11.          <xsd:maxExclusive value="100"/>
12.        </xsd:restriction>
13.      </xsd:simpleType>
14.    </xsd:element>
15.  </xsd:sequence>
16.  <xsd:attribute name="itemNumber" type="SKU" use="required"/>
17. </xsd:complexType>
18.
```

```

19. <xsd:simpleType name="SKU">
20.   <xsd:restriction base="xsd:string">
21.     <xsd:pattern value="\d{3}-[A-Z]{2}" />
22.   </xsd:restriction>
23. </xsd:simpleType>
24.
25. </xsd:schema>

```

The schema declares (line 3) that the root element is `stockItem`, which has a mandatory `itemNumber` attribute (line 16) of type `SKU`, and includes a sequence (lines 6 - 15) of other elements:

- An optional `itemName` element of type `string` (line 7)
- A required `quantityOnHand` element that has a constrained range of 1 - 99 based on the type `nonNegativeInteger` (lines 8 - 14)

Type declarations can be inline and unnamed, as in lines 9 - 13, which include the `maxExclusive` facet to specify the legal values for the `quantityOnHand` element.

For the `itemNumber` attribute, by contrast, the named type `SKU` is declared separately in lines 19 - 23, which include a pattern facet that uses regular expression syntax to specify that the legal values for that type consist of (in order): 3 digits, a hyphen-minus, then two uppercase letters.

The example referenced below shows a program that parses documents against this schema.

[“Example: parsing XML documents with validation” on page 648](#)

Related tasks

[“Parsing XML documents with validation” on page 622](#)

Related references

z/OS XML System Services User's Guide and Reference

Parsing XML documents one segment at a time

You can parse XML documents by passing the parser one *segment* (or record) of XML text at a time.

Processing very large documents, or processing XML documents that reside in a data set, are two possible major applications of this technique.

To use this feature, compile your program with the `XMLPARSE (XMLSS)` compiler option in effect.

You parse an XML document a segment at a time by initializing the parse data item to the first segment of the XML document, and then executing the `XML PARSE` statement. The parser processes the XML text and returns XML events to your processing procedure as usual.

At the end of the text segment, the parser signals an `END-OF-INPUT XML` event, with `XML - CODE` set to zero. If there is another segment of the document to process, in your processing procedure move the next segment of XML data to the parse data item, set `XML - CODE` to one, and return to the parser. To signal the end of XML segments to the parser, return to the parser with `XML - CODE` still set to zero.

The length of the parse data item is evaluated for each segment, and determines the segment length.

Variable-length segments: If the XML document segments are variable length, specify a variable-length item for the parse data item. For example, for variable-length XML segments, you can define the parse data item as one of the following items:

- A variable-length group item that contains an `OCCURS DEPENDING ON` clause
- A reference-modified item
- An FD record that specifies the `RECORD IS VARYING DEPENDING ON` clause, where the depending-on data item is used as the length in a reference modifier or ODO object for the FD record

When you send an XML document to the parser in multiple segments, document content is in some cases returned to the processing procedure in multiple fragments by means of multiple events, rather than as one large fragment in a single event.

For example, if the document is split into two segments with the split point in the middle of a string of content characters, the parser returns the content in two separate CONTENT-CHARACTERS events. In the processing procedure, you must reassemble the string of content as needed by the application.

Starting element tags, attribute names, namespace declarations, and ending element tags are always delivered to the processing procedure in a single event, even if those items are split between two segments of a document.

If a segment split occurs between the bytes of a multibyte character, the parser detects the split and reassembles the character for delivery in a single event.

If you are parsing an XML document with an unknown number of repetitive elements to be processed, use unbounded tables. For more information on unbounded tables, see [“Working with unbounded tables and groups” on page 88](#).

For each such element in a given document, manage the table size using one of the following methods:

- Calculating number of elements:
 1. Count the number of elements in the document during an initial parse.
 2. Set the OCCURS DEPENDING ON object for the table to that size
 3. Allocate storage for the table
 4. Parse the document a second time to process the XML
- Incremental expansion:
 1. Set an initial size in the OCCURS DEPENDING ON object for the unbounded table
 2. Parse the document normally. For each element
 - a. Check the limit and expand the unbounded table if necessary.
 3. Allocate a new, larger storage area:
 4. Copy the data from the smaller area
 5. Free the smaller area
 6. Set the table pointer to the address of the larger storage area.

QSAM and VSAM files: You can process XML documents stored in a QSAM or VSAM file as follows:

1. Open the file and read the first record of the XML document.
2. Execute the XML PARSE statement with the FD record as the parse data item.
3. In the processing-procedure logic for handling the END-OF-INPUT event, read the next record of the XML document into the parse data item. If not end-of-file (file status code 10), set XML-CODE to one and return to the parser. If end-of-file, return to the parser with XML-CODE still set to zero.
4. In your processing procedure logic for the END-OF-DOCUMENT event, close the file.

Miscellaneous information after the root element:

The root element of an XML document might be followed by zero or more occurrences of a comment or processing instruction, in any order. If you parse the document one segment at a time, the parser signals an END-OF-INPUT XML event after processing the end tag of the root element only if the last item in the segment is incomplete. If the segment ends with a complete XML item (such as the root element end tag, or after that tag, a complete comment or processing instruction), the next XML event after the event for the item itself is the END-OF-DOCUMENT XML event.

Tip: To provide successive segments of XML data after the end of the root element, include at least the first nonspace character of an XML item at the end of each segment. Include a complete item only on the last segment that you want the parser to process.

For instance, in the following example, in which each line represents a segment of an XML document, the segment that includes the text `This comment ends this segment` is the last segment to be parsed:

```
<Tagline>
```

```

COBOL is the language of the future!
</Tagline> <
!--First comment--
> <?pi data?> <!--
-This comment ends this segment-->
<!-- This segment is not included in the parse-->

```

[“Example: parsing an XML document one segment at a time” on page 646](#)

Related concepts

[“XML events” on page 617](#)

[“XML-CODE” on page 618](#)

Related tasks

[“Parsing XML documents one segment at a time” on page 625](#)

[“XML-CODE” on page 618](#)

Related references

[“XMLPARSE” on page 415 \(compiler option\)](#)

Handling splits using the XML-INFORMATION special register

You can parse large XML documents by using the XML - INFORMATION special register.

To use this feature, compile your program with the XMLPARSE (XMLSS) compiler option in effect.

Splits in character content might occur at arbitrary points in the XML data stream, even with unsegmented input. The XML - INFORMATION special register simplifies the reassembly of content. This register may be required for any and all attribute values and element character content.

The length of the parse data item is evaluated for each segment, and determines the segment length.

The example, [“Example: program for processing XML” on page 638](#), demonstrates various ways of assigning values obtained from the XML document to program data items for later processing.

The XML data is provided to the parser in 40-byte records, imitating the way an XML document might be acquired from an external source such as a data file. The record boundaries are designed so that all data splits but one are accommodated by the parser. For example, the sample treats as an error a split in any content except the content of the "filling" element.

In the example, the XML - INFORMATION special register is only used to simplify the reassembly of content for the "filling" element. This register could be used for any attribute values and element character content. An XML - INFORMATION value of 2 indicates that the character data for an ATTRIBUTE - CHARACTERS or CONTENT - CHARACTERS XML event is continued in a subsequent XML event, and should thus be buffered in order to accumulate the complete character string. A subsequent XML event of the same type with an XML - INFORMATION value of 1 indicates that XML - TEXT or XML - NTEXT contains the final piece of the character content, and that the complete string can be moved to the appropriate data item.

In the example, the STRING ... WITH POINTER statement accumulates and describes properly the complete character value for assignment to the "filling" identifier.

```

String xml-text delimited by size into
  content-buffer with pointer tally
On overflow
  Display 'content buffer ('
    length of content-buffer
    ' bytes) is too small'
  Move -1 to xml-code
End-string

```

Related concepts

[“XML events” on page 617](#)

[“XML-CODE” on page 618](#)

Related references

[“XMLPARSE” on page 415](#) (compiler option)

[“Example: program for processing XML” on page 638](#)

The encoding of XML documents

XML documents must be encoded in a supported code page.

XML documents generated in or parsed from national data items must be encoded in Unicode UTF-16 in big-endian format, CCSID 1200.

For XML GENERATE statements, documents generated in alphanumeric data items must be encoded in Unicode UTF-8 (CCSID 1208) or one of the single-byte EBCDIC encodings listed in the table below. You can use any CCSID from that table in the ENCODING phrase of the XML GENERATE statement.

For XML PARSE statements, documents in alphanumeric data items must be encoded as follows:

- If XMLPARSE (XMLSS) is in effect:
 - If the RETURNING NATIONAL phrase is specified in the XML PARSE statement, in any EBCDIC or ASCII encoding that is supported by z/OS Unicode Services for conversion to UTF-16
 - If the RETURNING NATIONAL phrase is not specified in the XML PARSE statement, in UTF-8 (CCSID 1208) or one of the single-byte EBCDIC encodings listed in the table below
- If XMLPARSE (COMPAT) is in effect: in one of the single-byte EBCDIC encodings listed in the table below

If XMLPARSE (XMLSS) is in effect, you can use any supported CCSID (as described above for XML PARSE) in the ENCODING phrase of the XML PARSE statement.

CCSID	Description
1208	UTF-8 ¹
1047	Latin 1 / Open Systems
1140, 37	USA, Canada, . . . Euro Country Extended Code Page (ECECP), Country Extended Code Page (CECP)
1141, 273	Austria, Germany ECECP, CECP
1142, 277	Denmark, Norway ECECP, CECP
1143, 278	Finland, Sweden ECECP, CECP
1144, 280	Italy ECECP, CECP
1145, 284	Spain, Latin America (Spanish) ECECP, CECP
1146, 285	UK ECECP, CECP
1147, 297	France ECECP, CECP
1148, 500	International ECECP, CECP
1149, 871	Iceland ECECP, CECP

1. Supported for the XML PARSE statement in the ENCODING phrase if XMLPARSE (XMLSS) is in effect

Related concepts

[“XML input document encoding” on page 629](#)

Related tasks

[“Specifying the encoding” on page 630](#)

[“Parsing XML documents encoded in UTF-8” on page 632](#)
[Chapter 38, “Producing XML output,” on page 651](#)

Related references

[“CODEPAGE” on page 343](#)

[“XMLPARSE” on page 415](#) (compiler option)

XML input document encoding

To parse an XML document using the XML PARSE statement, the document must be encoded in a supported encoding.

The supported encodings for a given parse operation depend on:

- The category of the data item that contains the XML document
- The setting of the XMLPARSE compiler option
- The optional phrases that are specified in the XML PARSE statement

For XML documents that are contained in a national data item, the supported encoding is Unicode UTF-16 in big-endian format, CCSID 1200.

For XML documents that are contained in an alphanumeric data item, the supported encodings if the XMLPARSE (XMLSS) compiler option is in effect are as follows:

- If the RETURNING NATIONAL phrase is specified in the XML PARSE statement: UTF-8 or any EBCDIC or ASCII encoding that is supported by the z/OS Unicode Services for conversion to UTF-16
- If the RETURNING NATIONAL phrase is not specified: UTF-8 or any of the single-byte EBCDIC CCSIDs listed in the related reference about the encoding of XML documents

For XML documents that are contained in an alphanumeric data item, the supported CCSIDs if XMLPARSE (COMPAT) is in effect are those specified in the related reference about the encoding of XML documents.

To parse an XML document that is encoded in an unsupported code page, first convert the document to national character data (UTF-16) by using the NATIONAL -OF intrinsic function. You can convert the individual pieces of document text that are passed to the processing procedure in special register XML -NTEXT back to the original code page by using the DISPLAY -OF intrinsic function.

XML declaration and white space:

XML documents can begin with *white space* only if they do not have an XML declaration:

- If an XML document begins with an XML declaration, the first angle bracket (<) in the document must be the first character in the document.
- If an XML document does not begin with an XML declaration, the first angle bracket in the document can be preceded only by white space.

White-space characters have the hexadecimal values shown in the following table.

White-space character	EBCDIC	Unicode
Space	X'40'	X'0020'
Horizontal tabulation	X'05'	X'0009'
Carriage return	X'0D'	X'000D'
Line feed	X'25'	X'000A'
New line / next line	X'15'	X'0085'

Determining the encoding of an input XML document

The parser must know the encoding of an XML document in order to process the document correctly.

If the specified encoding is not one of the supported coded character sets, the parser signals an XML exception event before beginning the parse operation. If the *actual document encoding* does not match the specified encoding, the parser signals an appropriate XML exception after beginning the parse operation.

Several sources are used in determining the encoding of an XML document:

- If the XMLPARSE (XMLSS) option is in effect:
 - The data type of the data item that contains the XML document
 - The ENCODING phrase (if used) of the XML PARSE statement
 - The CCSID specified in the CODEPAGE compiler option
- If the XMLPARSE (COMPAT) option is in effect:
 - The data type of the data item that contains the XML document
 - The actual encoding determined when the parser examines the first few bytes of the document
 - The *encoding declaration* specified within the XML document
 - The CCSID specified in the CODEPAGE compiler option

If XMLPARSE (XMLSS) is in effect:

- Any encoding declaration specified within the XML document is ignored.
- For XML documents that are contained in a national data item, the ENCODING phrase of the XML PARSE statement must be omitted or must specify CCSID 1200. The CCSID specified in the CODEPAGE compiler option is ignored. The parser signals an XML exception event if the actual document encoding is not UTF-16 in big-endian format.
- For XML documents that are contained in an alphanumeric data item, the CCSID specified in the ENCODING phrase overrides the CODEPAGE compiler option. The parser raises an XML exception event at the beginning of the parse operation if the actual document encoding is not consistent with the specified CCSID.

Related tasks

[“Converting to or from national \(Unicode\) representation” on page 134](#)

[“Specifying the encoding” on page 630](#)

[“Parsing XML documents encoded in UTF-8” on page 632](#)

[“Handling XML PARSE exceptions” on page 633](#)

Related references

[“XMLPARSE” on page 415 \(compiler option\)](#)

[“The encoding of XML documents” on page 628](#)

[“EBCDIC code-page-sensitive characters in XML markup” on page 631](#)

Specifying the encoding

You can choose how to specify the encoding for parsing an XML document that is in an alphanumeric data item.

The preferred way is to omit the encoding declaration from the document and to specify the encoding using one of the following means:

- If XMLPARSE (XMLSS) is in effect: the ENCODING phrase of the XML PARSE statement, or the CODEPAGE compiler option
- If XMLPARSE (COMPAT) is in effect: the CODEPAGE compiler option

Omitting the encoding declaration makes it possible to more easily transmit an XML document between heterogeneous systems. (If you included an encoding declaration, you would need to update it to reflect any code-page translation imposed by the transmission process.)

For XMLPARSE (COMPAT):

You can instead specify an encoding declaration in the XML declaration with which most XML documents begin. For example:

```
<?xml version="1.0" encoding="ibm-1140"?>
```

Note that the XML parser generates an exception if it encounters an XML declaration that does not begin in the first byte of an XML document.

If you specify an encoding declaration, do so in one of the following ways:

- Specify the CCSID number (with or without any number of leading zeros) prefixed by one of the following strings in any mixture of uppercase and lowercase letters:
 - IBM-
 - IBM_
 - CCSID-
 - CCSID_
- Use one of the aliases listed in the following table. You can code the aliases in any mixture of uppercase and lowercase letters.

CCSID	Supported aliases
037	EBCDIC-CP-US, EBCDIC-CP-CA, EBCDIC-CP-WT, EBCDIC-CP-NL
500	EBCDIC-CP-BE, EBCDIC-CP-CH
1200	UTF-16
1208	UTF-8

For more information about the CCSIDs that are supported for XML parsing, see the related reference about the encoding of XML documents.

Related concepts

[“XML input document encoding” on page 629](#)

Related tasks

[“Parsing XML documents encoded in UTF-8” on page 632](#)

[“Handling encoding conflicts” on page 636](#)

Related references

[“The encoding of XML documents” on page 628](#)

EBCDIC code-page-sensitive characters in XML markup

Several special characters that are used in XML markup have different hexadecimal representations in different EBCDIC code pages.

The following table shows those special characters and their hexadecimal values for various EBCDIC CCSIDs.

Table 80. Hexadecimal values of special characters for various EBCDIC CCSIDs

Character	1047	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149
[X'AD'	X'BA'	X'63'	X'9E'	X'B5'	X'90'	X'4A'	X'B1'	X'90'	X'4A'	X'AE'
]	X'BD'	X'BB'	X'FC'	X'9F'	X'9F'	X'51'	X'5A'	X'BB'	X'B5'	X'5A'	X'9E'
!	X'5A'	X'5A'	X'4F'	X'4F'	X'4F'	X'4F'	X'BB'	X'5A'	X'4F'	X'4F'	X'4F'
	X'4F'	X'4F'	X'BB'	X'BB'	X'BB'	X'BB'	X'4F'	X'4F'	X'BB'	X'BB'	X'BB'
#	X'7B'	X'7B'	X'7B'	X'4A'	X'63'	X'B1'	X'69'	X'7B'	X'B1'	X'7B'	X'7B'

Parsing XML documents encoded in UTF-8

If the XMLPARSE (XMLSS) compiler option is in effect, you can parse XML documents that are encoded in Unicode UTF-8 in a manner similar to parsing other XML documents. However, some additional requirements apply.

To parse a UTF-8 XML document, you must specify CCSID 1208 in the ENCODING phrase of the XML PARSE statement, as shown in the following code fragment:

```
XML PARSE xml-document
  WITH ENCODING 1208
  PROCESSING PROCEDURE xml-event-handler
.
.
END-XML
```

You define `xml-document` as an alphanumeric data item or alphanumeric group item in `WORKING-STORAGE` or `LOCAL-STORAGE`.

If you do not code the `RETURNING NATIONAL` phrase in the `XML PARSE` statement, the parser returns the XML document fragments in the alphanumeric special registers `XML-TEXT`, `XML-NAMESPACE`, and `XML-NAMESPACE-PREFIX`.

UTF-8 characters are encoded using a variable number of bytes per character. Most COBOL operations on alphanumeric data assume a single-byte encoding, in which each character is encoded in 1 byte. When you operate on UTF-8 characters as alphanumeric data, you must ensure that the data is processed correctly. Avoid operations (such as reference modification and moves that involve truncation) that can split a multibyte character between bytes. You cannot reliably use statements such as `INSPECT` to process multibyte characters in alphanumeric data.

You can more reliably process UTF-8 document fragments by specifying the `RETURNING NATIONAL` phrase in the `XML PARSE` statement. If you use the `RETURNING NATIONAL` phrase, XML document fragments are efficiently converted to UTF-16 encoding and are returned to the application in the national special registers `XML-NTEXT`, `XML-NNAMESPACE`, and `XMLNNAMESPACE-PREFIX`. Then you can process the XML text fragments in national data items. (The UTF-16 encoding in national data items greatly facilitates Unicode processing in COBOL.)

The following code fragment illustrates the use of both the `ENCODING` phrase and the `RETURNING NATIONAL` phrase for parsing a UTF-8 XML document:

```
XML PARSE xml-document
  WITH ENCODING 1208 RETURNING NATIONAL
  PROCESSING PROCEDURE xml-event-handler
ON EXCEPTION
  DISPLAY 'XML document error ' XML-CODE
  STOP RUN
NOT ON EXCEPTION
  DISPLAY 'XML document was successfully parsed.'
END-XML
```

Related concepts

[“XML-TEXT and XML-NTEXT” on page 620](#)

[“XML-NAMESPACE and XML-NNAMESPACE” on page 621](#)

[“XML-NAMESPACE-PREFIX and XML-NNAMESPACE-PREFIX” on page 621](#)

Related tasks

[“Processing UTF-8 data using UTF-16 \(national\) data types” on page 153](#)

[“Parsing XML documents” on page 613](#)

[“Specifying the encoding” on page 630](#)

Related references

[“XMLPARSE” on page 415 \(compiler option\)](#)

[“The encoding of XML documents” on page 628](#)

XML PARSE statement (*Enterprise COBOL for z/OS Language Reference*)

Handling XML PARSE exceptions

If the XML parser encounters an anomaly or error during parsing, it sets an exception code in the XML - CODE special register and signals an XML exception event. The specific exception codes that can occur and the subsequent actions that you can take differ depending on the setting of the XMLPARSE compiler option.

For XMLPARSE (XMLSS):

Return code and reason code: The exception code is formed from the return code and the reason code that the parser generates. The return code and the reason code are each a halfword binary value. The value in XML - CODE is a concatenation of these two values.

As an example, the following XML document is not well formed because the element end tag `mmsg` does not match the element start tag `msg`:

```
<msg>Hello</mmsg>
```

The return code is hexadecimal 000C (XRC_NOT_WELL_FORMED), and the reason code is hexadecimal 3035 (XRSN_ENDTAG_NAME_MISMATCH), if you parse the document without validation. The concatenation of these two values, hexadecimal 000C3035, is returned to the processing procedure in the XML - CODE special register.

If you parse a document with validation, the values returned in XML - CODE for any well-formedness errors differ from the values returned for the same errors when you parse without validation. The return code generated by the z/OS XML System Services parser for any validation error is 24 (hexadecimal 0018).

For more information about the return codes and reason codes that can be generated, see the related reference about exceptions with XMLPARSE (XMLSS) in effect.

If XMLPARSE (XMLSS) is in effect, processing procedures cannot handle exception events and cannot cause parsing to resume. When a processing procedure returns to the parser from an exception event, the parser does not signal any further events. The parser transfers control to the statement that is specified in the ON EXCEPTION phrase of the XML PARSE statement. If you did not code an ON EXCEPTION phrase, control is passed to the end of the XML PARSE statement. XML - CODE contains the original exception code set by the parser.

If no exception occurs during parsing, control is passed to the statement specified in the NOT ON EXCEPTION phrase. If you did not code a NOT ON EXCEPTION phrase, control is passed to the end of the XML PARSE statement. XML - CODE contains zero.

For XMLPARSE (COMPAT):

If the exception code is within a certain range, you might be able to handle the exception event within your processing procedure, and resume parsing.

To handle an exception in the processing procedure, follow these steps:

1. Check the contents of XML - CODE.
2. Handle the exception appropriately.

3. Set XML - CODE to zero to indicate that you handled the exception.
4. Return control to the parser.

The exception condition no longer exists.

You can handle exceptions in this way only if the exception code that is passed in XML - CODE is within one of the following ranges, which indicates that an encoding conflict was detected:

- 50 - 99
- 100,001 - 165,535

Exception codes 1 - 49: In the processing procedure, you can do limited handling of exceptions for which the exception code is within the range 1 - 49. After an exception in this range occurs, the parser does not signal any further normal events, except the END-OF-DOCUMENT event, even if you set XML - CODE to zero before returning. If you set XML - CODE to zero, the parser continues parsing the document and signals any exceptions that it finds. (Doing so can provide a useful way to discover multiple errors in the document.)

Restriction: The compatibility-mode COBOL XML parser might not signal all additional exception events. The number of exceptions is limited to the remaining space in the XML PARSE event token array, probably 8192 events.

At the end of parsing after an exception that has an exception code in the range 1 - 49, control is passed to the statement specified in the ON EXCEPTION phrase. If you did not code an ON EXCEPTION phrase, control is passed to the end of the XML PARSE statement. XML - CODE contains the code set by the parser for the most recent exception.

For all exceptions other than those having an exception code within one of the ranges described above, the parser does not signal any further events, but passes control to the statement specified in the ON EXCEPTION phrase. XML - CODE contains the original exception code even if you set XML - CODE in the processing procedure before returning control to the parser.

If you do not want to handle an exception, return control to the parser without changing the value of XML - CODE. The parser transfers control to the statement specified in the ON EXCEPTION phrase. If you did not code an ON EXCEPTION phrase, control is transferred to the end of the XML PARSE statement.

If no unhandled exceptions occur before the end of parsing, control is passed to the statement specified in the NOT ON EXCEPTION phrase. If you did not code a NOT ON EXCEPTION phrase, control is transferred to the end of the XML PARSE statement. XML - CODE contains zero.

Related concepts

[“XML-CODE” on page 618](#)

[“XML input document encoding” on page 629](#)

[“How the XML parser handles errors” on page 635](#)

Related tasks

[“Writing procedures to process XML” on page 615](#)

[“Parsing XML documents with validation” on page 622](#)

[“Handling encoding conflicts” on page 636](#)

Related references

[“XMLPARSE” on page 415 \(compiler option\)](#)

[“The encoding of XML documents” on page 628](#)

[“XML PARSE exceptions with XMLPARSE\(XMLSS\) in effect” on page 795](#)

[“XML PARSE exceptions with XMLPARSE\(COMPAT\) in effect” on page 797](#)

z/OS XML System Services User's Guide and Reference

How the XML parser handles errors

When the XML parser detects an error in an XML document, it generates an XML exception event and passes control to your processing procedure.

The parser passes the following information in special registers to the processing procedure:

- XML -EVENT contains 'EXCEPTION'.
- XML -CODE contains a numeric exception code.

The exception codes are described in the related references about XML PARSE exceptions.

- For fatal exceptions, XML -TEXT or XML -NTEXT contains the document text up to and including the point where the exception was detected.
- For the warning exceptions issued for using an undeclared prefix, XML -TEXT or XML -NTEXT contains the fully qualified attribute name or element name. That is, the name includes the undeclared prefix and the separator colon (:).
- If XMLPARSE (COMPAT) is in effect, XML -TEXT or XML -NTEXT contains the document text up to and including the point where the exception was detected.
- If XMLPARSE (XMLSS) is in effect, XML -TEXT or XML -NTEXT contains the document text up to the point where the error or anomaly was detected. If you process the XML document one segment at a time, the applicable special register contains only the current segment.

All other XML special registers are empty with length zero.

For XMLPARSE (XMLSS):

Parsing cannot continue after a fatal exception even if you set XML -CODE to zero in the processing procedure. Upon return to the parser from the processing procedure, the parser transfers control to the ON EXCEPTION phrase, if specified; otherwise the parser transfers control to the end of the XML PARSE statement. XML -CODE contains the original exception code set by the parser.

For XMLPARSE (COMPAT):

The processing procedure might be able to handle an exception so that parsing continues if the exception code is within one of the following ranges:

- 1 - 99
- 100,001 - 165,535

If the exception code has any other nonzero value, parsing cannot continue.

Encoding conflicts: The exceptions for encoding conflicts (50 - 99 and 300 - 399) are signaled before the parsing of the document begins. For these exceptions, XML -TEXT or XML -NTEXT is either length zero or contains only the encoding declaration value from the document.

Exception codes 1 - 49: An exception for which the exception code is in the range 1 - 49 is a fatal error according to the *XML specification*. Therefore, the parser does not continue normal parsing even if the processing procedure handles the exception. However, the parser does continue scanning for further errors until it reaches the end of the document, or until the existing XML EVENT token array is exhausted. For these exceptions, the parser does not signal any further normal events except the END-OF-DOCUMENT event.

Related concepts

[“XML events” on page 617](#)

[“XML-CODE” on page 618](#)

[“XML input document encoding” on page 629](#)

Related tasks

[“Parsing XML documents one segment at a time” on page 625](#)

[“Handling XML PARSE exceptions” on page 633](#)

[“Handling encoding conflicts” on page 636](#)

[“Terminating XML parsing” on page 637](#)

Related references

[“XMLPARSE” on page 415 \(compiler option\)](#)

[“The encoding of XML documents” on page 628](#)

[“XML PARSE exceptions with XMLPARSE\(XMLSS\) in effect” on page 795](#)

[“XML PARSE exceptions with XMLPARSE\(COMPAT\) in effect” on page 797](#)

[z/OS XML System Services User's Guide and Reference XML specification](#)

Handling encoding conflicts

The way that you handle encoding-conflict exceptions depends on the setting of the XMLPARSE compiler option.

For XMLPARSE (XMLSS):

The parser does not continue after an encoding-conflict exception or after any other type of exception. Any changes that you make in the processing procedure to the value of XML - CODE are ignored. The value in XML - CODE when the parser returns to the XML PARSE statement is the original exception code that the parser set.

For XMLPARSE (COMPAT):

Your processing procedure might be able to handle exceptions for document encoding conflicts. Exception events in which the parse data item is alphanumeric and the exception code in XML - CODE is within the range 100,001 - 165,535 indicate that the code page of the document (as specified by its encoding declaration) conflicts with the external code-page information.

In this special case, you can choose to parse using the code page of the document by subtracting 100,000 from the value in XML - CODE. For instance, if XML - CODE contains 101,140, the code page of the document is 1140. Alternatively, you can choose to parse using the *external code page* by setting XML - CODE to zero before returning to the parser.

The parser takes one of three actions after returning from a processing procedure for an encoding-conflict exception event:

- If you set XML - CODE to zero, the parser uses the external code page: the value of the CODEPAGE compiler option.
- If you set XML - CODE to the code page of the document (that is, the original XML - CODE value minus 100,000), the parser uses the code page of the document.

This is the only case in which the parser continues when XML - CODE has a nonzero value upon returning from a processing procedure.

- Otherwise, the parser stops processing the document and returns control to the XML PARSE statement with an exception condition. XML - CODE contains the exception code that was originally passed with the exception event.

Related concepts

[“XML-CODE” on page 618](#)

[“XML input document encoding” on page 629](#)

[“How the XML parser handles errors” on page 635](#)

Related tasks

[“Handling XML PARSE exceptions” on page 633](#)

Related references

[“XMLPARSE” on page 415 \(compiler option\)](#)

[“The encoding of XML documents” on page 628](#)

[“XML PARSE exceptions with XMLPARSE\(XMLSS\) in effect” on page 795](#)
[“XML PARSE exceptions with XMLPARSE\(COMPAT\) in effect” on page 797](#)
z/OS XML System Services User's Guide and Reference

Terminating XML parsing

You can terminate parsing immediately, without processing any remaining XML text, by setting XML - CODE to -1 in your processing procedure before the procedure returns to the parser from any normal XML event (that is, any event other than EXCEPTION).

You can use this technique when the processing procedure has examined enough of the document or has detected some irregularity in the document that precludes further meaningful processing.

If you terminate parsing in this way, the parser does not signal any further XML events, including the exception event. Control transfers to the ON EXCEPTION phrase of the XML PARSE statement, if that phrase was specified.

In the imperative statement of the ON EXCEPTION phrase, you can determine whether parsing was deliberately terminated by testing whether XML - CODE contains -1. If you do not specify the ON EXCEPTION phrase, control transfers to the end of the XML PARSE statement.

If the XMLPARSE (COMPAT) compiler option is in effect, you can also terminate parsing after any XML EXCEPTION event by returning to the parser from the processing procedure without changing the value in XML - CODE. The result is similar to the result of deliberate termination, except that the parser returns to the XML PARSE statement with XML - CODE containing the original exception code.

If the XMLPARSE (XMLSS) option is in effect, parsing always terminates after any exception event.

Related concepts

[“XML-CODE” on page 618](#)
[“How the XML parser handles errors” on page 635](#)

Related tasks

[“Writing procedures to process XML” on page 615](#)
[“Handling XML PARSE exceptions” on page 633](#)

XML PARSE examples

The examples that are referenced below illustrate various uses of the XML PARSE statement.

Use these examples to understand the basic use of XML PARSE and for XMLPARSE (XMLSS), specialized uses such as parsing documents that include namespaces, parsing documents one segment at a time, and parsing documents with validation against a schema.

[“Example: parsing a simple document” on page 637](#)
[“Example: program for processing XML” on page 638](#)
[“Example: parsing an XML document that uses namespaces” on page 642](#)
[“Example: parsing an XML document one segment at a time” on page 646](#)
[“Example: parsing XML documents with validation” on page 648](#)

Example: parsing a simple document

This example shows the flow of events and the contents of special register XML - TEXT that result from the parsing of a simple XML document.

Assume that the COBOL program contains the following XML document in data item Doc:

```
<?xml version="1.0"?><msg type="short">Hello, World!</msg>
```

The following code fragment shows an XML PARSE statement for parsing Doc, and a processing procedure, P, for handling the XML events:

```
XML Parse Doc
  Processing procedure P
  .
  .
  .
P. Display XML-Event XML-Text.
```

The processing procedure displays the content of XML - EVENT and XML - TEXT for each event that the parser signals during parsing. The following table shows the events and the text.

XML - EVENT	XML - TEXT
START-OF-DOCUMENT	
VERSION-INFORMATION	1.0
START-OF-ELEMENT	msg
ATTRIBUTE-NAME	type
ATTRIBUTE-CHARACTERS	short
CONTENT-CHARACTERS	Hello, World!
END-OF-ELEMENT	msg
END-OF-DOCUMENT	

Related concepts

“XML events” on page 617

“XML-TEXT and XML-NTEXT” on page 620

Example: program for processing XML

This example shows the parsing of an XML document, and a processing procedure that reports the various XML events and their associated text fragments.

The XML document is shown in the program source to make it easier to follow the flow of the parsing. The output of the program with XMLPARSE (XMLSS) and with XMLPARSE (COMPAT) in effect is shown after the example.

To understand the interaction of the parser and the processing procedure, and to match events to document fragments, compare the XML document to the output of the program.

```
Process codepage(1047)
  Identification division.
  Program-id. XMLSAMPL.
  Data division.
  Working-storage section.
*****
* XML document data, encoded as initial values of data items. *
*****
1 xml-document-data.
2 pic x(39) value '<?xml version="1.0" encoding="IBM-1047"'.
2 pic x(19) value ' standalone="yes"?>'.
2 pic x(39) value '<!--This document is just an example-->'.
2 pic x(10) value '<sandwich>'.
2 pic x(33) value '<bread type="baker&apos;s best"/>'.
2 pic x(36) value '<?spread We'll use real mayonnaise?>'.
2 pic x(29) value '<meat>Ham & turkey</meat>'.
2 pic x(34) value '<filling>Cheese, lettuce, tomato, '.
2 pic x(32) value 'and that's all, Folks!</filling>'.
2 pic x(25) value '<![CDATA[We should add a '.
2 pic x(20) value '<relish> element!]]>'.
2 pic x(28) value '<listprice>$4.99</listprice>'.
2 pic x(25) value '<discount>0.10</discount>'.
2 pic x(31) value '</sandwich>'.
```

```

*****
* XML document, represented as fixed-length records. *
*****
1 xml-document redefines xml-document-data.
2 xml-segment pic x(40) occurs 10 times.
1 xml-segment-no comp pic s9(4).
1 content-buffer pic x(100).
1 current-element-stack.
2 current-element pic x(30) occurs 10 times.
*****
* Sample data definitions for processing numeric XML content. *
*****
1 element-depth comp pic s9(4).
1 discount computational pic 9v99 value 0.
1 display-price pic $$9.99.
1 filling pic x(4095).
1 list-price computational pic 9v99 value 0.
1 ofr-ed pic x(9) justified.
1 ofr-ed-1 redefines ofr-ed pic 999999.99.
Procedure division.
Mainline section.
Move 1 to xml-segment-no
Display 'Initial segment {' xml-segment(xml-segment-no) '}'
Display ' '
XML parse xml-segment(xml-segment-no)
processing procedure XML-handler
On exception
Display 'XML processing error, XML-Code=' XML-Code '.'
Move 16 to return-code
Goback
Not on exception
Display 'XML document successfully parsed.'
End-XML
*****
* Process the transformed content and calculate promo price. *
*****
Display ' '
Display '-----++++***** Using information from XML '
'*****++++-----'
Display ' '
Move list-price to Display-price
Display ' Sandwich list price: ' Display-price
Compute Display-price = list-price * (1 - discount)
Display ' Promotional price: ' Display-price
Display ' Get one today!'
Goback.
XML-handler section.
Evaluate XML-Event
* ==> Order XML events most frequent first
When 'START-OF-ELEMENT'
Display 'Start element tag: {' XML-Text '}'
Add 1 to element-depth
Move XML-Text to current-element(element-depth)
When 'CONTENT-CHARACTERS'
Display 'Content characters: {' XML-Text '}'
* ==> In general, a split can occur for any element or attribute
* ==> data, but in this sample, it only occurs for "filling"...
If xml-information = 2 and
current-element(element-depth) not = 'filling'
Display 'Unexpected split in content for element '
current-element(element-depth)
Move -1 to xml-code
End-if
* ==> Transform XML content to operational COBOL data item...
Evaluate current-element(element-depth)
When 'filling'
* ==> After reassembling separate pieces of character content...
String xml-text delimited by size into
content-buffer with pointer tally
On overflow
Display 'content buffer ('
length of content-buffer
' bytes) is too small'
Move -1 to xml-code
End-string
Evaluate xml-information
When 2
Display ' Character data for element "filling" '
'is incomplete.'
Display ' The partial data was buffered for '
'content assembly.'
When 1

```

```

        subtract 1 from tally
        move content-buffer(1:tally) to filling
        Display ' Element "filling" data (' tally
          ' bytes) is now complete:'
        Display ' {' filling(1:tally) '}'
      End-evaluate
    When 'listprice'
  * ==> Using function NUMVAL-C...
        Move XML-Text to content-buffer
        Compute list-price =
          function numval-c(content-buffer)
    When 'discount'
  * ==> Using de-editing of a numeric edited item...
        Move XML-Text to ofr-ed
        Move ofr-ed-1 to discount
      End-evaluate
    When 'END-OF-ELEMENT'
        Display 'End element tag: {' XML-Text '}'
        Subtract 1 from element-depth
    When 'END-OF-INPUT'
        Display 'End of input'
        Add 1 to xml-segment-no
        Display ' Next segment: {' xml-segment(xml-segment-no)
          '}'
        Display ' '
        Move 1 to xml-code
    When 'START-OF-DOCUMENT'
        Display 'Start of document'
        Move 0 to element-depth
        Move 1 to tally
    When 'END-OF-DOCUMENT'
        Display 'End of document.'
    When 'VERSION-INFORMATION'
        Display 'Version: {' XML-Text '}'
    When 'ENCODING-DECLARATION'
        Display 'Encoding: {' XML-Text '}'
    When 'STANDALONE-DECLARATION'
        Display 'Standalone: {' XML-Text '}'
    When 'ATTRIBUTE-NAME'
        Display 'Attribute name: {' XML-Text '}'
    When 'ATTRIBUTE-CHARACTERS'
        Display 'Attribute value characters: {' XML-Text '}'
    When 'ATTRIBUTE-CHARACTER'
        Display 'Attribute value character: {' XML-Text '}'
    When 'START-OF-CDATA-SECTION'
        Display 'Start of CData section'
    When 'END-OF-CDATA-SECTION'
        Display 'End of CData section'
    When 'CONTENT-CHARACTER'
        Display 'Content character: {' XML-Text '}'
    When 'PROCESSING-INSTRUCTION-TARGET'
        Display 'PI target: {' XML-Text '}'
    When 'PROCESSING-INSTRUCTION-DATA'
        Display 'PI data: {' XML-Text '}'
    When 'COMMENT'
        Display 'Comment: {' XML-Text '}'
    When 'EXCEPTION'
        Compute tally = function length (XML-Text)
        Display 'Exception ' XML-Code ' at offset ' tally '.'
    When other
        Display 'Unexpected XML event: ' XML-Event '.'
      End-evaluate
    .
  End program XMLSAMPL.

```

Output from parsing with XMLPARSE (XMLSS)

From the following output you can see which fragments of the document were associated with the events that occurred during parsing:

```

Initial segment {<?xml version="1.0" encoding="ibm-1047" }

Start of document
End of input
  Next segment: {standalone="yes"?><!--This document is j}

Version: {1.0}
Encoding: {ibm-1047}
Standalone: {yes}

```

```

Comment: {This document is j}
End of input
  Next segment: {ust an example--><sandwich><bread type="}

Comment: {ust an example}
Start element tag: {sandwich}
End of input
  Next segment: {baker&apos;s best"/><?spread We'll use r}

Start element tag: {bread}
Attribute name: {type}
Attribute value characters: {baker's best}
End element tag: {bread}
PI target: {spread}
PI data: {We'll use r}
End of input
  Next segment: {eal mayonnaise?><meat>Ham & turkey</}

PI target: {spread}
PI data: {eal mayonnaise}
Start element tag: {meat}
Content characters: {Ham & turkey}
End of input
  Next segment: {meat}<filling>Cheese, lettuce, tomato, a}

End element tag: {meat}
Start element tag: {filling}
Content characters: {Cheese, lettuce, tomato, a}
  Character data for element "filling" is incomplete.
  The partial data was buffered for content assembly.
End of input
  Next segment: {nd that's all, Folks!</filling><![CDATA[}

Content characters: {nd that's all, Folks!}
  Element "filling" data (00047 bytes) is now complete:
  {Cheese, lettuce, tomato, and that's all, Folks!}
End element tag: {filling}
End of input
  Next segment: {We should add a <relish> element!]]><lis}

Start of CData section
Content characters: {We should add a <relish> element!}
End of CData section
End of input
  Next segment: {tprice>$4.99</listprice><discount>0.10</}

Start element tag: {listprice}
Content characters: {$4.99}
End element tag: {listprice}
Start element tag: {discount}
Content characters: {0.10}
End of input
  Next segment: {discount}</sandwich>          }

End element tag: {discount}
End element tag: {sandwich}
End of document.
XML document successfully parsed.

-----+***** Using information from XML *****-----

Sandwich list price: $4.99
Promotional price: $4.49
Get one today!

```

Output from parsing with XMLPARSE (COMPAT)

From the following output you can see which fragments of the document were associated with the events that occurred during parsing:

```

Start of document
Version: {1.0}
Encoding: {IBM-1047}
Standalone: {yes}
Comment: {This document is just an example}
Start element tag: {sandwich}
Content characters: { }
Start element tag: {bread}

```

```

Attribute name: {type}
Attribute value characters: {baker}
Attribute value character: {'}
Attribute value characters: {s best}
End element tag: {bread}
Content characters: { }
PI target: {spread}
PI data: {please use real mayonnaise }
Content characters: { }
Start element tag: {meat}
Content characters: {Ham }
Content character: {&}
Content characters: { turkey}
End element tag: {meat}
Content characters: { }
Start element tag: {filling}
Content characters: {Cheese, lettuce, tomato, etc.}
End element tag: {filling}
Content characters: { }
Start of CData: {<![CDATA[{}
Content characters: {We should add a <relish> element in future!}
End of CData: {]]>}
Content characters: { }
Start element tag: {listprice}
Content characters: {$4.99 }
End element tag: {listprice}
Content characters: { }
Start element tag: {discount}
Content characters: {0.10}
End element tag: {discount}
End element tag: {sandwich}
End of document.
XML document successfully parsed

```

```

-----***** Using information from XML *****-----

```

```

Sandwich list price:  $4.99
Promotional price:   $4.49
Get one today!

```

Related concepts

[“XML events” on page 617](#)

Related tasks

[“Handling splits using the XML-INFORMATION special register” on page 627](#)

Related references

[“XMLPARSE” on page 415 \(compiler option\)](#)

[XML-EVENT \(Enterprise COBOL for z/OS Language Reference\)](#)

Example: parsing an XML document that uses namespaces

This example shows the parsing of a document that uses namespaces and namespace prefixes. The program must be compiled using the XMLPARSE (XMLSS) compiler option.

Namespace identifiers and namespace prefixes are used in the program to qualify element names and attribute names. This qualification makes it possible to use the same name in more than one context: title is used both as an author's title (Mr) and as a book title (Writing COBOL for Fun and Profit).

Sample XML document

The following XML document contains several namespace declarations: a default namespace; then three namespace identifiers with prefixes (bk, pi, and isbn). Notice that the default namespace is set to the empty string for the element comment (xmlns= ' '). This setting "undeclares" the default namespace, with the result that there is no default namespace.

```

<section
  xmlns="http://www.ibm.com/events"

```

```

xmlns:bk="urn:loc.gov:books"
xmlns:pi="urn:personalInformation"
xmlns:isbn='urn:ISBN:0-395-36341-6'>
<title>Book-Signing Event</title>
<signing>
  <bk:author pi:title="Mr" pi:name="Jim Ross"/>
  <book bk:title="Writing COBOL for Fun and Profit" isbn:number="0426070806"/>
  <comment xmlns=''>What a great issue!</comment>
</signing>
</section>

```

Results from parsing

The following table shows the sequence of events that the processing procedure receives from the parser, and shows the content of the associated XML special registers.

<i>Table 82. XML events and special registers</i>			
XML - EVENT	XML - TEXT	XML - NAMESPACE - PREFIX	XML - NAMESPACE
START-OF-DOCUMENT			
START-OF-ELEMENT	section		http://www.ibm.com/events
NAMESPACE-DECLARATION			http://www.ibm.com/events
NAMESPACE-DECLARATION		bk	urn:loc.gov:books
NAMESPACE-DECLARATION		pi	urn:personalInformation
NAMESPACE-DECLARATION		isbn	urn:ISBN:0-395-36341-6
START-OF-ELEMENT	title		http://www.ibm.com/events
CONTENT-CHARACTERS	Book-Signing Event		
END-OF-ELEMENT	title		http://www.ibm.com/events
START-OF-ELEMENT	signing		http://www.ibm.com/events
START-OF-ELEMENT	author	bk	urn:loc.gov:books
ATTRIBUTE-NAME	title	pi	urn:personalInformation
ATTRIBUTE-CHARACTERS	Mr		
ATTRIBUTE-NAME	name	pi	urn:personalInformation
ATTRIBUTE-CHARACTERS	Jim Ross		
END-OF-ELEMENT	author	bk	urn:loc.gov:books
START-OF-ELEMENT	book		http://www.ibm.com/events

<i>Table 82. XML events and special registers (continued)</i>			
XML-EVENT	XML-TEXT	XML-NAMESPACE-PREFIX	XML-NAMESPACE
ATTRIBUTE-NAME	title	bk	urn:loc.gov:books
ATTRIBUTE-CHARACTERS	Writing COBOL for Fun and Profit		
ATTRIBUTE-NAME	number	isbn	urn:ISBN:0-395-36341-6
ATTRIBUTE-CHARACTERS	0426070806		
END-OF-ELEMENT	book		http://www.ibm.com/ events
START-OF-ELEMENT	comment		
NAMESPACE-DECLARATION			
CONTENT-CHARACTERS	What a great issue!		
END-OF-ELEMENT	comment		
END-OF-ELEMENT	signing		http://www.ibm.com/ events
END-OF-ELEMENT	section		http://www.ibm.com/ events
END-OF-DOCUMENT			

XML PARSE example with an undeclared namespace prefix

The following XML document contains undeclared namespace prefixes:

```

Identification division.
  Program-id. XMLup.
Data division.
  Working-storage section.
  1 d.
    2 pic x(40) value '<pfx0:root xmlns:pfx1="http://whatever">'.
    2 pic x(19) value '<pfx1:localElName1>'.
    2 pic x(20) value '<pfx2:localElName2/>'.
    2 pic x(40) value '<pfx3:localElName3 pfx4:localAtName4="">'.
    2 pic x(02) value 'c1'.
    2 pic x(41) value '<pfx5:localElName5 pfx6:localAtName6=""/>'.
    2 pic x(24) value 'c2</pfx3:localElName3>c3'.
    2 pic x(32) value '</pfx1:localElName1></pfx0:root>'.
Procedure division.
  main.
    display 'XML document: ' d
    display ' '
    xml parse d processing procedure h
    goback.
  h.
    if xml-event = 'EXCEPTION'
      display ' '
    end-if
    display xml-event xml-code '|' xml-text '|'
      xml-namespace-prefix '|'
      xml-namespace '|'
    if xml-event = 'EXCEPTION' and xml-code = 264192 or 264193
      move 0 to xml-code
    end-if
  .
End program XMLup.

```


Results from parsing XML document with an undeclared namespace prefix

The following table lists the sequence of events that the processing procedure receives from the parser, and shows the content of the associated XML special registers.

Table 83. XML events and special registers from parsing XML document with an undeclared namespace prefix

XML - EVENT	XML - CODE	XML - TEXT	XML - NAMESPACE - PREFIX	XML - NAMESPACE
START-OF-DOCUMENT	000000000			
EXCEPTION	000264193	pfx0:root		
START-OF-ELEMENT	000000000	root	pfx0	
NAMESPACE-DECLARATION	000000000		pfx1	http://whatever
START-OF-ELEMENT	000000000	localElName1	pfx1	http://whatever
EXCEPTION	000264193	pfx2:localElName2		
START-OF-ELEMENT	000000000	localElName2	pfx2	
END-OF-ELEMENT	000000000	localElName2	pfx2	
EXCEPTION	000264193	pfx3:localElName3		
START-OF-ELEMENT	000000000	localElName3	pfx3	
EXCEPTION	000264192	pfx4:localAtName4		
ATTRIBUTE-NAME	000000000	localAtName4	pfx4	
ATTRIBUTE-CHARACTERS	000000000			
CONTENT-CHARACTERS	000000000	c1		
EXCEPTION	000264193	pfx5:localElName5		
START-OF-ELEMENT	000000000	localElName5	pfx5	
EXCEPTION	000264192	pfx6:localAtName6		
ATTRIBUTE-NAME	000000000	localAtName6	pfx6	
ATTRIBUTE-CHARACTERS	000000000			
END-OF-ELEMENT	000000000	localElName5	pfx5	
CONTENT-CHARACTERS	000000000	c2		
END-OF-ELEMENT	000000000	localElName3	pfx3	
CONTENT-CHARACTERS	000000000	c3		

Table 83. XML events and special registers from parsing XML document with an undeclared namespace prefix (continued)

XML - EVENT	XML - CODE	XML - TEXT	XML - NAMESPACE - PREFIX	XML - NAMESPACE
END-OF-ELEMENT	000000000	localElName1	pfx1	http://whatever
END-OF-ELEMENT	000000000	root	pfx0	
END-OF-DOCUMENT	000000000			

For a detailed description of the set of XML events, see the related reference about XML - EVENT.

Related concepts

[“XML events” on page 617](#)

[“XML-TEXT and XML-NTEXT” on page 620](#)

[“XML-NAMESPACE and XML-NNAMESPACE” on page 621](#)

[“XML-NAMESPACE-PREFIX and XML-NNAMESPACE-PREFIX” on page 621](#)

Related references

[“XMLPARSE” on page 415 \(compiler option\)](#)

XML-EVENT (*Enterprise COBOL for z/OS Language Reference*)

Example: parsing an XML document one segment at a time

This example shows the parsing of a document one segment at a time. The program must be compiled using the XMLPARSE (XMLSS) compiler option.

The example shows the XML content of a file, the program that reads and submits XML text to the parser, and the sequence of events that results from parsing the input records.

Content of infile

The XML document that will be parsed a segment at a time is contained in file `infile`, shown below.

```
<?xml version='1.0'?>
<Tagline>
COBOL is the language of the future!
</Tagline>
```

Program PARSESEG

Program PARSESEG reads a segment (a record) of the XML document from file `infile`, then passes the record to the parser using the XML PARSE statement. The parser processes the XML text and transfers control to the processing procedure for each XML event. The processing procedure handles each event and returns to the parser.

At the end of the segment, the parser sets XML - EVENT to END-OF-INPUT, sets XML - CODE to zero, and transfers control to the processing procedure. The processing procedure reads the next XML record into the parse data item, sets XML - CODE to one, and returns to the parser.

The exchange between the processing procedure and the parser continues until the READ statement returns the end-of-file status code. The processing procedure returns to the parser with XML - CODE still set to zero to indicate the end of segment processing.

```
Identification division.
Program-id. PARSESEG.
Environment division.
Input-output section.
File-control.
Select Input-XML
```

```

Assign to infile
File status is Input-XML-status.
Data division.
File section.
FD Input-XML
Record is varying from 1 to 255 depending on Rec-length
Recording mode V.
1 fdrec.
2 pic X occurs 1 to 255 depending on Rec-length .
Working-storage section.
1 Event-number comp pic 99.
1 Rec-length comp-5 pic 9(4).
1 Input-XML-status pic 99.
Procedure division.
Open input Input-XML
If Input-XML-status not = 0
    Display 'Open failed, file status: ' Input-XML-status
    Goback
End-if
Read Input-XML
If Input-XML-status not = 0
    Display 'Read failed, file status: ' Input-XML-status
    Goback
End-if
Move 0 to Event-number
Display 'Starting with: ' fdrec
Display 'Event number and name      Content of XML-text'
XML parse fdrec processing procedure Handle-parse-events
Close Input-XML
Goback
.
Handle-parse-events.
Add 1 to Event-number
Display '      Event-number ': ' XML-event '{' XML-text '}'
Evaluate XML-event
    When 'END-OF-INPUT'
        Read Input-XML
        Evaluate Input-XML-status
            When 0
                Move 1 to XML-code
                Display 'Continuing with: ' fdrec
            When 10
                Display 'At EOF; no more input.'
            When other
                Display 'Read failed, file status:' Input-XML-status
                Goback
        End-evaluate
    When other
        Continue
End-evaluate
.
End program PARSESEG.

```

Results from parsing

To show parsing results, the processing procedure displayed each record of input, followed by the sequence of XML events and any associated text fragments in XML - TEXT. The content of XML - TEXT is displayed in braces ({}); empty braces signify that XML - TEXT is empty.

Notice the extra zero-length CONTENT - CHARACTERS XML event at event number 08. (Such anomalies are typical when supplying XML text piecemeal.)

```

Starting with:  <?xml version='1.0'?>
Event number and name      Content of XML-TEXT
01: START-OF-DOCUMENT      {}
02: VERSION-INFORMATION    {1.0}
03: END-OF-INPUT          {}
Continuing with:  <Tagline>
04: START-OF-ELEMENT      {Tagline}
05: END-OF-INPUT          {}
Continuing with:  COBOL is the language of the future!
06: CONTENT-CHARACTERS    {COBOL is the language of the future!}
07: END-OF-INPUT          {}
Continuing with:  </Tagline>
08: CONTENT-CHARACTERS    {}

```

```
09: END-OF-ELEMENT      {Tagline}
10: END-OF-DOCUMENT     {}
```

For a detailed description of the XML events that were detected, see the related reference about XML - EVENT.

Related references

“XMLPARSE” on page 415 (compiler option)

XML-EVENT (*Enterprise COBOL for z/OS Language Reference*)

Example: parsing XML documents with validation

This example shows the parsing of several XML documents with validation against a schema, and a processing procedure that captures the return code and reason code that the parser generates after parsing each document. All of the XML documents are well formed but not necessarily valid.

The program must be compiled using the XMLPARSE (XMLSS) compiler option.

The example uses the schema that was described in the related concept about XML schemas.

Assume that file `item.xsd` contains the schema in text format, and that the preprocessed schema was generated in file `item.osr` by means of the following z/OS UNIX command:

```
xsdosrg -v -o /u/HLQ/xml/item.osr /u/HLQ/xml/item.xsd
```

The example uses the XML - SCHEMA clause to associate the XML schema name `schema` with the ddname `ddschema`. The following DD statement associates the ddname with the external z/OS UNIX file that contains the schema:

```
//GO.DDSCHEMA DD PATH='/u/HLQ/xml/item.osr'
```

Program ValidCk

```
Identification division.
  Program-id. ValidCk.
Environment division.
  Configuration section.
    Special-names.
      xml-schema schema is 'ddschema'.
Data division.
  Working-storage section.
    1 xml-decode.
      2 rtn comp Pic 9(2).
      2 rsn comp-5 Pic 9(4).
    1 hv pic x(16) value '0123456789ABCDEF'.
    1 T Pic 999 COMP.
    1 xml-document-1.
      2 pic x(52) value
        '<!--Valid: the "itemName" element can be omitted-->'.
      2 pic x(31) value '<stockItem itemName="123-AB">'.
      2 pic x(36) value ' <quantityOnHand>1</quantityOnHand>'.
      2 pic x(12) value '</stockItem>'.
    1 xml-document-2.
      2 pic x(44)
        value '<!--Invalid: missing attribute itemName-->'.
      2 pic x(11) value '<stockItem>'.
      2 pic x(30) value ' <itemName>No name</itemName>'.
      2 pic x(36) value ' <quantityOnHand>1</quantityOnHand>'.
      2 pic x(12) value '</stockItem>'.
    1 xml-document-3.
      2 pic x(47)
        value '<!--Invalid: unexpected attribute warehouse-->'.
      2 pic x(46) value
        '<stockItem itemName="074-UN" warehouse="NJ">'.
      2 pic x(37) value ' <quantityOnHand>10</quantityOnHand>'.
      2 pic x(32) value ' <itemName>Not here!</itemName>'.
      2 pic x(12) value '</stockItem>'.
    1 xml-document-4.
      2 pic x(46)
```

```

    value '<!--Invalid: illegal attribute value 123-Ab-->'.
2 pic x(31) value '<stockItem itemNumber="123-Ab">'.
2 pic x(33) value ' <itemName>Paintbrush</itemName>'.
2 pic x(37) value ' <quantityOnHand>10</quantityOnHand>'.
2 pic x(12) value '</stockItem>'.
1 xml-document-5.
2 pic x(46)
    value '<!--Invalid: missing element quantityOnHand-->'.
2 pic x(31) value '<stockItem itemNumber="074-UN">'.
2 pic x(32) value ' <itemName>Not here!</itemName>'.
2 pic x(12) value '</stockItem>'.
1 xml-document-6.
2 pic x(42)
    value '<!--Invalid: unexpected element comment-->'.
2 pic x(31) value '<stockItem itemNumber="123-AB">'.
2 pic x(33) value ' <itemName>Paintbrush</itemName>'.
2 pic x(36) value ' <quantityOnHand>1</quantityOnHand>'.
2 pic x(35) value ' <comment>Nylon bristles</comment>'.
2 pic x(12) value '</stockItem>'.
1 xml-document-7.
2 pic x(46) value
    '<!--Invalid: out-of-range element value 100-->'.
2 pic x(31) value '<stockItem itemNumber="123-AB">'.
2 pic x(33) value ' <itemName>Paintbrush</itemName>'.
2 pic x(38) value ' <quantityOnHand>100</quantityOnHand>'.
2 pic x(12) value '</stockItem>'.
Procedure division.
m.
xml parse xml-document-1 validating with file schema
processing procedure p
xml parse xml-document-2 validating with file schema
processing procedure p
xml parse xml-document-3 validating with file schema
processing procedure p
xml parse xml-document-4 validating with file schema
processing procedure p
xml parse xml-document-5 validating with file schema
processing procedure p
xml parse xml-document-6 validating with file schema
processing procedure p
xml parse xml-document-7 validating with file schema
processing procedure p
goback
.
p.
evaluate xml-event
when 'COMMENT'
    display ' '
    display xml-text
when 'END-OF-DOCUMENT'
    display ' Document successfully parsed.'
when 'EXCEPTION'
    move xml-code to xml-decode
    Divide rsn by 16 giving tally remainder T
    display ' RC=' rtn ', reason=x'''
        hv(function mod(rsn / 4096 16) + 1:1)
        hv(function mod(rsn / 256 16) + 1:1)
        hv(function mod(rsn / 16 16) + 1:1)
        hv(T + 1:1) ''''
end-evaluate
.
End program ValidCk.

```

Output from program ValidCk

In the following output, you can see which XML documents in the source program failed validation against the schema.

For those documents that were not valid, the parser signaled an XML exception and passed control to the processing procedure with special register XML - EVENT containing 'EXCEPTION' and special-register XML - CODE containing the return code and a specific reason code.

```

Valid: the "itemName" element can be omitted
Document successfully parsed.

Invalid: missing attribute itemNumber
RC=24, reason=x'8613'

```

Invalid: unexpected attribute warehouse
RC=24, reason=x'8612'

Invalid: illegal attribute value 123-Ab
RC=24, reason=x'8809'

Invalid: missing element quantityOnHand
RC=24, reason=x'8611'

Invalid: unexpected element comment
RC=24, reason=x'8607'

Invalid: out-of-range element value 100
RC=24, reason=x'8803'

Related concepts

[“XML-CODE” on page 618](#)

[“XML schemas” on page 624](#)

Related tasks

[“Parsing XML documents with validation” on page 622](#)

[“Handling XML PARSE exceptions” on page 633](#)

Related references

[“XML PARSE exceptions with XMLPARSE\(XMLSS\) in effect” on page 795](#)

Chapter 38. Producing XML output

You can produce XML output from a COBOL program by using the XML GENERATE statement.

In the XML GENERATE statement, you identify the source and the output data items. You can optionally also identify:

- A field to receive a count of the XML characters generated
- A code page in which the generated XML document is to be encoded
- A *namespace* for the generated document
- A namespace prefix to qualify the start and end tag of each element, if you specify a namespace
- A user-defined element or attribute name in the generated XML document
- Attributes or elements to be suppressed according to some specified conditions
- Particular items to be specified as attributes, elements or content in the generated XML output.
- A statement to receive control if an exception occurs

Optionally, you can generate an XML declaration for the document, and can cause eligible source data items to be expressed as attributes in the output rather than as elements.

You can use the XML - CODE special register to determine the status of XML generation.

After you transform COBOL data items to XML, you can use the resulting XML output in various ways, such as deploying it in a web service, passing it as a message to WebSphere MQ, or transmitting it for subsequent conversion to a CICS communication area.

Link-edit considerations: COBOL programs that contain the XML GENERATE statement must be link-edited with AMODE 31.

Related tasks

[“Generating XML output” on page 651](#)

[“Controlling the encoding of generated XML output” on page 656](#)

[“Handling XML GENERATE exceptions” on page 656](#)

[“Enhancing XML output” on page 661](#)

Related references

[Extensible Markup Language \(XML\)](#)

XML GENERATE statement (*Enterprise COBOL for z/OS Language Reference*)

Generating XML output

To transform COBOL data to XML, use the XML GENERATE statement as in the example below.

```
XML GENERATE XML-OUTPUT FROM SOURCE-REC
COUNT IN XML-CHAR-COUNT
ON EXCEPTION
  DISPLAY 'XML generation error ' XML-CODE
  STOP RUN
NOT ON EXCEPTION
  DISPLAY 'XML document was successfully generated.'
END-XML
```

In the XML GENERATE statement, you first identify the data item (XML - OUTPUT in the example above) that is to receive the XML output. Define the data item to be large enough to contain the generated XML output, typically five to 10 times the size of the COBOL source data depending on the length of its data-name or data-names.

In the DATA DIVISION, you can define the receiving identifier as alphanumeric (either an alphanumeric group item or an elementary item of category alphanumeric) or as national (either a national group item or an elementary item of category national).

Next you identify the source data item that is to be transformed to XML format (SOURCE-REC in the example). The source data item can be an alphanumeric group item, national group item, or elementary data item of class alphanumeric or national.

Some COBOL data items are not transformed to XML, but are ignored. Subordinate data items of an alphanumeric group item or national group item that you transform to XML are ignored if they:

- Specify the REDEFINES clause, or are subordinate to such a redefining item
- Specify the RENAMES clause

These items in the source data item are also ignored when you generate XML:

- Elementary FILLER (or unnamed) data items
- Slack bytes inserted for SYNCHRONIZED data items

No extra white space (for example, new lines or indentation) is inserted to make the generated XML more readable.

Optionally, you can code the COUNT IN phrase to obtain the number of XML character encoding units that are filled during generation of the XML output. If the receiving identifier has category national, the count is in UTF-16 character encoding units. For all other encodings (including UTF-8), the count is in bytes.

You can use the count field as a reference modification length to obtain only that portion of the receiving data item that contains the generated XML output. For example, XML-OUTPUT(1:XML-CHAR-COUNT) references the first XML-CHAR-COUNT character positions of XML-OUTPUT.

Consider the following program excerpt:

```
01 doc pic x(512).
01 docSize pic 9(9) binary.
01 G.
   05 A pic x(3) value "aaa".
   05 B.
       10 C pic x(3) value "ccc".
       10 D pic x(3) value "ddd".
   05 E pic x(3) value "eee".
XML Generate Doc from G
```

The code above generates the following XML document, in which A, B, and E are expressed as child elements of element G, and C and D become child elements of element B:

```
<G><A>aaa</A><B><C>ccc</C><D>ddd</D></B><E>eee</E></G>
```

Alternatively, you can specify the ATTRIBUTES phrase of the XML GENERATE statement. The ATTRIBUTES phrase causes every eligible data item included in the generated XML document to be expressed as an attribute of the containing XML element, rather than as a child element of the containing XML element. To be eligible, the data item must be elementary, must have a name other than FILLER, and must not have an OCCURS clause in its data description entry. The containing XML element corresponds to the group data item that is immediately superordinate to the elementary data item. Optionally, you can specify more precise control of which data items should be expressed as attributes or elements by using the TYPE OF phrase.

For example, suppose that the XML GENERATE statement in the program excerpt above had instead been coded as follows:

```
XML Generate Doc from G with attributes
```

The code would then generate the following XML document, in which A and E are expressed as attributes of element G, and C and D become attributes of element B:

```
<G A="aaa" E="eee"><B C="ccc" D="ddd"></B></G>
```

Optionally, you can code the ENCODING phrase of the XML GENERATE statement to specify the CCSID of the generated XML document. If you do not use the ENCODING phrase, the document encoding is

determined by the category of the receiving data item and by the CODEPAGE compiler option. For further details, see the related task below about controlling the encoding of generated XML output.

Optionally, you can code the XML-DECLARATION phrase to cause the generated XML document to have an XML declaration that includes version information and an encoding declaration. If the receiving data item is of category:

- National: The encoding declaration has the value UTF-16 (encoding="UTF-16").
- Alphanumeric: The encoding declaration is derived from the ENCODING phrase, if specified, or from the CODEPAGE compiler option in effect for the program if the ENCODING phrase is not specified.

For example, the program excerpt below specifies the XML-DECLARATION phrase of XML GENERATE, and specifies encoding with CCSID 1208 (UTF-8):

```
01 Greeting.  
   05 msg pic x(80) value 'Hello, world!'.  
   XML Generate Doc from Greeting  
       with Encoding 1208  
       with XML-declaration  
   End-XML
```

The code above generates the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?><Greeting><msg>Hello, world!</msg></Greeting>
```

If you do not code the XML-DECLARATION phrase, an XML declaration is not generated.

Optionally, you can code the NAMESPACE phrase to specify a *namespace* for the generated XML document. The namespace value must be a valid *Uniform Resource Identifier (URI)*, for example, a URL (Uniform Resource Locator); for further details, see the related concept about URI syntax below.

Specify the namespace in an identifier or literal of either category national or alphanumeric.

If you specify a namespace, but do not specify a namespace prefix (described below), the namespace becomes the *default namespace* for the document. That is, the namespace define on the root element applies by default to each element name in the document, including the root element.

For example, consider the following data definitions and XML GENERATE statement:

```
01 Greeting.  
   05 msg pic x(80) value 'Hello, world!'.  
   01 NS pic x(20) value 'http://example'.  
   XML Generate Doc from Greeting  
       namespace is NS
```

The resulting XML document has a default namespace (http://example), as follows:

```
<Greeting xmlns="http://example"><msg>Hello, world!</msg></Greeting>
```

If you do not specify a namespace, the element names in the generated XML document are not in any namespace.

Optionally, you can code the NAMESPACE-PREFIX phrase to specify a prefix to be applied to the start and end tag of each element in the generated document. You can specify a prefix only if you have specified a namespace as described above.

When the XML GENERATE statement is executed, the prefix value must be a valid XML name, but without the colon (:); see the related reference below about namespaces for details. The value can have trailing spaces, which are removed before the prefix is used.

Specify the namespace prefix in an identifier or literal of either category national or alphanumeric.

It is recommended that the prefix be short, because it qualifies the start and end tag of each element.

For example, consider the following data definitions and XML GENERATE statement:

```
01 Greeting.  
   05 msg pic x(80) value 'Hello, world!'.  
01 NS pic x(20) value 'http://example'.  
01 NP pic x(5) value 'pre'.  
  
XML Generate Doc from Greeting  
   namespace is NS  
   namespace-prefix is NP
```

The resulting XML document has an explicit namespace (`http://example`), and the prefix `pre` is applied to the start and end tag of the elements `Greeting` and `msg`, as follows:

```
<pre:Greeting xmlns:pre="http://example"><pre:msg>Hello, world!</pre:msg></pre:Greeting>
```

Optionally, you can code the `NAME` phrase to specify attribute and element names in the generated XML document. The attribute and element names must be alphanumeric or national literals and must be legal names according to the XML 1.0 standard.

For example, consider the following data structure and XML GENERATE statement:

```
01 Msg.  
   02 Msg-Severity pic 9 value 1.  
   02 Msg-Date pic 9999/99/99 value "2012/04/12".  
   02 Msg-Text pic X(50) value "Sell everything!".  
01 Doc pic X(500).
```

```
XML Generate Doc from Msg  
With attributes  
Name of Msg is "Message"  
Msg-Severity is "Severity"  
Msg-Date is "Date"  
Msg-Text is "Text"  
  
End-XML
```

The resulting XML document is as follows:

```
<Message Severity="1" Date="2012/04/12" Text="Sell everything!"></Message>
```

Optionally, you can code the `SUPPRESS` phrase to specify whether individual data items are generated based on whether or not they meet certain criteria.

For example, consider the following data structure and XML GENERATE statement to suppress spaces and zeros:

```
01 G.  
   02 SensitiveInfo.  
       03 SSN pic x(11) value '123-45-6789'.  
       03 HomeAddress pic x(50) value '123 Main St, Anytown, USA'.  
   02 Aarray value spaces.  
       03 A pic AAA occurs 5.  
   02 Barray value spaces.  
       03 B pic XXX occurs 5.  
   02 Carray value zeros.  
       03 C pic 999 occurs 5.  
Move 'abc' to A(1)  
Move 123 to C(3)  
XML Generate Doc from G  
   Suppress SensitiveInfo  
       every nonnumeric element when space  
       every numeric element when zero  
  
End-XML
```

The resulting XML document is as follows:

```
<G>  
<Aarray><A>abc</A></Aarray>
```

```
<Carray><C>123</C></Carray>
</G>
```

Optionally, you can use the TYPE OF phrase to specify whether individual data items are expressed as attributes, elements or content.

For example, consider the following data structure and XML GENERATE statement:

```
01 Msg.
  02 Msg-Severity pic 9 value 1.
  02 Msg-Date pic 9999/99/99 value "2012/04/12".
  02 Msg-Text pic X(50) value "Sell everything!".
01 Doc pic X(500).
  XML Generate Doc from Msg
  With attributes
  Type of Msg-Severity is attribute
      Msg-Date is attribute
      Msg-Text is element
End-XML
```

The resulting XML document is as follows:

```
<Msg Msg-Severity="1" Msg-Date="2012/04/12">
  <Msg-Text>Sell everything!</Msg-Text></Msg>
```

In addition, you can specify either or both of the following phrases to receive control after generation of the XML document:

- ON EXCEPTION, to receive control if an error occurs during XML generation
- NOT ON EXCEPTION, to receive control if no error occurs

You can end the XML GENERATE statement with the explicit scope terminator END-XML. Code END-XML to nest an XML GENERATE statement that has the ON EXCEPTION or NOT ON EXCEPTION phrase in a conditional statement.

XML generation continues until either the COBOL source record has been transformed to XML or an error occurs. If an error occurs, the results are as follows:

- The XML-CODE special register contains a nonzero exception code.
- Control is passed to the ON EXCEPTION phrase, if specified, otherwise to the end of the XML GENERATE statement.

If no error occurs during XML generation, the XML-CODE special register contains zero, and control is passed to the NOT ON EXCEPTION phrase if specified or to the end of the XML GENERATE statement otherwise.

[“Example: generating XML” on page 657](#)

Related concepts

[Uniform Resource Identifier \(URI\): Generic Syntax](#)

Related tasks

[“Controlling the encoding of generated XML output” on page 656](#)

[“Handling XML GENERATE exceptions” on page 656](#)

[“Processing UTF-8 data using UTF-16 \(national\) data types” on page 153](#)

Related references

XML GENERATE statement (*Enterprise COBOL for z/OS Language Reference*)

[Extensible Markup Language \(XML\)](#)

[Namespaces in XML 1.0](#)

Controlling the encoding of generated XML output

When you generate XML output by using the XML GENERATE statement, you can control the encoding of the output by the category of the data item that receives the output, and by identifying the code page using the WITH ENCODING phrase of the XML GENERATE statement.

If you specify the WITH ENCODING *codepage* phrase to designate the coded character set identifier (CCSID) of the output document, *codepage* must specify an unsigned integer data item or unsigned integer literal that identifies one of the code pages supported for COBOL XML processing as described in the related reference below about the encoding of XML documents:

- If the data item that receives the generated XML is of category national, the WITH ENCODING phrase must specify 1200, the CCSID for Unicode UTF-16.
- If the receiving identifier is of category alphanumeric, the WITH ENCODING phrase must specify CCSID 1208 or the CCSID of a supported EBCDIC code page.

If you do not code the WITH ENCODING phrase, the generated XML output is encoded as shown in the table below.

If you define the receiving XML identifier as:	The generated XML output is encoded in:
Alphanumeric	The code page specified by the CODEPAGE compiler option in effect when the source was compiled
National	UTF-16 big-endian (UTF-16BE, CCSID 1200)

A byte order mark is not generated.

For details about how data items are converted to XML and how the XML element names and attributes names are formed from the COBOL data-names, see the related reference below about the operation of the XML GENERATE statement.

Related references

[“CODEPAGE” on page 343](#)

[“The encoding of XML documents” on page 628](#)

XML GENERATE statement (*Enterprise COBOL for z/OS Language Reference*)

Operation of XML GENERATE (*Enterprise COBOL for z/OS Language Reference*)

Handling XML GENERATE exceptions

When an error is detected during generation of XML output, an exception condition exists. You can write code to check the XML-CODE special register, which contains a numeric exception code that indicates the error type.

To handle errors, use either or both of the following phrases of the XML GENERATE statement:

- ON EXCEPTION
- COUNT IN

If you code the ON EXCEPTION phrase in the XML GENERATE statement, control is transferred to the imperative statement that you specify. You might code an imperative statement, for example, to display the XML-CODE value. If you do not code an ON EXCEPTION phrase, control is transferred to the end of the XML GENERATE statement.

When an error occurs, one problem might be that the data item that receives the XML output is not large enough. In that case, the XML output is not complete, and the XML-CODE special register contains error code 400®.

You can examine the generated XML output by doing these steps:

1. Code the COUNT IN phrase in the XML GENERATE statement.

The count field that you specify holds a count of the XML character encoding units that are filled during XML generation. If you define the XML output as national, the count is in UTF-16 character encoding units; for all other encodings (including for UTF-8), the count is in bytes.

2. Use the count field as a reference modification length to refer to the substring of the receiving data item that contains the XML characters that were generated until the point when the error occurred.

For example, if XML-OUTPUT is the data item that receives the XML output, and XML-CHAR-COUNT is the count field, then XML-OUTPUT(1:XML-CHAR-COUNT) references the XML output.

Use the contents of XML-CODE to determine what corrective action to take. For a list of the exceptions that can occur during XML generation, see the related reference below.

Related tasks

[“Referring to substrings of data items” on page 109](#)

Related references

[“XML GENERATE exceptions” on page 803](#)

XML-CODE (*Enterprise COBOL for z/OS Language Reference*)

Example: generating XML

The following example simulates the building of a purchase order in a group data item, and generates an XML version of that purchase order.

Program XGFX uses XML GENERATE to produce XML output in elementary data item xmlPO from the source record, group data item purchaseOrder. Elementary data items in the source record are converted to character format as necessary, and the characters are inserted as the values of XML attributes whose names are derived from the data-names in the source record.

XGFX calls program Pretty, which uses the XML PARSE statement with processing procedure p to format the XML output with new lines and indentation so that the XML content can more easily be verified.

Program XGFX

```
Identification division.
  Program-id. XGFX.
Data division.
  Working-storage section.
    01 numItems pic 99 global.
    01 purchaseOrder global.
      05 orderDate pic x(10).
      05 shipTo.
        10 country pic xx value 'US'.
        10 name pic x(30).
        10 street pic x(30).
        10 city pic x(30).
        10 state pic xx.
        10 zip pic x(10).
      05 billTo.
        10 country pic xx value 'US'.
        10 name pic x(30).
        10 street pic x(30).
        10 city pic x(30).
        10 state pic xx.
        10 zip pic x(10).
      05 orderComment pic x(80).
      05 items occurs 0 to 20 times depending on numItems.
        10 item.
          15 partNum pic x(6).
          15 productName pic x(50).
          15 quantity pic 99.
          15 USPrice pic 999v99.
          15 shipDate pic x(10).
          15 itemComment pic x(40).
    01 numChars comp pic 999.
    01 xmlPO pic x(999).
Procedure division.
```

```

m.
  Move 20 to numItems
  Move spaces to purchaseOrder

  Move '1999-10-20' to orderDate

  Move 'US' to country of shipTo
  Move 'Alice Smith' to name of shipTo
  Move '123 Maple Street' to street of shipTo
  Move 'Mill Valley' to city of shipTo
  Move 'CA' to state of shipTo
  Move '90952' to zip of shipTo

  Move 'US' to country of billTo
  Move 'Robert Smith' to name of billTo
  Move '8 Oak Avenue' to street of billTo
  Move 'Old Town' to city of billTo
  Move 'PA' to state of billTo
  Move '95819' to zip of billTo
  Move 'Hurry, my lawn is going wild!' to orderComment

  Move 0 to numItems
  Call 'addFirstItem'
  Call 'addSecondItem'
  Move space to xmlPO
  Xml generate xmlPO from purchaseOrder count in numChars
    with xml-declaration with attributes
      namespace 'http://www.example.com' namespace-prefix 'po'
  Call 'pretty' using xmlPO value numChars
  Goback
.

Identification division.
  Program-id. 'addFirstItem'.
Procedure division.
  Add 1 to numItems
  Move '872-AA' to partNum(numItems)
  Move 'Lawnmower' to productName(numItems)
  Move 1 to quantity(numItems)
  Move 148.95 to USPrice(numItems)
  Move 'Confirm this is electric' to itemComment(numItems)
  Goback.
End program 'addFirstItem'.

Identification division.
  Program-id. 'addSecondItem'.
Procedure division.
  Add 1 to numItems
  Move '926-AA' to partNum(numItems)
  Move 'Baby Monitor' to productName(numItems)
  Move 1 to quantity(numItems)
  Move 39.98 to USPrice(numItems)
  Move '1999-05-21' to shipDate(numItems)
  Goback.
End program 'addSecondItem'.

End program XGFX.

```

Program Pretty

```

Process xmlparse(xmlss), codepage(37)
Identification division.
  Program-id. Pretty.
Data division.
  Working-storage section.
    01 prettyPrint.
      05 pose pic 999.
      05 posd pic 999.
      05 depth pic 99.
      05 inx pic 999.
      05 elementName pic x(30).
      05 indent pic x(40).
      05 buffer pic x(998).
      05 lastitem pic 9.
      88 unknown value 0.
      88 xml-declaration value 1.
      88 element value 2.
      88 attribute value 3.

```

```

    88 charcontent value 4.
Linkage section.
1 doc.
  2 pic x occurs 16384 times depending on len.
  1 len comp-5 pic 9(9).
Procedure division using doc value len.
m.
  Move space to prettyPrint
  Move 0 to depth
  Move 1 to posd pose
  Xml parse doc processing procedure p
  Goback
.
p.
  Evaluate xml-event
  When 'VERSION-INFORMATION'
    String '<?xml version="' xml-text '"' delimited by size
      into buffer with pointer posd
    Set xml-declaration to true
  When 'ENCODING-DECLARATION'
    String ' encoding="' xml-text '"' delimited by size
      into buffer with pointer posd
  When 'STANDALONE-DECLARATION'
    String ' standalone="' xml-text '"' delimited by size
      into buffer with pointer posd
  When 'START-OF-ELEMENT'
    Evaluate true
    When xml-declaration
      String '?>' delimited by size into buffer
        with pointer posd
      Set unknown to true
      Perform printline
      Move 1 to posd
    When element
      String '>' delimited by size into buffer
        with pointer posd
    When attribute
      String '">' delimited by size into buffer
        with pointer posd
    End-evaluate
    If elementName not = space
      Perform printline
    End-if
    Move xml-text to elementName
    Add 1 to depth
    Move 1 to pose
    Set element to true
    If xml-namespace-prefix = space
      String '<' xml-text delimited by size
        into buffer with pointer pose
    Else
      String '<' xml-namespace-prefix ':' xml-text
        delimited by size into buffer with pointer pose
    End-if
    Move pose to posd
  When 'ATTRIBUTE-NAME'
    If element
      String ' ' delimited by size into buffer
        with pointer posd
    Else
      String '" ' delimited by size into buffer
        with pointer posd
    End-if
    If xml-namespace-prefix = space
      String xml-text '=' delimited by size into buffer
        with pointer posd
    Else
      String xml-namespace-prefix ':' xml-text '='
        delimited by size into buffer with pointer posd
    End-if
    Set attribute to true
  When 'NAMESPACE-DECLARATION'
    If element
      String ' ' delimited by size into buffer
        with pointer posd
    Else
      String '" ' delimited by size into buffer
        with pointer posd
    End-if
    If xml-namespace-prefix = space
      String 'xmlns="' xml-namespace delimited by size
        into buffer with pointer posd

```

```

Else
  String 'xmlns:' xml-namespace-prefix '=' xml-namespace
  delimited by size into buffer with pointer posd
End-if
Set attribute to true
When 'ATTRIBUTE-CHARACTERS'
  String xml-text delimited by size into buffer
  with pointer posd
When 'ATTRIBUTE-CHARACTER'
  String xml-text delimited by size into buffer
  with pointer posd
When 'CONTENT-CHARACTERS'
  Evaluate true
  When element
    String '>' delimited by size into buffer
    with pointer posd
  When attribute
    String '">' delimited by size into buffer
    with pointer posd
  End-evaluate
  String xml-text delimited by size into buffer
  with pointer posd
  Set charcontent to true
When 'CONTENT-CHARACTER'
  Evaluate true
  When element
    String '>' delimited by size into buffer
    with pointer posd
  When attribute
    String '">' delimited by size into buffer
    with pointer posd
  End-evaluate
  String xml-text delimited by size into buffer
  with pointer posd
  Set charcontent to true
When 'END-OF-ELEMENT'
  Move space to elementName
  Evaluate true
  When element
    String '/>' delimited by size into buffer
    with pointer posd
  When attribute
    String '">' delimited by size into buffer
    with pointer posd
  When other
    If xml-namespace-prefix = space
      String '</' xml-text '>' delimited by size
      into buffer with pointer posd
    Else
      String '</' xml-namespace-prefix ':' xml-text '>'
      delimited by size into buffer with pointer posd
    End-if
  End-evaluate
  Set unknown to true
  Perform printline
  Subtract 1 from depth
  Move 1 to posd
When other
  Continue
End-evaluate
.
printline.
Compute inx = function max(0 2 * depth - 2) + posd - 1
If inx > 120
  compute inx = 117 - function max(0 2 * depth - 2)
  If depth > 1
    Display indent(1:2 * depth - 2) buffer(1:inx) '...'
  Else
    Display buffer(1:inx) '...'
  End-if
Else
  If depth > 1
    Display indent(1:2 * depth - 2) buffer(1:posd - 1)
  Else
    Display buffer(1:posd - 1)
  End-if
End-if
.
End program Pretty.

```


Output from program XGFX

```
<?xml version="1.0" encoding="IBM-037"?>
<po:purchaseOrder xmlns:po="http://www.example.com" orderDate="1999-10-20" orderComment="Hurry, my lawn
is going wild!">
  <po:shipTo country="US" name="Alice Smith" street="123 Maple Street" city="Mill Valley" state="CA"
zip="90952"/>
  <po:billTo country="US" name="Robert Smith" street="8 Oak Avenue" city="Old Town" state="PA"
zip="95819"/>
  <po:items>
    <po:item partNum="872-AA" productName="Lawnmower" quantity="1" USPrice="148.95" shipDate=" "
itemComment="Confirm..."
  </po:items>
  <po:items>
    <po:item partNum="926-AA" productName="Baby Monitor" quantity="1" USPrice="39.98"
shipDate="1999-05-21" itemComme...
  </po:items>
</po:purchaseOrder>
```

Related tasks

Chapter 37, “Processing XML input,” on page 611

Related references

“XMLPARSE” on page 415 (compiler option)

Operation of XML GENERATE (*Enterprise COBOL for z/OS Language Reference*)

Enhancing XML output

It might happen that the information that you want to express in XML format already exists in a group item in the DATA DIVISION, but you are unable to use that item directly to generate an XML document because of one or more factors.

For example:

- In addition to the required data, the item has subordinate data items that contain values that are irrelevant to the XML output document.
- The names of the required data items are unsuitable for external presentation, and are possibly meaningful only to programmers.
- The required data items are broken up into too many components, and should be output as the content of the containing group.

There are various ways that you can deal with such situations. One possible technique is to define a new data item that has the appropriate characteristics, and move the required data to the appropriate fields of this new data item. However, this approach is somewhat laborious and requires careful maintenance to keep the original and new data items synchronized.

A superior approach that addresses most such problems is to use the new optional phrases of the XML GENERATE statement in order to:

- Provide more meaningful and appropriate names for the selected elementary items and for the group items that contain them.
- Exclude irrelevant data items from the generated XML by suppressing them based on their values.

The example that is referenced below shows a way to do so.

[“Example: enhancing XML output” on page 662](#)

Related references

Operation of XML GENERATE (*Enterprise COBOL for z/OS Language Reference*)

Example: enhancing XML output

The following example shows how you can modify XML output.

Consider the following data structure. The XML that is generated from the structure suffers from several problems that can be corrected.

```
01  CDR-LIFE-BASE-VALUES-BOX.
    15  CDR-LIFE-BASE-VAL-DATE      PIC X(08).
    15  CDR-LIFE-BASE-VALUE-LINE  OCCURS 2 TIMES.
        20  CDR-LIFE-BASE-DESC.
            25  CDR-LIFE-BASE-DESC1 PIC X(15).
            25  FILLER                PIC X(01).
            25  CDR-LIFE-BASE-LIT     PIC X(08).
            25  CDR-LIFE-BASE-DTE     PIC X(08).
        20  CDR-LIFE-BASE-PRICE.
            25  CDR-LIFE-BP-SPACE    PIC 9(08).
            25  CDR-LIFE-BP-DASH     PIC X.
            25  CDR-LIFE-BP-SPACE1   PIC X(02).
        20  CDR-LIFE-BASE-PRICE-ED  REDEFINES
            CDR-LIFE-BASE-PRICE     PIC $$$.$$.
        20  CDR-LIFE-BASE-QTY.
            25  CDR-LIFE-QTY-SPACE   PIC X(08).
            25  CDR-LIFE-QTY-DASH    PIC X.
            25  CDR-LIFE-QTY-SPACE1  PIC X(03).
            25  FILLER                PIC X(02).
        20  CDR-LIFE-BASE-VALUE     PIC $$$9.99
            BLANK WHEN ZERO.
    15  CDR-LIFE-BASE-TOT-VALUE     PIC X(15)
```

When this data structure is populated with some sample values, and XML is generated directly from it and then formatted using program Pretty (shown in [“Example: generating XML”](#) on page 657), the result is as follows:

```
<CDR-LIFE-BASE-VALUES-BOX>
  <CDR-LIFE-BASE-VAL-DATE>01/02/03</CDR-LIFE-BASE-VAL-DATE>
  <CDR-LIFE-BASE-VALUE-LINE>
    <CDR-LIFE-BASE-DESC>
      <CDR-LIFE-BASE-DESC1>First</CDR-LIFE-BASE-DESC1>
      <CDR-LIFE-BASE-LIT> </CDR-LIFE-BASE-LIT>
      <CDR-LIFE-BASE-DTE>01/01/01</CDR-LIFE-BASE-DTE>
    </CDR-LIFE-BASE-DESC>
    <CDR-LIFE-BASE-PRICE>
      <CDR-LIFE-BP-SPACE>23</CDR-LIFE-BP-SPACE>
      <CDR-LIFE-BP-DASH>.</CDR-LIFE-BP-DASH>
      <CDR-LIFE-BP-SPACE1>00</CDR-LIFE-BP-SPACE1>
    </CDR-LIFE-BASE-PRICE>
    <CDR-LIFE-BASE-QTY>
      <CDR-LIFE-QTY-SPACE>123</CDR-LIFE-QTY-SPACE>
      <CDR-LIFE-QTY-DASH>.</CDR-LIFE-QTY-DASH>
      <CDR-LIFE-QTY-SPACE1>000</CDR-LIFE-QTY-SPACE1>
    </CDR-LIFE-BASE-QTY>
    <CDR-LIFE-BASE-VALUE>$765.00</CDR-LIFE-BASE-VALUE>
  </CDR-LIFE-BASE-VALUE-LINE>
  <CDR-LIFE-BASE-VALUE-LINE>
    <CDR-LIFE-BASE-DESC>
      <CDR-LIFE-BASE-DESC1>Second</CDR-LIFE-BASE-DESC1>
      <CDR-LIFE-BASE-LIT> </CDR-LIFE-BASE-LIT>
      <CDR-LIFE-BASE-DTE>02/02/02</CDR-LIFE-BASE-DTE>
    </CDR-LIFE-BASE-DESC>
    <CDR-LIFE-BASE-PRICE>
      <CDR-LIFE-BP-SPACE>34</CDR-LIFE-BP-SPACE>
      <CDR-LIFE-BP-DASH>.</CDR-LIFE-BP-DASH>
      <CDR-LIFE-BP-SPACE1>00</CDR-LIFE-BP-SPACE1>
    </CDR-LIFE-BASE-PRICE>
    <CDR-LIFE-BASE-QTY>
      <CDR-LIFE-QTY-SPACE>234</CDR-LIFE-QTY-SPACE>
      <CDR-LIFE-QTY-DASH>.</CDR-LIFE-QTY-DASH>
      <CDR-LIFE-QTY-SPACE1>000</CDR-LIFE-QTY-SPACE1>
    </CDR-LIFE-BASE-QTY>
    <CDR-LIFE-BASE-VALUE>$654.00</CDR-LIFE-BASE-VALUE>
  </CDR-LIFE-BASE-VALUE-LINE>
  <CDR-LIFE-BASE-TOT-VALUE>Very high!</CDR-LIFE-BASE-TOT-VALUE>
</CDR-LIFE-BASE-VALUES-BOX>
```

This generated XML suffers from several problems:

- The element names are long and not very meaningful. There may also be an XML schema that specifies required tag names.
- The XML schema may require some tag names that are COBOL reserved words such as DATE/TIME
- Some fields that are elements should be attributes such as, CDR-LIFE-BASE-VAL-DATE and CDR-LIFE-BASE-DESC1.
- There is unwanted data, for example, CDR-LIFE-BASE-LIT and CDR-LIFE-BASE-DTE.
- Other required fields are split into too many subcomponents. For example, CDR-LIFE-BASE-PRICE has three subcomponents for one amount.

These and other characteristics of the XML output can be remedied by using additional phrases of the XML GENERATE statement as follows:

- Use the NAME OF phrase to provide appropriate tag or attribute names.
- Use the TYPE OF ... IS ATTRIBUTE phrase to select the fields which should be XML attributes rather than elements.
- Use the TYPE OF ... IS CONTENT phrase to suppress tags for excessive subcomponents.
- Use the SUPPRESS ... WHEN phrase to exclude fields that contain uninteresting values.

Here is an example of the XML GENERATE statement to address those problems:

```
XML generate Doc from CDR-LIFE-BASE-VALUES-BOX
Count in tally
Name of
  CDR-LIFE-BASE-VALUES-BOX
  is 'Base_Values'
  CDR-LIFE-BASE-VAL-DATE
  is 'Date'
  CDR-LIFE-BASE-DTE
  is 'Date'
  CDR-LIFE-BASE-VALUE-LINE
  is 'BaseValueLine'
  CDR-LIFE-BASE-DESC1
  is 'Description'
  CDR-LIFE-BASE-PRICE
  is 'BasePrice'
  CDR-LIFE-BASE-QTY
  is 'BaseQuantity'
  CDR-LIFE-BASE-VALUE
  is 'BaseValue'
  CDR-LIFE-BASE-TOT-VALUE
  is 'TotalValue'
Type of
  CDR-LIFE-BASE-VAL-DATE is attribute
  CDR-LIFE-BASE-DESC1 is attribute
  CDR-LIFE-BP-SPACE is content
  CDR-LIFE-BP-DASH is content
  CDR-LIFE-BP-SPACE1 is content
  CDR-LIFE-QTY-SPACE is content
  CDR-LIFE-QTY-DASH is content
  CDR-LIFE-QTY-SPACE1 is content
Suppress every nonnumeric when space
every numeric when zero
```

The result of generating and formatting XML from the statement shown above is more usable:

```
<Base_Values Date="01/02/03">
  <BaseValueLine Description="First">
    <Date>01/01/01</Date>
    <BasePrice>23.00</BasePrice>
    <BaseQuantity>123.000</BaseQuantity>
    <BaseValue>$765.00</BaseValue>
  </BaseValueLine>
  <BaseValueLine Description="Second">
    <Date>02/02/02</Date>
    <BasePrice>34.00</BasePrice>
    <BaseQuantity>234.000</BaseQuantity>
    <BaseValue>$654.00</BaseValue>
```

```
</BaseValueLine>  
<TotalValue>Very high!</TotalValue>  
</Base_Values>
```

Note that the COBOL reserved word DATE can now be used as an XML tag name in the output. Characters such as accented letters and period . that are illegal in single-byte data names can also be used.

Related references

Operation of XML GENERATE (*Enterprise COBOL for z/OS Language Reference*)

REPLACE statement (*Enterprise COBOL for z/OS Language Reference*)

Part 6. Developing object-oriented programs

Chapter 39. Writing object-oriented programs

When you write an object-oriented (OO) program, you have to determine what classes you need and the methods and data that the classes need to do their work.

OO programs are based on *objects* (entities that encapsulate state and behavior) and their classes, methods, and data. A *class* is a template that defines the state and the capabilities of an object. Usually a program creates and works with multiple *object instances* (or simply, *instances*) of a class, that is, multiple objects that are members of that class. The state of each instance is stored in data known as *instance data*, and the capabilities of each instance are called *instance methods*. A class can define data that is shared by all instances of the class, known as *factory* or *static* data, and methods that are supported independently of any object instance, known as *factory* or *static* methods.

Using Enterprise COBOL, you can:

- Define classes, with methods and data implemented in COBOL.
- Create instances of Java and COBOL classes.
- Invoke methods on Java and COBOL objects.
- Write classes that inherit from Java classes or other COBOL classes.
- Define and invoke overloaded methods.

In Enterprise COBOL programs, you can call the services provided by the Java Native Interface (JNI) to obtain Java-oriented capabilities in addition to the basic OO capabilities available directly in the COBOL language.

In Enterprise COBOL classes, you can code CALL statements to interface with procedural COBOL programs. Thus COBOL class definition syntax can be especially useful for writing *wrapper* classes for procedural COBOL logic, enabling existing COBOL code to be accessed from Java.

Java code can create instances of COBOL classes, invoke methods of these classes, and can extend COBOL classes.

It is recommended that you develop and run OO COBOL programs and Java programs in the z/OS UNIX environment.

Restrictions:

- COBOL class definitions and methods cannot contain EXEC SQL statements and cannot be compiled using the SQL compiler option.
- COBOL class definitions and methods cannot contain EXEC SQLIMS statements and cannot be compiled using the SQLIMS compiler option.
- COBOL programs that use object-oriented syntax for Java interoperability cannot contain EXEC CICS statements, and cannot be run in CICS. They cannot be compiled using the CICS compiler option.

[“Example: accounts” on page 668](#)

Related tasks

[“Defining a class” on page 670](#)

[“Defining a class instance method” on page 674](#)

[“Defining a client” on page 682](#)

[“Defining a subclass” on page 692](#)

[“Defining a factory section” on page 695](#)

[Chapter 16, “Compiling, linking, and running OO applications,” on page 291](#)

Upgrading IBM COBOL source programs

(*Enterprise COBOL for z/OS Migration Guide*)

Related references

[The Java Language Specification](#)

Example: accounts

Consider the example of a bank in which customers can open accounts and make deposits to and withdrawals from their accounts. You could represent an account by a general-purpose class, called Account. Because there are many customers, multiple instances of the Account class could exist simultaneously.

After you determine the classes that you need, the next step is to determine the methods that the classes need to do their work. An Account class must provide the following services:

- Open the account.
- Get the current balance.
- Deposit to the account.
- Withdraw from the account.
- Report account status.

The following methods for an Account class meet those needs:

init

Open an account and assign it an account number.

getBalance

Return the current balance of the account.

credit

Deposit a given sum to the account.

debit

Withdraw a given sum from the account.

print

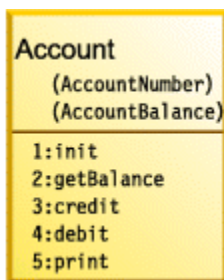
Display account number and account balance.

As you design an Account class and its methods, you discover the need for the class to keep some instance data. Typically, an Account object needs the following instance data:

- Account number
- Account balance
- Customer information: name, address, home phone, work phone, social security number, and so forth

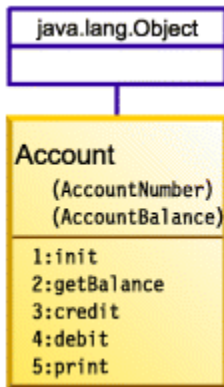
To keep the example simple, however, it is assumed that the account number and account balance are the only instance data that the Account class needs.

Diagrams are helpful when you design classes and methods. The following diagram depicts a first attempt at a design of the Account class:



The words in parentheses in the diagrams are the names of the instance data, and the words that follow a number and colon are the names of the instance methods.

The structure below shows how the classes relate to each other, and is known as the *inheritance hierarchy*. The Account class inherits directly from the class `java.lang.Object`.



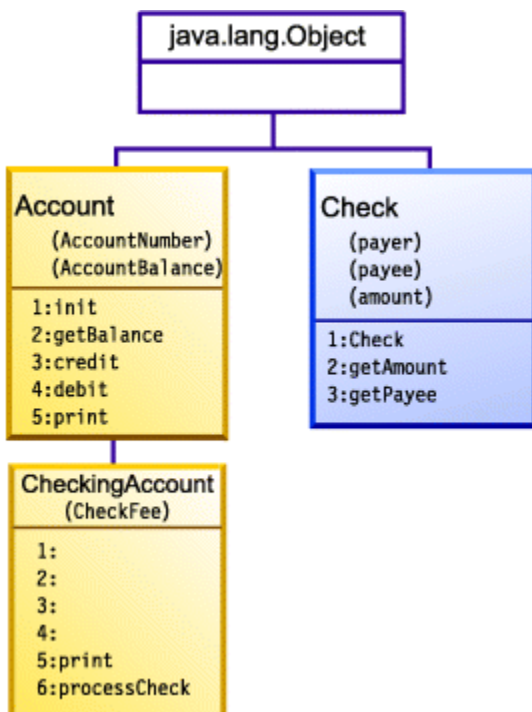
Subclasses

In the account example, Account is a general-purpose class. However, a bank could have many different types of accounts: checking accounts, savings accounts, mortgage loans, and so forth, all of which have all the general characteristics of accounts but could have additional characteristics not shared by all types of accounts.

For example, a CheckingAccount class could have, in addition to the account number and account balance that all accounts have, a check fee that applies to each check written on the account. A CheckingAccount class also needs a method to process checks (that is, to read the amount, debit the payer, credit the payee, and so forth). So it makes sense to define CheckingAccount as a subclass of Account, and to define in the subclass the additional instance data and instance methods that the subclass needs.

As you design the CheckingAccount class, you discover the need for a class that models checks. An instance of class Check needs, at a minimum, instance data for payer, payee, and the check amount.

Many additional classes (and database and transaction-processing logic) would need to be designed in a real-world OO account system, but have been omitted to keep the example simple. The updated inheritance diagram is shown below.



A number and colon with no method-name following them indicate that the method with that number is inherited from the superclass.

Multiple inheritance: You cannot use *multiple inheritance* in OO COBOL applications. All classes that you define must have exactly one parent, and java.lang.Object must be at the root of every inheritance hierarchy. The class structure of any object-oriented system defined in an OO COBOL application is thus a tree.

[“Example: defining a method” on page 680](#)

Related tasks

[“Defining a class” on page 670](#)

[“Defining a class instance method” on page 674](#)

[“Defining a subclass” on page 692](#)

Defining a class

A COBOL class definition consists of an IDENTIFICATION DIVISION and ENVIRONMENT DIVISION, followed by an optional factory definition and optional object definition, followed by an END CLASS marker.

Table 85. Structure of class definitions

Section	Purpose	Syntax
IDENTIFICATION DIVISION (required)	Name the class. Provide inheritance information for it.	“CLASS-ID paragraph for defining a class” on page 671 (required) AUTHOR paragraph (optional) INSTALLATION paragraph (optional) DATE-WRITTEN paragraph (optional) DATE-COMPILED paragraph (optional)
ENVIRONMENT DIVISION (required)	Describe the computing environment. Relate class-names used within the class definition to the corresponding external class-names known outside the compilation unit.	CONFIGURATION SECTION (required) “REPOSITORY paragraph for defining a class” on page 672 (required) SOURCE-COMPUTER paragraph (optional) OBJECT-COMPUTER paragraph (optional) SPECIAL-NAMES paragraph (optional)
Factory definition (optional)	Define data to be shared by all instances of the class, and methods supported independently of any object instance.	<pre>IDENTIFICATION DIVISION. FACTORY. DATA DIVISION. WORKING-STORAGE SECTION. * (Factory data here) PROCEDURE DIVISION. * (Factory methods here) END FACTORY.</pre>
Object definition (optional)	Define instance data and instance methods.	<pre>IDENTIFICATION DIVISION. OBJECT. DATA DIVISION. WORKING-STORAGE SECTION. * (Instance data here) PROCEDURE DIVISION. * (Instance methods here) END OBJECT.</pre>

If you specify the SOURCE-COMPUTER, OBJECT-COMPUTER, or SPECIAL-NAMES paragraphs in a class CONFIGURATION SECTION, they apply to the entire class definition including all methods that the class introduces.

A class CONFIGURATION SECTION can consist of the same entries as a program CONFIGURATION SECTION, except that a class CONFIGURATION SECTION cannot contain an INPUT-OUTPUT SECTION. You define an INPUT-OUTPUT SECTION only in the individual methods that require it rather than defining it at the class level.

As shown above, you define instance data and methods in the DATA DIVISION and PROCEDURE DIVISION, respectively, within the OBJECT paragraph of the class definition. In classes that require data and methods that are to be associated with the class itself rather than with individual object instances, define a separate DATA DIVISION and PROCEDURE DIVISION within the FACTORY paragraph of the class definition.

Each COBOL class definition must be in a separate source file.

[“Example: defining a class” on page 674](#)

Related tasks

[“WORKING-STORAGE SECTION for defining class instance data” on page 673](#)

[“Defining a class instance method” on page 674](#)

[“Defining a subclass” on page 692](#)

[“Defining a factory section” on page 695](#)

[“Describing the computing environment” on page 5](#)

[Chapter 16, “Compiling, linking, and running OO applications,” on page 291](#)

Related references

COBOL class definition structure (*Enterprise COBOL for z/OS Language Reference*)

CLASS-ID paragraph for defining a class

Use the CLASS-ID paragraph in the IDENTIFICATION DIVISION to name a class and provide inheritance information for it.

```
Identification Division.      Required
Class-id. Account inherits Base. Required
```

Use the CLASS-ID paragraph to identify these classes:

- The class that you are defining (Account in the example above).
- The immediate superclass from which the class that you are defining inherits its characteristics. The superclass can be implemented in Java or COBOL.

In the example above, `inherits Base` indicates that the Account class inherits methods and data from the class known within the class definition as Base. It is recommended that you use the name Base in your OO COBOL programs to refer to `java.lang.Object`.

A class-name must use single-byte characters and must conform to the normal rules of formation for a COBOL user-defined word.

Use the REPOSITORY paragraph in the CONFIGURATION SECTION of the ENVIRONMENT DIVISION to associate the superclass name (Base in the example) with the name of the superclass as it is known externally (`java.lang.Object` for Base). You can optionally also specify the name of the class that you are defining (Account in the example) in the REPOSITORY paragraph and associate it with its corresponding external class-name.

You must derive all classes directly or indirectly from the `java.lang.Object` class.

Related tasks

[“REPOSITORY paragraph for defining a class” on page 672](#)

Related references

CLASS-ID paragraph (*Enterprise COBOL for z/OS Language Reference*)

User-defined words (*Enterprise COBOL for z/OS Language Reference*)

REPOSITORY paragraph for defining a class

Use the REPOSITORY paragraph to declare to the compiler that the specified words are class-names when you use them within a class definition, and to optionally relate the class-names to the corresponding external class-names (the class-names as they are known outside the compilation unit).

External class-names are case sensitive and must conform to Java rules of formation. For example, in the Account class definition you might code this:

```
Environment Division.           Required  
Configuration Section.         Required  
Repository.                    Required  
  Class Base is "java.lang.Object" Required  
  Class Account is "Account".     Optional
```

The REPOSITORY paragraph entries indicate that the external class-names of the classes referred to as Base and Account within the class definition are java.lang.Object and Account, respectively.

In the REPOSITORY paragraph, you must code an entry for each class-name that you explicitly reference in the class definition. For example:

- Base
- A superclass from which the class that you are defining inherits
- The classes that you reference in methods within the class definition

In a REPOSITORY paragraph entry, you must specify the external class-name if the name contains non-COBOL characters. You must also specify the external class-name for any referenced class that is part of a Java *package*. For such a class, specify the external class-name as the fully qualified name of the package, followed by period (.), followed by the simple name of the Java class. For example, the Object class is part of the java.lang package, so specify its external name as java.lang.Object as shown above.

An external class-name that you specify in the REPOSITORY paragraph must be an alphanumeric literal that conforms to the rules of formation for a fully qualified Java class-name.

If you do not include the external class-name in a REPOSITORY paragraph entry, the external class-name is formed from the class-name in the following manner:

- The class-name is converted to uppercase.
- Each hyphen is changed to zero.
- The first character, if a digit, is changed:
 - 1-9 are changed to A-I.
 - 0 is changed to J.
- Underscores are not changed.

In the example above, class Account is known externally as Account (in mixed case) because the external name is spelled using mixed case.

You can optionally include in the REPOSITORY paragraph an entry for the class that you are defining (Account in this example). You must include an entry for the class that you are defining if the external class-name contains non-COBOL characters, or to specify a fully package-qualified class-name if the class is to be part of a Java package.

[“Example: external class-names and Java packages” on page 673](#)

Related tasks

[“Declaring arrays and strings for Java” on page 712](#)

Related references

REPOSITORY paragraph (*Enterprise COBOL for z/OS Language Reference*)

[The Java Language Specification \(Identifiers\)](#)

[The Java Language Specification \(Packages\)](#)

Example: external class-names and Java packages

The following example shows how external class-names are determined from entries in a REPOSITORY paragraph.

```
Environment division.  
Configuration section.  
Repository.  
    Class Employee is "com.acme.Employee"  
    Class JavaException is "java.lang.Exception"  
    Class Orders.
```

The local class-names (the class-names as used within the class definition), the Java packages that contain the classes, and the associated external class-names are as shown in the table below.

Local class-name	Java package	External class-name
Employee	com.acme	com.acme.Employee
JavaException	java.lang	java.lang.Exception
Orders	(unnamed)	ORDERS

The external class-name (the name after the class-name and optional IS in the REPOSITORY paragraph entry) is composed of the fully qualified name of the package (if any) followed by a period, followed by the simple name of the class.

Related tasks

[“REPOSITORY paragraph for defining a class” on page 672](#)

Related references

REPOSITORY paragraph (*Enterprise COBOL for z/OS Language Reference*)

WORKING-STORAGE SECTION for defining class instance data

Use the WORKING-STORAGE SECTION in the DATA DIVISION of the OBJECT paragraph to describe the *instance data* that a COBOL class needs, that is, the data to be allocated for each instance of the class.

The OBJECT keyword, which you must immediately precede with an IDENTIFICATION DIVISION declaration, indicates the beginning of the definitions of the instance data and instance methods for the class. For example, the definition of the instance data for the Account class might look like this:

```
IDENTIFICATION DIVISION.  
Object.  
DATA DIVISION.  
WORKING-STORAGE SECTION.  
01 AccountNumber pic 9(6).  
01 AccountBalance pic S9(9) value zero.  
.  
.  
End Object.
```

The instance data is allocated when an object instance is created, and exists until garbage collection of the instance by the Java run time.

You can initialize simple instance data by using VALUE clauses as shown above. You can initialize more complex instance data by coding customized methods to create and initialize instances of classes.

COBOL instance data is equivalent to Java private nonstatic member data. No other class or subclass (nor factory method in the same class, if any) can reference COBOL instance data directly. Instance data is global to all instance methods that the OBJECT paragraph defines. If you want to make instance data accessible from outside the OBJECT paragraph, define attribute (get or set) instance methods for doing so.

The syntax of the WORKING-STORAGE SECTION for instance data definition is generally the same as in a program, with these exceptions:

- You cannot use the EXTERNAL attribute.
- You can use the GLOBAL attribute, but it has no effect.

Related tasks

[“Creating and initializing instances of classes” on page 689](#)

[“Freeing instances of classes” on page 691](#)

[“Defining a factory method” on page 697](#)

[“Coding attribute \(get and set\) methods” on page 679](#)

Example: defining a class

The following example shows a first attempt at the definition of the Account class, excluding method definitions.

```

cb1 dll,thread,pgmname(longmixed)
IDENTIFICATION DIVISION.
Class-id. Account inherits Base.
ENVIRONMENT DIVISION.
Configuration section.
Repository.
    Class Base    is "java.lang.Object"
    Class Account is "Account".
*
IDENTIFICATION DIVISION.
Object.
DATA DIVISION.
WORKING-STORAGE SECTION.
    01 AccountNumber pic 9(6).
    01 AccountBalance pic S9(9) value zero.
*
PROCEDURE DIVISION.
*
*   (Instance method definitions here)
*
End Object.
*
End class Account.

```

Related tasks

Chapter 16, [“Compiling, linking, and running OO applications,” on page 291](#)

[“Defining a client” on page 682](#)

Defining a class instance method

Define COBOL *instance methods* in the PROCEDURE DIVISION of the OBJECT paragraph of a class definition. An instance method defines an operation that is supported for each object instance of a class.

A COBOL instance method definition consists of four divisions (like a COBOL program), followed by an END METHOD marker.

Division	Purpose	Syntax
IDENTIFICATION (required)	Name a method.	<p>“METHOD-ID paragraph for defining a class instance method” on page 675 (required)</p> <p>AUTHOR paragraph (optional) INSTALLATION paragraph (optional) DATE-WRITTEN paragraph (optional) DATE-COMPILED paragraph (optional)</p>

Table 86. **Structure of instance method definitions** (continued)

Division	Purpose	Syntax
ENVIRONMENT (optional)	Relate the file-names used in a method to the corresponding file-names known to the operating system.	“INPUT-OUTPUT SECTION for defining a class instance method” on page 676 (optional)
DATA (optional)	Define external files. Allocate a copy of the data.	“DATA DIVISION for defining a class instance method” on page 676 (optional)
PROCEDURE (optional)	Code the executable statements to complete the service provided by the method.	“PROCEDURE DIVISION for defining a class instance method” on page 677 (optional)

Definition: The *signature* of a method consists of the name of the method and the number and type of its formal parameters. (You define the formal parameters of a COBOL method in the USING phrase of the method's PROCEDURE DIVISION header.)

Within a class definition, you do not need to make each method-name unique, but you do need to give each method a unique signature. (You *overload* methods by giving them the same name but a different signature.)

COBOL instance methods are equivalent to Java public nonstatic methods.

[“Example: defining a method” on page 680](#)

Related tasks

[“PROCEDURE DIVISION for defining a class instance method” on page 677](#)

[“Overloading an instance method” on page 678](#)

[“Overriding an instance method” on page 678](#)

[“Invoking methods \(INVOKE\)” on page 685](#)

[“Defining a subclass instance method” on page 694](#)

[“Defining a factory method” on page 697](#)

METHOD-ID paragraph for defining a class instance method

Use the METHOD-ID paragraph to name an instance method. Immediately precede the METHOD-ID paragraph with an IDENTIFICATION DIVISION declaration to indicate the beginning of the method definition.

For example, the definition of the `credit` method in the Account class begins like this:

```
Identification Division.
Method-id. "credit".
```

Code the method-name as an alphanumeric or national literal. The method-name is processed in a case-sensitive manner and must conform to the rules of formation for a Java method-name.

Other Java or COBOL methods or programs (that is, clients) use the method-name to invoke a method.

Related tasks

[“Invoking methods \(INVOKE\)” on page 685](#)

[“Using national data \(Unicode\) in COBOL” on page 125](#)

Related references

[The Java Language Specification](#) (Meaning of method names)

INPUT-OUTPUT SECTION for defining a class instance method

The ENVIRONMENT DIVISION of an instance method can have only one section, the INPUT-OUTPUT SECTION. This section relates the file-names used in a method definition to the corresponding file-names as they are known to the operating system.

For example, if the Account class defined a method that read information from a file, the Account class might have an INPUT-OUTPUT SECTION that is coded like this:

```
Environment Division.  
Input-Output Section.  
File-Control.  
    Select account-file Assign AcctFile.
```

The syntax for the INPUT-OUTPUT SECTION of a method is the same as the syntax for the INPUT-OUTPUT SECTION of a program.

Related tasks

[“Describing the computing environment” on page 5](#)

Related references

INPUT-OUTPUT section (*Enterprise COBOL for z/OS Language Reference*)

DATA DIVISION for defining a class instance method

The DATA DIVISION of an instance method consists of any of the following four sections: FILE SECTION, LOCAL-STORAGE SECTION, WORKING-STORAGE SECTION, and LINKAGE SECTION.

FILE SECTION

The same as a program FILE SECTION, except that a method FILE SECTION can define EXTERNAL files only.

LOCAL-STORAGE SECTION

A separate copy of the LOCAL-STORAGE data is allocated for each invocation of the method, and is freed on return from the method. The method LOCAL-STORAGE SECTION is similar to a program LOCAL-STORAGE SECTION.

If you specify the VALUE clause on a data item, the item is initialized to that value on each invocation of the method.

WORKING-STORAGE SECTION

A single copy of the WORKING-STORAGE data is allocated. The data persists in its last-used state until the run unit ends. The same copy of the data is used whenever the method is invoked, regardless of the invoking object or thread. The method WORKING-STORAGE SECTION is similar to a program WORKING-STORAGE SECTION.

If you specify the VALUE clause on a data item, the item is initialized to that value on the first invocation of the method. You can specify the EXTERNAL clause for the data items.

LINKAGE SECTION

The same as a program LINKAGE SECTION.

If you define a data item with the same name in both the DATA DIVISION of an instance method and the DATA DIVISION of the OBJECT paragraph, a reference in the method to that data-name refers only to the method data item. The method DATA DIVISION takes precedence.

Related tasks

[“Describing the data” on page 11](#)

[“Sharing data by using the EXTERNAL clause” on page 560](#)

Related references

DATA DIVISION overview (*Enterprise COBOL for z/OS Language Reference*)

PROCEDURE DIVISION for defining a class instance method

Code the executable statements to implement the service that an instance method provides in the PROCEDURE DIVISION of the instance method.

You can code most COBOL statements in the PROCEDURE DIVISION of a method that you can code in the PROCEDURE DIVISION of a program. You cannot, however, code the following statements in a method:

- ENTRY
- EXIT PROGRAM
- The following obsolete elements of the 85 COBOL Standard:
 - ALTER
 - GOTO without a specified procedure-name
 - SEGMENT-LIMIT
 - USE FOR DEBUGGING

Additionally, because you must compile all COBOL class definitions with the THREAD compiler option, you cannot use SORT or MERGE statements in a COBOL method.

You can code the EXIT METHOD or GOBACK statement in an instance method to return control to the invoking client. Both statements have the same effect. If you specify the RETURNING phrase upon invocation of the method, the EXIT METHOD or GOBACK statement returns the value of the data item to the invoking client.

An implicit EXIT METHOD is generated as the last statement in the PROCEDURE DIVISION of each method.

You can specify STOP RUN in a method; doing so terminates the entire run unit including all threads executing within it.

You must terminate a method definition with an END METHOD marker. For example, the following statement marks the end of the credit method:

```
End method "credit".
```

USING phrase for obtaining passed arguments: Specify the formal parameters to a method, if any, in the USING phrase of the method's PROCEDURE DIVISION header. You must specify that the arguments are passed BY VALUE. Define each parameter as a level-01 or level-77 item in the method's LINKAGE SECTION. The data type of each parameter must be one of the types that are interoperable with Java.

RETURNING phrase for returning a value: Specify the data item to be returned as the method result, if any, in the RETURNING phrase of the method's PROCEDURE DIVISION header. Define the data item as a level-01 or level-77 item in the method's LINKAGE SECTION. The data type of the return value must be one of the types that are interoperable with Java.

Related tasks

[“Coding interoperable data types in OO COBOL and Java” on page 711](#)

[“Overriding an instance method” on page 678](#)

[“Overloading an instance method” on page 678](#)

[“Comparing and setting object references” on page 684](#)

[“Invoking methods \(INVOKE\)” on page 685](#)

[Chapter 16, “Compiling, linking, and running OO applications,” on page 291](#)

Related references

[“THREAD” on page 408](#)

The procedure division header (*Enterprise COBOL for z/OS Language Reference*)

Overriding an instance method

An instance method that is defined in a subclass is said to *override* an inherited instance method that would otherwise be accessible in the subclass if the two methods have the same signature.

To override a superclass instance method `m1` in a COBOL subclass, define an instance method `m1` in the subclass that has the same name and whose `PROCEDURE DIVISION USING` phrase (if any) has the same number and type of formal parameters as the superclass method has. (If the superclass method is implemented in Java, you must code formal parameters that are interoperable with the data types of the corresponding Java parameters.) When a client invokes `m1` on an instance of the subclass, the subclass method rather than the superclass method is invoked.

For example, the `Account` class defines a method `debit` whose `LINKAGE SECTION` and `PROCEDURE DIVISION` header look like this:

```
Linkage section.  
01 inDebit    pic S9(9) binary.  
Procedure Division using by value inDebit.
```

If you define a `CheckingAccount` subclass and want it to have a `debit` method that overrides the `debit` method defined in the `Account` superclass, define the subclass method with exactly one input parameter also specified as `pic S9(9) binary`. If a client invokes `debit` using an object reference to a `CheckingAccount` instance, the `CheckingAccount` `debit` method (rather than the `debit` method in the `Account` superclass) is invoked.

The presence or absence of a method return value and the data type of the return value used in the `PROCEDURE DIVISION RETURNING` phrase (if any) must be identical in the subclass instance method and the overridden superclass instance method.

An instance method must not override a factory method in a COBOL superclass nor a static method in a Java superclass.

[“Example: defining a method” on page 680](#)

Related tasks

[“PROCEDURE DIVISION for defining a class instance method” on page 677](#)

[“Coding interoperable data types in OO COBOL and Java” on page 711](#)

[“Invoking methods \(INVOKE\)” on page 685](#)

[“Invoking overridden superclass methods” on page 689](#)

[“Defining a subclass” on page 692](#)

[“Hiding a factory or static method” on page 698](#)

Related references

[The Java Language Specification \(Inheritance, overriding, and hiding\)](#)

Overloading an instance method

Two methods that are supported in a class (whether defined in the class or inherited from a superclass) are said to be *overloaded* if they have the same name but different signatures.

You overload methods when you want to enable clients to invoke different versions of a method, for example, to initialize data using different sets of parameters.

To overload a method, define a method whose `PROCEDURE DIVISION USING` phrase (if any) has a different number or type of formal parameters than an identically named method that is supported in the same class. For example, the `Account` class defines an instance method `init` that has exactly one formal

parameter. The LINKAGE SECTION and PROCEDURE DIVISION header of the `init` method look like this:

```
Linkage section.  
01 inAccountNumber pic S9(9) binary.  
Procedure Division using by value inAccountNumber.
```

Clients invoke this method to initialize an Account instance with a given account number (and a default account balance of zero) by passing exactly one argument that matches the data type of `inAccountNumber`.

But the Account class could define, for example, a second instance method `init` that has an additional formal parameter that allows the opening account balance to also be specified. The LINKAGE SECTION and PROCEDURE DIVISION header of this `init` method could look like this:

```
Linkage section.  
01 inAccountNumber pic S9(9) binary.  
01 inBalance       pic S9(9) binary.  
Procedure Division using by value inAccountNumber  
                               inBalance.
```

Clients could invoke either `init` method by passing arguments that match the signature of the required method.

The presence or absence of a method return value does not have to be consistent in overloaded methods, and the data type of the return value given in the PROCEDURE DIVISION RETURNING phrase (if any) does not have to be identical in overloaded methods.

You can overload factory methods in exactly the same way that you overload instance methods.

The rules for overloaded method definition and resolution of overloaded method invocations are based on the corresponding rules for Java.

Related tasks

[“Invoking methods \(INVOKE\)” on page 685](#)

[“Defining a factory method” on page 697](#)

Related references

[The Java Language Specification \(Overloading\)](#)

Coding attribute (get and set) methods

You can provide access to an instance variable `X` from outside the class in which `X` is defined by coding accessor (`get`) and mutator (`set`) methods for `X`.

Instance variables in COBOL are *private*: the class that defines instance variables fully encapsulates them, and only the instance methods defined in the same OBJECT paragraph can access them directly. Normally a well-designed object-oriented application does not need to access instance variables from outside the class.

COBOL does not directly support the concept of a public instance variable as defined in Java and other object-oriented languages, nor the concept of a class attribute as defined by CORBA. (A CORBA *attribute* is an instance variable that has an automatically generated `get` method for accessing the value of the variable, and an automatically generated `set` method for modifying the value of the variable if the variable is not read-only.)

[“Example: coding a get method” on page 680](#)

Related tasks

[“WORKING-STORAGE SECTION for defining class instance data” on page 673](#)

[“Processing the data” on page 17](#)

Example: coding a get method

The following example shows the definition in the Account class of an instance method, `getBalance`, to return the value of the instance variable `AccountBalance` to a client. `getBalance` and `AccountBalance` are defined in the OBJECT paragraph of the Account class definition.

```
Identification Division.
Class-id. Account inherits Base.
* (ENVIRONMENT DIVISION not shown)
* (FACTORY paragraph not shown)
*
Identification division.
Object.
Data division.
WORKING-STORAGE SECTION.
01 AccountBalance pic S9(9) value zero.
* (Other instance data not shown)
*
Procedure Division.
*
Identification Division.
Method-id. "getBalance".
Data division.
Linkage section.
01 outBalance pic S9(9) binary.
*
Procedure Division returning outBalance.
Move AccountBalance to outBalance.
End method "getBalance".
*
* (Other instance methods not shown)
End Object.
*
End class Account.
```

Example: defining a method

The following example adds to the previous example the instance method definitions of the Account class, and shows the definition of the Java Check class.

(The previous example was [“Example: defining a class”](#) on page 674.)

Account class

```
cb1 dll,thread,pgmname(longmixed)
Identification Division.
Class-id. Account inherits Base.
Environment Division.
Configuration section.
Repository.
Class Base is "java.lang.Object"
Class Account is "Account".
*
* (FACTORY paragraph not shown)
*
Identification division.
Object.
Data division.
Working-storage section.
01 AccountNumber pic 9(6).
01 AccountBalance pic S9(9) value zero.
*
Procedure Division.
*
* init method to initialize the account:
Identification Division.
Method-id. "init".
Data division.
Linkage section.
01 inAccountNumber pic S9(9) binary.
Procedure Division using by value inAccountNumber.
Move inAccountNumber to AccountNumber.
```

```

End method "init".
*
*   getBalance method to return the account balance:
Identification Division.
Method-id. "getBalance".
Data division.
Linkage section.
01 outBalance pic S9(9) binary.
Procedure Division returning outBalance.
    Move AccountBalance to outBalance.
End method "getBalance".
*
*   credit method to deposit to the account:
Identification Division.
Method-id. "credit".
Data division.
Linkage section.
01 inCredit   pic S9(9) binary.
Procedure Division using by value inCredit.
    Add inCredit to AccountBalance.
End method "credit".
*
*   debit method to withdraw from the account:
Identification Division.
Method-id. "debit".
Data division.
Linkage section.
01 inDebit   pic S9(9) binary.
Procedure Division using by value inDebit.
    Subtract inDebit from AccountBalance.
End method "debit".
*
*   print method to display formatted account number and balance:
Identification Division.
Method-id. "print".
Data division.
Local-storage section.
01 PrintableAccountNumber pic ZZZZZ999999.
01 PrintableAccountBalance pic $$$,$$$,$$9CR.
Procedure Division.
    Move AccountNumber to PrintableAccountNumber
    Move AccountBalance to PrintableAccountBalance
    Display " Account: " PrintableAccountNumber
    Display " Balance: " PrintableAccountBalance.
End method "print".
*
End Object.
*
End class Account.

```

Check class

```

/**
 * A Java class for check information
 */
public class Check {
    private CheckingAccount payer;
    private Account         payee;
    private int             amount;

    public Check(CheckingAccount inPayer, Account inPayee, int inAmount) {
        payer=inPayer;
        payee=inPayee;
        amount=inAmount;
    }

    public int getAmount() {
        return amount;
    }

    public Account getPayee() {
        return payee;
    }
}

```

Related tasks

[Chapter 16, "Compiling, linking, and running OO applications," on page 291](#)

Defining a client

A program or method that requests services from one or more methods in a class is called a *client* of that class.

In a COBOL or Java client, you can:

- Create object instances of Java and COBOL classes.
- Invoke instance methods on Java and COBOL objects.
- Invoke COBOL factory methods and Java static methods.

In a COBOL client, you can also call services provided by the Java Native Interface (JNI).

A COBOL client program consists of the usual four divisions:

Division	Purpose	Syntax
IDENTIFICATION (required)	Name a client.	Code as usual, except that a client program must be: <ul style="list-style-type: none">• Recursive (declared RECURSIVE in the PROGRAM- ID paragraph)• Thread-enabled (compiled with the THREAD option, and conforming to the coding guidelines for threaded applications)
ENVIRONMENT (required)	Describe the computing environment. Relate class-names used in the client to the corresponding external class-names known outside the compilation unit.	CONFIGURATION SECTION (required) “REPOSITORY paragraph for defining a client” on page 683 (required)
DATA (optional)	Describe the data that the client needs.	“DATA DIVISION for defining a client” on page 683 (optional)
PROCEDURE (optional)	Create instances of classes, manipulate object reference data items, and invoke methods.	Code using INVOKE, IF, and SET statements.

Because you must compile all COBOL programs that contain object-oriented syntax or that interoperate with Java with the THREAD compiler option, you cannot use the following language elements in a COBOL client:

- SORT or MERGE statements
- Nested programs

Any programs that you compile with the THREAD compiler option must be recursive. You must specify the RECURSIVE clause in the PROGRAM- ID paragraph of each OO COBOL client program.

[“Example: defining a client” on page 691](#)

Related tasks

[Chapter 16, “Compiling, linking, and running OO applications,” on page 291](#)

[Chapter 31, “Preparing COBOL programs for multithreading,” on page 577](#)

[Chapter 40, “Communicating with Java methods,” on page 707](#)

[“Coding interoperable data types in OO COBOL and Java” on page 711](#)

[“Creating and initializing instances of classes” on page 689](#)

[“Comparing and setting object references” on page 684](#)

[“Invoking methods \(INVOKE\)” on page 685](#)

[“Invoking factory or static methods” on page 698](#)

Related references

[“THREAD” on page 408](#)

REPOSITORY paragraph for defining a client

Use the REPOSITORY paragraph to declare to the compiler that the specified words are class-names when you use them in a COBOL client, and to optionally relate the class-names to the corresponding external class-names (the class-names as they are known outside the compilation unit).

External class-names are case sensitive, and must conform to Java rules of formation. For example, in a client program that uses the Account and Check classes you might code this:

```
Environment division.           Required
Configuration section.         Required
  Source-Computer.  IBM-390.
  Object-Computer.  IBM-390.
Repository.                   Required
  Class Account is "Account"
  Class Check  is "Check".
```

The REPOSITORY paragraph entries indicate that the external class-names of the classes referred to as Account and Check within the client are Account and Check, respectively.

In the REPOSITORY paragraph, you must code an entry for each class-name that you explicitly reference in the client. In a REPOSITORY paragraph entry, you must specify the external class-name if the name contains non-COBOL characters.

You must specify the external class-name for any referenced class that is part of a Java package. For such a class, specify the external class-name as the fully qualified name of the package, followed by period (.), followed by the simple name of the Java class.

An external class-name that you specify in the REPOSITORY paragraph must be an alphanumeric literal that conforms to the rules of formation for a fully qualified Java class-name.

If you do not include the external class-name in a REPOSITORY paragraph entry, the external class-name is formed from the class-name in the same manner as it is when an external class-name is not included in a REPOSITORY paragraph entry in a class definition. In the example above, class Account and class Check are known externally as Account and Check (in mixed case), respectively, because the external names are spelled using mixed case.

The SOURCE - COMPUTER, OBJECT - COMPUTER, and SPECIAL - NAMES paragraphs of the CONFIGURATION SECTION are optional.

Related tasks

[“REPOSITORY paragraph for defining a class” on page 672](#)

Related references

REPOSITORY paragraph (*Enterprise COBOL for z/OS Language Reference*)

[The Java Language Specification \(Identifiers\)](#)

[The Java Language Specification \(Packages\)](#)

DATA DIVISION for defining a client

You can use any of the sections of the DATA DIVISION to describe the data that the client needs.

```
Data Division.
Local-storage section.
01  anAccount      usage object reference Account.
01  aCheckingAccount usage object reference CheckingAccount.
01  aCheck         usage object reference Check.
```

```
01 payee          usage object reference Account.  
  . . .
```

Because a client references classes, it needs one or more special data items called *object references*, that is, references to instances of those classes. All requests to instance methods require an object reference to an instance of a class in which the method is supported (that is, either defined or available by inheritance). You code object references to refer to instances of Java classes using the same syntax as you use to refer to instances of COBOL classes. In the example above, the phrase `usage object reference` indicates an object reference data item.

All four object references in the code above are called *typed* object references because a class-name appears after the `OBJECT REFERENCE` phrase. A typed object reference can refer only to an instance of the class named in the `OBJECT REFERENCE` phrase or to one of its subclasses. Thus `anAccount` can refer to instances of the `Account` class or one of its subclasses, but cannot refer to instances of any other class. Similarly, `aCheck` can refer only to instances of the `Check` class or any subclasses that it might have.

Another type of object reference, not shown above, does not have a class-name after the `OBJECT REFERENCE` phrase. Such a reference is called a *universal* object reference, which means that it can refer to instances of any class. Avoid coding universal object references, because they are interoperable with Java in only very limited circumstances (when used in the `RETURNING` phrase of the `INVOKE class-name NEW . . . statement`).

You must define, in the `REPOSITORY` paragraph of the `CONFIGURATION SECTION`, class-names that you use in the `OBJECT REFERENCE` phrase.

Related tasks

[“Choosing LOCAL-STORAGE or WORKING-STORAGE” on page 684](#)
[“Coding interoperable data types in OO COBOL and Java” on page 711](#)
[“Invoking methods \(INVOKE\)” on page 685](#)
[“REPOSITORY paragraph for defining a client” on page 683](#)

Related references

`RETURNING` phrase (*Enterprise COBOL for z/OS Language Reference*)

Choosing LOCAL-STORAGE or WORKING-STORAGE

You can in general use the `WORKING-STORAGE SECTION` to define working data that a client program needs. However, if the program could simultaneously run on multiple threads, you might instead want to define the data in the `LOCAL-STORAGE SECTION`.

Each thread has access to a separate copy of `LOCAL-STORAGE` data but shares access to a single copy of `WORKING-STORAGE` data. If you define the data in the `WORKING-STORAGE SECTION`, you need to synchronize access to the data or ensure that no two threads can access it simultaneously.

Related tasks

[Chapter 31, “Preparing COBOL programs for multithreading,” on page 577](#)

Comparing and setting object references

You can compare object references by coding conditional statements or a call to the JNI service `IsSameObject`, and you can set object references by using the `SET` statement.

For example, code either `IF` statement below to check whether the object reference `anAccount` refers to no object instance:

```
If anAccount = Null . . .  
If anAccount = Nulls . . .
```

You can code a call to `IsSameObject` to check whether two object references, `object1` and `object2`, refer to the same object instance or whether each refers to no object instance. To ensure that the arguments

and return value are interoperable with Java and to establish addressability to the callable service, code the following data definitions and statements before the call to `IsSameObject`:

```
Local-storage Section.  
01 is-same Pic X.  
   88 is-same-false Value X'00'.  
   88 is-same-true  Value X'01' Through X'FF'.  
Linkage Section.  
Copy JNI.  
Procedure Division.  
Set Address Of JNIEnv To JNIEnvPtr  
Set Address Of JNINativeInterface To JNIEnv  
Call IsSameObject Using By Value JNIEnvPtr object1 object2  
   Returning is-same  
If is-same-true . . .
```

Within a method you can check whether an object reference refers to the object instance on which the method was invoked by coding a call to `IsSameObject` that compares the object reference and `SELF`.

You can instead invoke the Java `equals` method (inherited from `java.lang.Object`) to determine whether two object references refer to the same object instance.

You can make an object reference refer to no object instance by using the `SET` statement. For example:

```
Set anAccount To Null.
```

You can also make one object reference refer to the same instance as another object reference does by using the `SET` statement. For example:

```
Set anotherAccount To anAccount.
```

This `SET` statement causes `anotherAccount` to refer to the same object instance as `anAccount` does. If the receiver (`anotherAccount`) is a universal object reference, the sender (`anAccount`) can be either a universal or a typed object reference. If the receiver is a typed object reference, the sender must be a typed object reference bound to the same class as the receiver or to one of its subclasses.

Within a method you can make an object reference refer to the object instance on which the method was invoked by setting it to `SELF`. For example:

```
Set anAccount To Self.
```

Related tasks

[“Coding interoperable data types in OO COBOL and Java” on page 711](#)

[“Accessing JNI services” on page 707](#)

Related references

[The Java Native Interface \(IsSameObject\)](#)

Invoking methods (INVOKE)

In a Java client, you can create object instances of classes that were implemented in COBOL and invoke methods on those objects using standard Java syntax. In a COBOL client, you can invoke methods that are defined in Java or COBOL classes by coding the `INVOKE` statement.

```
Invoke Account "createAccount"  
   using by value 123456  
   returning anAccount  
Invoke anAccount "credit" using by value 500.
```

The first example INVOKE statement above uses the class-name Account to invoke a method called createAccount. This method must be either defined or inherited in the Account class, and must be one of the following types:

- A Java static method
- A COBOL factory method

The phrase using by value 123456 indicates that 123456 is an input argument to the method, and is passed by value. The input argument 123456 and the returned data item anAccount must conform to the definition of the formal parameters and return type, respectively, of the (possibly overloaded) createAccount method.

The second INVOKE statement uses the returned object reference anAccount to invoke the instance method credit, which is defined in the Account class. The input argument 500 must conform to the definition of the formal parameters of the (possibly overloaded) credit method.

Code the name of the method to be invoked either as a literal or as an identifier whose value at run time matches the method-name in the signature of the target method. The method-name must be an alphanumeric or national literal or a category alphabetic, alphanumeric, or national data item, and is interpreted in a case-sensitive manner.

When you code an INVOKE statement using an object reference (as in the second example statement above), the statement begins with one of the following two forms:

```
Invoke objRef "literal-name" . . .  
Invoke objRef identifier-name . . .
```

When the method-name is an identifier, you must define the object reference (objRef) as USAGE OBJECT REFERENCE with no specified type, that is, as a universal object reference.

If an invoked method is not supported in the class to which the object reference refers, a severity-3 Language Environment condition is raised at run time unless you code the ON EXCEPTION phrase in the INVOKE statement.

You can use the optional scope terminator END-INVOKE with the INVOKE statement.

The INVOKE statement does not set the RETURN-CODE special register.

Related tasks

[“USING phrase for passing arguments” on page 686](#)

[“RETURNING phrase for obtaining a returned value” on page 688](#)

[“PROCEDURE DIVISION for defining a class instance method” on page 677](#)

[“Coding interoperable data types in OO COBOL and Java” on page 711](#)

[“Invoking overridden superclass methods” on page 689](#)

[“Invoking factory or static methods” on page 698](#)

Related references

INVOKE statement (*Enterprise COBOL for z/OS Language Reference*)

USING phrase for passing arguments

If you pass arguments to a method, specify the arguments in the USING phrase of the INVOKE statement. Code the data type of each argument so that it conforms to the type of the corresponding formal parameter in the intended target method.

Table 88. Conformance of arguments in a COBOL client

Programming language of the target method	Is the argument an object reference?	Then code the DATA DIVISION definition of the argument as:	Restriction
COBOL	No	The same as the definition of the corresponding formal parameter	
Java	No	Interoperable with the corresponding Java parameter	
COBOL or Java	Yes	An object reference that is typed to the same class as the corresponding parameter in the target method	In a COBOL client (unlike in a Java client), the class of an argument cannot be a subclass of the class of the corresponding parameter.

See the example referenced below for a way to make an object-reference argument conform to the type of a corresponding formal parameter by using the SET statement or the REDEFINES clause.

[“Example: passing conforming object-reference arguments from a COBOL client” on page 687](#)

If the target method is overloaded, the data types of the arguments are used to select from among the methods that have the same name.

You must specify that the arguments are passed BY VALUE. In other words, the arguments are not affected by any change to the corresponding formal parameters in the invoked method.

The data type of each argument must be one of the types that are interoperable with Java.

Related tasks

[“PROCEDURE DIVISION for defining a class instance method” on page 677](#)

[“Overloading an instance method” on page 678](#)

[“Coding interoperable data types in OO COBOL and Java” on page 711](#)

[“Passing data” on page 551](#)

Related references

INVOKE statement (*Enterprise COBOL for z/OS Language Reference*)

SET statement (*Enterprise COBOL for z/OS Language Reference*)

REDEFINES clause (*Enterprise COBOL for z/OS Language Reference*)

Example: passing conforming object-reference arguments from a COBOL client

The following example shows a way to make an object-reference argument in a COBOL client conform to the expected class of the corresponding formal parameter in an invoked method.

Class C defines a method M that has one parameter, a reference to an object of class java.lang.Object:

```

. . .
Class-id. C inherits Base.
. . .
Repository.
  Class Base      is "java.lang.Object"
  Class JavaObject is "java.lang.Object".
Identification division.
Factory.
. . .
Procedure Division.
  Identification Division.
  Method-id. "M".
  Data division.

```

```

Linkage section.
01 obj object reference JavaObject.
Procedure Division using by value obj.
. . .

```

To invoke method M, a COBOL client must pass an argument that is a reference to an object of class java.lang.Object. The client below defines a data item aString, which cannot be passed as an argument to M because aString is a reference to an object of class java.lang.String. The client first uses a SET statement to assign aString to a data item, anObj, that is a reference to an object of class java.lang.Object. (This SET statement is legal because java.lang.String is a subclass of java.lang.Object.) The client then passes anObj as the argument to M.

```

. . .
Repository.
  Class jstring    is "java.lang.String"
  Class JavaObject is "java.lang.Object".
Data division.
Local-storage section.
01 aString object reference jstring.
01 anObj    object reference JavaObject.
*
Procedure division.
. . . (statements here assign a value to aString)
Set anObj to aString
Invoke C "M"
  using by value anObj

```

Instead of using a SET statement to obtain anObj as a reference to an object of class java.lang.Object, the client could define aString and anObj with the REDEFINES clause as follows:

```

. . .
01 aString object reference jstring.
01 anObj    redefines aString object reference JavaObject.

```

After the client assigns a value to data item aString (that is, a valid reference to an object of class java.lang.String), anObj can be passed as the argument to M. For an example of the use of the REDEFINES clause to obtain argument conformance, see the example referenced below.

[“Example: J2EE client written in COBOL” on page 719](#)

Related tasks

[“Coding interoperable data types in OO COBOL and Java” on page 711](#)

[“PROCEDURE DIVISION for defining a class instance method” on page 677](#)

Related references

INVOKE statement (*Enterprise COBOL for z/OS Language Reference*)

SET statement (*Enterprise COBOL for z/OS Language Reference*)

REDEFINES clause (*Enterprise COBOL for z/OS Language Reference*)

RETURNING phrase for obtaining a returned value

If a data item is to be returned as the method result, specify the item in the RETURNING phrase of the INVOKE statement. Define the returned item in the DATA DIVISION of the client.

The item that you specify in the RETURNING phrase of the INVOKE statement must conform to the type returned by the target method, as shown in the table below.

<i>Table 89. Conformance of the returned data item in a COBOL client</i>		
Programming language of the target method	Is the returned item an object reference?	Then code the DATA DIVISION definition of the returned item as:
COBOL	No	The same as the definition of the RETURNING item in the target method

Table 89. *Conformance of the returned data item in a COBOL client (continued)*

Programming language of the target method	Is the returned item an object reference?	Then code the DATA DIVISION definition of the returned item as:
Java	No	Interoperable with the returned Java data item
COBOL or Java	Yes	An object reference that is typed to the same class as the object reference that is returned by the target method

In all cases, the data type of the returned value must be one of the types that are interoperable with Java.

Related tasks

[“Coding interoperable data types in OO COBOL and Java” on page 711](#)

Related references

INVOKE statement (*Enterprise COBOL for z/OS Language Reference*)

Invoking overridden superclass methods

Sometimes within a class you need to invoke an overridden superclass method instead of invoking a method that has the same signature and is defined in the current class.

For example, suppose that the `CheckingAccount` class overrides the `debit` instance method defined in its immediate superclass, `Account`. You could invoke the `Account` `debit` method within a method in the `CheckingAccount` class by coding this statement:

```
Invoke Super "debit" Using By Value amount.
```

You would define `amount` as `PIC S9(9) BINARY` to match the signature of the `debit` methods.

The `CheckingAccount` class overrides the `print` method that is defined in the `Account` class. Because the `print` method has no formal parameters, a method in the `CheckingAccount` class could invoke the superclass `print` method with this statement:

```
Invoke Super "print".
```

The keyword `SUPER` indicates that you want to invoke a superclass method rather than a method in the current class. (`SUPER` is an implicit reference to the object used in the invocation of the currently executing method.)

[“Example: accounts” on page 668](#)

Related tasks

[“Overriding an instance method” on page 678](#)

Related references

INVOKE statement (*Enterprise COBOL for z/OS Language Reference*)

Creating and initializing instances of classes

Before you can use the instance methods that are defined in a Java or COBOL class, you must first create an instance of the class.

To create a new instance of class `class-name` and to obtain a reference `object-reference` to the created object, code a statement of the following form, where `object-reference` is defined in the DATA DIVISION of the client:

```
INVOKE class-name NEW . . . RETURNING object-reference
```

When you code the INVOKE . . . NEW statement within a method, and the use of the returned object reference is not limited to the duration of the method invocation, you must convert the returned object reference to a global reference by calling the JNI service NewGlobalRef:

```
Call NewGlobalRef using by value JNIEnvPtr object-reference
returning object-reference
```

If you do not call NewGlobalRef, the returned object reference is only a local reference, which means that it is automatically freed after the method returns.

Related tasks

[“Instantiating Java classes” on page 690](#)

[“Instantiating COBOL classes” on page 690](#)

[“Accessing JNI services” on page 707](#)

[“Managing local and global references” on page 709](#)

[“DATA DIVISION for defining a client” on page 683](#)

[“Invoking methods \(INVOKE\)” on page 685](#)

[“Coding interoperable data types in OO COBOL and Java” on page 711](#)

Related references

INVOKE statement (*Enterprise COBOL for z/OS Language Reference*)

Instantiating Java classes

To instantiate a Java class, invoke any parameterized constructor that the class supports by coding the USING phrase in the INVOKE . . . NEW statement immediately before the RETURNING phrase, passing BY VALUE the number and types of arguments that match the signature of the constructor.

The data type of each argument must be one of the types that are interoperable with Java. To invoke the default (parameterless) constructor, omit the USING phrase.

For example, to create an instance of the Check class, initialize its instance data, and obtain reference aCheck to the Check instance created, you could code this statement in a COBOL client:

```
Invoke Check New
using by value aCheckingAccount, payee, 125
returning aCheck
```

Related tasks

[“Invoking methods \(INVOKE\)” on page 685](#)

[“Coding interoperable data types in OO COBOL and Java” on page 711](#)

Related references

VALUE clause (*Enterprise COBOL for z/OS Language Reference*)

INVOKE statement (*Enterprise COBOL for z/OS Language Reference*)

Instantiating COBOL classes

To instantiate a COBOL class, you can specify either a typed or universal object reference in the RETURNING phrase of the INVOKE . . . NEW statement. However, you cannot code the USING phrase: the instance data is initialized as specified in the VALUE clauses in the class definition.

Thus the INVOKE . . . NEW statement is useful for instantiating COBOL classes that have only simple instance data. For example, the following statement creates an instance of the Account class, initializes the instance data as specified in VALUE clauses in the WORKING-STORAGE SECTION of the OBJECT paragraph of the Account class definition, and provides reference outAccount to the new instance:

```
Invoke Account New returning outAccount
```

To make it possible to initialize COBOL instance data that cannot be initialized using VALUE clauses alone, when designing a COBOL class you must define a parameterized creation method in the FACTORY paragraph and a parameterized initialization method in the OBJECT paragraph:

1. In the parameterized factory creation method, do these steps:
 - a) Code `INVOKE class-name NEW RETURNING objectRef` to create an instance of `class-name` and to give initial values to the instance data items that have VALUE clauses.
 - b) Invoke the parameterized initialization method on the instance (`objectRef`), passing BY VALUE the arguments that were supplied to the factory method.
2. In the initialization method, code logic to complete the instance data initialization using the values supplied through the formal parameters.

To create an instance of the COBOL class and properly initialize it, the client invokes the parameterized factory method, passing BY VALUE the required arguments. The object reference returned to the client is a local reference. If the client code is within a method, and the use of the returned object reference is not limited to the duration of that method, the client code must convert the returned object reference to a global reference by calling the JNI service `NewGlobalRef`.

[“Example: defining a factory \(with methods\)” on page 699](#)

Related tasks

[“Accessing JNI services” on page 707](#)

[“Managing local and global references” on page 709](#)

[“Invoking methods \(INVOKE\)” on page 685](#)

[“Defining a factory section” on page 695](#)

Related references

VALUE clause (*Enterprise COBOL for z/OS Language Reference*)

INVOKE statement (*Enterprise COBOL for z/OS Language Reference*)

Freeing instances of classes

You do not need to take any action to free individual object instances of any class. No syntax is available for doing so. The Java runtime system automatically performs *garbage collection*, that is, it reclaims the memory for objects that are no longer in use.

There could be times, however, when you need to explicitly free local or global references to objects within a native COBOL client in order to permit garbage collection of the referenced objects to occur.

Related tasks

[“Managing local and global references” on page 709](#)

Example: defining a client

The following example shows a small client program of the Account class.

The program does this:

- Invokes a factory method `createAccount` to create an Account instance with a default balance of zero
- Invokes the instance method `credit` to deposit \$500 to the new account
- Invokes the instance method `print` to display the account status

(The Account class was shown in [“Example: defining a method” on page 680.](#))

```
cb1 dll,thread,pgmname(longmixed)
Identification division.
Program-id. "TestAccounts" recursive.
Environment division.
Configuration section.
Repository.
    Class Account is "Account".
Data Division.
* Working data is declared in LOCAL-STORAGE instead of
```

```

* WORKING-STORAGE so that each thread has its own copy:
  Local-storage section.
  01 anAccount usage object reference Account.
*
  Procedure division.
  Test-Account-section.
  Display "Test Account class"
* Create account 123456 with 0 balance:
  Invoke Account "createAccount"
  using by value 123456
  returning anAccount
* Deposit 500 to the account:
  Invoke anAccount "credit" using by value 500
  Invoke anAccount "print"
  Display space
*
  Stop Run.
  End program "TestAccounts".

```

[“Example: defining a factory \(with methods\)” on page 699](#)

Related tasks

[“Defining a factory method” on page 697](#)

[“Invoking factory or static methods” on page 698](#)

[Chapter 16, “Compiling, linking, and running OO applications,” on page 291](#)

Defining a subclass

You can make a class (called a *subclass*, derived class, or child class) a specialization of another class (called a *superclass*, base class, or parent class).

A subclass inherits the methods and instance data of its superclasses, and is related to its superclasses by an *is-a* relationship. For example, if subclass P inherits from superclass Q, and subclass Q inherits from superclass S, then an instance of P is an instance of Q and also (by transitivity) an instance of S. An instance of P inherits the methods and data of Q and S.

Using subclasses has several advantages:

- Reuse of code: Through inheritance, a subclass can reuse methods that already exist in a superclass.
- Specialization: In a subclass you can add new methods to handle cases that the superclass does not handle. You can also add new data items that the superclass does not need.
- Change in action: A subclass can override a method that it inherits from a superclass by defining a method of the same signature as that in the superclass. When you override a method, you might make only a few minor changes or completely change what the method does.

Restriction: You cannot use *multiple inheritance* in your COBOL programs. Each COBOL class that you define must have exactly one immediate superclass that is implemented in Java or COBOL, and each class must be derived directly or indirectly from `java.lang.Object`. The semantics of inheritance are as defined by Java.

The structure and syntax of a subclass definition are identical to those of a class definition: Define instance data and methods in the DATA DIVISION and PROCEDURE DIVISION, respectively, within the OBJECT paragraph of the subclass definition. In subclasses that require data and methods that are to be associated with the subclass itself rather than with individual object instances, define a separate DATA DIVISION and PROCEDURE DIVISION within the FACTORY paragraph of the subclass definition.

COBOL instance data is private. A subclass can access the instance data of a COBOL superclass only if the superclass defines attribute (get or set) instance methods for doing so.

[“Example: accounts” on page 668](#)

[“Example: defining a subclass \(with methods\)” on page 694](#)

Related tasks

[“Defining a class” on page 670](#)

[“Overriding an instance method” on page 678](#)

[“Coding attribute \(get and set\) methods” on page 679](#)

[“Defining a subclass instance method” on page 694](#)

[“Defining a factory section” on page 695](#)

Related references

The Java Language Specification (Inheritance, overriding, and hiding)

COBOL class definition structure (*Enterprise COBOL for z/OS Language Reference*)

CLASS-ID paragraph for defining a subclass

Use the CLASS-ID paragraph to name the subclass and indicate from which immediate Java or COBOL superclass it inherits its characteristics.

```
Identification Division.                               Required
Class-id. CheckingAccount inherits Account.          Required
```

In the example above, CheckingAccount is the subclass being defined. CheckingAccount inherits all the methods of the class known within the subclass definition as Account. CheckingAccount methods can access Account instance data only if the Account class provides attribute (get or set) methods for doing so.

You must specify the name of the immediate superclass in the REPOSITORY paragraph in the CONFIGURATION SECTION of the ENVIRONMENT DIVISION. You can optionally associate the superclass name with the name of the class as it is known externally. You can also specify the name of the subclass that you are defining (here, CheckingAccount) in the REPOSITORY paragraph and associate it with its corresponding external class-name.

Related tasks

[“CLASS-ID paragraph for defining a class” on page 671](#)

[“Coding attribute \(get and set\) methods” on page 679](#)

[“REPOSITORY paragraph for defining a subclass” on page 693](#)

REPOSITORY paragraph for defining a subclass

Use the REPOSITORY paragraph to declare to the compiler that the specified words are class-names when you use them within a subclass definition, and to optionally relate the class-names to the corresponding external class-names (the class-names as they are known outside the compilation unit).

For example, in the CheckingAccount subclass definition, these REPOSITORY paragraph entries indicate that the external class-names of the classes referred to as CheckingAccount, Check, and Account within the subclass definition are CheckingAccount, Check, and Account, respectively.

```
Environment Division.                               Required
Configuration Section.                             Required
Repository.                                         Required
  Class CheckingAccount is "CheckingAccount"       Optional
  Class Check           is "Check"                 Required
  Class Account         is "Account".              Required
```

In the REPOSITORY paragraph, you must code an entry for each class-name that you explicitly reference in the subclass definition. For example:

- A user-defined superclass from which the subclass that you are defining inherits
- The classes that you reference in methods within the subclass definition

The rules for coding REPOSITORY paragraph entries in a subclass are identical to those for coding REPOSITORY paragraph entries in a class.

Related tasks

[“REPOSITORY paragraph for defining a class” on page 672](#)

Related references

REPOSITORY paragraph (*Enterprise COBOL for z/OS Language Reference*)

WORKING-STORAGE SECTION for defining subclass instance data

Use the WORKING-STORAGE SECTION in the DATA DIVISION of the subclass OBJECT paragraph to describe any instance data that the subclass needs in addition to the instance data defined in its superclasses. Use the same syntax that you use to define instance data in a class.

For example, the definition of the instance data for the CheckingAccount subclass of the Account class might look like this:

```
IDENTIFICATION DIVISION.  
Object.  
DATA DIVISION.  
WORKING-STORAGE SECTION.  
01 CheckFee pic S9(9) value 1.  
.  
.  
End Object.
```

Related tasks

[“WORKING-STORAGE SECTION for defining class instance data” on page 673](#)

Defining a subclass instance method

A subclass inherits the methods of its superclasses. In a subclass definition, you can override any instance method that the subclass inherits by defining an instance method with the same signature as the inherited method. You can also define new methods that the subclass needs.

The structure and syntax of a subclass instance method are identical to those of a class instance method. Define subclass instance methods in the PROCEDURE DIVISION of the OBJECT paragraph of the subclass definition.

[“Example: defining a subclass \(with methods\)” on page 694](#)

Related tasks

[“Defining a class instance method” on page 674](#)

[“Overriding an instance method” on page 678](#)

[“Overloading an instance method” on page 678](#)

Example: defining a subclass (with methods)

The following example shows the instance method definitions for the CheckingAccount subclass of the Account class.

The processCheck method invokes the Java instance methods getAmount and getPayee of the Check class to get the check data. It invokes the credit and debit instance methods inherited from the Account class to credit the payee and debit the payer of the check.

The print method overrides the print instance method defined in the Account class. It invokes the overridden print method to display account status, and also displays the check fee. CheckFee is an instance data item defined in the subclass.

(The Account class was shown in [“Example: defining a method” on page 680.](#))

CheckingAccount class (subclass of Account)

```
cb1 dll,thread,pgmname(longmixed)  
Identification Division.  
Class-id. CheckingAccount inherits Account.
```

```

Environment Division.
Configuration section.
Repository.
  Class CheckingAccount is "CheckingAccount"
  Class Check          is "Check"
  Class Account        is "Account".
*
* (FACTORY paragraph not shown)
*
Identification division.
Object.
  Data division.
  Working-storage section.
  01 CheckFee pic S9(9) value 1.
  Procedure Division.
*
*   processCheck method to get the check amount and payee,
*   add the check fee, and invoke inherited methods debit
*   to debit the payer and credit to credit the payee:
  Identification Division.
  Method-id. "processCheck".
  Data division.
  Local-storage section.
  01 amount pic S9(9) binary.
  01 payee usage object reference Account.
  Linkage section.
  01 aCheck usage object reference Check.
*
*   Procedure Division using by value aCheck.
  Invoke aCheck "getAmount" returning amount
  Invoke aCheck "getPayee" returning payee
  Invoke payee "credit" using by value amount
  Add checkFee to amount
  Invoke self "debit" using by value amount.
  End method "processCheck".
*
*   print method override to display account status:
  Identification Division.
  Method-id. "print".
  Data division.
  Local-storage section.
  01 printableFee pic $$,$$$,$$9.
  Procedure Division.
  Invoke super "print"
  Move CheckFee to printableFee
  Display " Check fee: " printableFee.
  End method "print".
*
* End Object.
*
* End class CheckingAccount.

```

Related tasks

[Chapter 16, “Compiling, linking, and running OO applications,” on page 291](#)

[“Invoking methods \(INVOKE\)” on page 685](#)

[“Overriding an instance method” on page 678](#)

[“Invoking overridden superclass methods” on page 689](#)

Defining a factory section

Use the FACTORY paragraph in a class definition to define data and methods that are to be associated with the class itself rather than with individual object instances.

COBOL *factory data* is equivalent to Java private static data. A single copy of the data is instantiated for the class and is shared by all object instances of the class. You most commonly use factory data when you want to gather data from all the instances of a class. For example, you could define a factory data item to keep a running total of the number of instances of the class that are created.

COBOL *factory methods* are equivalent to Java public static methods. The methods are supported by the class independently of any object instance. You most commonly use factory methods to customize object creation when you cannot use VALUE clauses alone to initialize instance data.

By contrast, you use the OBJECT paragraph in a class definition to define data that is created for each object instance of the class, and methods that are supported for each object instance of the class.

A factory definition consists of three divisions, followed by an END FACTORY statement:

<i>Table 90. Structure of factory definitions</i>		
Division	Purpose	Syntax
IDENTIFICATION (required)	Identify the start of the factory definition.	IDENTIFICATION DIVISION. FACTORY.
DATA (optional)	Describe data that is allocated once for the class (as opposed to data allocated for each instance of a class).	“WORKING-STORAGE SECTION for defining factory data” on page 696 (optional)
PROCEDURE (optional)	Define factory methods.	Factory method definitions: “Defining a factory method” on page 697

[“Example: defining a factory \(with methods\)” on page 699](#)

Related tasks

[“Defining a class” on page 670](#)

[“Instantiating COBOL classes” on page 690](#)

[“Wrapping procedure-oriented COBOL programs” on page 703](#)

[“Structuring OO applications” on page 703](#)

WORKING-STORAGE SECTION for defining factory data

Use the WORKING-STORAGE SECTION in the DATA DIVISION of the FACTORY paragraph to describe the *factory data* that a COBOL class needs, that is, statically allocated data to be shared by all object instances of the class.

The FACTORY keyword, which you must immediately precede with an IDENTIFICATION DIVISION declaration, indicates the beginning of the definitions of the factory data and factory methods for the class. For example, the definition of the factory data for the Account class might look like this:

```
IDENTIFICATION DIVISION.
Factory.
DATA DIVISION.
WORKING-STORAGE SECTION.
  01 NumberOfAccounts pic 9(6) value zero.
End Factory.
```

You can initialize simple factory data by using VALUE clauses as shown above.

COBOL factory data is equivalent to Java private static data. No other class or subclass (nor instance method in the same class, if any) can reference COBOL factory data directly. Factory data is global to all factory methods that the FACTORY paragraph defines. If you want to make factory data accessible from outside the FACTORY paragraph, define factory attribute (get or set) methods for doing so.

Related tasks

[“Coding attribute \(get and set\) methods” on page 679](#)

[“Instantiating COBOL classes” on page 690](#)

Defining a factory method

Define COBOL *factory methods* in the PROCEDURE DIVISION of the FACTORY paragraph of a class definition. A factory method defines an operation that is supported by a class independently of any object instance of the class. COBOL factory methods are equivalent to Java public static methods.

You typically define factory methods for classes whose instances require complex initialization, that is, to values that you cannot assign by using VALUE clauses alone. Within a factory method you can invoke instance methods to initialize the instance data. A factory method cannot directly access instance data.

You can code factory attribute (get and set) methods to make factory data accessible from outside the FACTORY paragraph, for example, to make the data accessible from instance methods in the same class or from a client program. For example, the Account class could define a factory method getNumberOfAccounts to return the current tally of the number of accounts.

You can use factory methods to wrap procedure-oriented COBOL programs so that they are accessible from Java programs. You can code a factory method called main to enable you to run an OO application by using the java command, and to structure your applications in keeping with standard Java practice. See the Related tasks for details.

In defining factory methods, you use the same syntax that you use to define instance methods. A COBOL factory method definition consists of four divisions (like a COBOL program), followed by an END METHOD marker:

Division	Purpose	Syntax
IDENTIFICATION (required)	Same as for a class instance method	Same as for a class instance method (required)
ENVIRONMENT (optional)	Same as for a class instance method	Same as for a class instance method
DATA (optional)	Same as for a class instance method	Same as for a class instance method
PROCEDURE (optional)	Same as for a class instance method	Same as for a class instance method

Within a class definition, you do not need to make each factory method-name unique, but you do need to give each factory method a unique signature. You can overload factory methods in exactly the same way that you overload instance methods. For example, the CheckingAccount subclass provides two versions of the factory method createCheckingAccount: one that initializes the account to have a default balance of zero, and one that allows the opening balance to be passed in. Clients can invoke either createCheckingAccount method by passing arguments that match the signature of the intended method.

If you define a data item with the same name in both the DATA DIVISION of a factory method and the DATA DIVISION of the FACTORY paragraph, a reference in the method to that data-name refers only to the method data item. The method DATA DIVISION takes precedence.

[“Example: defining a factory \(with methods\)” on page 699](#)

Related tasks

[“Structuring OO applications” on page 703](#)

[“Wrapping procedure-oriented COBOL programs” on page 703](#)

[“Instantiating COBOL classes” on page 690](#)

[“Defining a class instance method” on page 674](#)

[“Coding attribute \(get and set\) methods” on page 679](#)

[“Overloading an instance method” on page 678](#)

[“Hiding a factory or static method” on page 698](#)

[“Invoking factory or static methods” on page 698](#)

[“Using object-oriented COBOL and Java under IMS” on page 509](#)

Hiding a factory or static method

A factory method defined in a subclass is said to *hide* an inherited COBOL or Java method that would otherwise be accessible in the subclass if the two methods have the same signature.

To hide a superclass factory method `f1` in a COBOL subclass, define a factory method `f1` in the subclass that has the same name and whose `PROCEDURE DIVISION USING` phrase (if any) has the same number and type of formal parameters as the superclass method has. (If the superclass method is implemented in Java, you must code formal parameters that are interoperable with the data types of the corresponding Java parameters.) When a client invokes `f1` using the subclass name, the subclass method rather than the superclass method is invoked.

The presence or absence of a method return value and the data type of the return value used in the `PROCEDURE DIVISION RETURNING` phrase (if any) must be identical in the subclass factory method and the hidden superclass method.

A factory method must not hide an instance method in a Java or COBOL superclass.

[“Example: defining a factory \(with methods\)” on page 699](#)

Related tasks

[“Coding interoperable data types in OO COBOL and Java” on page 711](#)

[“Overriding an instance method” on page 678](#)

[“Invoking methods \(INVOKE\)” on page 685](#)

Related references

The Java Language Specification (Inheritance, overriding, and hiding)

The procedure division header (*Enterprise COBOL for z/OS Language Reference*)

Invoking factory or static methods

To invoke a COBOL factory method or Java static method in a COBOL method or client program, code the class-name as the first operand of the `INVOKE` statement.

For example, a client program could invoke one of the overloaded `CheckingAccount` factory methods called `createCheckingAccount` to create a checking account with account number `777777` and an opening balance of `$300` by coding this statement:

```
Invoke CheckingAccount "createCheckingAccount"  
  using by value 777777 300  
  returning aCheckingAccount
```

To invoke a factory method from within the same class in which you define the factory method, you also use the class-name as the first operand in the `INVOKE` statement.

Code the name of the method to be invoked either as a literal or as an identifier whose value at run time is the method-name. The method-name must be an alphanumeric or national literal or a category alphabetic, alphanumeric, or national data item, and is interpreted in a case-sensitive manner.

If an invoked method is not supported in the class that you name in the `INVOKE` statement, a severity-3 Language Environment condition is raised at run time unless you code the `ON EXCEPTION` phrase in the `INVOKE` statement.

The conformance requirements for passing arguments to a COBOL factory method or Java static method in the `USING` phrase, and receiving a return value in the `RETURNING` phrase, are the same as those for invoking instance methods.

[“Example: defining a factory \(with methods\)” on page 699](#)

Related tasks

[“Invoking methods \(INVOKE\)” on page 685](#)

[“Using national data \(Unicode\) in COBOL” on page 125](#)

[“Coding interoperable data types in OO COBOL and Java” on page 711](#)

Related references

INVOKE statement (*Enterprise COBOL for z/OS Language Reference*)

Example: defining a factory (with methods)

The following example updates the previous examples to show the definition of factory data and methods.

These updates are shown:

- The Account class adds factory data and a parameterized factory method, createAccount, which allows an Account instance to be created using an account number that is passed in.
- The CheckingAccount subclass adds factory data and an overloaded parameterized factory method, createCheckingAccount. One implementation of createCheckingAccount initializes the account with a default balance of zero, and the other allows the opening balance to be passed in. Clients can invoke either method by passing arguments that match the signature of the required method.
- The TestAccounts client invokes the services provided by the factory methods of the Account and CheckingAccount classes, and instantiates the Java Check class.
- The output from the TestAccounts client program is shown.

(The previous examples were [“Example: defining a method”](#) on page 680, [“Example: defining a client”](#) on page 691, and [“Example: defining a subclass \(with methods\)”](#) on page 694.)

You can also find the complete source code for this example in the cobol/demo/oosample subdirectory in the z/OS UNIX file system. Typically the complete path for the source is /usr/lpp/cobol/demo/oosample. You can use the makefile there to compile and link the code.

Account class

```
cbl dll,thread,pgmname(longmixed)
Identification Division.
Class-id. Account inherits Base.
Environment Division.
Configuration section.
Repository.
    Class Base      is "java.lang.Object"
    Class Account is "Account".
*
Identification division.
Factory.
Data division.
Working-storage section.
01 NumberOfAccounts pic 9(6) value zero.
*
Procedure Division.
*
*   createAccount method to create a new Account
*   instance, then invoke the OBJECT paragraph's init
*   method on the instance to initialize its instance data:
Identification Division.
Method-id. "createAccount".
Data division.
Linkage section.
01 inAccountNumber pic S9(6) binary.
01 outAccount object reference Account.
*   Facilitate access to JNI services:
    Copy JNI.
Procedure Division using by value inAccountNumber
    returning outAccount.
*   Establish addressability to JNI environment structure:
    Set address of JNIEnv to JNIEnvPtr
    Set address of JNINativeInterface to JNIEnv
    Invoke Account New returning outAccount
    Invoke outAccount "init" using by value inAccountNumber
    Add 1 to NumberOfAccounts.
    End method "createAccount".
*
End Factory.
*
Identification division.
Object.
Data division.
```

```

Working-storage section.
01 AccountNumber pic 9(6).
01 AccountBalance pic S9(9) value zero.
*
* Procedure Division.
*
*   init method to initialize the account:
Identification Division.
Method-id. "init".
Data division.
Linkage section.
01 inAccountNumber pic S9(9) binary.
Procedure Division using by value inAccountNumber.
    Move inAccountNumber to AccountNumber.
End method "init".
*
*   getBalance method to return the account balance:
Identification Division.
Method-id. "getBalance".
Data division.
Linkage section.
01 outBalance pic S9(9) binary.
Procedure Division returning outBalance.
    Move AccountBalance to outBalance.
End method "getBalance".
*
*   credit method to deposit to the account:
Identification Division.
Method-id. "credit".
Data division.
Linkage section.
01 inCredit pic S9(9) binary.
Procedure Division using by value inCredit.
    Add inCredit to AccountBalance.
End method "credit".
*
*   debit method to withdraw from the account:
Identification Division.
Method-id. "debit".
Data division.
Linkage section.
01 inDebit pic S9(9) binary.
Procedure Division using by value inDebit.
    Subtract inDebit from AccountBalance.
End method "debit".
*
*   print method to display formatted account number and balance:
Identification Division.
Method-id. "print".
Data division.
Local-storage section.
01 PrintableAccountNumber pic ZZZZZZ999999.
01 PrintableAccountBalance pic $$$,$$$,$$9CR.
Procedure Division.
    Move AccountNumber to PrintableAccountNumber
    Move AccountBalance to PrintableAccountBalance
    Display " Account: " PrintableAccountNumber
    Display " Balance: " PrintableAccountBalance.
End method "print".
*
* End Object.
*
* End class Account.

```

CheckingAccount class (subclass of Account)

```

cbl dll,thread,pgmname(longmixed)
Identification Division.
Class-id. CheckingAccount inherits Account.
Environment Division.
Configuration section.
Repository.
    Class CheckingAccount is "CheckingAccount"
    Class Check is "Check"
    Class Account is "Account".
*
Identification division.
Factory.
Data division.

```



```

Working-storage section.
01 NumberOfCheckingAccounts pic 9(6) value zero.
*
Procedure Division.
*
*   createCheckingAccount overloaded method to create a new
*   CheckingAccount instance with a default balance, invoke
*   inherited instance method init to initialize the account
*   number, and increment factory data tally of checking accounts:
Identification Division.
Method-id. "createCheckingAccount".
Data division.
Linkage section.
01 inAccountNumber pic S9(6) binary.
01 outCheckingAccount object reference CheckingAccount.
*   Facilitate access to JNI services:
Copy JNI.
Procedure Division using by value inAccountNumber
    returning outCheckingAccount.
*   Establish addressability to JNI environment structure:
Set address of JNIEnv to JNIEnvPtr
Set address of JNINativeInterface to JNIEnv
Invoke CheckingAccount New returning outCheckingAccount
Invoke outCheckingAccount "init"
    using by value inAccountNumber
Add 1 to NumberOfCheckingAccounts.
End method "createCheckingAccount".
*
*   createCheckingAccount overloaded method to create a new
*   CheckingAccount instance, invoke inherited instance methods
*   init to initialize the account number and credit to set the
*   balance, and increment factory data tally of checking accounts:
Identification Division.
Method-id. "createCheckingAccount".
Data division.
Linkage section.
01 inAccountNumber pic S9(6) binary.
01 inInitialBalance pic S9(9) binary.
01 outCheckingAccount object reference CheckingAccount.
Copy JNI.
Procedure Division using by value inAccountNumber
    inInitialBalance
    returning outCheckingAccount.
Set address of JNIEnv to JNIEnvPtr
Set address of JNINativeInterface to JNIEnv
Invoke CheckingAccount New returning outCheckingAccount
Invoke outCheckingAccount "init"
    using by value inAccountNumber
Invoke outCheckingAccount "credit"
    using by value inInitialBalance
Add 1 to NumberOfCheckingAccounts.
End method "createCheckingAccount".
*
* End Factory.
*
Identification division.
Object.
Data division.
Working-storage section.
01 CheckFee pic S9(9) value 1.
Procedure Division.
*
*   processCheck method to get the check amount and payee,
*   add the check fee, and invoke inherited methods debit
*   to debit the payer and credit to credit the payee:
Identification Division.
Method-id. "processCheck".
Data division.
Local-storage section.
01 amount pic S9(9) binary.
01 payee usage object reference Account.
Linkage section.
01 aCheck usage object reference Check.
Procedure Division using by value aCheck.
Invoke aCheck "getAmount" returning amount
Invoke aCheck "getPayee" returning payee
Invoke payee "credit" using by value amount
Add checkFee to amount
Invoke self "debit" using by value amount.
End method "processCheck".
*
*   print method override to display account status:

```

```

Identification Division.
Method-id. "print".
Data division.
Local-storage section.
01 printableFee pic $$,$$$,$$9.
Procedure Division.
    Invoke super "print"
    Move CheckFee to printableFee
    Display " Check fee: " printableFee.
End method "print".
*
End Object.
*
End class CheckingAccount.

```

Check class

```

/**
 * A Java class for check information
 */
public class Check {
    private CheckingAccount payer;
    private Account payee;
    private int amount;

    public Check(CheckingAccount inPayer, Account inPayee, int inAmount) {
        payer=inPayer;
        payee=inPayee;
        amount=inAmount;
    }

    public int getAmount() {
        return amount;
    }

    public Account getPayee() {
        return payee;
    }
}

```

TestAccounts client program

```

cbl dll,thread,pgmname(longmixed)
Identification division.
Program-id. "TestAccounts" recursive.
Environment division.
Configuration section.
Repository.
    Class Account          is "Account"
    Class CheckingAccount is "CheckingAccount"
    Class Check           is "Check".
Data Division.
* Working data is declared in Local-storage
* so that each thread has its own copy:
Local-storage section.
01 anAccount          usage object reference Account.
01 aCheckingAccount  usage object reference CheckingAccount.
01 aCheck            usage object reference Check.
01 payee             usage object reference Account.
*
Procedure division.
Test-Account-section.
    Display "Test Account class"
* Create account 123456 with 0 balance:
    Invoke Account "createAccount"
        using by value 123456
        returning anAccount
* Deposit 500 to the account:
    Invoke anAccount "credit" using by value 500
    Invoke anAccount "print"
    Display space
*
    Display "Test CheckingAccount class"
* Create checking account 777777 with balance of 300:
    Invoke CheckingAccount "createCheckingAccount"
        using by value 777777 300

```

```

        returning aCheckingAccount
* Set account 123456 as the payee:
  Set payee to anAccount
* Initialize check for 125 to be paid by account 777777 to payee:
  Invoke Check New
    using by value aCheckingAccount, payee, 125
    returning aCheck
* Debit the payer, and credit the payee:
  Invoke aCheckingAccount "processCheck"
    using by value aCheck
  Invoke aCheckingAccount "print"
  Invoke anAccount "print"
*
  Stop Run.
End program "TestAccounts".

```

Output produced by the TestAccounts client program

```

Test Account class
Account:      123456
Balance:      $500

Test CheckingAccount class
Account:      777777
Balance:      $174
Check fee:    $1
Account:      123456
Balance:      $625

```

Related tasks

[“Creating and initializing instances of classes” on page 689](#)

[“Defining a factory method” on page 697](#)

[“Invoking factory or static methods” on page 698](#)

[Chapter 16, “Compiling, linking, and running OO applications,” on page 291](#)

Wrapping procedure-oriented COBOL programs

A *wrapper* is a class that provides an interface between object-oriented code and procedure-oriented code. Factory methods provide a convenient means for writing wrappers for existing procedural COBOL code to make it accessible from Java programs.

To wrap COBOL code, do these steps:

1. Create a simple COBOL class that contains a FACTORY paragraph.
2. In the FACTORY paragraph, code a factory method that uses a CALL statement to call the procedural program.

A Java program can invoke the factory method by using a static method invocation expression, thus invoking the COBOL procedural program.

Related tasks

[“Defining a class” on page 670](#)

[“Defining a factory section” on page 695](#)

[“Defining a factory method” on page 697](#)

Structuring OO applications

You can structure applications that use object-oriented COBOL syntax in one of three ways.

An OO application can begin with:

- A COBOL program, which can have any name.

Under z/OS UNIX, you can run the application by specifying the name of the linked module (which should match the program name) at the command prompt. You can also bind the program as a module in a PDSE and run it in JCL using the EXEC PGM statement.

- A Java class definition that contains a method called `main`. Declare `main` as `public`, `static`, and `void`, with a single parameter of type `String[]`.

You can run the application with the `java` command, specifying the name of the class that contains `main`, and zero or more strings as command-line arguments.

- A COBOL class definition that contains a factory method called `main`. Declare `main` with no `RETURNING` phrase and a single `USING` parameter, an object reference to a class that is an array with elements of type `java.lang.String`. (Thus `main` is in effect `public`, `static`, and `void`, with a single parameter of type `String[]`.)

You can run the application with the `java` command, specifying the name of the class that contains `main`, and zero or more strings as command-line arguments.

Structure an OO application this way if you want to:

- Run the application by using the `java` command.
- Run the application in an environment where applications must start with the `main` method of a Java class (such as a Java dependent region).
- Follow standard Java programming practice.

[“Examples: COBOL applications that run using the java command” on page 704](#)

Related tasks

[Chapter 16, “Compiling, linking, and running OO applications,” on page 291](#)

[“Defining a factory method” on page 697](#)

[“Declaring arrays and strings for Java” on page 712](#)

[Chapter 24, “Developing COBOL programs for IMS,” on page 505](#)

Examples: COBOL applications that run using the java command

The following examples show COBOL class definitions that contain a factory method called `main`.

In each case, `main` has no `RETURNING` phrase and has a single `USING` parameter, an object reference to a class that is an array with elements of type `java.lang.String`. You can run these applications by using the `java` command.

Displaying a message

```

cbl dll,thread
Identification Division.
Class-id. CBLmain inherits Base.
Environment Division.
Configuration section.
Repository.
    Class Base is "java.lang.Object"
    Class stringArray is "jobjectArray:java.lang.String"
    Class CBLmain is "CBLmain".
*
Identification Division.
Factory.
Procedure division.
*
    Identification Division.
    Method-id. "main".
    Data division.
    Linkage section.
    01 SA usage object reference stringArray.
    Procedure division using by value SA.
        Display " >> COBOL main method entered"
        .
    End method "main".
End factory.
End class CBLmain.

```

Echoing the input strings

```

cbl dll,thread,pgmname(longmixed),ssrange
Identification Division.
Class-id. Echo inherits Base.
Environment Division.
Configuration section.
Repository.
  Class Base is "java.lang.Object"
  Class stringArray is "jobjectArray:java.lang.String"
  Class jstring is "java.lang.String"
  Class Echo is "Echo".
*
Identification Division.
Factory.
  Procedure division.
*
  Identification Division.
  Method-id. "main".
  Data division.
  Local-storage section.
  01 SAlen          pic s9(9) binary.
  01 I              pic s9(9) binary.
  01 SAElement     object reference jstring.
  01 SAElementlen pic s9(9) binary.
  01 Sbuffer        pic X(65535).
  01 P              pointer.
  Linkage section.
  01 SA             object reference stringArray.
  Copy "JNI.cpy" suppress.
  Procedure division using by value SA.
  Set address of JNIEnv to JNIEnvPtr
  Set address of JNINativeInterface to JNIEnv
  Call GetArrayLength using by value JNIEnvPtr SA
  returning SAlen
  Display "Input string array length: " SAlen
  Display "Input strings:"
  Perform varying I from 0 by 1 until I = SAlen
  Call GetObjectArrayElement
  using by value JNIEnvPtr SA I
  returning SAElement
  Call "GetStringPlatformLength"
  using by value JNIEnvPtr
  SAElement
  address of SAElementlen
  0
  Call "GetStringPlatform"
  using by value JNIEnvPtr
  SAElement
  address of Sbuffer
  length of Sbuffer
  0
  Display Sbuffer(1:SAElementlen)
  End-perform
  .
  End method "main".
  End factory.
  End class Echo.

```

Related tasks

[Chapter 16, "Compiling, linking, and running OO applications," on page 291](#)

["Defining a factory method" on page 697](#)

[Chapter 40, "Communicating with Java methods," on page 707](#)

Chapter 40. Communicating with Java methods

To achieve interlanguage interoperability with Java, you need to follow certain rules and guidelines for using services in the Java Native Interface (JNI), coding data types, and compiling COBOL programs.

You can invoke methods that are written in Java from COBOL programs, and you can invoke methods that are written in COBOL from Java programs. You need to code COBOL object-oriented language for basic Java object capabilities. For additional Java capabilities, you can call JNI services.

Because Java programs might be multithreaded and use asynchronous signals, compile COBOL programs with the THREAD option.

[“Example: J2EE client written in COBOL” on page 719](#)

[“Example: invoking Java from a batch COBOL program” on page 722](#)

Related tasks

Chapter 16, [“Compiling, linking, and running OO applications,” on page 291](#)

[“Accessing JNI services” on page 707](#)

[“Sharing data with Java” on page 711](#)

Chapter 39, [“Writing object-oriented programs,” on page 667](#)

Chapter 31, [“Preparing COBOL programs for multithreading,” on page 577](#)

Related references

[JDK 5.0 Documentation](#)

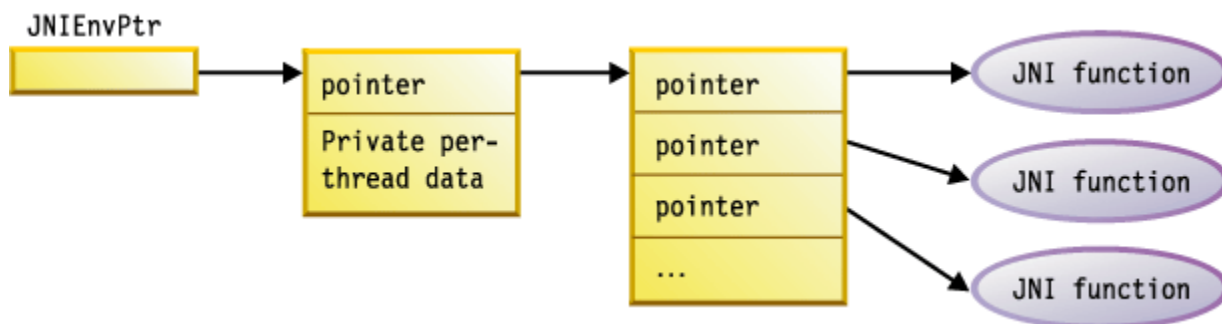
Accessing JNI services

The Java Native Interface (JNI) provides many callable services that you can use when you develop applications that mix COBOL and Java. To facilitate access to these services, copy `JNI.cpy` into the LINKAGE SECTION of your COBOL program.

The `JNI.cpy` copybook contains these definitions:

- COBOL data definitions that correspond to the Java JNI types
- `JNINativeInterface`, the JNI environment structure that contains function pointers for accessing the callable service functions

You obtain the JNI environment structure by two levels of indirection from the JNI environment pointer, as the following illustration shows:



Use the special register `JNIEnvPtr` to reference the JNI environment pointer to obtain the address for the JNI environment structure. `JNIEnvPtr` is implicitly defined as `USAGE POINTER`; do not use it as a receiving data item. Before you reference the contents of the JNI environment structure, you must code the following statements to establish its addressability:

```
Linkage section.  
COPY JNI  
...
```

```
Procedure division.  
  Set address of JNIEnv to JNIEnvPtr  
  Set address of JNINativeInterface to JNIEnv  
  . . .
```

The code above sets the addresses of the following items:

- JNIEnv, a pointer data item that JNI.cpy provides. JNIEnvPtr is the COBOL special register that contains the environment pointer.
- JNINativeInterface, the COBOL group structure that JNI.cpy contains. This structure maps the JNI environment structure, which contains an array of function pointers for the JNI callable services.

After you code the statements above, you can access the JNI callable services with CALL statements that reference the function pointers. You can pass the JNIEnvPtr special register as the first argument to the services that require the environment pointer, as shown in the following example:

```
01 InputArrayObj usage object reference jlongArray.  
01 ArrayLen pic S9(9) comp-5.  
 . . .  
  Call GetArrayLength using by value JNIEnvPtr InputArrayObj  
  returning ArrayLen
```

Important: Pass all arguments to the JNI callable services by value.

Some JNI callable services require a Java class-object reference as an argument. To obtain a reference to the class object that is associated with a class, use one of the following JNI callable services:

- GetObjectClass
- FindClass

Restriction: The JNI environment pointer is thread specific. Do not pass it from one thread to another.

Related tasks

[“Managing local and global references” on page 709](#)

[“Handling Java exceptions” on page 708](#)

[“Coding interoperable data types in OO COBOL and Java” on page 711](#)

[“Defining a client” on page 682](#)

Related references

[Appendix F, “JNI.cpy copybook,” on page 833](#)

[The Java Native Interface](#)

Handling Java exceptions

Use JNI services to throw and catch Java exceptions.

Throwing an exception: Use one of the following services to throw a Java exception from a COBOL method:

- Throw
- ThrowNew

You must make the thrown object an instance of a subclass of java.lang.Throwable.

The Java virtual machine (JVM) does not recognize and process the thrown exception until the method that contains the call has completed and returned to the JVM.

Catching an exception: After you invoke a method that might have thrown a Java exception, you can do these steps:

1. Test whether an exception occurred.
2. If an exception occurred, process the exception.
3. Clear the exception, if clearing is appropriate.

Use the following JNI services:

- ExceptionOccurred
- ExceptionCheck
- ExceptionDescribe
- ExceptionClear

To do error analysis, use the methods supported by the exception object that is returned. This object is an instance of the `java.lang.Throwable` class.

[“Example: handling Java exceptions” on page 709](#)

Example: handling Java exceptions

The following example shows the use of JNI services for catching an exception from Java and the use of the `printStackTrace` method of `java.lang.Throwable` for error analysis.

```
Repository.
  Class JavaException is "java.lang.Exception".
. . .
Local-storage section.
01 ex usage object reference JavaException.
Linkage section.
COPY "JNI.cpy".
. . .
Procedure division.
  Set address of JNIEnv to JNIEnvPtr
  Set address of JNINativeInterface to JNIEnv

  Invoke anObj "someMethod"
  Perform ErrorCheck
. . .
ErrorCheck.
  Call ExceptionOccurred
    using by value JNIEnvPtr
    returning ex
  If ex not = null then
    Call ExceptionClear using by value JNIEnvPtr
    Display "Caught an unexpected exception"
    Invoke ex "printStackTrace"
  Stop run
End-if
```

Managing local and global references

The Java virtual machine tracks the object references that you use in native methods, such as COBOL methods. This tracking ensures that the objects are not prematurely released during garbage collection.

There are two classes of such references:

Local references

Local references are valid only while the method that you invoke runs. Automatic freeing of the local references occurs after the native method returns.

Global references

Global references remain valid until you explicitly delete them. You can create global references from local references by using the JNI service `NewGlobalRef`.

The following object references are always local:

- Object references that are received as method parameters
- Object references that are returned as the method RETURNING value from a method invocation
- Object references that are returned by a call to a JNI function
- Object references that you create by using the `INVOKE . . . NEW` statement

You can pass either a local reference or a global reference as an object reference argument to a JNI service.

You can code methods to return either local or global references as RETURNING values. However, in either case, the reference that is received by the invoking program is a local reference.

You can pass either local or global references as USING arguments in a method invocation. However, in either case, the reference that is received by the invoked method is a local reference.

Local references are valid only in the thread in which you create them. Do not pass them from one thread to another.

Related tasks

[“Accessing JNI services” on page 707](#)

[“Deleting, saving, and freeing local references” on page 710](#)

Deleting, saving, and freeing local references

You can manually delete local references at any point within a method. Save local references only in object references that you define in the LOCAL-STORAGE SECTION of a method.

Use a SET statement to convert a local reference to a global reference if you want to save a reference in any of these data items:

- An object instance variable
- A factory variable
- A data item in the WORKING-STORAGE SECTION of a method

Otherwise, an error occurs. These storage areas persist when a method returns; therefore a local reference is no longer valid.

In most cases you can rely on the automatic freeing of local references that occurs when a method returns. However, in some cases you should explicitly free a local reference within a method by using the JNI service DeleteLocalRef. Here are two situations where explicit freeing is appropriate:

- In a method you access a large object, thereby creating a local reference to the object. After extensive computations, the method returns. Free the large object if you do not need it for the additional computations, because the local reference prevents the object from being released during garbage collection.
- You create a large number of local references in a method, but do not use all of them at the same time. Because the Java virtual machine requires space to keep track of each local reference, you should free those that you no longer need. Freeing the local references helps prevent the system from running out of memory.

For example, in a COBOL method you loop through a large array of objects, retrieve the elements as local references, and operate on one element at each iteration. You can free the local reference to the array element after each iteration.

Use the following callable services to manage local references and global references.

Service	Input arguments	Return value	Purpose
NewGlobalRef	<ul style="list-style-type: none"> • The JNI environment pointer • A local or global object reference 	The global reference, or NULL if the system is out of memory	To create a new global reference to the object that the input object reference refers to

Service	Input arguments	Return value	Purpose
DeleteGlobalRef	<ul style="list-style-type: none"> The JNI environment pointer A global object reference 	None	To delete a global reference to the object that the input object reference refers to
DeleteLocalRef	<ul style="list-style-type: none"> The JNI environment pointer A local object reference 	None	To delete a local reference to the object that the input object reference refers to

Related tasks

[“Accessing JNI services” on page 707](#)

Java access controls

The Java access modifiers `protected` and `private` are not enforced when you use the Java Native Interface. Therefore a COBOL program could invoke a protected or private Java method that is not invocable from a Java client. This usage is not recommended.

Sharing data with Java

You can share the COBOL data types that have Java equivalents. (Some COBOL data types have Java equivalents, but others do not.)

Share data items with Java in these ways:

- Pass them as arguments in the USING phrase of an INVOKE statement.
- Receive them as parameters in the USING phrase from a Java method.
- Receive them as the RETURNING value in an INVOKE statement.
- Return them as the value in the RETURNING phrase of the PROCEDURE DIVISION header in a COBOL method.

To pass or receive arrays and strings, declare them as object references:

- Declare an array as an object reference that contains an instance of one of the special array classes.
- Declare a string as an object reference that contains an instance of the `jstring` class.

Related tasks

[“Coding interoperable data types in OO COBOL and Java” on page 711](#)

[“Declaring arrays and strings for Java” on page 712](#)

[“Manipulating Java arrays” on page 714](#)

[“Manipulating Java strings” on page 716](#)

[“Invoking methods \(INVOKE\)” on page 685](#)

[Chapter 29, “Sharing data,” on page 551](#)

Coding interoperable data types in OO COBOL and Java

Your OO COBOL program can use only certain data types when communicating with Java.

Note: This section provides information about COBOL/Java data type compatibility for OO COBOL programs. For information about COBOL/Java data type compatibility for non-OO COBOL/Java interoperability, see “Mapping between non-OO COBOL and Java data types” in the *Enterprise COBOL Language Reference*.

Primitive Java data type	Corresponding COBOL data type
boolean ¹	PIC X followed by exactly two condition-names of this form: <pre> level-number data-name PIC X. 88 data-name-false value X'00'. 88 data-name-true value X'01' through X'FF'. </pre>
byte ¹	Single-byte alphanumeric: PIC X or PIC A
short	USAGE BINARY, COMP, COMP-4, or COMP-5, with PICTURE clause of the form S9(n), where 1<=n<=4
int	USAGE BINARY, COMP, COMP-4, or COMP-5, with PICTURE clause of the form S9(n), where 5<=n<=9
long	USAGE BINARY, COMP, COMP-4, or COMP-5, with PICTURE clause of the form S9(n), where 10<=n<=18
float ²	USAGE COMP-1
double ²	USAGE COMP-2
char	Single-character elementary national: PIC N USAGE NATIONAL. (Cannot be a national group.)
class types (object references)	USAGE OBJECT REFERENCE <i>class-name</i>
<p>1. You must distinguish boolean from byte, because they each correspond to PIC X. PIC X is interpreted as boolean only if you define an argument or a parameter with the two condition-names as shown. Otherwise, a PIC X data item is interpreted as the Java byte type.</p> <p>2. Java floating-point data is formatted according to the <i>IEEE Standard for Binary Floating Point Arithmetic</i>. Enterprise COBOL, however, uses hexadecimal floating-point representation. When you pass floating-point arguments by using an INVOKE statement, or you receive floating-point data from a Java method, the arguments and data are automatically converted as needed.</p>	

Related tasks

[“Using national data \(Unicode\) in COBOL” on page 125](#)

Declaring arrays and strings for Java

When you communicate with Java, declare arrays by using the special array classes, and declare strings by using jstring. Code the COBOL data types shown in the table below.

Java data type	Corresponding COBOL data type
boolean[]	object reference jbooleanArray
byte[]	object reference jbyteArray
short[]	object reference jshortArray
int[]	object reference jintArray
long[]	object reference jlongArray
char[]	object reference jcharArray

Table 94. **Interoperable arrays and strings in COBOL and Java** (continued)

Java data type	Corresponding COBOL data type
Object[]	object reference jobjectArray
String	object reference jstring

To use one of these classes for interoperability with Java, you must code an entry in the REPOSITORY paragraph. For example:

```
Configuration section.
Repository.
  Class jbooleanArray is "jbooleanArray".
```

The REPOSITORY paragraph entry for an object array type must specify an external class-name in one of these forms:

```
"jobjectArray"
"jobjectArray:external-classname-2"
```

In the first case, the REPOSITORY entry specifies an array class in which the elements of the array are objects of type java.lang.Object. In the second case, the REPOSITORY entry specifies an array class in which the elements of the array are objects of type *external-classname-2*. Code a colon as the separator between the specification of the jobjectArray type and the external class-name of the array elements.

The following example shows both cases. In the example, oa defines an array of elements that are objects of type java.lang.Object. aDepartment defines an array of elements that are objects of type com.acme.Employee.

```
Environment Division.
Configuration Section.
Repository.
  Class jobjectArray is "jobjectArray"
  Class Employee    is "com.acme.Employee"
  Class Department  is "jobjectArray:com.acme.Employee".
. . .
Linkage section.
01 oa              usage object reference jobjectArray.
01 aDepartment     usage object reference Department.
. . .
Procédure division using by value aDepartment.
. . .
```

[“Examples: COBOL applications that run using the java command” on page 704](#)

The following Java array types are currently not supported for interoperation with COBOL programs.

Table 95. **Noninteroperable array types in COBOL and Java**

Java data type	Corresponding COBOL data type
float[]	object reference jfloatArray
double[]	object reference jdoubleArray

Related tasks

[“REPOSITORY paragraph for defining a class” on page 672](#)

Manipulating Java arrays

To represent an array in a COBOL program, code a group item that contains a single elementary item that is of the data type that corresponds to the Java type of the array. Specify an OCCURS or OCCURS DEPENDING ON clause that is appropriate for the array.

For example, the following code specifies a structure to receive 500 or fewer integer values from a jlongArray object:

```
01 jlongArray.
  02 X pic S9(10) comp-5 occurs 1 to 500 times depending on N.
```

To operate on objects of the special Java-array classes, call the services that the JNI provides. You can use services to access and set individual elements of an array and for the following purposes, using the services cited:

Service	Input arguments	Return value	Purpose
GetArrayLength	<ul style="list-style-type: none"> The JNI environment pointer The array object reference 	The array length as a binary fullword integer	To get the number of elements in a Java array object
NewBooleanArray, NewByteArray, NewCharArray, NewShortArray, NewIntArray, NewLongArray	<ul style="list-style-type: none"> The JNI environment pointer The number of elements in the array, as a binary fullword integer 	The array object reference, or NULL if the array cannot be constructed	To create a new Java array object
GetBooleanArrayElements, GetByteArrayElements, GetCharArrayElements, GetShortArrayElements, GetIntArrayElements, GetLongArrayElements	<ul style="list-style-type: none"> The JNI environment pointer The array object reference A pointer to a boolean item. If the pointer is not null, the boolean item is set to true if a copy of the array elements was made. If a copy was made, the corresponding ReleasexxxArrayElements service must be called if changes are to be written back to the array object. 	A pointer to the storage buffer	To extract the array elements from a Java array into a storage buffer. The services return a pointer to the storage buffer, which you can use as the address of a COBOL group data item defined in the LINKAGE SECTION.
ReleaseBooleanArrayElements, ReleaseByteArrayElements, ReleaseCharArrayElements, ReleaseShortArrayElements, ReleaseIntArrayElements, ReleaseLongArrayElements	<ul style="list-style-type: none"> The JNI environment pointer The array object reference A pointer to the storage buffer The release mode, as a binary fullword integer. See Java JNI documentation for details. (Recommendation: Specify 0 to copy back the array content and free the storage buffer.) 	None; the storage for the array is released.	To release the storage buffer that contains elements that have been extracted from a Java array, and conditionally map the updated array values back into the array object

Table 96. **JNI array services** (continued)

Service	Input arguments	Return value	Purpose
NewObjectArray	<ul style="list-style-type: none"> The JNI environment pointer The number of elements in the array, as a binary fullword integer An object reference for the array element class An object reference for the initial element value. All array elements are set to this value. 	The array object reference, or NULL if the array cannot be constructed ¹	To create a new Java object array
GetObjectArrayElement	<ul style="list-style-type: none"> The JNI environment pointer The array object reference An array element index, as a binary fullword integer using origin zero 	An object reference ²	To return the element at a given index within an object array
SetObjectArrayElement	<ul style="list-style-type: none"> The JNI environment pointer The array object reference The array element index, as a binary fullword integer using origin zero The object reference for the new value 	None ³	To set an element within an object array

1. NewObjectArray throws an exception if the system runs out of memory.
2. GetObjectArrayElement throws an exception if the index is not valid.
3. SetObjectArrayElement throws an exception if the index is not valid or if the new value is not a subclass of the element class of the array.

[“Examples: COBOL applications that run using the java command” on page 704](#)

[“Example: processing a Java integer array” on page 715](#)

Related tasks

[“Coding interoperable data types in OO COBOL and Java” on page 711](#)

[“Declaring arrays and strings for Java” on page 712](#)

[“Accessing JNI services” on page 707](#)

Example: processing a Java integer array

The following example shows the use of the Java-array classes and JNI services to process a Java integer array in COBOL.

```

cbl thread,dll
Identification division.
Class-id. OOARRAY inherits Base.
Environment division.
Configuration section.
Repository.
    Class Base is "java.lang.Object"
    Class jintArray is "jintArray".
    
```

```

Identification division.
Object.
Procedure division.
  Identification division.
  Method-id. "ProcessArray".
  Data Division.
  Local-storage section.
  01 intArrayPtr pointer.
  01 intArrayLen pic S9(9) comp-5.
  Linkage section.
  COPY JNI.
  01 inIntArrayObj usage object reference jintArray.
  01 intArrayGroup.
  02 X pic S9(9) comp-5
  occurs 1 to 1000 times depending on intArrayLen.
  Procedure division using by value inIntArrayObj.
  Set address of JNIEnv to JNIEnvPtr
  Set address of JNINativeInterface to JNIEnv

  Call GetArrayLength
  using by value JNIEnvPtr inIntArrayObj
  returning intArrayLen
  Call GetIntArrayElements
  using by value JNIEnvPtr inIntArrayObj 0
  returning IntArrayPtr
  Set address of intArrayGroup to intArrayPtr

* . . . process the array elements X(I) . . .

  Call ReleaseIntArrayElements
  using by value JNIEnvPtr inIntArrayObj intArrayPtr 0.
End method "ProcessArray".
End Object.
End class OOARRAY.

```

Manipulating Java strings

COBOL represents Java String data in Unicode. To represent a Java String in a COBOL program, declare the string as an object reference of the `jstring` class. Then use JNI services to set or extract COBOL alphanumeric or national (Unicode) data from the object.

Services for Unicode: Use the following standard services to convert between `jstring` object references and COBOL USAGE NATIONAL data items. Use these services for applications that you intend to be portable between the workstation and the mainframe. Access these services by using function pointers in the `JNINativeInterface` environment structure.

<i>Table 97. Services that convert between jstring references and national data</i>		
Service	Input arguments	Return value
NewString ¹	<ul style="list-style-type: none"> The JNI environment pointer A pointer to a Unicode string, such as a COBOL national data item The number of characters in the string; binary fullword 	jstring object reference
GetStringLength	<ul style="list-style-type: none"> The JNI environment pointer A jstring object reference 	The number of Unicode characters in the jstring object reference; binary fullword

Table 97. *Services that convert between jstring references and national data (continued)*

Service	Input arguments	Return value
GetStringChars ¹	<ul style="list-style-type: none"> • The JNI environment pointer • A jstring object reference • A pointer to a boolean data item, or NULL 	<ul style="list-style-type: none"> • A pointer to the array of Unicode characters extracted from the jstring object, or NULL if the operation fails. The pointer is valid until it is released with ReleaseStringChars. • If the pointer to the boolean data item is not null, the boolean value is set to true if a copy is made of the string and to false if no copy is made.
ReleaseStringChars	<ul style="list-style-type: none"> • The JNI environment pointer • A jstring object reference • A pointer to the array of Unicode characters that was returned from GetStringChars 	None; the storage for the array is released.

1. This service throws an exception if the system runs out of memory.

Services for EBCDIC: Use the following z/OS services, an extension of the JNI, to convert between jstring object references and COBOL alphanumeric data (PIC X(*n*)).

Table 98. *Services that convert between jstring references and alphanumeric data*

Service	Input arguments	Return value
NewStringPlatform	<ul style="list-style-type: none"> • The JNI environment pointer • Pointer to the null-terminated EBCDIC character string that you want to convert to a jstring object • Pointer to the jstring object reference in which you want the result • Pointer to the Java encoding name for the string, represented as a null-terminated EBCDIC character string¹ 	Return code as a binary fullword integer: <ul style="list-style-type: none"> 0 Success. -1 Malformed input or illegal input character. -2 Unsupported encoding; the jstring object reference pointer is set to NULL.

Table 98. *Services that convert between jstring references and alphanumeric data (continued)*

Service	Input arguments	Return value
GetStringPlatformLength	<ul style="list-style-type: none"> The JNI environment pointer jstring object reference for which you want the length Pointer to a binary fullword integer for the result Pointer to the Java encoding name for the string, represented as a null-terminated EBCDIC character string¹ 	<p>Return code as a binary fullword integer:</p> <p>0 Success.</p> <p>-1 Malformed input or illegal input character.</p> <p>-2 Unsupported encoding; the jstring object reference pointer is set to NULL.</p> <p>Returns, in the third argument, the needed length in bytes of the output buffer to hold the converted Java string, including the terminating null byte referenced by the second argument.</p>
GetStringPlatform	<ul style="list-style-type: none"> The JNI environment pointer jstring object reference that you want to convert to a null-terminated string Pointer to the output buffer in which you want the converted string Length of the output buffer as a binary fullword integer Pointer to the Java encoding name for the string, represented as a null-terminated EBCDIC character string¹ 	<p>Return code as a binary fullword integer:</p> <p>0 Success.</p> <p>-1 Malformed input or illegal input character.</p> <p>-2 Unsupported encoding; the output string is set to a null string.</p> <p>-3 Conversion buffer is full.</p>
<p>1. If the pointer is NULL, the encoding from the Java file.encoding property is used.</p>		

These EBCDIC services are packaged as a DLL that is part of your IBM Java Software Development Kit. For details about the services, see `jni_convert.h` in the IBM Java Software Development Kit.

Use CALL *literal* statements to call the services. The calls are resolved through the `libjvm.x` DLL side file, which you must include in the link step of any COBOL program that uses object-oriented language.

For example, the following code creates a Java String object from the EBCDIC string 'MyConverter'. (This code fragment is from the J2EE client program, which is shown in full in [“Example: J2EE client written in COBOL”](#) on page 719.)

```

Move z"MyConverter" to stringBuffer
Call "NewStringPlatform"
  using by value JNIEnvPtr
                address of stringBuffer
                address of jstring1
                0
returning rc

```

If the EBCDIC services are the only JNI services that you call from a COBOL program, you do not need to copy the JNI.cpy copybook. You also do not need to establish addressability with the JNI environment pointer.

Services for UTF-8: The Java Native Interface also provides services for conversion between jstring object references and UTF-8 strings. These services are not recommended for use in COBOL programs due to the difficulty in handling UTF-8 character strings on the z/OS platform.

Related tasks

[“Accessing JNI services” on page 707](#)

[“Coding interoperable data types in OO COBOL and Java” on page 711](#)

[“Declaring arrays and strings for Java” on page 712](#)

[“Using national data \(Unicode\) in COBOL” on page 125](#)

[Chapter 16, “Compiling, linking, and running OO applications,” on page 291](#)

Example: J2EE client written in COBOL

The following example shows a COBOL client program that can access enterprise beans that run on a J2EE-compliant EJB server.

The COBOL client is equivalent to the J2EE client program in the Getting Started section of the *Java 2 Enterprise Edition Developer's Guide*. For your convenience in comparing implementations, the second example shows the equivalent Java client from the guide. (The enterprise bean is the Java implementation of the simple currency-converter enterprise bean, and is in the same guide.)

You can find an alternate version of the Java enterprise bean and client code in *The Java EE 5 Tutorial*, referenced below.

COBOL client (ConverterClient.cbl)

```
Process pgmname(longmixed),dll,thread
*****
* Demo J2EE client written in COBOL.
*
* Based on the sample J2EE client written in Java, which is
* given in the "Getting Started" chapter of "The Java(TM) 2
* Enterprise Edition Developer's Guide."
*
* The client:
* - Locates the home interface of a session enterprise bean
* (a simple currency converter bean)
* - Creates an enterprise bean instance
* - Invokes a business method (currency conversion)
*****
Identification division.
Program-id. "ConverterClient" is recursive.
Environment Division.
Configuration section.
Repository.
    Class InitialCtx is "javax.naming.InitialContext"
    Class PortableRemoteObject
        is "javax.rmi.PortableRemoteObject"
    Class JavaObject is "java.lang.Object"
    Class JavaClass is "java.lang.Class"
    Class JavaException is "java.lang.Exception"
    Class jstring is "jstring"
    Class Converter is "Converter"
    Class ConverterHome is "ConverterHome".
Data division.
Working-storage section.
01 initialCtx object reference InitialCtx.
01 obj object reference JavaObject.
01 classObj object reference JavaClass.
01 ex object reference JavaException.
01 currencyConverter object reference Converter.
01 home object reference ConverterHome.
01 homeObject redefines home object reference JavaObject.
01 jstring1 object reference jstring.
01 stringBuf pic X(500) usage display.
01 len pic s9(9) comp-5.
```

```

01 rc          pic s9(9) comp-5.
01 amount     comp-2.
Linkage section.
  Copy JNI.
Procedure division.
  Set address of JNIenv to JNIEnvPtr
  Set address of JNINativeInterface to JNIenv

*****
* Create JNDI naming context. *
*****
  Invoke InitialContext New returning initialCtx
  Perform JavaExceptionCheck

*****
* Create a jstring object for the string "MyConverter" for use *
* as argument to the lookup method. *
*****
  Move z"MyConverter" to stringBuffer
  Call "NewStringPlatform"
    using by value JNIEnvPtr
    address of stringBuffer
    address of jstring1
    0

  returning rc
  If rc not = zero then
    Display "Error occurred creating jstring object"
  Stop run
  End-if

*****
* Use the lookup method to obtain a reference to the home *
* object bound to the name "MyConverter". (This is the JNDI *
* name specified when deploying the J2EE application.) *
*****
  Invoke initialCtx "lookup" using by value jstring1
  returning obj
  Perform JavaExceptionCheck

*****
* Narrow the home object to be of type ConverterHome. *
* First obtain class object for the ConverterHome class, by *
* passing the null-terminated ASCII string "ConverterHome" to *
* the FindClass API. Then use this class object as the *
* argument to the static method "narrow". *
*****
  Move z"ConverterHome" to stringBuffer
  Call "__etoa"
    using by value address of stringBuffer
    returning len
  If len = -1 then
    Display "Error occurred on ASCII conversion"
  Stop run
  End-if
  Call FindClass
    using by value JNIEnvPtr
    address of stringBuffer
    returning classObj
  If classObj = null
    Display "Error occurred locating ConverterHome class"
  Stop run
  End-if
  Invoke PortableRemoteObject "narrow"
    using by value obj
    classObj
    returning homeObject
  Perform JavaExceptionCheck

*****
* Create the ConverterEJB instance and obtain local object *
* reference for its remote interface *
*****
  Invoke home "create" returning currencyConverter
  Perform JavaExceptionCheck

*****
* Invoke business methods *
*****
  Invoke currencyConverter "dollarToYen"
    using by value +100.00E+0
    returning amount
  Perform JavaExceptionCheck

```

```

Display amount

Invoke currencyConverter "yenToEuro"
    using by value +100.00E+0
    returning amount
Perform JavaExceptionCheck

Display amount

*****
* Remove the object and return.
*****
Invoke currencyConverter "remove"
Perform JavaExceptionCheck

Goback
.

*****
* Check for thrown Java exceptions
*****
JavaExceptionCheck.
    Call ExceptionOccurred using by value JNIEnvPtr
    returning ex
    If ex not = null then
        Call ExceptionClear using by value JNIEnvPtr
        Display "Caught an unexpected exception"
        Invoke ex "printStackTrace"
        Stop run
    End-if
.
End program "ConverterClient".

```

Java client (ConverterClient.java)

```

/*
 *
 * Copyright 2000 Sun Microsystems, Inc. All Rights Reserved.
 *
 * This software is the proprietary information of Sun Microsystems, Inc.
 * Use is subject to license terms.
 *
 */

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

import Converter;
import ConverterHome;

public class ConverterClient {

    public static void main(String[] args) {
        try {
            Context initial = new InitialContext();
            Object objref = initial.lookup("MyConverter");

            ConverterHome home =
                (ConverterHome)PortableRemoteObject.narrow(objref,
                                                            ConverterHome.class);

            Converter currencyConverter = home.create();

            double amount = currencyConverter.dollarToYen(100.00);
            System.out.println(String.valueOf(amount));
            amount = currencyConverter.yenToEuro(100.00);
            System.out.println(String.valueOf(amount));

            currencyConverter.remove();

        } catch (Exception ex) {
            System.err.println("Caught an unexpected exception!");
            ex.printStackTrace();
        }
    }
}

```

```
}  
}
```

Related tasks

Chapter 16, “Compiling, linking, and running OO applications,” on page 291

WebSphere for z/OS: Applications

Java 2 Enterprise Edition Developer's Guide (Getting Started)

The Java EE 5 Tutorial (Getting Started with Enterprise Beans)

Example: invoking Java from a batch COBOL program

You can invoke Java from a batch COBOL program by using the Java Batch Launcher and Toolkit for z/OS (JZOS). The following example contains JCL and source for a COBOL program that invokes a Java program in a batch job step. Numbers in parentheses refer to notes that follow the example.

In this environment, it is often desirable to direct the standard Java `System.out` and `System.err` files to z/OS data sets or spool files. You can do this by calling the `redirectStandardStreams` method in the `com.ibm.jzos.ZUtil` class after starting the Java virtual machine (JVM). For details about the `ZUtil` class, see `ZUtil` in the *z/OS security and legacy services API Reference*.

In this example COBOL program, the `main()` method from the `com.ibm.jzos.sample.HelloWorld` class is invoked, but you can change this to invoke other Java class methods.

```
//COB2JAV JOB (), 'Dovetail',  
// MSGCLASS=H, REGION=128M,  
// NOTIFY=&SYSUID  
//*  
//* Tested on z/OS 2.2 with Ent Cobol 5.1 and Java 7.0  
// SET COBPRFX='SYSPROG.MNT.COBOL51' (1)  
// SET LIBPRFX='CEE'  
// SET SYSLIB1=&COBPRFX..SIGYLIB, DISP=SHR (2)  
//* See also CLASSPATH below  
//*  
//COMPILE EXEC PGM=IGYCRCTL,  
// PARM='SIZE(5000K)'  
//SYSLIB DD DISP=SHR, DSN=&SYSLIB1 (JNI) CPY  
//SYSLIN DD DSN=&&OBJECT (TSTHELLO), UNIT=3390, DISP=(NEW, PASS),  
// SPACE=(CYL, (1, 1, 1)), DCB=(LRECL=80, RECFM=FB)  
//SYSPRINT DD SYSOUT=*  
//STEPLIB DD DSN=&COBPRFX..SIGYCOMP, DISP=SHR  
// DD DSN=&LIBPRFX..SCEERUN, DISP=SHR  
// DD DSN=&LIBPRFX..SCEERUN2, DISP=SHR  
//SYSUT1 DD UNIT=VIO, SPACE=(CYL, (1, 1))  
//SYSUT2 DD UNIT=VIO, SPACE=(CYL, (1, 1))  
//SYSUT3 DD UNIT=VIO, SPACE=(CYL, (1, 1))  
//SYSUT4 DD UNIT=VIO, SPACE=(CYL, (1, 1))  
//SYSUT5 DD UNIT=VIO, SPACE=(CYL, (1, 1))  
//SYSUT6 DD UNIT=VIO, SPACE=(CYL, (1, 1))  
//SYSUT7 DD UNIT=VIO, SPACE=(CYL, (1, 1))  
//SYSUT8 DD UNIT=SYSALLDA, SPACE=(CYL, (1, 1))  
//SYSUT9 DD UNIT=SYSALLDA, SPACE=(CYL, (1, 1))  
//SYSUT10 DD UNIT=SYSALLDA, SPACE=(CYL, (1, 1))  
//SYSUT11 DD UNIT=SYSALLDA, SPACE=(CYL, (1, 1))  
//SYSUT12 DD UNIT=SYSALLDA, SPACE=(CYL, (1, 1))  
//SYSUT13 DD UNIT=SYSALLDA, SPACE=(CYL, (1, 1))  
//SYSUT14 DD UNIT=SYSALLDA, SPACE=(CYL, (1, 1))  
//SYSUT15 DD UNIT=SYSALLDA, SPACE=(CYL, (1, 1))  
//SYSMDECK DD UNIT=SYSALLDA, SPACE=(CYL, (1, 1))  
//SYSIN DD *  
    cbl dll,thread  
    Identification division.  
    Program-id. "TSTHELLO" recursive.  
    Environment division.  
    Configuration section.  
    Repository.  
        Class ZUtil is "com.ibm.jzos.ZUtil" (3)  
        Class HelloWorld is "com.ibm.jzos.sample.HelloWorld" (4)  
        Class HelloWorldException is  
            "com.ibm.jzos.test.helper.HelloException"  
        Class JavaException is "java.lang.Exception"  
        Class JavaObject is "java.lang.Object"  
        Class JavaString is "java.lang.String"  
        Class JavaClass is "java.lang.Class"  
        Class stringArray is "jobjectArray:java.lang.String".
```

```

Data Division.
Working-storage section.
01 args          object reference stringArray.
01 argsLen      pic s9(9) binary value 0.
01 jstring1     object reference JavaString.
01 stringClass  object reference JavaClass.
01 ex           object reference JavaException.
01 stringBuffer pic X(256) usage display.
Linkage section.
COPY "JNI" SUPPRESS.

Procedure division.
    Display "COBOL program TSTHELLO entered"
    Set address of JNIEnv to JNIEnvPtr
    Set address of JNINativeInterface to JNIENV
*
* This static JZOS method will redirect Java stdout/stderr
* to DD:STDOUT and DD:STDERR, which may be spool files or data sets
*
    Invoke ZUtil "redirectStandardStreams"          (3)
    Perform ErrorCheck
    Display "Returned from ZUtil.redirectStandardStreams"

*
* We invoke com.ibm.jzos.sample.HelloWorld,
* but this could be any arbitrary Java code
*
    Perform BuildEmptyArgsArray.
    Invoke HelloWorld "main"          (4)
        using by value args
    Perform ErrorCheck
    Display "Returned from HelloWorld.main"

* We invoke com.ibm.jzos.test.HelloException
* which we expect to throw an Exception and exit RC=32
    Invoke HelloException "main"
        using by value args
    Perform ErrorCheck
    Display "Returned from HelloException.main"
    Goback.

ErrorCheck.
    Call ExceptionOccurred
        using by value JNIEnvPtr
        returning ex
    If ex not = null then
        Call ExceptionClear using by value JNIEnvPtr
        Display "Caught a Java exception"
        Invoke ex "printStackTrace"
        Stop run
    End-if.

BuildEmptyArgsArray.
* Create a new empty string
    Call NewString
        using by value JNIEnvPtr
        address of stringBuffer
        0
        returning jstring1
    If jstring1 not = null then
        Display "NewString returned OK"
    Else
        Display "NewString returned null!"
        Stop run
    End-if

* Get a reference to the String class object
    Call GetObjectClass
        using by value JNIEnvPtr jstring1
        returning stringClass
    If stringClass not = null then
        Display "GetObjectClass returned OK"
    Else
        Display "GetObjectClass returned null!"
        Stop run
    End-if

* Create a zero-length String[] array
    move 0 to argsLen
    Call NewObjectArray
        using by value JNIEnvPtr

```

```

        argLen stringClass jstring1
        returning args
    If args not = null then
        Display "NewObjectArray returned OK"
    Else
        Display "NewObjectArray returned null!"
        Stop run
    End-if.

    End program "TSTHELLO".

/*
//LKED EXEC PGM=IEWL,COND=(4,LT,COMPILE),
//          PARM='RENT,LIST,LET,DYNAM(DLL),CASE(MIXED)'
//SYSLIB DD DSN=&LIBPRFX..SCEELKED,DISP=SHR
// DD DSN=&LIBPRFX..SCEELKEX,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTEM DD SYSOUT=*
//SYSLMOD DD DSN=&&GOSET(TSTHELLO),DISP=(MOD,PASS),UNIT=3390,
// SPACE=(CYL,(1,1,1)),DSNTYPE=LIBRARY
//SYSDEFS DD DUMMY
//OBJMOD DD DSN=&&OBJECT,DISP=(OLD,DELETE)
//SYSLIN DD *
INCLUDE OBJMOD(TSTHELLO)
INCLUDE '/usr/lpp/java/J7.0/bin/j9vm/libjvm.x'           (5)
INCLUDE '/usr/lpp/cobol/V5R1/lib/igzcjava.x'           (5)
/*
/* Note: we expect RC=32 since we should Stop run for exception
/*
//GO EXEC PGM=TSTHELLO,COND=(4,LT,LKED)
//CEEOPST DD *
* Be careful when editing: quoted ENVARS wrap at col 72
ENVAR(                                                (6)
"PATH=bin:/usr/lpp/java/J7.0/bin",
"LIBPATH=lib:/usr/lib:/usr/lpp/java/J7.0/bin:/usr/lpp/java/J7.0/lib/s390
:/usr/lpp/java/J7.0/lib/s390/j9vm",
"CLASSPATH=/home/g1java1/jzostest/jzos_test.jar")
POSIX(ON) XPLINK(ON)
*
* Add this ENVAR to send stdout/stderr to DD:SYSOUT
* "COBJVMINITOPTIONS=-Djzos.merge.sysout=true",
* Debugging options:
* "COBJVMINITOPTIONS=-Xdump:ceedump -Xcheck:jni:trace -Xjit:verbose")
//STEPLIB DD DSN=*.LKED.SYSLMOD,DISP=(OLD,PASS)
// DD DSN=&LIBPRFX..SCEERUN2,DISP=SHR
// DD DSN=&LIBPRFX..SCEERUN,DISP=SHR
//SYSOUT DD SYSOUT=*
//CEEDUMP DD SYSOUT=*
//SYSUDUMP DD DUMMY
/*
/** ZUtil.redirectStandardStreams will point to these for sdtout/stderr
/** Unless you add the -Djzos.merge.sysout=true option above.
/** Using that option, both Java stdout/stderr with go to DD:SYSOUT
//STDOUT DD SYSOUT=*
//STDERR DD SYSOUT=*
/*
/** JAVAOUT/JAVAERR should not be used unless redirectStandardStreams fails
/** so you may choose to point these to DUMMY
//JAVAOUT DD PATH='/tmp/cob2jav.javaout',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=(SIRUSR,SIWUSR,SIRGRP)
//JAVAERR DD PATH='/tmp/cob2jav.javaerr',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=(SIRUSR,SIWUSR,SIRGRP)           (8)

```

- (1) Set the JCL symbols to match your environment.
- (2) The JNI.cpy file is shipped with the compiler and is available as member JNI in data set SIGYLIB.
- (3) The ZUtil redirectStandardStreams method will redirect Java System.out and System.err to DD:STDOUT and DD:STDERR respectively.
- (4) The com.ibm.jzos.sample.HelloWorld class prints “Hello World! (stdout)” to System.out and “Hello World! (stderr)” to System.err. You can download the com.ibm.jzos.sample.HelloWorld class with the [JZOS samples](#).

- (5) Set the INCLUDEs to point to the locations where your Java and COBOL are installed.
- (6) Set the Language Environment ENVARs to point to your Java home directories. Note that individual environment variable settings wrap at column 72.
- (7) You can add the *COBJVMINITOPTIONS* environment variable as shown to set Java system properties for the JVM. The `jzos.merge.sysout=true` property can be used to merge both `System.out` and `System.err` to go to `DD:SYSOUT`.
- (8) The `JAVAOUT DD` and `JAVAERR DD` statements are not used if `ZUtil.redirectStandardStreams()` works properly, so you can point these to `DD DUMMY`.

About JZOS

Java Batch Launcher and Toolkit for z/OS (JZOS) is a set of tools that helps you develop z/OS Java applications that run in a traditional batch environment, and that access z/OS system services. For details, see the [*JZOS Installation and User's Guide*](#).

Part 7. COBOL/Java interoperability outside of the object-oriented (OO) COBOL framework

Enterprise COBOL supports a more comprehensive and flexible form of Java interoperability than that provided by the object-oriented (OO) COBOL framework.

The main goals of the stand-alone COBOL/Java interoperability feature are as follows:

- Provide an easy way for Java applications to call existing COBOL programs without requiring substantial changes to the COBOL programs and without imposing substantial new build requirements on them. Instead, user programs will be called dynamically and can continue to reside in typical user program object libraries.
- Provide an easy way to modify existing COBOL programs so that they can conveniently make calls to static Java methods using a standard CALL statement.
- Provide comprehensive support for automatic conversion between a wide variety of popular Java data types and their corresponding COBOL types so that parameter passing and returned value handling require no special processing in user code. This substantially reduces the need for COBOL programmers to know anything about Java Native Interface (JNI) programming. By comparison, in the OO COBOL framework, even dealing with an incoming String object from Java requires knowledge of JNI to get, and possibly convert, the Java string data so that it can be moved into a data item that is defined with a PIC X, PIC N, or PIC U clause.

COBOL data types that are automatically mapped to Java types are referred to as Java-compatible COBOL data types. For details, see Mapping between COBOL and Java types in the *Enterprise COBOL for z/OS Language Reference*.

- Provide Java applications a way to easily access COBOL WORKING-STORAGE memory directly, for both read and write purposes.
- Make it easy for AMODE 31 COBOL programs to interoperate with 64-bit Java applications.

To meet these objectives, the stand-alone Java interoperability feature of Enterprise COBOL consists of the following elements:

- JAVA-CALLABLE directive that makes an existing COBOL program automatically callable by Java.
- JAVA-SHAREABLE directive that identifies one or more 01/77-level WORKING-STORAGE data items that are to be made accessible to Java applications for both reading and writing.
- An enhanced CALL statement that interprets literal CALL targets of the form: '*Java.java-class-name.java-static-method-name*' as calls to a static Java method.

Related references

Example: COBPROD application

JAVA-CALLABLE (*Enterprise COBOL for z/OS Language Reference*)

JAVA-SHAREABLE (*Enterprise COBOL for z/OS Language Reference*)

Chapter 41. Making an existing program callable from Java

The JAVA-CALLABLE directive is provided to make an existing COBOL program callable from Java.

You can use the JAVA-CALLABLE directive to make an existing COBOL program callable from Java if the program has a well-defined interface that consists of zero or more Java-compatible parameters and optionally a returned value that must also be Java-compatible.

You need to add the JAVA-CALLABLE directive to your existing program anywhere before the PROCEDURE DIVISION header. When the program is compiled, the compiler will generate a file called a "COBOL native method stub" program which is written to a z/OS UNIX directory. You can specify the location of the directory via the OUTPATH suboption of the compiler option JAVAIOP. For more details, see ["JAVAIOP" on page 369](#).

The main purpose of the COBOL native method stub file is as follows:

- Convert any incoming Java arguments to the appropriate format of the COBOL parameter that will receive them
- Make a dynamic call to the existing COBOL program, which must reside in your program object library included in the STEPLIB when the program runs
- Convert an outgoing returned value from a COBOL data format to an appropriate Java format before returning to Java

There are other stub files associated with a COBOL program. See in *Enterprise COBOL for z/OS Language Reference* for more details.

A COBOL/Java interoperable application may involve many Java-callable COBOL programs. All stub files needed for the application must be built into a DLL for the application using the cjbuild utility that exists in the bin sub-directory of the COBOL install directory in the z/OS UNIX file system. The resulting DLL will be loaded into the Java Virtual Machine (JVM) at run time before a call from Java to COBOL is made or when Java is accessing the WORKING-STORAGE memory of COBOL, and will also be accessed whenever a COBOL program makes a call to a static Java method using the CALL statement. See ["Building non-OO applications that interoperate with Java" on page 301](#) for more details.

Example

```
identification division.
program-id. COBPROG.
data division.
working-storage section.
linkage-section.
01 arg1 pic s9(9) comp-5.
01 arg2.
   03 pic s9(18) comp-5 occurs 10 times.
01 retval pic x(20).
>>JAVA-CALLABLE
procedure division using arg1 arg2 arg3
                    returning retval
* COBOL code follows
:
```

In this example, the JAVA-CALLABLE directive makes this program automatically callable by Java.

Note:

- If a COBOL program with the JAVA-CALLABLE directive has parameters that are not one of the Java-compatible types, an error will be generated.
- If a general COBOL group item is specified as a parameter and is not a legal Java-compatible array type, then it will be mapped to a byte array in Java (byte[]). In this case, you need to fill the byte array with data that matches the layout of the COBOL group that will be used to map those bytes.

For more details, see in *Enterprise COBOL for z/OS Language Reference*.

The COBPROG program should be compiled as usual but with the JAVAIOP(OUTPATH(zos-unix-directory)) option specified to indicate an output directory on the z/OS UNIX file system where the generated "stub" programs for the application are to be generated.

The above program will produce at least one stub program in the specified output directory which is the so-called "native method" stub program. In this example, the name of the stub program is as follows:

```
COBPROG_java_native.cbl
```

When running the cjbuild utility to build the DLL for this application, you can specify a package name (-p option) that will indicate the Java package with which the COBOL native methods are to be associated in your Java application. The default is enterprise.COBOL.

For an application that consists of just one COBOL native method, you can invoke cjbuild utility as follows:

```
cjbuild -v methods app1
```

where "methods" refers to a text file in the current directory that contains the name of the COBOL program to be made Java-callable. For example, COBPROG, and where "app1" is used to construct the name of the resulting DLL. For a DLL written to z/OS, the DLL will be called libapp1.so. If the DLL is output to an MVS data set, the DLL will appear as a member in that data set with the name LIBAPP1. See the -d option of cjbuild utility for details.

Note: If the DLL resides on the z/OS UNIX file system, you can use the java.library.path property of the JVM on the java command that starts the application to indicate the location of the DLL.

For example, to run a Java application that starts with the "main" method of class TestApp and calls a COBOL program with an associated stub program DLL located in /home/myid/libs, issue the following command:

```
java -Djava.library.path="/home/myid/libs" TestApp
```

Otherwise, if the DLL is output to a data set, ensure that data set is in your STEPLIB at run time.

The actual Java code to invoke this native method is as follows:

```
import enterprise.COBOL.*;

:
int arg1 = 5;
int[] arg2 = new int[10];
:
String retval = enterprise.COBOL.progs.COBPROG(arg1, arg2);
```

Note: If the name of your COBOL native method program is not defined using a literal, that is, the name is not case-sensitive, then you must refer to it in Java in its form of upper case, as in the above example with COBPROG. However, if the COBPROG program was compiled with the PGMNAME (LONGMIXED) option in effect, then the program name should be specified as "cobprog" and the program should be referred to in Java with the name exactly as it appears in the quotes. See the following example:

```
String retval = enterprise.COBOL.progs.cobprog(arg1, arg2);
```

Related references

Example: COBPROD application

JAVA-CALLABLE (*Enterprise COBOL for z/OS Language Reference*)

Chapter 42. Calling static Java methods from COBOL

In the same way that Java applications can make calls to COBOL and have the arguments be automatically converted between Java and COBOL data formats, COBOL can make calls to Java and have the same automatic conversions performed. Only calls to static Java methods are supported.

You can make a call to Java using a CALL statement. The call target should be a literal in the following form:

```
'Java.java-class-name.java-static-method-name'
```

where `java-class-name` is a Java class name that is fully-qualified with its package name if it is part of a package, and where `java-static-method-name` is the name of the static method being called.

If the method is part of a group of nested classes, each nested class name must be introduced with a `$` character in the literal.

For example, to call a method `myMethod()` that exists in nested class `Util` in outer class `TestApp` as part of package `com.acme`, use the following CALL statement:

```
CALL 'Java.com.acme.TestApp$Util.myMethod' USING ...
```

Note: The 'Java.' prefix of the literal is not case-sensitive, that is, 'JAVA.' and 'jAvA.' would both be accepted, but the remaining portion of the literal is case-sensitive and must match the names as they are declared in Java.

Example

The following example consists of Java class `TestApp` with static method `MyMethod` having the indicated signature:

```
package com.acme;
import java.math.*;

public class TestApp
{
    public static void myMethod(float f1, long l1, BigDecimal[] bdarr) {
        :
    }
}
```

The method can be called from COBOL as follows:

```
working-storage section.
01 f1 comp-1.
01 l1 pic s9(18) comp-5.
01 bdarr.
   03 pd pic s9(9)v9(2) comp-3 occurs 3 times.
Procedure division.
:
CALL 'Java.com.acme.TestApp.myMethod' USING f1 l1 bdarr
```

To build the DLL for this COBOL application that calls Java, `cjbuild` can be invoked as follows:

```
cjbuild -v methods app1
```

where `methods` is the name of a file in the current z/OS UNIX directory containing the following line:

```
Java.com.acme.TestApp.myMethod
```

Related references

Example: COBPROD application

Chapter 43. Using Java-compatible array types in COBOL

A COBOL table is compatible with a single-dimension Java array when its definition follows a well-defined pattern.

Specifically, for a COBOL table to be considered compatible with a Java array, you should define the table in the following way:

```
01 array.  
  03 array-data <java-compatible-elementary-type> occurs X times.
```

where `java-compatible-elementary-type` is one of the items in the Table: Java-compatible elementary COBOL types in the *Enterprise COBOL for z/OS Language Reference*.

Note: The `array` and `array-data` names are placeholders and any legal COBOL data names are acceptable.

For example, a COBOL table that corresponds to an array of values of Java primitive type `double` (`double []`) would be defined as follows:

```
01 array-double.  
  03 double-data comp-2 occurs 10 times.
```

An example of a COBOL table that corresponds to a `java.math.BigDecimal` object is:

```
01 array-packed.  
  03 packed-data pic s9(5)v9(4) usage comp-3 occurs 20 times.
```

Note: `USAGE DISPLAY` can also be used here instead of `USAGE COMP-3` or `USAGE PACKED-DECIMAL`, that is, both packed decimal and zoned decimal types in COBOL map to `java.math.BigDecimal` in Java. Any legal packed or zoned decimal definition is compatible with `java.math.BigDecimal`. All conversions between native COBOL packed or zoned decimal formats and `java.math.BigDecimal` objects are handled by conversion routines that are part of the IBM JZOS Web Development Toolkit product.

An example of an array that corresponds to a `java.lang.String` object is:

```
01 array-string  
  03 string-data pic n(20) occurs 15 times.
```

Note: For string arrays, `PIC X`, `PIC N`, or `PIC U` can all be used. However, as Java strings are stored in the UTF-16 Unicode encoding, `PIC N` will be the most efficient way of communicating string data between Java and COBOL because codepage conversion is not needed.

Byte arrays and general group parameters

The COBOL compiler looks for group parameters in Java-callable programs and group arguments in calls to Java that might match the definition of a Java-compatible array type.

For any groups that do not meet the definition of a Java-compatible array type, the COBOL compiler assumes that you want to treat the group as a flat array of bytes in Java (`byte []`) and that the Java code will extract data from the byte array or write data to the byte array in a way that matches the layout of the data in COBOL. For details about Java-compatible array types, see the Table: Java-compatible array COBOL types in the *Enterprise COBOL for z/OS Language Reference*.

Note: For any group parameter or argument that is treated as corresponding to an array of bytes in Java, an informational message `IGYPA3415I` will be generated by the compiler.

For example, consider the following COBOL group:

```
01 g1.  
03 g1-data1 pic s9(9) comp-5 value 12.  
03 g1-data2 pic s9(18) comp-5 value 15.
```

This group will be identified by the compiler as corresponding to an array of bytes in Java. If `g1` is used as a parameter in a COBOL program with the `JAVA-CALLABLE` directive, then it is assumed that the incoming data from Java is in a byte array (`byte[]`) and it is the Java program's responsibility to set up the array correctly.

Meanwhile, if the COBOL program passes `g1` as an argument in a `CALL` to a static Java method, the corresponding Java method must receive the group as an array of bytes.

Here is an example of a static Java method that can be called from COBOL and can receive group argument `g1` as a byte array and access the two subordinate data items of `g1`:

```
import java.nio.*;  
  
public class TestApp  
{  
    public static void myMethod(byte[] g1)  
    {  
        ByteBuffer bb = ByteBuffer.wrap(g1, 0, 12);  
        System.out.println("data1 = " + bb.getInt(0));  
        System.out.println("data2 = " + bb.getLong(4));  
    }  
}
```

Array update semantics and restrictions in Java-interoperable applications

In Java, arrays are always considered objects, even when they are arrays of Java primitive types (such as `short[]`, `int[]`, `long[]`), so when arrays are passed between Java methods, changes made by a called method are always reflected in the caller method, which is the normal behavior in Java for all object arguments.

This behavior is preserved in the Enterprise COBOL Java interoperability model for arrays of primitive types. For example, if a Java method passes an array of a Java primitive type to COBOL and COBOL updates the COBOL table that receives the data, those changes are always reflected back to the Java caller. Similarly, when COBOL calls Java and passes a table of data that is received in the called Java method as an array object, changes to the array data made in Java will be reflected in the COBOL table when the Java method returns.

Note: For performance reasons, when COBOL receives an array of `java.lang.String` objects or an array of `java.math.BigDecimal` objects from Java, any changes to that data in COBOL will not be reflected on the Java side of the application. Similarly, when COBOL passes a table of PIC X/N/U items to Java, any changes to that data in Java is not reflected on the COBOL side of the application.

Array restrictions

At present, only fixed-length tables in COBOL can receive Java arrays (no ODO tables), and the number of elements in the incoming Java array and the number of elements in the COBOL table as indicated in its `OCCURS` clause must match exactly. Otherwise, a Java exception of class `java.lang.IllegalArgumentException` is thrown. This exception can be caught in Java code with `try/catch` and can be caught by a COBOL program calling a static Java method using the `ON EXCEPTION` phrase of the `CALL` statement.

Note: In Java, array parameters of a method are always dynamic and can receive an incoming array of any length. Therefore, calls from COBOL to static Java methods are more flexible because a mismatch in the number of elements of array arguments is not possible. For an array returned from Java to COBOL, however, the length must still match exactly or an exception will be thrown.

Chapter 44. Accessing COBOL WORKING-STORAGE data items from Java

This topic explains how data items in COBOL WORKING-STORAGE can be accessed from Java.

Any Java-compatible data items in a COBOL program that fall under the scope of a `JAVA-SHAREABLE ON` directive are accessible from Java. The directive applies only to 01/77-level items, thus if a `JAVA-SHAREABLE ON` directive appears in the middle of a group, it will apply to the next 01/77-level item that follows. However, if an 01-level group item falls under the scope of a `JAVA-SHAREABLE ON` directive, all data items subordinate to the 01-level item in the group can be accessed from Java.

When a data item is shareable with Java, the compiler generates a Java class that defines an interface to both read and write to the memory of the data item from Java code. The interface class is instantiated at run time and accessible through a special class called `strg` that belongs to the package indicated by the `-p` option when `cjbuild` is run, where `enterprise.COBOL` is the default package name. The `strg` class will provide an interface for each COBOL program in the application that shares WORKING-STORAGE memory with Java, and for each program there will be an interface for each of the WORKING-STORAGE 01/77-level items that are shareable.

Note:

1. Because the mechanism for sharing data between COBOL and Java involves EXTERNAL data items in COBOL, data items that are Java-shareable should have a unique name across the entire COBOL part of the application; otherwise, there is a possibility for a name collision. The name of Java-shareable data items must also be 26 characters or less.
2. Before the WORKING-STORAGE memory of a COBOL program can be accessed from Java, the WORKING-STORAGE of the program should be initialized first. This can only happen if the program is called directly or indirectly from Java.
3. Any data item that falls under the scope of a `JAVA-SHAREABLE ON` directive that is not a Java-compatible type will be ignored.

Example

Consider the following Java-shareable group defined in Java-callable COBOL program PROGA:

```
JAVA-SHAREABLE ON
01 g1.
   03 data1 pic s9(9) comp-5.
   03 data2 pic s9(5)v9(4) comp-3.
01 f1 comp-1.
JAVA-SHAREABLE OFF
```

And consider the following Java-shareable group that is defined in Java-callable COBOL program PROGB:

```
JAVA-SHAREABLE ON
01 g2.
   03 data1 pic x(20).
   03 data2 pic comp-2.
01 i1 pic s9(9) comp-5.
JAVA-SHAREABLE OFF
```

Assume the default package `enterprise.COBOL` was used when building the DLL for the application that uses `cjbuild`. You can use the following Java code to access these data items:

```
import enterprise.COBOL.*;
import java.math.*;

public class TestApp
{
    public static void main(String[] args)
    {
        // Make sure COBOL programs have been called at least once
    }
}
```

```

// to initialize their WORKING-STORAGE
enterprise.COBOL.progs.PROGA(); // assume no arguments to pass
enterprise.COBOL.progs.PROGB(); // assume no arguments to pass

// get data item values from PROGA
int g1_data1 = enterprise.COBOL.strg.PROGA.G1.data1.get();
BigDecimal g1_data2 = enterprise.COBOL.strg.PROGA.G1.data2.get();
float f1 = enterprise.COBOL.strg.PROGA.F1.get();

// put data items from PROGA
enterprise.COBOL.strg.PROGA.G1.data1.put(25);
enterprise.COBOL.strg.PROGA.G1.data2.put(new BigDecimal(25.3645));
enterprise.COBOL.strg.PROGA.F1.put(3.1415f);

// get data item values from PROGB
String g2_data1 = enterprise.COBOL.strg.PROGB.data1.put();
double g2_data2 = enterprise.COBOL.strg.PROGB.data2.put();
int i1 = enterprise.COBOL.strg.PROGB.I1.get();

// put data items from PROGB
enterprise.COBOL.strg.PROGB.data1.put("Hello, world!");
enterprise.COBOL.strg.PROGB.data2.put(3.1415926);
enterprise.COBOL.strg.PROGB.I1.put(26);
}
}

```

Related references

JAVA-CALLABLE (*Enterprise COBOL for z/OS Language Reference*)

JAVA-SHAREABLE (*Enterprise COBOL for z/OS Language Reference*)

Related tasks

[“Step 2: Running the cjbuild utility to build a DLL of Java stub programs” on page 304](#)

Chapter 45. Handling errors in Java-interoperable COBOL applications

This topic provides suggestions on COBOL/Java interoperability error handling.

For cases where COBOL calls out to a static Java method using the CALL statement, exceptions can occur on the Java side of the application or in the Java call stub program. All exceptions occurring during the call can be handled using the ON EXCEPTION phrase of the CALL statement. If an exception occurs and the ON EXCEPTION phrase is specified, the exception will be cleared before the code in the ON EXCEPTION phrase is executed. If desired, user exception handling code can access the Java exception object that caused the exception through the IGY-JAVAIOP-CALL-EXCEPTION special register. To propagate the exception up the chain of calling programs, the JNI function Throw() can be used to re-throw the exception object. If an exception occurs and the ON EXCEPTION phrase is not specified, the COBOL run unit will be stopped due to the unhandled exception. An alternative method of handling exceptions that occur on the Java side of the application when COBOL calls a static Java method is for the static Java method to catch any exceptions and convert them to a return code that is returned to the COBOL caller. On the COBOL side, the COBOL application can decide what to do based on the return code.

For cases where Java calls COBOL, if a Java exception occurs in the native method call stub program when executing JNI functions, the stub program will not clear the exception and will immediately return to the Java side of the application where the exception can be caught and handled appropriately. If the exception is not handled, the Java application will be terminated due to the unhandled exception.

Chapter 46. AMODE considerations

This topic describes different possible AMODE configurations for non-OO COBOL applications that interoperate with Java.

Non-OO COBOL applications that interoperate with Java can run in any one of the following three AMODE configurations:

1. [“AMODE 31 \(31-bit\) COBOL and AMODE 31 \(31-bit\) Java” on page 739](#)
2. [AMODE 31 \(31-bit\) COBOL and AMODE 64 \(64-bit\) Java](#)
3. [AMODE 64 \(64-bit\) COBOL and AMODE 64 \(64-bit\) Java](#)

AMODE 31 (31-bit) COBOL and AMODE 31 (31-bit) Java

To build a COBOL/Java interoperable application that consists of AMODE 31 COBOL and AMODE 31 Java, compile COBOL user programs with the LP(32) option in effect, which is the default when the LP option is not specified, and specify the `-m PURE31` option when running the `cjbuild` utility. Compile all Java files in the application with the Java compiler from the 31-bit Java SDK.

AMODE 31 (31-bit) COBOL and AMODE 64 (64-bit) Java

To build a COBOL/Java interoperable application that consists of AMODE 31 COBOL and AMODE 64 Java, compile COBOL user programs with the LP(32) option in effect, which is the default when the LP option is not specified, and specify the `-m MIX_31_64` option when running the `cjbuild` utility. Compile all Java files in the application with the Java compiler from the 64-bit Java SDK.

AMODE 64 (64-bit) COBOL and AMODE 64 (64-bit) Java

To build a COBOL/Java interoperable application that consists of AMODE 64 COBOL and AMODE 64 Java, compile COBOL user programs with the LP(64) option in effect, and specify the `-m PURE64` option when running the `cjbuild` utility. Compile all Java files in the application with the Java compiler from the 64-bit Java SDK.

Related tasks

[Chapter 17, “Compiling, linking, and running non-OO COBOL applications that interoperate with Java,” on page 301](#)

Part 8. Specialized processing

Chapter 47. 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 z/OS let an interrupted program 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.

The checkpoint routine starts from the COBOL program object that contains your program. While your program is running, the checkpoint routine creates records at points that 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” on page 743](#)

[“Restarting programs” on page 746](#)

[“Resubmitting jobs for restart” on page 748](#)

z/OS DFSMS: Checkpoint/Restart

Related references

[“DD statements for defining checkpoint data sets” on page 744](#)

[“Messages generated during checkpoint” on page 746](#)

[“Formats for requesting deferred restart” on page 747](#)

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 to a checkpoint data set whenever a specified number of records in the COBOL file have been processed or when END OF VOLUME is reached. You cannot use the RERUN clause with files that are defined with the EXTERNAL attribute.

You can write checkpoint records from several COBOL files to 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.

Restrictions: A checkpoint data set must have sequential organization. You cannot write checkpoints in VSAM data sets or in data sets that are 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 that is open.

Checkpoint records are written in 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 in 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 if 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 with the 85 COBOL 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* that indicates the checkpoint data set must not specify an ASCII file.

Related tasks

[“Using checkpoint/restart with DFSORT” on page 237](#)

[“Designing checkpoints” on page 744](#)

[“Testing for a successful checkpoint” on page 744](#)

Related references

[“DD statements for defining checkpoint data sets” on page 744](#)

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 that uses disk files, design the program so that you can identify previously processed records. For example, consider a disk file that contains 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 that are 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 references

z/OS DFSMS: Checkpoint/Restart (Return codes)

DD statements for defining checkpoint data sets

To define checkpoint data sets, use DD statements.

For tape:

```
//ddname DD DSNAME=data-set-name,  
//          [VOLUME=SER=volser],UNIT=device-type,  
//          DISP=({NEW|MOD},PASS)
```

For direct-access devices:

```
//ddname DD DSNAME=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” on page 745](#)

Examples: defining checkpoint data sets

The following examples show the JCL and COBOL coding you can use to define checkpoint data sets.

Writing single checkpoint records, using tape:

```
//CHECKPT DD DSNAME=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 is successfully completed, a message is displayed that associates the jobname (*ddname*, *unit*, *volser*) with the checkpoint taken (*checkid*).

The control program assigns *checkid* as an eight-character string. The first character is the letter C, followed by a decimal number that indicates the checkpoint. For example, the following message indicates the fourth checkpoint taken in the job step:

```
checkid C0000004
```

Restarting programs

The system restart routine retrieves the information recorded in a checkpoint record, restores the contents of main storage and all registers, and 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 in 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, . . .).

Related tasks

[“Using checkpoint/restart with DFSORT” on page 237](#)

[“Requesting automatic restart” on page 746](#)

[“Requesting deferred restart” on page 747](#)

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:

- 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 that 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 748](#)

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 was 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 748](#)

Related references

[“Formats for requesting deferred restart” on page 747](#)

Formats for requesting deferred restart

The formats for the RESTART parameter of the JOB statement and the SYSCHK DD statements are as shown below.

```
//jobname JOB MSGLEVEL=1,RESTART=(request[,checkid])
//SYSCHK DD DSN=DSNAME=data-set-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 that contains the checkpoint data set.

[“Example: requesting a deferred restart” on page 748](#)

Example: requesting a deferred restart

This example shows JCL to restart the GO step of an IGYWCLG procedure at checkpoint identifier (CHECKID) C0000003.

```
//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.

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). 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 in a DD statement) in the previous run, or change the DD statement to mark the data set as OLD.

The system automatically repositions input data sets that are on tape or disk.

[“Example: resubmitting a job for a step restart” on page 749](#)

[“Example: resubmitting a job for a checkpoint restart” on page 749](#)

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 DSN=PRIV.LIB3,DISP=OLD
//SYSCHK DD DSN=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, which requests step restart for any abnormally terminated job step.

```
//J1234 JOB 386,SMITH,MSGLEVEL=1,RD=R
//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)),
```



```
//
//DDCKPNT DD VOLUME=(PRIVATE,RETAIN,,6)
// UNIT=TAPE,DISP=(MOD,PASS,CATLG),
// DSNNAME=C91468,LABEL=(,NL)
```

The DDCKPNT DD statement 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.

Example: resubmitting a job for a step restart

This example shows the changes that you might make to the JCL before you resubmit a job for step restart.

```
//J3412 JOB 386,SMITH,MSGLEVEL=1,RD=R,RESTART=*
//S1 EXEC PGM=MYPROG
//INDATA DD DSNNAME=INVENT[,UNIT=TAPE],DISP=OLD,
// [VOLUME=SER=91468,]LABEL=RETPD=14
//REPORT DD SYSOUT=A
//WORK DD DSNNAME=S91468,
// DISP=(,KEEP),UNIT=SYSDA,
// SPACE=(3000,(5000,500)),
// VOLUME=(PRIVATE,RETAIN,,6)
//DDCHKPNT DD UNIT=TAPE,DISP=(MOD,PASS,CATLG),
// DSNNAME=R91468,LABEL=(,NL)
```

The following changes were made in the example above:

- 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), 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.

[“Example: requesting a step restart” on page 748](#)

Example: resubmitting a job for a checkpoint restart

This example shows the changes that you might make to JCL before you resubmit a job for checkpoint restart.

```
//J3412 JOB 386,SMITH,MSGLEVEL=1,RD=R,
// RESTART=(*,C00000002)
//SYSCHK DD DSNNAME=C91468,DISP=OLD
//S1 EXEC PGM=MYPROG
//INDATA DD DSNNAME=INVENT,UNIT=TAPE,DISP=OLD,
// VOLUME=SER=91468,LABEL=RETPD=14
//REPORT DD SYSOUT=A
//WORK DD DSNNAME=T91468,DISP=(,KEEP),
// UNIT=SYSDA,SPACE=(3000,(5000,500)),
// VOLUME=(PRIVATE,RETAIN,,6)
//DDCKPNT DD UNIT=TAPE,DISP=(MOD,KEEP,CATLG),
// DSNNAME=C91468,LABEL=(,NL)
```

The following changes were made in the example above:

- 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 are DSNAME and DISP as shown.

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.

Chapter 48. Using zlib compression from a COBOL program

Refer to the following example including instructions and tricks about using zlib compression from a COBOL program.

```
//PROCLIB JCLLIB ORDER=IGYV6R40.SIGYPROC
//EXAMPLE1 EXEC IGYWCLG
//COBOL.STEPLIB DD DISP=SHR,DSNAME=IGYV6R40.SIGYCOMP
//SYSADATA DD SYSOUT=*
//COBOL.SYSIN DD *
  CBL PGMNAME(LONGMIXED)
*****
*
* This is a sample testcase to show how you can call zlib.
* It initializes compression (deflate) and then deflates some
* data, then initializes decompression (inflate) and inflates
* the same data.
*
* Some of the key differences from 'typical' COBOL are:
* - Compiler option PGMNAME(LONGMIXED) is required.
*   This means that the Program-Id name has to be a literal
* - This program uses COMP-5 data items so that the program
*   can work with any setting of the TRUNC compiler option
* - Rather than passing the zstream structure by REFERENCE like
*   most COBOL programs, we pass the ADDRESS OF using the
*   BY VALUE phrase to avoid high-order bit getting set
* - This sample has DISPLAY statements to tell you if it was
*   successful or not, and they should be removed for production*
* - The binder (linkage editor) needs to have LIBRARY
*   statements in order to access the functions in the
*   /usr/lpp/hzc/lib/libz.a archive file. See the sample
*   LKED.SYSIN statements below
*
*****
  Id Division.
  Program-id. 'ZLIB'.
*
  Data Division.

  Working-Storage Section.
**> Memory for tests. 'data' to be compressed
01 zinput pic X(65536) value x'00'.
01 zoutput pic X(65536) value x'00'.
01 z.
*
* next_in is a Pointer to zstring
02 next_in Pointer.
02 avail_in Pic S9(9) Comp-5.
02 total_in Pic S9(9) Comp-5.
*
* next_out is a Pointer to zstring
02 next_out Pointer.
02 avail_out Pic S9(9) Comp-5.
02 total_out Pic S9(9) Comp-5.
*
* msg is a Pointer to zstring
02 msg Pointer.
02 state Pointer.
02 zalloc Function-pointer.
02 zfree Function-pointer.
02 opaque Function-pointer.
02 data_type Pic S9(9) Comp-5.
02 Adler Pic S9(9) Comp-5.
02 reserved Pic S9(9) Comp-5.
01 y.
*
* next_in is a Pointer to zstring
02 next_in Pointer.
02 avail_in Pic S9(9) Comp-5.
02 total_in Pic S9(9) Comp-5.
*
* next_out is a Pointer to zstring
02 next_out Pointer.
02 avail_out Pic S9(9) Comp-5.
02 total_out Pic S9(9) Comp-5.
*
* msg is a Pointer to zstring
02 msg Pointer.
02 state Pointer.
```

```

02  zalloc      Function-pointer.
02  zfree       Function-pointer.
02  opaque      Function-pointer.
02  data_type   Pic S9(9)   Comp-5.
02  Adler       Pic S9(9)   Comp-5.
02  reserved    Pic S9(9)   Comp-5.

77  rc-disp    Pic S9(9) Display Sign leading separate.

Linkage Section.
77  zstring    Pic x(1000).
77  rc         Pic S9(9) Comp-5.

Procedure Division returning rc.
**>*****
**>                               */
**> Setup the z_stream structure with defaults                               */
**>                               */
**>*****
    Set zalloc of z to Null
    Set zfree  of z to Null
    Set opaque of z to Null

*   *>*****/
*   *>                               */
*   *> Call deflateInit to initialize the deflate stream                    */
*   *> service using a GZIP wrapper                                       */
*   *>                               */
*   *>*****/
*   *> Call 'DEIN2' Using By Value Address of z,
                                By Value  -1, 8, 31, 8, 0,
                                By Content '1.2.7',
                                By Value  Length Of z
    returning rc

If rc NOT = 0 Then
    Move rc to rc-disp
    Display 'Error: deflateInit failed with Return Code '
                                rc-disp

    If msg of z NOT = Null Then
        Set Address of zstring to msg of z
        Display 'Message = ' zstring
    Else
        Display 'Message pointer is NULL '
    End-if
    Move -1 To Return-code rc

    Goback
Else
    Display 'deflatinit Successful ! '
End-if

*****
* Initialize available input, output, total in for deflate
*****
    Compute avail_in of z = 65536
    Compute avail_out of z = 65536
    Compute total_in of z = 0
*****
* Set input and output pointers
*****
    Set next_out of z to Address of zoutput
    Set next_in of z to Address of zinput

*   *>*****/
*   *>                               */
*   *> Call deflate to compress the data. We only call it                 */
*   *> once with a flush mode of Z_FINISH (4) which indicates             */
*   *> to end the stream                                                   */
*   *>                               */
*   *>*****/
*   *> Call 'deflate' Using By Value Address of z
                                BY Value 4 Returning rc
    If rc NOT = 1 Then
        Move rc to rc-disp
        Display ' Error: deflate returned ' rc-disp

        If msg of z NOT = Null Then
            Set Address of zstring to msg of z
            Display zstring
        End-if

```

```

    Display "avail_in of z= " avail_in of z
    Display "total_in of z= " total_in of z
    Display "avail_out of z= " avail_out of z
    Display "total_out of z= " total_out of z

    Call 'DEEND' Using By Value address of z returning rc

    Move -1 To Return-code rc
    Goback
  Else
    Display 'deflate Successful ! '
  End-if

  Display 'After deflate '
  Display 'avail_in of z= ' avail_in of z
  Display 'total_in of z= ' total_in of z
  Display 'avail_out of z= ' avail_out of z
  Display 'total_out of z= ' total_out of z

* /*****
* /*
* /* Now that we are done , call deflateEnd to cleanup the */
* /* deflate internal state. */
* /*
* /*****
  Call 'DEEND' Using By Value Address of z returning rc
  If rc Not = 0 Then
    Move rc to rc-disp
    Display 'Error: deflateEnd returned ' rc-disp

    If msg of z NOT = Null Then
      Move -1 To Return-code rc
      Goback
    End-if
  Else
    Display 'deflateEnd Successful ! '
  End-if

* /*****
* /*
* /* Inflate the data we just deflated. Call inflateInit to */
* /* initialize the inflate stream */
* /*
* /*****
  Set zalloc of y to Null
  Set zfree of y to Null

  Set opaque of y to Null

  Call 'ININ2' Using By Value Address of y,
                                     By Value 31,
                                     By Content '1.2.7',
                                     By VALUE Length Of y

    returning rc

  If rc NOT = 0 Then
    Move rc to rc-disp
    Display 'Error: inflateInit failed with Return Code '
                                     rc-disp

    If msg of y NOT = Null Then
      Set Address of zstring to msg of y
      Display zstring
    End-if
    Move -1 To Return-code rc
    Goback
  Else
    Display 'inflateInit Successful ! '
  End-if

* /*****
* /*
* /* Set the amount of input based on what deflate returned */
* /* and what we expect the output size to be. */
* /*
* /*****
  Compute avail_in of y = total_out of z
  Compute avail_out of y = 65536
  Compute total_in of y = 0

  Set next_out of y to Address of zinput

```

```

        Set next_in of y to Address of zoutput

* /*****
* /*
* /* Call inflate to decompress the data. Note that we expect */
* /* this to end with a Z_STREAM_END (1) since we provided the */
* /* full stream above. */
* /*
* /*****
    Call 'inflate' Using By Value Address of y
                        By Value 0 Returning rc

    If rc Not = 1 Then
        Move rc to rc-disp
        Display 'Error: inflate returned ' rc-disp

        If msg of z NOT = Null Then
            Set Address of zstring to msg of z
            Display zstring
        End-if

        Display "avail_in of y= " avail_in of z
        Display "total_in of y= " total_in of z
        Display "avail_out of y= " avail_out of z
        Display "total_out of y= " total_out of z

        Call 'INEND' Using By Value Address of y Returning rc
        Move -1 To Return-code rc
        Goback

    Else
        Display 'inflate Successful ! '
    End-if

    Display 'After inflate '
    Display 'avail_in of y= ' avail_in of y
    Display 'total_in of y= ' total_in of y
    Display 'avail_out of y= ' avail_out of y
    Display 'total_out of y= ' total_out of y

* /*****
* /*
* /* Now that we are done call inflateEnd to cleanup the */
* /* internal inflate state of the stream. */
* /*
* /*****
    Call 'INEND' Using By Value Address of y Returning rc
    If rc Not = 0 Then

        If rc Not = 0 Then
            Move rc to rc-disp
            Display 'Error: inflateEnd returned ' rc-disp

            If msg of z NOT = Null Then
                Move -1 To Return-code rc
                Goback
            End-if

        Else
            Display 'inflateEnd Successful ! '
        End-if

        Move zero to return-code rc
        Goback.

/*
//LKED.SYSIN DD *
LIBRARY '/usr/lpp/hzc/lib/libzz.a'
/*
//LKED.SYSLIB DD DSN=CEEZ240.SCEELKED,DISP=SHR
//LKED.SYSLMOD DD DSN=MYLOAD.COBOL.LOAD(ZLIB),DISP=SHR
//GO.STEPLIB DD DSNAME=CEEV240.SCEERUN,DISP=SHR
//GO.SYSUDUMP DD DUMMY
//GO.SYSSORT DD DUMMY
//GO.SYSOUT DD SYSOUT=*
//

```

Part 9. Improving performance and productivity

Chapter 49. Tuning your program

When a program is comprehensible, you can assess its performance. A tangled control flow makes a program difficult to understand and maintain, and inhibits the optimization of its code.

To improve the performance of your program, examine at least these aspects:

- Underlying algorithms: For best performance, using sound algorithms is essential. For example:
 - A sophisticated algorithm for sorting a million items might be hundreds of thousands of times faster than a simple algorithm.
 - If the program frequently accesses data, reduce the number of steps to access the data.
- Data structures: Using data structures that are appropriate for the algorithms is essential.

You can write programs that result in better generated code sequences and use system services more efficiently. These additional aspects can affect performance:

- Coding techniques: Use a programming style that enables the optimizer to choose efficient data types and handle tables efficiently.
- Optimization: You can optimize code by using the OPTIMIZE compiler option.
- Compiler options and USE FOR DEBUGGING ON ALL PROCEDURES: Some compiler options and language affect program efficiency.
- Runtime environment: Consider your choice of runtime options.
- Running under CICS, IMS, or VSAM: Heeding various tips can help make these programs run more efficiently.

Related concepts

[“Optimization” on page 762](#)

Related tasks

[“Using an optimal programming style” on page 757](#)

[“Choosing efficient data types” on page 758](#)

[“Handling tables efficiently” on page 760](#)

[“Optimizing your code” on page 762](#)

[“Choosing compiler features to enhance performance” on page 763](#)

[“Running efficiently with CICS, IMS, or VSAM” on page 769](#)

Language Environment Programming Guide (Specifying runtime options)

Related references

[“Performance-related compiler options” on page 764](#)

Language Environment Programming Guide (Storage performance considerations)

Enterprise COBOL for z/OS Performance Tuning Guide

Using an optimal programming style

The coding style you use can affect how the optimizer handles your code. You can improve optimization by using structured programming techniques, factoring expressions, using symbolic constants, and grouping constant and duplicate computations.

Related tasks

[“Using structured programming” on page 758](#)

[“Factoring expressions” on page 758](#)

[“Using symbolic constants” on page 758](#)

Using structured programming

Using structured programming statements, such as EVALUATE and inline PERFORM, makes your program more comprehensible and generates a more linear control flow. As a result, the optimizer can operate over larger regions of the program, which gives you more efficient code.

Use top-down programming constructs. Out-of-line PERFORM statements are a natural means of doing top-down programming. Out-of-line PERFORM statements can often be as efficient as inline PERFORM statements, because the optimizer can simplify or remove the linkage code.

Avoid using the following constructs:

- ALTER statements
- Explicit GO TO statements
- PERFORM procedures that involve irregular control flow (such as preventing control from passing to the end of the procedure and returning to the PERFORM statement)

Factoring expressions

By factoring expressions in your programs, you can potentially eliminate a lot of unnecessary computation.

For example, the first block of code below is more efficient than the second block of code:

```
MOVE ZERO TO TOTAL
PERFORM VARYING I FROM 1 BY 1 UNTIL I = 10
  COMPUTE TOTAL = TOTAL + ITEM(I)
END-PERFORM
COMPUTE TOTAL = TOTAL * DISCOUNT
```

```
MOVE ZERO TO TOTAL
PERFORM VARYING I FROM 1 BY 1 UNTIL I = 10
  COMPUTE TOTAL = TOTAL + ITEM(I) * DISCOUNT
END-PERFORM
```

The optimizer does not factor expressions across multiple statements. For details, see Factoring expressions in the *Enterprise COBOL for z/OS Performance Tuning Guide*.

Using symbolic constants

To have the optimizer recognize a data item as a constant throughout the program, initialize it with a VALUE clause and do not change it anywhere in the program.

If you pass a data item to a subprogram BY REFERENCE, the optimizer treats it as an external data item and assumes that it is changed at every subprogram call.

Choosing efficient data types

Using the SYNCHRONIZED clause can produce more efficient code.

Consistent data types can reduce the need for conversions during operations on data items. You can also improve program performance by carefully determining when to use fixed-point and floating-point data types.

Related concepts

[“Formats for numeric data” on page 47](#)

Related tasks

[“Choosing efficient computational data items” on page 759](#)

[“Using consistent data types” on page 759](#)

[“Making arithmetic expressions efficient” on page 759](#)

[“Making exponentiations efficient” on page 760](#)

Choosing efficient computational data items

When you use a data item mainly for arithmetic or as a subscript, code `USAGE BINARY` on the data description entry for the item. The operations for manipulating binary data are faster than those for manipulating decimal data.

However, if a fixed-point arithmetic statement has intermediate results with a large precision (number of significant digits), the compiler uses decimal arithmetic anyway, after converting the operands to packed-decimal, decimal floating point, or vector decimal form. For fixed-point arithmetic statements, the compiler normally uses binary arithmetic for simple computations with binary operands if the precision is eight or fewer digits. Above 18 digits, the compiler always uses decimal arithmetic. With a precision of nine to 18 digits, the compiler uses either form.

To produce the most efficient code for a `BINARY` data item, ensure that it has:

- A sign (an `S` in its `PICTURE` clause)
- Eight or fewer digits

For a data item that is larger than eight digits or is used with `DISPLAY` or `NATIONAL` data items, use `PACKED-DECIMAL`. 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 on `ARCH(7)` machines. If a `PACKED-DECIMAL` data item has more than 31 digits, library routines are used. For a `PACKED-DECIMAL` data item with 16-31 digits on `ARCH(8)` or higher level machines, the compiler uses instructions that are more efficient than library calls, but not as fast as if the data item has 15 or fewer digits.

Using consistent data types

In operations on operands of different types, one of the operands must be converted to the same type as the other. Each conversion requires several instructions. For example, one of the operands might need to be scaled to give it the appropriate number of decimal places.

You can largely avoid conversions by using consistent data types and by giving both operands the same usage and also appropriate `PICTURE` specifications. That is, you should ensure that two numbers to be compared, added, or subtracted not only have the same usage but also the same number of decimal places (9s after the `V` in the `PICTURE` clause).

Making arithmetic expressions efficient

Computation of arithmetic expressions that are evaluated in floating point is most efficient when the operands need little or no conversion. Use operands that are `COMP-1` or `COMP-2` to produce the most efficient code.

Define integer items as `BINARY` or `PACKED-DECIMAL` with nine or fewer digits to afford quick conversion to floating-point data. Also, conversion from a `COMP-1` or `COMP-2` item to a fixed-point integer with nine or fewer digits, without `SIZE ERROR` in effect, is efficient when the value of the `COMP-1` or `COMP-2` item is less than 1,000,000,000.

Making exponentiations efficient

Use floating point for exponentiations for large exponents to achieve faster evaluation and more accurate results.

For example, the first statement below is computed more quickly and accurately than the second statement:

```
COMPUTE fixed-point1 = fixed-point2 ** 100000.E+00  
COMPUTE fixed-point1 = fixed-point2 ** 100000
```

A floating-point exponent causes floating-point arithmetic to be used to compute the exponentiation.

Using VOLATILE clauses efficiently

Optimization of data items that are defined with the VOLATILE clause is significantly restricted. Therefore, use the VOLATILE clause only when appropriate.

In particular, it is important to understand that when the VOLATILE clause is used on a group item, the compiler treats all data items subordinate to the group item as volatile, and all higher-level group items that contain the volatile group item are treated as volatile, too. If a particular member of a group needs to be treated as volatile, specify the VOLATILE clause on the data description entry for that item only, where possible.

At present, the primary reason to use the VOLATILE clause is for data items that are set or referenced inside an LE condition handler but are defined outside the LE condition handler program. The VOLATILE clause guarantees that such items are handled correctly by the optimizer. For more information on when to use VOLATILE, see *VOLATILE clause* in the *Enterprise COBOL for z/OS Language Reference*.

Handling tables efficiently

You can use several techniques to improve the efficiency of table-handling operations, and to influence the optimizer. The return for your efforts can be significant, particularly when table-handling operations are a major part of an application.

The following two guidelines affect your choice of how to refer to table elements:

- Use indexing rather than subscripting.

Although the compiler can eliminate duplicate indexes and subscripts, the original reference to a table element is more efficient with indexes (even if the subscripts were BINARY). The value of an index has the element size factored into it, whereas the value of a subscript must be multiplied by the element size when the subscript is used. The index already contains the displacement from the start of the table, and this value does not have to be calculated at run time. However, subscripting might be easier to understand and maintain.

- Use relative indexing.

Relative index references (that is, references in which an unsigned numeric literal is added to or subtracted from the index-name) are executed at least as fast as direct index references, and sometimes faster. There is no merit in keeping alternative indexes with the offset factored in.

Whether you use indexes or subscripts, the following coding guidelines can help you get better performance:

- Specify the element length so that it matches that of related tables.

When you index or subscript tables, it is most efficient if all the tables have the same element length. That way, the stride for the last dimension of the tables is the same, and the optimizer can reuse the rightmost index or subscript computed for one table. If both the element lengths and the number of occurrences in each dimension are equal, then the strides for dimensions other than the last are also equal, resulting in greater commonality between their subscript computations. The optimizer can then reuse indexes or subscripts other than the rightmost.

- Avoid errors in references by coding index and subscript checks into your program.

If you need to validate indexes and subscripts, it might be faster to code your own checks than to use the SSRANGE compiler option.

You can also improve the efficiency of tables by using these guidelines:

- Use binary data items for all subscripts.

When you use subscripts to address a table, use a BINARY signed data item with eight or fewer digits. In some cases, using four or fewer digits for the data item might also improve processing time.

- Use binary data items for variable-length table items.

For tables with variable-length items, you can improve the code for OCCURS DEPENDING ON (ODO). To avoid unnecessary conversions each time the variable-length items are referenced, specify BINARY for OCCURS . . . DEPENDING ON objects.

- Use fixed-length data items whenever possible.

Copying variable-length data items into a fixed-length data item before a period of high-frequency use can reduce some of the overhead associated with using variable-length data items.

- Organize tables according to the type of search method used.

If the table is searched sequentially, put the data values most likely to satisfy the search criteria at the beginning of the table. If the table is searched using a binary search algorithm, put the data values in the table sorted alphabetically on the search key field.

Related concepts

[“Optimization of table references” on page 761](#)

Related tasks

[“Referring to an item in a table” on page 70](#)

[“Choosing efficient data types” on page 758](#)

Related references

[“SSRANGE” on page 401](#)

Optimization of table references

The COBOL compiler optimizes table references in several ways.

For the table element reference ELEMENT(S1 S2 S3), where S1, S2, and S3 are subscripts, the compiler evaluates the following expression:

```
comp_s1 * d1 + comp_s2 * d2 + comp_s3 * d3 + base_address
```

Here comp_s1 is the value of S1 after conversion to binary, comp_s2 is the value of S2 after conversion to binary, and so on. The strides for each dimension are d1, d2, and d3. The *stride* of a given dimension is the distance in bytes between table elements whose occurrence numbers in that dimension differ by 1 and whose other occurrence numbers are equal. For example, the stride d2 of the second dimension in the above example is the distance in bytes between ELEMENT(S1 1 S3) and ELEMENT(S1 2 S3).

Index computations are similar to subscript computations, except that no multiplication needs to be done. Index values have the stride factored into them. They involve loading the indexes into registers, and these data transfers can be optimized, much as the individual subscript computation terms are optimized.

Optimization of variable-length items

A group item that contains a subordinate OCCURS DEPENDING ON data item has a variable length. The program must perform special code every time a variable-length data item is referenced.

Because this code is out-of-line, it might interrupt optimization. Furthermore, the code to manipulate variable-length data items is much less efficient than that for fixed-size data items and can significantly

increase processing time. For instance, the code to compare or move a variable-length data item might involve calling a library routine and is much slower than the same code for fixed-length data items.

Comparison of direct and relative indexing

Relative index references are as fast as or faster than direct index references.

The direct indexing in ELEMENT (I5, J3, K2) requires this preprocessing:

```
SET I5 TO I
SET I5 UP BY 5
SET J3 TO J
SET J3 DOWN BY 3
SET K2 TO K
SET K2 UP BY 2
```

This processing makes the direct indexing less efficient than the relative indexing in ELEMENT (I + 5, J - 3, K + 2).

Related concepts

[“Optimization” on page 762](#)

Related tasks

[“Handling tables efficiently” on page 760](#)

Optimizing your code

When your program is ready for final testing, specify the OPTIMIZE(1|2) compiler option so that the tested code and the production code are identical. Note that IBM recommends that all users use OPT(2) for the best performance.

If you frequently run a program without recompiling it during development, you might also want to use OPTIMIZE(1|2). However, if you recompile frequently, the overhead for OPTIMIZE(1|2) might outweigh its benefits unless you are using the assembler language expansion (LIST compiler option) to fine-tune the program.

For unit-testing a program, you will probably find it easier to debug code that has not been optimized.

To see how the optimizer works on a program, compile it with different levels of optimization and compare the generated code. (Use the LIST compiler option to request the assembler listing of the generated code.)

Related concepts

[“Optimization” on page 762](#)

Related references

[“LIST” on page 372](#)

[“OPTIMIZE” on page 385](#)

Optimization

To improve the efficiency of the generated code, you can use the OPTIMIZE(1) or OPTIMIZE(2) compiler option.

OPTIMIZE(1) causes the COBOL optimizer to do the following optimizations:

- Eliminate unnecessary transfers of control and inefficient branches, including those generated by the compiler that are not evident from looking at the source program.
- Simplify the compiled code for a PERFORM statement. The compiler replicates the PERFORM a number of times to avoid linkage code.
- 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 STGOPT option.)

OPTIMIZE (2) causes the COBOL optimizer to do further optimizations:

- Simplify operations more aggressively and schedule instructions.
- Do interblock optimizations such as global value propagation and loop invariant code motion.

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 then chooses the next best optimization for the CALL statement.

Related concepts

[“PERFORM procedure integration” on page 763](#)

Related references

[“OPTIMIZE” on page 385](#)

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 orderly 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, after which it no longer integrates these procedures.

Related references

[“INLINE” on page 365](#)

INLINE directive (*Enterprise COBOL for z/OS Language Reference*)

Choosing compiler features to enhance performance

Your choice of performance-related compiler options and your use of the USE FOR DEBUGGING ON ALL PROCEDURES statement can affect how well your program is optimized.

You might have a customized system that requires certain options for optimum performance. Do these steps:

1. To see what your system defaults are, get a short listing for any program and review the listed option settings.
2. Determine which options are fixed as nonoverridable at your installation by checking with your system programmer.

- For the options not fixed at installation, select performance-related options for compiling your programs.

Important: Confer with your system programmer about how to tune COBOL programs. Doing so will ensure that the options you choose are appropriate for programs at your site.

Another compiler feature to consider is the `USE FOR DEBUGGING ON ALL PROCEDURES` statement. It can greatly affect the compiler optimizer. The `ON ALL PROCEDURES` option generates extra code at each transfer to a procedure name. Although very useful for debugging, it can make the program significantly larger and inhibit optimization substantially.

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 762](#)

Related tasks

[“Optimizing your code” on page 762](#)

[“Getting listings” on page 443](#)

Related references

[“Performance-related compiler options” on page 764](#)

Performance-related compiler options

In the table below you can see a description of the purpose of each option, its performance advantages and disadvantages, and usage notes where applicable.

Compiler option	Purpose	Performance advantages	Performance disadvantages	Usage notes
AFP (NOVOLATILE) (see “AFP” on page 336)	To control the compiler usage of the Additional Floating Point (AFP) registers that are provided by IBM z/Architecture processors	AFP (NOVOLATILE) lets the compiler generate more efficient code sequences for programs with floating point operations.	None	Poorly behaved assembler code might not adhere to the standard calling convention and might fail to correctly preserve values in Floating Point registers. With AFP (NOVOLATILE), COBOL programs can safely call such routines.
ARCH (see “ARCH” on page 337)	To specify the machine architecture for which the executable program instructions are to be generated	If you specify a higher ARCH level, the machine generates code that uses newer and faster instructions instead of the sequences of common instructions.	None	Your application might abend if it runs on a processor with an architecture level lower than that specified for the ARCH option.
ARITH(EXTEND) (see “ARITH” on page 339)	To increase the maximum number of digits allowed for decimal numbers	None	ARITH(EXTEND) causes some degradation in performance for all decimal data types because of larger intermediate results.	The amount of degradation that you experience depends directly on the amount of decimal data that you use.
AWO (see “AWO” on page 340)	To get optimum use of buffer and device space for QSAM files	Can result in performance savings, because this option results in fewer calls to data management services to handle input and output	None	If you use AWO, the <code>APPLY WRITE-ONLY</code> clause is in effect for all QSAM files in the program that have V-mode records.

Table 99. *Performance-related compiler options (continued)*

Compiler option	Purpose	Performance advantages	Performance disadvantages	Usage notes
BLOCK0 (see “BLOCK0” on page 340)	To take advantage of system-determined block size for QSAM output files	Can result in enhanced processing speed and minimized storage requirements for QSAM output files	None	If you use BLOCK0, a BLOCK CONTAINS 0 clause is activated for all QSAM files in the program that specify neither BLOCK CONTAINS nor RECORDING MODE U in the file description entry.
DATA(31) (see “DATA” on page 349)	To have DFSMS allocate QSAM buffers above the 16 MB line (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.	None	On a z/OS 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 storage above the 16 MB line.
DYNAM (see “DYNAM” on page 356)	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.
FASTSRT (see “FASTSRT” on page 359)	To specify that the IBM DFSORT product (or equivalent) will handle all of the input and output	Eliminates the overhead of returning to Enterprise COBOL after each record is processed.	None	FASTSRT is recommended if direct work files are used for the sort work files. Not all sorts are eligible for this option.
HGPR (see “HGPR” on page 362)	To control the compiler usage of the 64-bit registers provided by IBM z/Architecture processors.	If you specify HGPR(NOPRESERVE), the compiler omits preserving the high-halves of the 64-bit GPRs that a program is using, which improves performance.	None	If your program modifies and does not save the high-halves of the registers, but calling programs depend on the unchanged values, the application might give incorrect results. Exception: It does not apply if the caller of this program is Enterprise COBOL, Enterprise PL/I or z/OS XL C/C++ programs.
INLINE (see “INLINE” on page 365)	To control the compiler usage of inlining procedures (paragraphs or sections) referenced by PERFORM statements in the source program.	Specifying INLINE allows the compiler to decide whether to inline procedures referenced by PERFORM statements. This typically improves performance when the application contains commonly occurring and often-executed procedures.	Specifying INLINE usually increases the size of the module. For commonly occurring but rarely executed procedures, you can use the >>INLINE OFF directive to prevent the compiler from inlining that procedure and increasing the module size.	You can use the INLINE compiler option to indicate that a procedure should be considered eligible for inlining, but the decision whether to inline a procedure in a specific PERFORM statement or not is made by the compiler.
INITCHECK (see “INITCHECK” on page 363)	To have the compiler check for uninitialized data items and issue warning messages when they are used without being initialized.	None	Use of the INITCHECK option might increase compile time and memory consumption.	All of the INITCHECK analyses occur at compile time only. The INITCHECK option has no effect on the behavior or performance of the program after it has been compiled.

Table 99. *Performance-related compiler options (continued)*

Compiler option	Purpose	Performance advantages	Performance disadvantages	Usage notes
MAXPCF (see “MAXPCF” on page 375)	To reduce optimization in programs that require excessive compilation time or excessive storage requirements because of large sizes or complexity.	None	If you specify MAXPCF(<i>n</i>) and <i>n</i> is not zero, when the program complexity factor exceeds <i>n</i> , any specification of OPTIMIZE(1) or OPTIMIZE(2) is reset to OPTIMIZE(0), and a warning message is generated.	None
NUMCHECK (see “NUMCHECK” on page 378)	To have the compiler generate extra code to validate data items when they are used as sending data items.	None	NUMCHECK is much slower than NONUMCHECK, depending on how many zoned decimal (numeric USAGE DISPLAY) data items, packed decimal (COMP-3) data items, and binary data items are used in a COBOL program.	None
NUMPROC (PFD) (see “NUMPROC” on page 382)	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 (NOPFD).	If you use NUMPROC (PFD), the compiler assumes and requires that all decimal items contain the preferred sign values and bypasses the sign "fix-up" process. However, because not all external data files contain the proper signs for COMP-3 or DISPLAY numeric data, and programs might use REDEFINES, group moves, or parameter passing in ways that do not ensure preferred signs, the NUMPROC (PFD) might not be appropriate for many programs.
OPTIMIZE (0 1 2) (see “OPTIMIZE” on page 385)	To optimize generated code for better performance	Generally results in more efficient runtime code	Longer compile time: OPTIMIZE (1 2) requires more processing time for compiles than OPTIMIZE (0).	OPTIMIZE (0) is generally used during program development when frequent compiles are needed; it also allows for symbolic debugging. For production runs, OPTIMIZE (1 2) is recommended.
STGOPT (see “STGOPT” on page 402)	To optimize storage allocation in DATA DIVISION	Generally results in less storage usage	None	STGOPT deletes unused data items, which might be undesirable in the case of time stamps or data items that are used only as markers for dump reading.
PARMCHECK (see “PARMCHECK” on page 387)	To have the compiler generate an extra data item following the last item in WORKING-STORAGE. This buffer data item is then used at run time to check whether a called subprogram corrupted data beyond the end of WORKING-STORAGE.	None	PARMCHECK will cause the compiler to generate slower code for programs with CALL statements. NOPARMCHECK should be in effect for good performance.	None
RENT (see “RENT” on page 391)	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	None

Table 99. *Performance-related compiler options (continued)*

Compiler option	Purpose	Performance advantages	Performance disadvantages	Usage notes
RMODE (ANY) (see “RMODE” on page 392)	To let the program be loaded anywhere	None	None	None
SMARTBIN (see “SMARTBIN” on page 396)	To instruct the compiler to generate modules containing additional binary metadata that enables them to be optimized by IBM Automatic Binary Optimizer (ABO) for z/OS (ABO) 2.2	Enables your COBOL modules to be eligible for ABO optimization in the future to maximize benefits of IBM Z hardware enhancements without the need to recompile and test	Use of this option increases compile time and module size on disk, requiring larger load modules. Size in memory is not impacted as the metadata is created in a NOLOAD segment.	None
SSRANGE (see “SSRANGE” on page 401)	To verify that all table references and reference modification expressions are in proper bounds	SSRANGE generates additional code for verifying table references. Using NOSSRANGE causes that code not to be generated.	With SSRANGE specified, checks for valid ranges do affect compiler performance.	In general, if you need to verify the table references only a few times instead of at every reference, coding your own checks might be faster than using SSRANGE. For performance-sensitive applications, NOSSRANGE is recommended.
TEST (see “TEST” on page 404)	To get full debugging capability when using Debug Tool and to get a symbolic dump of the data items in CEEDUMP. You can also get a symbolic dump of the data items in CEEDUMP with NOTEST (DWARF).	None	Some reduction in optimization occurs when the TEST option is used. More reduction in optimization occurs when the EJPD suboption of TEST is used.	For production runs, using NOTEST or TEST (NOEJPD) is recommended. If during a production run, you want a symbolic dump of the data items in a formatted dump if the program abends, compile using TEST or with NOTEST (DWARF).
THREAD (see “THREAD” on page 408)	To enable programs for execution in a Language Environment enclave that has multiple POSIX threads or PL/I tasks	None	There is a slight performance penalty because of the overhead of serialization logic.	A slight performance penalty occurs in either a threaded or nonthreaded environment.
TRUNC (OPT) (see “TRUNC” on page 409)	To avoid having code generated to truncate the receiving fields of arithmetic operations	Does not generate extra code and generally improves performance	Both TRUNC (BIN) and TRUNC (STD) generate extra code whenever a BINARY data item is changed. TRUNC (BIN) is usually the slowest of these options.	TRUNC (STD) conforms to the 85 COBOL Standard, but TRUNC (BIN) and TRUNC (OPT) do not. With TRUNC (OPT), the compiler assumes that the data conforms to the PICTURE and USAGE specifications. TRUNC (OPT) is recommended where possible.
TUNE (see “TUNE” on page 411)	To specify the architecture for which the executable program will be optimized	Tunes instruction selection and other implementation-dependent performance enhancements for a specific implementation of a hardware architecture	None	If the specified TUNE level is lower than the specified ARCH level, the compiler adjusts the TUNE level to match the ARCH level.

Related concepts

[“Optimization” on page 762](#)

[“Storage and its addressability” on page 39](#)

Related tasks

[“Generating a list of compiler messages” on page 281](#)

[“Evaluating performance” on page 768](#)

[“Optimizing buffer and device space” on page 10](#)

[“Choosing compiler features to enhance performance” on page 763](#)

- [“Improving sort performance with FASTSRT” on page 232](#)
- [“Using striped extended-format QSAM data sets” on page 185](#)
- [“Handling tables efficiently” on page 760](#)

Related references

- [“Sign representation of zoned and packed-decimal data” on page 53](#)
- [“Allocation of buffers for QSAM files” on page 186](#)
- [“COBOL compiler options” on page 329](#)
- [“Conflicting compiler options” on page 334](#)

Evaluating performance

Fill in the following worksheet to help you evaluate the performance of your program. If you answer yes to each question, you are probably improving the performance.

In thinking about the performance tradeoff, be sure you understand the function of each option as well as the performance advantages and disadvantages. You might prefer function over increased performance in many instances.

<i>Table 100. Performance-tuning worksheet</i>	
Compiler option	Consideration
ARCH	Do you use the highest architecture level possible for all environments in which your programs will run? For example, if the lowest level architecture you have including your disaster recovery machines is zEC12 or zBC12, are you using ARCH(10)?
AWO	Do you use the AWO option when possible?
BLOCK0	Do you use BLOCK0 for QSAM files?
DATA	When you use QSAM striped data sets, do you use the RENT and DATA(31) options? Is the program object AMODE 31? Are you running with ALL31(ON)?
DYNAM	Can you use NODYNAM? Consider the performance tradeoffs.
FASTSRT	When you use direct work files for the sort work files, did you use the FASTSRT option?
INLINE	Do you use INLINE when possible?
INVDATA	Do you use INVDATA when possible?
NUMCHECK	Do you use NONUMCHECK for production runs?
NUMPROC	Do you use NUMPROC(PFD) when possible?
OPTIMIZE	Do you use a non-zero OPTIMIZE level for production runs?
PARMCHECK	Do you use NOPARMCHECK for production runs?
SSRANGE	Do you use NOSSRANGE for production runs?
TEST	Do you use NOTEST or TEST(NOJPD) for production runs?
TRUNC	Do you use TRUNC(OPT) when possible?
TUNE	Do you use the highest architecture level possible for the environment in which your programs will run most often? For example, if the machine in your production environment is zEC12 or zBC12, are you using TUNE(10)?

Related concepts

[“Storage and its addressability” on page 39](#)

Related tasks

[“Choosing compiler features to enhance performance” on page 763](#)

Related references

[“Performance-related compiler options” on page 764](#)

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 LINK commands to COBOL CALL statements to improve transaction response time.

IMS: If your application runs under IMS, preloading the application program and the library routines can help reduce the overhead of loading and searching. 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 Language Environment 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) that is appropriate for the application. A smaller CISZ results in faster retrieval for random processing at the expense of inserts. 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 use alternate indexes, access method services builds them more efficiently than the AIXBLD runtime option.

Related tasks

[“Coding COBOL programs to run under CICS” on page 483](#)

Chapter 24, [“Developing COBOL programs for IMS,” on page 505](#)

[“Improving VSAM performance” on page 211](#)

Language Environment Customization

Related references

Language Environment Programming Guide (Specifying runtime options)

Choosing static or dynamic calls

If you can arrange your modules, and the programs that frequently call each other are in one module, static calls are faster than dynamic calls. On the other hand, dynamic calls can make updates to applications easier because multiple users of a subprogram would not have to be rebound (re-link-edited).

For more information, see [“Performance considerations of static and dynamic calls” on page 540](#).

Related concepts

[“Performance considerations of static and dynamic calls” on page 540](#)

Chapter 50. Simplifying coding

You can use coding techniques to improve your productivity. By using the COPY statement, the format 2 SORT statement, COBOL intrinsic functions, Language Environment callable services, COBOL/Java communication features, calls between AMODE 31 (31-bit) and AMODE 64 (64-bit) COBOL programs, and user-defined functions, you can avoid repetitive coding and having to code many arithmetic calculations or other complex tasks. In addition, compiler options including SSRANGE, NUMCHECK, PARMCHECK, and INITCHECK can help detect invalid data issues to ease your migration efforts from COBOL 4 or earlier to COBOL 6 or later.

COPY statement

If your program contains frequently used code sequences (such as blocks of common data items, input-output routines, error routines, or even entire COBOL programs), write the code sequences once and put them in a COBOL copy library. You can use the COPY statement to retrieve these code sequences and have them included in your program at compile time. Using copybooks in this manner eliminates repetitive coding. For details, see [“Eliminating repetitive coding”](#) on page 772.

Format 2 SORT statement

To sort a table, you can use the format 2 SORT statement to simplify coding. It provides a much simpler way compared to the format 1 SORT statement. For details, see [“Using the format 2 SORT statement to sort a table”](#) on page 777.

Intrinsic functions

COBOL provides various capabilities for manipulating strings and numbers. These capabilities can help you simplify your coding. For details, see [“Converting data items \(intrinsic functions\)”](#) on page 113 and [“Evaluating data items \(intrinsic functions\)”](#) on page 117.

Language Environment callable services

The Language Environment date and time callable services store dates as fullword binary integers and store time stamps as long (64-bit) floating-point values. These formats let you do arithmetic calculations on date and time values simply and efficiently. You do not need to write special subroutines that use services outside the language library to perform such calculations. For details, see [“Using Language Environment callable services”](#) on page 773.

COBOL/Java interoperability

COBOL programs can interoperate with Java programs without using the object-oriented programming framework. Therefore, you can easily extend your COBOL applications with Java. For details, see [Chapter 17, “Compiling, linking, and running non-OO COBOL applications that interoperate with Java,”](#) on page 301 and [Part 7, “COBOL/Java interoperability outside of the object-oriented \(OO\) COBOL framework,”](#) on page 727.

AMODE 31 (31-bit) and AMODE 64 (64-bit) COBOL programs interoperability

You can create AMODE 64 COBOL applications that can interoperate with your existing AMODE 31 COBOL applications. AMODE 64 COBOL applications can access data items greater than the existing AMODE 31 data size limits, without changes to the program logic. For details, see [“COBOL applications with mixed AMODE 31 and AMODE 64 programs”](#) on page 528.

User-defined functions

As with many popular programming languages, COBOL supports user-defined functions, which gives you a familiar structure. Write your own functions using the user-defined function definition and invoke them like intrinsic functions, improving code modularity and maintainability. For details, see [Function identification division in the *Enterprise COBOL for z/OS Language Reference*](#) and [Chapter 32, “Using user-defined functions,”](#) on page 585.

Compiler options to detect invalid data during COBOL migration

One of the most common problems when migrating to Enterprise COBOL 6 is caused by invalid COBOL data that cannot be detected by inspecting source code. To identify these problems more easily, consider using the 2-step compile process, that is, use the SSRANGE, NUMCHECK, PARMCHECK, INITCHECK, and OPT(0) compiler options for initial code changes and unit tests, and then recompile with NOSSRANGE, NONUMCHECK, NOPARMCHECK, and OPT(2) for quality assurance tests and production. For details, see FAQs about compiling with Enterprise COBOL in the *Enterprise COBOL Migration Guide*.

For all the COBOL migration-related information, visit the [COBOL Migration Portal](#), which includes case studies, COBOL experts interview videos, the cloud-based COBOL Migration Assistant for a navigation through the migration process, COBOL Migration and Performance Tuning Webinars, FAQs, other IBM products to support your migration, and many other resources.

Eliminating repetitive coding

To include stored source statements in a program, use the COPY statement in any program division and at any code sequence level. You can nest COPY statements to any depth.

To specify more than one copy library, use either multiple system definitions or a combination of multiple definitions and the IN/OF phrase (IN/OF *library-name*):

MVS batch

Use JCL to concatenate data sets in 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.

z/OS UNIX

Use the SYSLIB environment variable to define multiple paths to your copybooks. Alternatively, use multiple environment variables 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 cannot itself be a debugging line; if it contains a D, and WITH DEBUGGING mode is not specified, the COPY statement is nevertheless processed.

[“Example: using the COPY statement” on page 772](#)

Related references

[Chapter 20, “Compiler-directing statements,” on page 429](#)

Example: using the COPY statement

These examples show how you can use the COPY statement to include library text in a program.

Suppose the library entry CFILEA consists of the following FD entries:

```
BLOCK CONTAINS 20 RECORDS
RECORD CONTAINS 120 CHARACTERS
LABEL RECORDS ARE STANDARD
DATA RECORD IS FILE-OUT.
01 FILE-OUT          PIC X(120).
```


You can retrieve the text-name CFILEA by using the COPY statement in a source program as follows:

```
FD FILEA
   COPY CFILEA.
```

The library entry is copied into your program, and the resulting program listing looks like this:

```
FD FILEA
   COPY CFILEA.
C   BLOCK CONTAINS 20 RECORDS
C   RECORD CONTAINS 120 CHARACTERS
C   LABEL RECORDS ARE STANDARD
C   DATA RECORD IS FILE-OUT.
C   01 FILE-OUT PIC X(120).
```

In the compiler source listing, the COPY statement prints on a separate line. C precedes copied lines.

Assume that a copybook with the text-name DOWORK is stored by using the following statements:

```
COMPUTE QTY-ON-HAND = TOTAL-USED-NUMBER-ON-HAND
MOVE QTY-ON-HAND to PRINT-AREA
```

To retrieve the copybook identified as DOWORK, code:

```
paragraph-name.
  COPY DOWORK.
```

The statements that are in the DOWORK procedure will follow *paragraph-name*.

If you use the EXIT compiler option to provide a LIBEXIT module, your results might differ from those shown here.

Note: To save compile time, you might group related items in a copybook, but not necessarily have a single large copybook with unrelated items in it.

Related tasks

[“Eliminating repetitive coding” on page 772](#)

Related references

[Chapter 20, “Compiler-directing statements,” on page 429](#)

Using Language Environment callable services

Language Environment callable services make many types of programming tasks easier. You call them by using the CALL statement.

Language Environment services help you with the following tasks:

- Handling conditions

The Language Environment condition-handling facilities enable COBOL applications to react to unexpected errors. You can use language constructs or runtime options to select the level at which to handle each condition. For example, you can 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. You can also create your own storage pools.

- Calculating dates and times

If you use 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 09 for 2009.

- 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 a few exceptions. You should be familiar with these differences before deciding which to use.

- Handling messages

Message-handling services include services for 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 that application users want. 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 09 or as 6,23,09.

- General services such as starting z/OS Debugger and obtaining a Language Environment formatted dump

z/OS Debugger provides advanced debugging functions for COBOL applications, including both batch and interactive debugging of CICS programs. z/OS Debugger enables you to debug a COBOL application from the host or, in conjunction with the Debug Perspective of IBM Developer for z/OS, from a workstation.

Depending on the options that you select, the Language Environment formatted dump might contain the names and values of data items, and information about conditions, program tracebacks, control blocks, storage, and files. All Language Environment dumps have a common, well-labeled, easy-to-read format.

[“Example: Language Environment callable services” on page 776](#)

Related concepts

[“Sample list of Language Environment callable services” on page 774](#)

Related tasks

[“Using numeric intrinsic functions” on page 56](#)

[“Using math-oriented callable services” on page 58](#)

[“Using date callable services” on page 59](#)

[“Calling Language Environment services” on page 775](#)

[“Using procedure and function pointers” on page 545](#)

Sample list of Language Environment callable services

The following table shows some examples of the callable services that are available with Language Environment. Many more services are available than those listed.

Table 101. Language Environment callable services

Function type	Service	Purpose
Condition handling	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 finished
Dynamic storage	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
Math	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	CEEMOUT	To dispatch a message
	CEEMGET	To retrieve a message
National language support	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	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 service. Define the variables for the CALL statement in the DATA DIVISION with the definitions that are required by that service.

```

77 argument          comp-1.
77 feedback-code    pic x(12) display.
77 result           comp-1.

CALL "CEESSQT" using argument, feedback-code, result

```

In the example above, Language Environment service CEESSQT calculates the value of the square root of the variable `argument` and returns this value in the variable `result`.

You can choose whether to specify the feedback code parameter. If you specify it, 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 let the default Language Environment processing for unhandled conditions occur. In either case, you avoid having to write logic to check the feedback code explicitly after each call.

If 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 776](#)

Related concepts

Language Environment Programming Guide (General callable services)

Related references

Language Environment Programming Reference (General callable services)

CALL statement (*Enterprise COBOL for z/OS Language Reference*)

Example: Language Environment callable services

This example shows a COBOL program that uses the 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 coding that would be required without Language Environment.

```

ID DIVISION.
PROGRAM-ID. HOHOHO.
*****
* FUNCTION:  DISPLAY TODAY'S DATE IN THE FOLLOWING FORMAT: *
*           WWWWWWWWWW, MMMMMMMM DD, YYYY                *
*           For example: TUESDAY, SEPTEMBER 15, 2009      *
*           *****                                       *
*****
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 LANGUAGE ENVIRONMENT 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 " WWWWWWWWWWZ, MMMMMMMMMZ DD, YYYY " TO PICSTR-STRING.
MOVE 50 TO PICSTR-LENGTH.
CALL "CEEDATE" USING LILIAN , PICSTR , FORMATTED-DATE ,
OMITTED.
*
DISPLAY "*****".
DISPLAY FORMATTED-DATE.
DISPLAY "*****".

```

*

STOP RUN.

Using the format 2 SORT statement to sort a table

It is recommended to use the format 2 SORT statement to sort a table. It provides the following benefits when compared to the format 1 SORT statement.

Characteristics	Format 1 SORT statements	Format 2 SORT statements
Can be used to sort a file or a table	Yes	No, it is for tables only
Requires DFSORT or equivalent sorting program	Yes	No
Supported in CICS	Limited	Yes
Supported in UNIX System Services	No	Yes
Supported in programs that are compiled with the THREAD option	No	Yes
Table can be sorted by using a single SORT statement, which simplifies coding	No, it requires the SELECT clauses, SD entries with record descriptions, and input and output procedures	Yes
Keys for sorting can be specified as part of the table definition, which can also be used in the SEARCH ALL statement	No, keys must be specified in the SORT statement. If the table is to be searched by using SEARCH ALL as well, the keys must also be redundantly specified as part of the table definition.	Yes, and it also supports specifying keys in the SORT statement if needed
Can filter or preprocess table elements during the sorting process	Yes, using input and output procedures	No, all of the table elements are passed to SORT as-is
Uses special registers that include SORT-CONTROL, SORT-CORE-SIZE, SORT-FILE-SIZE, SORT-MESSAGE, SORT-MODE-SIZE, and SORT-RETURN	Yes	No
Can be executed within the range of an input or output procedure	No	Yes

Note: Do not use the format 2 SORT with large tables in an environment where storage is constrained, because the format 2 SORT uses heap storage to do the sort.

Related references

SORT statement (*Enterprise COBOL for z/OS Language Reference*)

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 to reserve.

Intermediate results are possible in the following cases:

- In an ADD or SUBTRACT statement that contains more than one operand immediately after the verb
- In a COMPUTE statement that specifies a series of arithmetic operations or multiple result fields
- In an arithmetic expression contained in a conditional statement or in a reference-modification specification
- In an ADD, SUBTRACT, MULTIPLY, or DIVIDE statement that uses the GIVING option and multiple result fields
- In a statement that uses an intrinsic function as an operand
- In a statement that contains the ROUNDED phrase

[“Example: calculation of intermediate results” on page 780](#)

The precision of intermediate results depends on whether you compile using the default option ARITH(COMPAT) (referred to as *compatibility mode*) or using ARITH(EXTEND) (referred to as *extended mode*).

In compatibility mode, evaluation of arithmetic operations is unchanged from that in releases of IBM COBOL before COBOL for OS/390 & VM 2.2:

- A maximum of 30 digits is used for fixed-point intermediate results.
- Floating-point intrinsic functions return long-precision (64-bit) floating-point results.
- Expressions that contain floating-point operands, fractional exponents, or floating-point intrinsic functions are evaluated as if all operands that are not in floating point are converted to long-precision floating point and floating-point operations are used to evaluate the expression.
- Floating-point literals and external floating-point data items are converted to long-precision floating point for processing.

In extended mode, evaluation of arithmetic operations has the following characteristics:

- A maximum of 31 digits is used for fixed-point intermediate results.
- Floating-point intrinsic functions return extended-precision (128-bit) floating-point results.
- Expressions that contain floating-point operands, fractional exponents, or floating-point intrinsic functions are evaluated as if all operands that are not in floating point are converted to extended-precision floating point and floating-point operations are used to evaluate the expression.
- Floating-point literals and external floating-point data items are converted to extended-precision floating point for processing.

Related concepts

[“Formats for numeric data” on page 47](#)

[“Fixed-point contrasted with floating-point arithmetic” on page 62](#)

Related references

[“Fixed-point data and intermediate results” on page 781](#)

[“Floating-point data and intermediate results” on page 786](#)

[“Arithmetic expressions in nonarithmetic statements” on page 787](#)

[“ARITH” on page 339](#)

Terminology used for intermediate results

To understand this information about intermediate results, you need to understand the following terminology.

i

The number of integer places carried for an intermediate result. (If you use the **ROUNDED** phrase, one more integer place might be carried for accuracy if necessary.)

d

The number of decimal places carried for an intermediate result. (If you use the **ROUNDED** phrase, one more decimal place might be carried for accuracy if necessary.)

dmax

In a particular statement, the largest of the following items:

- The number of decimal places needed for the final result field or fields
- The maximum number of decimal places defined for any operand, except divisors or exponents
- The *outer-dmax* for any function operand

inner-dmax

In reference to a function, the largest of the following items:

- The number of decimal places defined for any of its elementary arguments
- The *dmax* for any of its arithmetic expression arguments
- The *outer-dmax* for any of its embedded functions

outer-dmax

The number of decimal places that a function result contributes to operations outside of its own evaluation (for example, if the function is an operand in an arithmetic expression, or an argument to another function).

op1

The first operand in a generated arithmetic statement (in division, the divisor).

op2

The second operand in a generated arithmetic statement (in division, the dividend).

i1 , i2

The number of integer places in *op1* and *op2*, respectively.

d1 , d2

The number of decimal places in *op1* and *op2*, respectively.

ir

The intermediate result when a generated arithmetic statement or operation is performed. (Intermediate results are generated either in registers or storage locations.)

ir1 , ir2

Successive intermediate results. (Successive intermediate results might have the same storage location.)

Related references

ROUNDED phrase (*Enterprise COBOL for z/OS Language Reference*)

Example: calculation of intermediate results

The following example shows how the compiler performs an arithmetic statement as a succession of operations, storing intermediate results as needed.

```
COMPUTE Y = A + B * C - D / E + F ** G
```

The result is calculated in the following order:

1. Exponentiate F by G yielding $ir1$.
2. Multiply B by C yielding $ir2$.
3. Divide E into D yielding $ir3$.
4. Add A to $ir2$ yielding $ir4$.
5. Subtract $ir3$ from $ir4$ yielding $ir5$.
6. Add $ir5$ to $ir1$ yielding Y.

Related tasks

[“Using arithmetic expressions” on page 56](#)

Related references

[“Terminology used for intermediate results” on page 780](#)

Fixed-point data and intermediate results

The compiler determines the number of integer and decimal places in an intermediate result.

Addition, subtraction, multiplication, and division

The following table shows the precision theoretically possible as the result of addition, subtraction, multiplication, or division.

Operation	Integer places	Decimal places
+ or -	$(i1 \text{ or } i2) + 1$, whichever is greater	$d1$ or $d2$, whichever is greater
*	$i1 + i2$	$d1 + d2$
/	$i2 + d1$	$(d2 - d1)$ or $dmax$, whichever is greater

You must define the operands of any arithmetic statements with enough decimal places to obtain the accuracy you want in the final result.

The following table shows the number of places the compiler carries for fixed-point intermediate results of arithmetic operations that involve addition, subtraction, multiplication, or division in *compatibility mode* (that is, when the default compiler option ARITH (COMPAT) is in effect):

Value of $i + d$	Value of d	Value of $i + dmax$	Number of places carried for ir
<30 or =30	Any value	Any value	i integer and d decimal places
>30	< $dmax$ or = $dmax$	Any value	$30 - d$ integer and d decimal places
		<30 or =30	i integer and $30 - i$ decimal places
	> $dmax$	>30	$30 - dmax$ integer and $dmax$ decimal places

The following table shows the number of places the compiler carries for fixed-point intermediate results of arithmetic operations that involve addition, subtraction, multiplication, or division in *extended mode* (that is, when the compiler option ARITH (EXTEND) is in effect):

Value of $i + d$	Value of d	Value of $i + dmax$	Number of places carried for ir
<31 or =31	Any value	Any value	i integer and d decimal places

Value of $i + d$	Value of d	Value of $i + d_{max}$	Number of places carried for ir
>31	< d_{max} or = d_{max}	Any value	31- d integer and d decimal places
	> d_{max}	<31 or =31	i integer and 31- i decimal places
		>31	31- d_{max} integer and d_{max} decimal places

Exponentiation

Exponentiation is represented by the expression $op1 ** op2$. Based on the characteristics of $op2$, the compiler handles exponentiation of fixed-point numbers in one of three ways:

- When $op2$ is expressed with decimals, floating-point instructions are used.
- When $op2$ is an integral literal or constant, the value d is computed as

$$d = d1 * |op2|$$

and the value i is computed based on the characteristics of $op1$:

- When $op1$ is a data-name or variable,

$$i = i1 * |op2|$$

- When $op1$ is a literal or constant, i is set equal to the number of integers in the value of $op1 ** |op2|$.

In compatibility mode (compilation using ARITH(COMPAT)), the compiler having calculated i and d takes the action indicated in the table below to handle the intermediate results ir of the exponentiation.

Value of $i + d$	Other conditions	Action taken
<30	Any	i integer and d decimal places are carried for ir .
=30	$op1$ has an odd number of digits.	i integer and d decimal places are carried for ir .
	$op1$ has an even number of digits.	Same action as when $op2$ is an integral data-name or variable (shown below). Exception: for a 30-digit integer raised to the power of literal 1, i integer and d decimal places are carried for ir .
>30	Any	Same action as when $op2$ is an integral data-name or variable (shown below)

In extended mode (compilation using ARITH(EXTEND)), the compiler having calculated i and d takes the action indicated in the table below to handle the intermediate results ir of the exponentiation.

Value of $i + d$	Other conditions	Action taken
<31	Any	i integer and d decimal places are carried for ir .
=31 or >31	Any	Same action as when $op2$ is an integral data-name or variable (shown below). Exception: for a 31-digit integer raised to the power of literal 1, i integer and d decimal places are carried for ir .

If $op2$ is negative, the value of 1 is then divided by the result produced by the preliminary computation. The values of i and d that are used are calculated following the division rules for fixed-point data already shown above.

- When *op2* is an integral data-name or variable, *dmax* decimal places and 30-*dmax* (compatibility mode) or 31-*dmax* (extended mode) integer places are used. *op1* is multiplied by itself ($|op2| - 1$) times for nonzero *op2*.

If *op2* is equal to 0, the result is 1. Division-by-0 and exponentiation SIZE ERROR conditions apply.

Fixed-point exponents with more than nine significant digits are always truncated to nine digits. If the exponent is a literal or constant, an E-level compiler diagnostic message is issued; otherwise, an informational message is issued at run time.

[“Example: exponentiation in fixed-point arithmetic” on page 783](#)

Related references

[“Terminology used for intermediate results” on page 780](#)

[“Truncated intermediate results” on page 783](#)

[“Binary data and intermediate results” on page 784](#)

[“Floating-point data and intermediate results” on page 786](#)

[“Intrinsic functions evaluated in fixed-point arithmetic” on page 784](#)

[“ARITH” on page 339](#)

SIZE ERROR phrases (*Enterprise COBOL for z/OS Language Reference*)

Example: exponentiation in fixed-point arithmetic

The following example shows how the compiler performs an exponentiation to a nonzero integer power as a succession of multiplications, storing intermediate results as needed.

```
COMPUTE Y = A ** B
```

If B is equal to 4, the result is computed as shown below. The values of *i* and *d* that are used are calculated according to the multiplication rules for fixed-point data and intermediate results (referred to below).

1. Multiply A by A yielding an internal intermediate result *iir1*.
2. Multiply *iir1* by A yielding an internal intermediate result *iir2*.
3. Multiply *iir2* by A yielding an internal intermediate result *iir3*.
4. Move *iir3* to *ir4*.

ir4 has *dmax* decimal places. Because B is positive, *ir4* is moved to Y. If B were equal to -4, however, an additional fifth step would be performed:

5. Divide *ir4* into 1 yielding *ir5*.

ir5 has *dmax* decimal places, and would then be moved to Y.

Note: The internal intermediate results (*iir1*, *iir2*, and *iir3*) obtained by the internal library routine performing the exponential calculation in steps 1, 2, and 3 above do not use the same decimal precision as *ir4* and *ir5* above. Instead, those intermediate results are much more precise, ensuring the most accurate result possible in *ir4* or *ir5*.

Related references

[“Terminology used for intermediate results” on page 780](#)

[“Fixed-point data and intermediate results” on page 781](#)

Truncated intermediate results

Whenever the number of digits in an intermediate result exceeds 30 in compatibility mode or 31 in extended mode, the compiler truncates to 30 (compatibility mode) or 31 (extended mode) digits and

issues a warning. If truncation occurs at run time, a message is issued and the program continues running.

If you want to avoid the truncation of intermediate results that can occur in fixed-point calculations, use floating-point operands (COMP-1 or COMP-2) instead.

Related concepts

[“Formats for numeric data” on page 47](#)

Related references

[“Fixed-point data and intermediate results” on page 781](#)

[“ARITH” on page 339](#)

Binary data and intermediate results

If an operation that involves binary operands requires intermediate results longer than 18 digits, the compiler converts the operands to internal decimal before performing the operation. If the result field is binary, the compiler converts the result from internal decimal to binary.

Binary operands are most efficient when intermediate results will not exceed nine digits.

Related references

[“Fixed-point data and intermediate results” on page 781](#)

[“ARITH” on page 339](#)

Intrinsic functions evaluated in fixed-point arithmetic

The compiler determines the *inner-dmax* and *outer-dmax* values for an intrinsic function from the characteristics of the function.

Integer functions

Integer intrinsic functions return an integer; thus their *outer-dmax* is always zero. For those integer functions whose arguments must all be integers, the *inner-dmax* is thus also always zero.

The following table summarizes the *inner-dmax* and the precision of the function result.

Function	Inner-dmax	Digit precision of function result
DATE-OF-INTEG	0	8
DATE-TO-YYYYMMDD	0	8
DAY-OF-INTEG	0	7
DAY-TO-YYYYDDD	0	7
FACTORIAL	0	30 in compatibility mode, 31 in extended mode
INTEGER-OF-DATE	0	7
INTEGER-OF-DAY	0	7
LENGTH	n/a	9
MOD	0	min(<i>i1 i2</i>)
ORD	n/a	3
ORD-MAX		9
ORD-MIN		9
YEAR-TO-YYYY	0	4

Function	Inner-dmax	Digit precision of function result
INTEGER		For a fixed-point argument: one more digit than in the argument. For a floating-point argument: 30 in compatibility mode, 31 in extended mode.
INTEGER - PART		For a fixed-point argument: same number of digits as in the argument. For a floating-point argument: 30 in compatibility mode, 31 in extended mode.

Mixed functions

A *mixed* intrinsic function is a function whose result type depends on the type of its arguments. A mixed function is fixed point if all of its arguments are numeric and none of its arguments is floating point. (If any argument of a mixed function is floating point, the function is evaluated with floating-point instructions and returns a floating-point result.) When a mixed function is evaluated with fixed-point arithmetic, the result is integer if all of the arguments are integer; otherwise, the result is fixed point.

For the mixed functions MAX, MIN, RANGE, REM, and SUM, the *outer-dmax* is always equal to the *inner-dmax* (and both are thus zero if all the arguments are integer). To determine the precision of the result returned for these functions, apply the rules for fixed-point arithmetic and intermediate results (as referred to below) to each step in the algorithm.

MAX

1. Assign the first argument to the function result.
2. For each remaining argument, do the following steps:
 - a. Compare the algebraic value of the function result with the argument.
 - b. Assign the greater of the two to the function result.

MIN

1. Assign the first argument to the function result.
2. For each remaining argument, do the following steps:
 - a. Compare the algebraic value of the function result with the argument.
 - b. Assign the lesser of the two to the function result.

RANGE

1. Use the steps for MAX to select the maximum argument.
2. Use the steps for MIN to select the minimum argument.
3. Subtract the minimum argument from the maximum.
4. Assign the difference to the function result.

REM

1. Divide argument one by argument two.
2. Remove all noninteger digits from the result of step 1.
3. Multiply the result of step 2 by argument two.
4. Subtract the result of step 3 from argument one.
5. Assign the difference to the function result.

SUM

1. Assign the value 0 to the function result.
2. For each argument, do the following steps:
 - a. Add the argument to the function result.
 - b. Assign the sum to the function result.

Related references

[“Terminology used for intermediate results” on page 780](#)

[“Fixed-point data and intermediate results” on page 781](#)

[“Floating-point data and intermediate results” on page 786](#)

[“ARITH” on page 339](#)

Floating-point data and intermediate results

If any operation in an arithmetic expression is computed in floating-point arithmetic, the entire expression is computed as if all operands were converted to floating point and the operations were performed using floating-point instructions.

Floating-point instructions are used to compute an arithmetic expression if any of the following conditions is true of the expression:

- A receiver or operand is COMP-1, COMP-2, external floating point, or a floating-point literal.
- An exponent contains decimal places.
- An exponent is an expression that contains an exponentiation or division operator, and *dmax* is greater than zero.
- An intrinsic function is a floating-point function.

In compatibility mode, if an expression is computed in floating-point arithmetic, the precision used to evaluate the arithmetic operations is determined as follows:

- Single precision is used if all receivers and operands are COMP-1 data items and the expression contains no multiplication or exponentiation operations.
- In all other cases, long precision is used.

Whenever long-precision floating point is used for one operation in an arithmetic expression, all operations in the expression are computed as if long floating-point instructions were used.

In extended mode, if an expression is computed in floating-point arithmetic, the precision used to evaluate the arithmetic operations is determined as follows:

- Single precision is used if all receivers and operands are COMP-1 data items and the expression contains no multiplication or exponentiation operations.
- Long precision is used if all receivers and operands are COMP-1 or COMP-2 data items, at least one receiver or operand is a COMP-2 data item, and the expression contains no multiplication or exponentiation operations.
- In all other cases, extended precision is used.

Whenever extended-precision floating point is used for one operation in an arithmetic expression, all operations in the expression are computed as if extended-precision floating-point instructions were used.

Alert: If a floating-point operation has an intermediate result field in which exponent overflow occurs, the job is abnormally terminated.

Exponentiations evaluated in floating-point arithmetic

In compatibility mode, floating-point exponentiations are always evaluated using long floating-point arithmetic. In extended mode, floating-point exponentiations are always evaluated using extended-precision floating-point arithmetic.

The value of a negative number raised to a fractional power is undefined in COBOL. For example, $(-2)^{**} 3$ is equal to -8, but $(-2)^{**} (3.000001)$ is undefined. When an exponentiation is evaluated in floating point and there is a possibility that the result is undefined, the exponent is evaluated at run time to determine if it has an integral value. If not, a diagnostic message is issued.

Intrinsic functions evaluated in floating-point arithmetic

In compatibility mode, floating-point intrinsic functions always return a long (64-bit) floating-point value. In extended mode, floating-point intrinsic functions always return an extended-precision (128-bit) floating-point value.

Mixed functions that have at least one floating-point argument are evaluated using floating-point arithmetic.

Related references

[“Terminology used for intermediate results” on page 780](#)

[“ARITH” on page 339](#)

Arithmetic expressions in nonarithmetic statements

Arithmetic expressions can appear in contexts other than arithmetic statements. For example, you can use an arithmetic expression with the IF or EVALUATE statement.

In such statements, the rules for intermediate results with fixed-point data and for intermediate results with floating-point data apply, with the following changes:

- Abbreviated IF statements are handled as though the statements were not abbreviated.
- In an explicit relation condition where at least one of the comparands is an arithmetic expression, *dmax* is the maximum number of decimal places for any operand of either comparand, excluding divisors and exponents. The rules for floating-point arithmetic apply if any of the following conditions is true:
 - Any operand in either comparand is COMP-1, COMP-2, external floating point, or a floating-point literal.
 - An exponent contains decimal places.
 - An exponent is an expression that contains an exponentiation or division operator, and *dmax* is greater than zero.

For example:

```
IF operand-1 = expression-1 THEN . . .
```

If *operand-1* is a data-name defined to be COMP-2, the rules for floating-point arithmetic apply to *expression-1* even if it contains only fixed-point operands, because it is being compared to a floating-point operand.

- When the comparison between an arithmetic expression and another data item or arithmetic expression does not use a relational operator (that is, there is no explicit relation condition), the arithmetic expression is evaluated without regard to the attributes of its comparand. For example:

```
EVALUATE expression-1  
  WHEN expression-2 THRU expression-3  
  WHEN expression-4  
END-EVALUATE
```

In the statement above, each arithmetic expression is evaluated in fixed-point or floating-point arithmetic based on its own characteristics.

Related concepts

[“Fixed-point contrasted with floating-point arithmetic” on page 62](#)

Related references

[“Terminology used for intermediate results” on page 780](#)

[“Fixed-point data and intermediate results” on page 781](#)

[“Floating-point data and intermediate results” on page 786](#)

IF statement (*Enterprise COBOL for z/OS Language Reference*)

EVALUATE statement (*Enterprise COBOL for z/OS Language Reference*)
Conditional expressions (*Enterprise COBOL for z/OS Language Reference*)

Appendix B. Converting double-byte character set (DBCS) data

The Language Environment service routines IGZCA2D and IGZCD2A were intended for converting alphanumeric data items that contain DBCS data to and from pure DBCS data items in order to reliably perform operations such as STRING, UNSTRING, and reference modification.

These service routines continue to be provided for compatibility; however, using national data items and the national conversion operations is now recommended instead for this purpose.

The service routines do not support a code-page argument and are not sensitive to the code page specified by the CODEPAGE compiler option. The DBCS compiler option does not affect their operation.

Related tasks

[“Converting to or from national \(Unicode\) representation” on page 134](#)

[“Processing alphanumeric data items that contain DBCS data” on page 160](#)

Related references

[“DBCS notation” on page 789](#)

[“Alphanumeric to DBCS data conversion \(IGZCA2D\)” on page 789](#)

[“DBCS to alphanumeric data conversion \(IGZCD2A\)” on page 791](#)

[“CODEPAGE” on page 343](#)

DBCS notation

The symbols shown below are used in the DBCS data conversion examples to 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 that correspond to single-byte EBCDIC characters
.A, .B, .C, . . .	Any double-byte EBCDIC character that corresponds to a single-byte EBCDIC character. The period (.) represents the value X'42'.
A single letter, such as A, B, or s	Any single-byte EBCDIC character

Alphanumeric to DBCS data conversion (IGZCA2D)

The Language Environment IGZCA2D service routine converts alphanumeric 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 by using the CALL statement:

parameter-1

The sending field for the conversion, handled as an alphanumeric 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 in the CALL statement to the conversion routine. 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 (nonspace) EBCDIC character X'*nn*' to a character represented by X'42*nn*'
 - Converts the single-byte 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 for the contents of *parameter-4* is 1 to 134,217,727.

[“Example: IGZCA2D” on page 791](#)

Related references

[“IGZCA2D return codes” on page 790](#)

IGZCA2D return codes

IGZCA2D sets the RETURN-CODE special register to reflect the status of the conversion.

Return code	Explanation
0	<i>parameter-1</i> was converted and the results were placed in <i>parameter-2</i> .
2	<i>parameter-1</i> was converted and the results were placed in <i>parameter-2</i> . <i>parameter-2</i> was padded on the right with DBCS spaces.
4	<i>parameter-1</i> was converted and the results were placed in <i>parameter-2</i> . The DBCS data placed in <i>parameter-2</i> was truncated on the right.
6	<i>parameter-1</i> was converted and the results were placed in <i>parameter-2</i> . A single-byte character in the range X'00' to X'3F' or X'FF' was encountered. The valid single-byte character was converted into an out-of-range DBCS character.
8	<i>parameter-1</i> was converted and the results were placed in <i>parameter-2</i> . A single-byte character in the range X'00' to X'3F' or X'FF' was encountered. The valid single-byte character was converted into an out-of-range DBCS character. <i>parameter-2</i> was padded on the right with DBCS spaces.

Table 103. **IGZCA2D return codes** (continued)

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

Example: IGZCA2D

This example CALL statement converts the alphanumeric data in *alpha-item* to DBCS data. The results of the conversion are placed in *dbcs-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 *dbcs-item* and the lengths before the conversion are:

```
alpha-item = AB<D1D2D3>CD
dbcs-item = D4D5D6D7D8D9D0
```

```
LENGTH OF alpha-item = 12
LENGTH OF dbc-item = 14
```

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

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

The content of the RETURN-CODE register is 0.

Related references

[“DBCS notation” on page 789](#)

DBCS to alphanumeric data conversion (IGZCD2A)

The Language Environment IGZCD2A routine converts pure DBCS data to alphanumeric 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 an alphanumeric 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 do not correspond to single-byte EBCDIC characters
 - Converts DBCS characters to single-byte characters when the DBCS characters correspond to single-byte EBCDIC characters
 - Converts the DBCS space (X'4040') to a single-byte 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 for the contents of *parameter-4* is 1 to 134,217,727.

[“Example: IGZCD2A” on page 793](#)

Related references

[“IGZCD2A return codes” on page 792](#)

IGZCD2A return codes

IGZCD2A sets the RETURN-CODE special register to reflect the status of the conversion.

<i>Table 104. IGZCD2A return codes</i>	
Return code	Explanation
0	<i>parameter-1</i> was converted and the results were placed in <i>parameter-2</i> .
2	<i>parameter-1</i> was converted and the results were placed in <i>parameter-2</i> . <i>parameter-2</i> was padded on the right with single-byte spaces.
4	<i>parameter-1</i> was converted and the results were placed in <i>parameter-2</i> . <i>parameter-2</i> was truncated on the right. ¹
14	<i>parameter-1</i> and <i>parameter-2</i> were overlapping. No conversion occurred.
15	The value of <i>parameter-3</i> or <i>parameter-4</i> was out of range. No conversion occurred.
16	An odd number of bytes was coded in <i>parameter-3</i> . No conversion occurred.

Table 104. **IGZCD2A return codes** (continued)

Return code	Explanation
	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 alphanumeric data is padded with a single-byte space after the shift-in.

Example: IGZCD2A

This example CALL statement converts the DBCS data in `dbcs-item` to alphanumeric 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 789](#)

Appendix C. XML reference material

The following information describes the XML exception codes that might be returned during XML parsing or XML generation.

Related references

[“XML PARSE exceptions with XMLPARSE\(XMLSS\) in effect” on page 795](#)

[“XML PARSE exceptions with XMLPARSE\(COMPAT\) in effect” on page 797](#)

[“XML GENERATE exceptions” on page 803](#)
[XML specification](#)

XML PARSE exceptions with XMLPARSE(XMLSS) in effect

When the z/OS XML System Services parser passes control to your processing procedure for an exception event, the XML - CODE special register contains the exception code, which is formed from a return code and a reason code.

The return code and reason code are each a halfword binary value. The exception code is the concatenation of those two values: the return code in the high-order halfword, and the reason code in the low-order halfword.

The return codes and reason codes are documented as hexadecimal values in the *z/OS XML System Services User's Guide and Reference*, referenced below, and in [Table 105 on page 795](#) below.

After most exception events, the parser does not continue processing; the value in XML - CODE at the end of the XML PARSE statement is the original exception code set by the parser.

When the processing procedure returns to the parser after the exception event, control transfers to the statement specified in the ON EXCEPTION phrase, or to the end of the XML PARSE statement if you did not code an ON EXCEPTION phrase.

Validation exceptions:

If you code an XML PARSE statement that contains the VALIDATING phrase, and the z/OS XML System Services parser determines that the document is not valid, the parser generates return code 24 (hexadecimal 18, XRC_NOT_VALID).

Exceptions that are unique to Enterprise COBOL:

Some exceptions are unique to Enterprise COBOL and thus are not documented in the *z/OS XML System Services User's Guide and Reference*, for example, errors that occur during XML schema retrieval. The return code for exceptions with reason codes in the hexadecimal range 800 to 899 is 4 (hexadecimal 0004, XRC_WARNING). For other exceptions, the return code is 16 (hexadecimal 0010, XRC_FATAL). The exception code (the value in special register XML - CODE), is formed from this return code concatenated with one of the reason codes shown in the following table.

Reason code (hexadecimal)	Description
700	VALIDATING WITH FILE is not supported under CICS.
701	The optimized XML schema that was read in was too short, or the file was empty.
702	The file identifier for the schema was not a ddname or environment-variable name.
703	The DSN value contained a space character in a position where a space is not allowed.
704	The DSN value specified a temporary data set.

Table 105. Reason codes for XML PARSE exceptions that are unique to Enterprise COBOL (continued)

Reason code (hexadecimal)	Description
705	The PATH value contained an unescaped space character.
706	The PATH value contained a path name that was not an absolute path.
707	Memory allocation for the XML schema buffer failed.
708	The environment variable was null or contained only spaces.
709	The environment variable contained an invalid keyword.
710	The DSN value contained an invalid character after the member name.
711	The DSN value did not specify a member name.
712	The DSN value did not specify a data set name, or parentheses were not specified correctly.
713	The PATH value did not specify a path name, or parentheses were not specified correctly.
714	The DSN value contained an extra parenthesis.
715	The PATH value contained an extra parenthesis.
716	The DSN value was missing the closing parenthesis.
717	The PATH value was missing the closing parenthesis.
718	The DSN value contained an escape character.
720	A character reference for an unrepresentable character was not resolved.
721	An unrepresentable character reference in the document type declaration is not supported.
800	The attribute name used an undeclared prefix.
801	The START-OF-ELEMENT name used an undeclared prefix. (The END-OF-ELEMENT name must match, so using the same undeclared prefix does not cause another exception.)
900	Internal error. Report the error to your service representative.

For any of the reason codes except 900, correct the error and then retry your program.

Related concepts

[“XML-CODE” on page 618](#)

[“XML events” on page 617](#)

Related tasks

[“Handling XML PARSE exceptions” on page 633](#)

Related references

[“XMLPARSE” on page 415 \(compiler option\)](#)

XML PARSE statement (*Enterprise COBOL for z/OS Language Reference*)
z/OS XML System Services User's Guide and Reference

XML PARSE exceptions with XMLPARSE(COMPAT) in effect

When an exception event occurs, the XML parser that is provided with the Enterprise COBOL library sets special register XML - CODE to a value that identifies the exception. Depending on the value in XML - CODE, the parser might or might not be able to continue processing after the exception, as detailed in the information referenced below.

Related references

[“XML PARSE exceptions that allow continuation” on page 797](#)

[“XML PARSE exceptions that do not allow continuation” on page 800](#)

XML PARSE exceptions that allow continuation

If the XMLPARSE (COMPAT) compiler option is in effect, whether the XML parser can continue processing after an exception event depends upon the value of the exception code.

The parser can continue processing if the exception code, which is in special register XML - CODE, is within one of the following ranges:

- 1 - 99
- 100,001 - 165,535

The following table describes each exception, and identifies the actions that the parser takes if you request that it continue after the exception. Some of the descriptions use the following terms:

- *Actual document encoding*
- *Document encoding declaration*

For definitions of the terms, see the related concept about XML input document encoding.

Exception code (decimal)	Description	Parser action on continuation
1	The parser found an invalid character while scanning white space outside element content. For further information about white space, see the related concept about XML input document encoding.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
2	The parser found an invalid start of a processing instruction, element, comment, or document type declaration outside element content.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
3	The parser found a duplicate attribute name.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
4	The parser found the markup character '<' in an attribute value.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.

Table 106. XML PARSE exceptions that allow continuation (continued)

Exception code (decimal)	Description	Parser action on continuation
5	The start and end tag names of an element did not match.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
6	The parser found an invalid character in element content.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
7	The parser found an invalid start of an element, comment, processing instruction, or CDATA section in element content.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
8	The parser found in element content the CDATA closing character sequence ']]>' without the matching opening character sequence '<![CDATA['.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
9	The parser found an invalid character in a comment.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
10	The parser found in a comment the character sequence '- -' (two hyphens) not followed by '>'.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
11	The parser found an invalid character in a processing instruction data segment.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
12	The XML declaration was not at the beginning of the document.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
13	The parser found an invalid digit in a hexadecimal character reference (of the form <code>&#xddd;</code>).	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.

Table 106. XML PARSE exceptions that allow continuation (continued)

Exception code (decimal)	Description	Parser action on continuation
14	The parser found an invalid digit in a decimal character reference (of the form &#dxxx;).	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
15	The encoding declaration value in the XML declaration did not begin with lowercase or uppercase A through Z.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
16	A character reference did not refer to a legal XML character.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
17	The parser found an invalid character in an entity reference name.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
18	The parser found an invalid character in an attribute value.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
70	The actual document encoding was EBCDIC, and the CODEPAGE compiler option specified a supported EBCDIC code page, but the document encoding declaration did not specify a supported EBCDIC code page.	The parser uses the encoding specified by the CODEPAGE compiler option.
71	The actual document encoding was EBCDIC, and the document encoding declaration specified a supported EBCDIC encoding, but the CODEPAGE compiler option did not specify a supported EBCDIC code page.	The parser uses the encoding specified by the document encoding declaration.
72	The actual document encoding was EBCDIC, the CODEPAGE compiler option did not specify a supported EBCDIC code page, and the document did not contain an encoding declaration.	The parser uses EBCDIC code page 1140 (USA, Canada, . . . Euro Country Extended Code Page).

Exception code (decimal)	Description	Parser action on continuation
73	The actual document encoding was EBCDIC, but neither the CODEPAGE compiler option nor the document encoding declaration specified a supported EBCDIC code page.	The parser uses EBCDIC code page 1140 (USA, Canada, . . . Euro Country Extended Code Page).
82	The actual document encoding was ASCII, but the document did not contain an encoding declaration.	The parser uses ASCII code page 819 (ISO-8859-1 Latin 1/Open Systems).
83	The actual document encoding was ASCII, but the document encoding declaration did not specify code page 813, 819, or 920.	The parser uses ASCII code page 819 (ISO-8859-1 Latin 1/Open Systems).
92	The document data item was alphanumeric, but the actual document encoding was Unicode UTF-16.	The parser uses code page 1200 (Unicode UTF-16).
100,001 - 165,535	The CODEPAGE compiler option and the document encoding declaration specified different supported EBCDIC code pages. XML - CODE contains the code page CCSID for the encoding declaration plus 100,000.	If you set XML - CODE to zero before returning from the EXCEPTION event, the parser uses the encoding specified by the CODEPAGE compiler option. If you set XML - CODE to the CCSID for the document encoding declaration (by subtracting 100,000), the parser uses this encoding.

Related concepts

[“XML-CODE” on page 618](#)

[“XML input document encoding” on page 629](#)

Related tasks

[“Handling XML PARSE exceptions” on page 633](#)

Related references

[“XMLPARSE” on page 415 \(compiler option\)](#)

XML PARSE exceptions that do not allow continuation

If the XMLPARSE (COMPAT) compiler option is in effect, the XML parser cannot continue processing if any of the exceptions described below occurs.

No further events are returned from the parser for any of these exceptions even if the processing procedure sets XML - CODE to zero before passing control back to the parser. The parser transfers control to the statement in the ON EXCEPTION phrase, if specified, otherwise to the end of the XML PARSE statement.

Exception code (decimal)	Description
100	The parser reached the end of the document while scanning the start of the XML declaration.

Table 107. XML PARSE exceptions that do not allow continuation (for XMLPARSE (COMPAT)) (continued)

Exception code (decimal)	Description
101	The parser reached the end of the document while looking for the end of the XML declaration.
102	The parser reached the end of the document while looking for the root element.
103	The parser reached the end of the document while looking for the version information in the XML declaration.
104	The parser reached the end of the document while looking for the version information value in the XML declaration.
106	The parser reached the end of the document while looking for the encoding declaration value in the XML declaration.
108	The parser reached the end of the document while looking for the standalone declaration value in the XML declaration.
109	The parser reached the end of the document while scanning an attribute name.
110	The parser reached the end of the document while scanning an attribute value.
111	The parser reached the end of the document while scanning a character reference or entity reference in an attribute value.
112	The parser reached the end of the document while scanning an empty element tag.
113	The parser reached the end of the document while scanning the root element name.
114	The parser reached the end of the document while scanning an element name.
115	The parser reached the end of the document while scanning character data in element content.
116	The parser reached the end of the document while scanning a processing instruction in element content.
117	The parser reached the end of the document while scanning a comment or CDATA section in element content.
118	The parser reached the end of the document while scanning a comment in element content.
119	The parser reached the end of the document while scanning a CDATA section in element content.
120	The parser reached the end of the document while scanning a character reference or entity reference in element content.
121	The parser reached the end of the document while scanning after the close of the root element.
122	The parser found a possible invalid start of a document type declaration.
123	The parser found a second document type declaration.
124	The first character of the root element name was not a letter, '_', or ':'.
125	The first character of the first attribute name of an element was not a letter, '_', or ':'.
126	The parser found an invalid character either in or following an element name.
127	The parser found a character other than '=' following an attribute name.

Table 107. XML PARSE exceptions that do not allow continuation (for XMLPARSE (COMPAT)) (continued)

Exception code (decimal)	Description
128	The parser found an invalid attribute value delimiter.
130	The first character of an attribute name was not a letter, '_', or ':'.
131	The parser found an invalid character either in or following an attribute name.
132	An empty element tag was not terminated by a '>' following the '/'.
133	The first character of an element end tag name was not a letter, '_', or ':'.
134	An element end tag name was not terminated by a '>'.
135	The first character of an element name was not a letter, '_', or ':'.
136	The parser found an invalid start of a comment or CDATA section in element content.
137	The parser found an invalid start of a comment.
138	The first character of a processing instruction target name was not a letter, '_', or ':'.
139	The parser found an invalid character in or following a processing instruction target name.
140	A processing instruction was not terminated by the closing character sequence '?>'.
141	The parser found an invalid character following '&' in a character reference or entity reference.
142	The version information was not present in the XML declaration.
143	'version' in the XML declaration was not followed by '='.
144	The version declaration value in the XML declaration is either missing or improperly delimited.
145	The version information value in the XML declaration specified a bad character, or the start and end delimiters did not match.
146	The parser found an invalid character following the version information value closing delimiter in the XML declaration.
147	The parser found an invalid attribute instead of the optional encoding declaration in the XML declaration.
148	'encoding' in the XML declaration was not followed by '='.
149	The encoding declaration value in the XML declaration is either missing or improperly delimited.
150	The encoding declaration value in the XML declaration specified a bad character, or the start and end delimiters did not match.
151	The parser found an invalid character following the encoding declaration value closing delimiter in the XML declaration.
152	The parser found an invalid attribute instead of the optional standalone declaration in the XML declaration.
153	standalone in the XML declaration was not followed by =.
154	The standalone declaration value in the XML declaration is either missing or improperly delimited.
155	The standalone declaration value was neither 'yes' nor 'no' only.

Table 107. XML PARSE exceptions that do not allow continuation (for XMLPARSE (COMPAT)) (continued)

Exception code (decimal)	Description
156	The standalone declaration value in the XML declaration specified a bad character, or the start and end delimiters did not match.
157	The parser found an invalid character following the standalone declaration value closing delimiter in the XML declaration.
158	The XML declaration was not terminated by the proper character sequence '>', or contained an invalid attribute.
159	The parser found the start of a document type declaration after the end of the root element.
160	The parser found the start of an element after the end of the root element.
315	The actual document encoding was UTF-16 little-endian, which the parser does not support on this platform.
316	The actual document encoding was UCS4, which the parser does not support.
317	The parser cannot determine the document encoding. The document might be damaged.
318	The actual document encoding was UTF-8, which the parser does not support.
320	The document data item was national, but the actual document encoding was EBCDIC.
321	The document data item was national, but the actual document encoding was ASCII.
500 - 599	Internal error. Report the error to your service representative.

Related concepts

[“XML-CODE” on page 618](#)

Related tasks

[“Handling XML PARSE exceptions” on page 633](#)

XML GENERATE exceptions

One of several exception codes might be returned in the XML - CODE special register during XML generation. If one of these exceptions occurs, control is passed to the statement in the ON EXCEPTION phrase, or to the end of the XML GENERATE statement if you did not code an ON EXCEPTION phrase.

Table 108. XML GENERATE exceptions

Exception code (decimal)	Description
400	The receiver was too small to contain the generated XML document. The COUNT IN data item, if specified, contains the count of character positions that were actually generated.
401	A DBCS data-name contained a character that, when converted to Unicode, was not valid in an XML element or attribute name.
402	The first character of a DBCS data-name, when converted to Unicode, was not valid as the first character of an XML element or attribute name.
403	The value of an OCCURS DEPENDING ON variable exceeded 16,777,215.

Table 108. XML GENERATE exceptions (continued)

Exception code (decimal)	Description
410	The CCSID page specified by the CODEPAGE compiler option is not supported for conversion to Unicode.
411	The CCSID specified by the CODEPAGE compiler option is not a supported single-byte EBCDIC CCSID.
414	The CCSID specified for the XML document was invalid or was not supported.
415	The receiver was national, but the encoding specified for the document was not UTF-16.
416	The XML namespace identifier contained invalid XML characters.
417	<p>Element character content or an attribute value contained characters that are illegal in XML content. XML generation has continued, with the element tag name or the attribute name prefixed with 'hex.' and the original data value represented in the document in hexadecimal.</p> <p>Any TYPE IS CONTENT specification is ignored, and the item is treated as an element.</p>
418	Substitution characters were generated by encoding conversion.
419	The XML namespace prefix was invalid.
420	The receiver was alphanumeric and the input included national or DBCS data or names, but the encoding specified for the document was not 1208.
600-699	Internal error. Report the error to your service representative.

Related tasks

[“Handling XML GENERATE exceptions” on page 656](#)

Appendix D. JSON reference material

The following information describes the JSON exception codes that might be returned during JSON parsing or JSON generation.

Related references

[“JSON GENERATE exceptions” on page 805](#)

[“JSON PARSE conditions and associated codes and runtime messages” on page 805](#)

[“Nonexception conditions and corresponding values of JSON-STATUS” on page 806](#)

[“Exception conditions, and corresponding values of JSON-CODE” on page 806](#)

[“Nonexception condition runtime messages” on page 807](#)

[“Exception condition runtime messages” on page 808](#)

JSON GENERATE exceptions

One of several exception codes might be returned in the JSON-CODE special register during JSON generation. If one of these exceptions occurs, control is passed to the statement in the ON EXCEPTION phrase, or to the end of the JSON GENERATE statement if you did not code an ON EXCEPTION phrase.

Exception code (decimal)	Description
1	The receiver was too small to contain the generated JSON text. The COUNT IN data item, if specified, contains the count of character positions that were actually generated.
2	The CCSID of the ENCODING phrase in the JSON GENERATE statement was invalid or unsupported.
504	The storage is insufficient for the JSON GENERATE work area. Increase the region size and rerun the application.
507	Internal error. The LE runtime level might not be up-to-date. Install the latest LE runtime PTFs. If the error persists, report the error to your service representative.
500 - 599 (excluding exception codes defined above)	Internal error. Report the error to your service representative.

JSON PARSE conditions and associated codes and runtime messages

Two kinds of conditions might occur during the execution of a JSON PARSE statement, and might result in the receiver being partially modified:

- Nonexception conditions result in a reason code set in the special register JSON-STATUS, but do not terminate execution of the statement.
- Exception conditions result in the exception code set in the special register JSON-CODE, and cause execution of the statement to be terminated.

The JSON-STATUS reason code values in the following table are additive. For example, execution of the JSON PARSE statement might encounter conditions with codes 1 and 4, resulting in a combined JSON-STATUS value of 5. You can determine if a given condition occurred, and the corresponding code is present in JSON-STATUS, by using a statement such as the following:

```

IF FUNCTION MOD(JSON-STATUS 2 * code) / code = 1
  DISPLAY 'JSON-STATUS condition ' code ' occurred.'
END-IF

```

where code is one of the individual JSON-STATUS codes.

The runtime messages are issued only if the WITH DETAIL phrase was specified on the JSON PARSE statement. Special registers JSON-STATUS and JSON-CODE are always set.

Nonexception conditions and corresponding values of JSON-STATUS

<i>Table 110. Reason codes for JSON nonexception conditions</i>	
Reason code in the JSON-STATUS register	Description
1	One or more data items had no matching JSON name/value pair, and thus were not changed.
2	One or more JSON name/value pairs did not match any data item.
4	One or more data items had multiple matching JSON name/value pairs with duplicate values.
8	One or more table data items had more elements than the matching JSON array.
16	One or more JSON arrays had more values than the matching data item.
32	One or more data items were not changed because the corresponding JSON name/value pair had the value null.
64	One or more table data items had elements that were not changed because the corresponding JSON value was null.
128	A "SIZE ERROR" condition was detected in one or more numeric assignments. The data items were modified anyway.
256	A loss of information occurred in one or more alphanumeric assignments. The data items were modified anyway.
512	One or more JSON name/value pairs had a value that resulted in one or more substitution characters when translated from Unicode to the CCSID specified by the CODEPAGE compiler option.

Exception conditions, and corresponding values of JSON-CODE

<i>Table 111. Reason codes for JSON exception conditions</i>	
Reason code in the JSON-CODE register	Description
100	The JSON text was invalid.
101	The JSON text was zero-length, or consisted only of whitespace.

Table 111. Reason codes for JSON exception conditions (continued)

Reason code in the JSON-CODE register	Description
102	Superfluous nonwhitespace characters were found after the closing brace of the outermost JSON object.
103	One or more data items had multiple matching JSON name/value pairs with different values, and were set to the leftmost value encountered in the JSON text.
104	One or more JSON name/value pairs had a value that was incompatible with the matching data item.
105	One or more matching JSON name/value pairs had the value <code>true</code> or <code>false</code> .
106	No JSON name/value pair matched any data item.
107	One or more matching JSON name/value pairs had the value <code>true</code> or <code>false</code> but the matching COBOL data item was not specified on a 'CONVERTING BOOLEAN' phrase.
108	JSON PARSE statement without NAME OMITTED phrase and OCCURS clause attempted to parse anonymous JSON array in the JSON text.
109	The CCSID of the ENCODING phrase in the JSON PARSE statement was invalid or unsupported.

Nonexception condition runtime messages

These messages are issued if the WITH DETAIL phrase was specified.

IGZ0321I

During execution of the JSON PARSE statement on line *line-number* of program *program-name*, no JSON name/value pair matched data item *data-name*, which was thus not modified.

IGZ0321I

During execution of the JSON PARSE statement on line *line-number* of program *program-name*, no data item matched JSON name *JSON-name* at offset *offset*.

IGZ0323I

During execution of the JSON PARSE statement on line *line-number* of program *program-name*, a duplicate JSON name/value pair at *offset* matched data item *data-name*. The duplicate value was accepted.

IGZ0324I

During execution of the JSON PARSE statement on line *line-number* of program *program-name*, JSON array with name *JSON-name* at offset *offset* had fewer elements than the matching table item *data-name*. The additional table elements were not modified.

IGZ0325I

During execution of the JSON PARSE statement on line *line-number* of program *program-name*, JSON array with name *JSON-name* at offset *offset* had more values than the matching table item *data-name*. The additional values were ignored.

IGZ0326I

During execution of the JSON PARSE statement on line *line-number* of program *program-name*, data item *data-name* was not changed because the value of the JSON name/value pair at offset *offset* was the special value `null`.

IGZ0327I

During execution of the JSON PARSE statement on line *line-number* of program *program-name*, JSON array with name *JSON-name* at offset *offset* included one or more null values. Corresponding elements in the matching table item *data-name* were not changed.

IGZ0328I

During execution of the JSON PARSE statement on line *line-number* of program *program-name*, assignment of the value of the JSON name/value pair at offset *offset* to data item *data-name* resulted in loss of significance ("SIZE ERROR").

IGZ0329I

During execution of the JSON PARSE statement on line *line-number* of program *program-name*, assignment of the value of the JSON name/value pair at offset *offset* to data item *data-name* resulted in a loss of information.

IGZ0330I

During execution of the JSON PARSE statement on line *line-number* of program *program-name*, the value of JSON name/value pair with name *JSON-name* at offset *offset* resulted in one or more substitution characters when translated from Unicode to the CCSID specified by the CODEPAGE compiler option. The translated value was assigned to data-item *data-name*.

Exception condition runtime messages

These messages are issued if the WITH DETAIL phrase was specified.

IGZ0335W

During execution of the JSON PARSE statement on line *line-number* of program *program-name*, the JSON text in *data-name* was found to be invalid. After JSON text *JSON-text-fragment* at offset *offset*, *JSON-token* was found, but one of *JSON-tokens* was expected.

IGZ0336W

During execution of the JSON PARSE statement on line *line-number* of program *program-name*, the JSON text in *data-name* was found to be invalid. The JSON text was zero-length, or consisted only of whitespace.

IGZ0337W

During execution of the JSON PARSE statement on line *line-number* of program *program-name*, the JSON text in *data-name* was found to be invalid. Superfluous characters *text* were found following the closing brace of the outermost JSON object.

IGZ0338W

During execution of the JSON PARSE statement on line *line-number* of program *program-name*, JSON name/value pair at offset *offset* was a duplicate match to data item *data-name*, but with a value different from the first (leftmost) matching name/value pair. The value from the first matching JSON name/value pair was retained.

IGZ0339W

During execution of the JSON PARSE statement on line *line-number* of program *program-name*, the value of JSON name/value pair with name *JSON-name* at offset *offset* was found to be incompatible with the matching data item.

IGZ0340W

During execution of the JSON PARSE statement on line *line-number* of program *program-name*, the value of JSON name/value pair with name *JSON-name* at offset *offset* was found to be one of the special values true or false.

IGZ0341W

During execution of the JSON PARSE statement on line *line-number* of program *program-name*, no JSON name/value pair matched any data item in the receiver. The receiver *data-name* was not modified.

IGZ0342W

During execution of the JSON PARSE statement on line *line-number* of program *program-name*, the value of JSON name/value pair with name *JSON-name* at offset *offset* was found to be one of the

special values `true` or `false`. But, the matching COBOL data item *data-name* is not specified on a 'CONVERTING BOOLEAN' phrase of the JSON PARSE statement.

IGZ0343W

During execution of the JSON PARSE statement on line *line-number* of program *program-name*, the JSON text in *data-name* was found to be an anonymous JSON array, but 'NAME *data-name* OMITTED' phrase of the JSON PARSE statement was not specified or *data-name* did not contain an OCCURS clause.

IGZ0344W

During execution of the JSON PARSE statement on line *line-number* of program *program-name*, the CCSID *ccsid-number* of an ENCODING phrase was found to be invalid or unsupported.

Appendix E. EXIT compiler option

You can use the EXIT compiler option to provide user-supplied modules in place of various compiler functions. For details about processing of each exit module, error handling for exit modules, or using the EXIT option with CICS, SQL and SQLIMS statements, see the following topics.

Note: All user exit routines must be AMODE 31 or AMODE ANY. You should not compile exit routines with LP(64).

Related tasks

[“Using the user-exit work area” on page 811](#)

[“Calling from exit modules” on page 812](#)

[Using the EXIT compiler option with CICS, SQL and SQLIMS statements](#)

Related references

[“EXIT” on page 357](#)

[“Processing of INEXIT” on page 812](#)

[“Processing of LIBEXIT” on page 813](#)

[“Processing of PRTEXIT” on page 816](#)

[“Processing of ADEXIT” on page 818](#)

[“Processing of MSGEXIT” on page 819](#)

[“Error handling for exit modules” on page 828](#)

Using the user-exit work area

When you use one of the user exits, the compiler provides a work area in which you can save the address of GETMAIN storage obtained by the exit module. Having such a work area lets the module be reentrant.

The user-exit work area consists of 6 fullwords that reside 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.

The words in the user-exit work area are used by the individual exit modules as shown in the following table.

Word number	Used by module:
1	INEXIT
2	LIBEXIT
3	PRTEXIT
4	ADEXIT
5	(Reserved)
6	MSGEXIT

Related references

[“Processing of INEXIT” on page 812](#)

[“Processing of LIBEXIT” on page 813](#)

[“Processing of PRTEXIT” on page 816](#)

[“Processing of ADEXIT” on page 818](#)

[“Processing of MSGEXIT” on page 819](#)

Calling from exit modules

To call COBOL programs or library routines within your exit modules, use standard COBOL linkage. You need to be aware of the register conventions in order to trace the call chain correctly.

When a call is made to a program or to a routine in an exit module, 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 RMODE attribute 24 and AMODE attribute ANY.

Related concepts

[“Storage and its addressability” on page 39](#)

Processing of INEXIT

The INEXIT exit module is used to read the primary source code instead of obtaining it directly from SYSIN.

<i>Table 113. INEXIT processing</i>	
Action by compiler	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

INEXIT parameters

The compiler uses 10 parameters, passed by reference, to communicate with the exit module. The return code, data length, and data parameters are set by the exit module for return to the compiler; the other items are passed from the compiler to the exit module.

<i>Table 114. INEXIT parameters</i>		
Parameter number	Parameter item	Description of item
1	User-exit type	Halfword that identifies which user exit is to perform the operation. 1=INEXIT

Table 114. **INEXIT parameters** (continued)

Parameter number	Parameter item	Description of item
2	Operation code	Halfword that indicates the type of operation: <ul style="list-style-type: none"> • 0=OPEN • 1=CLOSE • 2=GET
3	Return code	Fullword, set by the exit module, that indicates the success of the requested operation: <ul style="list-style-type: none"> • 0=Operation was successful • 4=End-of-data • 12=Operation failed
4	User-exit work area	Six-fullword work area provided by the compiler for use by the user-exit module. First word: for use by INEXIT
5	Data length	Fullword, set by the exit module, that specifies the length of the record being returned by the GET operation (must be 80)
6	Data or <i>str1</i>	Fullword, set by the exit module, that contains 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.
7	Not used	(Used only by LIBEXIT and MSGEXIT)
8	Not used	(Used only by LIBEXIT)
9	Not used	(Used only by LIBEXIT)
10	Not used	(Used only by LIBEXIT)

Related tasks

[“Using the user-exit work area” on page 811](#)

[“Calling from exit modules” on page 812](#)

[Using the EXIT compiler option with CICS, SQL and SQLIMS statements](#)

Processing of LIBEXIT

The LIBEXIT 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 copybooks whenever COPY or BASIS statements are encountered.

Table 115. **LIBEXIT processing**

Action by compiler	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.

<i>Table 115. LIBEXIT processing (continued)</i>	
Action by compiler	Action by exit module
Calls the exit module with a FIND op code if the <i>library-name</i> was 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 copybook. 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 copybook 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 copybook can contain a COPY statement.

You cannot make recursive calls to *text-name*. That is, a copybook can be named only once in a set of nested COPY statements until the end-of-data for that copybook is reached.

The following table shows how the processing of LIBEXIT changes when there are one or more valid COPY statements that are not nested.

<i>Table 116. LIBEXIT processing with nonnested COPY statements</i>	
Action by compiler	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> was 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 copybook. Passes an appropriate return code to the compiler when positioning is complete.
Calls the exit module with a FIND op code if the <i>library-name</i> was successfully opened	Reestablishes positioning at the previous active copybook. 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 copybook. After verification, passes either the length and address of the record to be copied from the active copybook 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

The following table shows how the processing of LIBEXIT changes when the compiler encounters a valid nested COPY statement.

Action by compiler	Action by exit module
If the requested <i>library-name</i> from the nested <i>COPY</i> statement was not previously opened, calls the exit module with an <i>OPEN</i> op code	Pushes its control information about the active copybook onto a stack. Completes the requested action (<i>OPEN</i>). The newly requested <i>text-name</i> (or <i>basis-name</i>) becomes the active copybook.
Calls the exit module with a <i>FIND</i> op code for the requested new <i>text-name</i>	Pushes its control information about the active copybook onto a stack. Completes the requested action (<i>FIND</i>). The newly requested <i>text-name</i> (or <i>basis-name</i>) becomes the active copybook.
Calls the exit module with a <i>GET</i> op code	Passes the compiler either the length and address of the record to be copied from the active copybook or the end-of-data indicator. At end-of-data, pops its control information from the stack.

LIBEXIT parameters

The compiler uses 10 parameters, passed by reference, to communicate with the exit module. The return code, data length, and data parameters are set by the exit module for return to the compiler; the other items are passed from the compiler to the exit module.

Parameter number	Parameter item	Description of item
1	User-exit type	Halfword that identifies which user exit is to perform the operation. 2=LIBEXIT
2	Operation code	Halfword that indicates the type of operation: <ul style="list-style-type: none"> • 0=OPEN • 1=CLOSE • 2=GET • 4=FIND
3	Return code	Fullword, set by the exit module, that indicates the success of the requested operation: <ul style="list-style-type: none"> • 0=Operation was successful • 4=End-of-data • 12=Operation failed
4	User-exit work area	Six-fullword work area provided by the compiler for use by the user-exit module. Second word: for use by LIBEXIT
5	Data length	Fullword, set by the exit module, that specifies the length of the record being returned by the <i>GET</i> operation (must be 80)

Table 118. **LIBEXIT** parameters (continued)

Parameter number	Parameter item	Description of item
6	Data or <i>str2</i>	Fullword, set by the exit module, that contains 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.
7	System <i>library-name</i>	Eight-character area that contains 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.
8	System <i>text-name</i>	Eight-character area that contains the <i>text-name</i> from the COPY statement (<i>basis-name</i> from BASIS statement). Processing and conversion rules for a program-name are applied. Padded with blanks if required. Applies only to FIND.
9	Library-name	Thirty-character area that contains 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.
10	Text-name	Thirty-character area that contains 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 tasks

[“Using the user-exit work area” on page 811](#)

[“Calling from exit modules” on page 812](#)

[Using the EXIT compiler option with CICS, SQL and SQLIMS statements](#)

Processing of PRTEXTIT

The PRTEXTIT exit module is used in place of the SYSPRINT data set.

Table 119. **PRTEXTIT** processing

Action by compiler	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

PRTEXIT parameters

The compiler uses 10 parameters, passed by reference, to communicate with the exit module. The return code, data length, and data buffer parameters are set by the exit module for return to the compiler; the other items are passed from the compiler to the exit module.

Parameter number	Parameter item	Description of item
1	User-exit type	Halfword that identifies which user exit is to perform the operation. 3=PRTEXIT
2	Operation code	Halfword that indicates the type of operation: <ul style="list-style-type: none"> • 0=OPEN • 1=CLOSE • 3=PUT
3	Return code	Fullword, set by the exit module, that indicates the success of the requested operation: <ul style="list-style-type: none"> • 0=Operation was successful • 12=Operation failed
4	User-exit work area	Six-fullword work area provided by the compiler for use by the user-exit module. Third word: for use by PRTEXIT
5	Data length	Fullword that specifies the length of the record being supplied by the PUT operation (the compiler sets this value to 133)
6	Data buffer or <i>str3</i>	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.
7	Not used	(Used only by LIBEXIT and MSGEXIT)
8	Not used	(Used only by LIBEXIT)
9	Not used	(Used only by LIBEXIT)
10	Not used	(Used only by LIBEXIT)

Related tasks

[“Using the user-exit work area” on page 811](#)

[“Calling from exit modules” on page 812](#)

[Using the EXIT compiler option with CICS, SQL and SQLIMS statements](#)

Processing of ADEXIT

The ADEXIT module is called for each SYSADATA record immediately after the record has been written out to the file.

To use an ADEXIT module, you must compile using the ADATA option to produce SYSADATA output, and code the SYSADATA DD statement.

Action by compiler	Action by exit module
Loads the exit module (<i>mod4</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 module with a PUT op code when the compiler has written a SYSADATA record, supplying the address and length of the SYSADATA record	Passes the status of the PUT request to the compiler by a return code
Calls the exit module with a CLOSE op code when the end-of-data is presented	Releases any resources

ADEXIT parameters

The compiler uses 10 parameters, passed by reference, to communicate with the exit module. The return code, data length, and data buffer parameters are set by the exit module for return to the compiler; the other items are passed from the compiler to the exit module.

Parameter number	Parameter item	Description of item
1	User-exit type	Halfword that identifies which user exit is to perform the operation. 4=ADEXIT
2	Operation code	Halfword that indicates the type of operation: <ul style="list-style-type: none"> • 0=OPEN • 1=CLOSE • 3=PUT
3	Return code	Fullword, set by the exit module, that indicates the success of the requested operation: <ul style="list-style-type: none"> • 0=Operation was successful • 12=Operation failed
4	User-exit work area	Six-fullword work area provided by the compiler for use by the user-exit module. Fourth word: for use by ADEXIT
5	Data length	Fullword that specifies the length of the record being supplied by the PUT operation

Table 122. **ADEXIT parameters** (continued)

Parameter number	Parameter item	Description of item
6	Data buffer or <i>str4</i>	Fullword that contains 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.
7	Not used	(Used only by LIBEXIT and MSGEXIT)
8	Not used	(Used only by LIBEXIT)
9	Not used	(Used only by LIBEXIT)
10	Not used	(Used only by LIBEXIT)

Related tasks

[“Using the user-exit work area” on page 811](#)

[“Calling from exit modules” on page 812](#)

[Using the EXIT compiler option with CICS, SQL and SQLIMS statements](#)

Related references

[“ADATA” on page 335](#)

Processing of MSGEXIT

The MSGEXIT module is used to customize compiler diagnostic messages and FIPS messages. The module can customize a message either by changing its severity or suppressing it.

If the MSGEXIT module assigns a severity to a FIPS message, the message is converted into a diagnostic message. (The message is shown in the summary of diagnostic messages in the listing.)

A MSGEXIT summary at the end of the compiler listing shows how many messages were changed in severity and how many messages were suppressed.

Table 123. **MSGEXIT processing**

Action by compiler	Action by exit module
Loads the exit module (<i>mod5</i>) during initialization	
Calls the exit module with an OPEN operation code (op code)	Optionally processes <i>str5</i> and passes the status of the OPEN request to the compiler
Calls the exit module with a MSGSEV operation code (op code) when the compiler is about to issue a diagnostic message or FIPS message	One of the following actions: <ul style="list-style-type: none"> • Indicates no customization of the message (by setting return code to 0) • Specifies a new severity for (or suppression of) the message, and sets return code to 4 • Indicates that the operation failed (by setting return code to 12)
Calls the exit module with a CLOSE op code	Optionally frees storage and passes the status of the CLOSE request to the compiler

<i>Table 123. MSGEXIT processing (continued)</i>	
Action by compiler	Action by exit module
Deletes the exit module (<i>mod5</i>) during compiler termination	

MSGEXIT parameters

The compiler uses 10 parameters, passed by reference, to communicate with the exit module. The return code and user-requested severity parameters are set by the exit module for return to the compiler; the other items are passed from the compiler to the exit module.

<i>Table 124. MSGEXIT parameters</i>		
Parameter number	Parameter item	Description of item
1	User-exit type	Halfword that identifies which user exit is to perform the operation. 6=MSGEXIT
2	Operation code	Halfword that indicates the type of operation: <ul style="list-style-type: none"> • 0=OPEN • 1=CLOSE • 5=MSGSEV: customize message severity
3	Return code	Fullword, set by the exit module, that indicates the success of the requested operation. For op code MSGSEV: <ul style="list-style-type: none"> • 0=Message not customized • 4=Message found and customized • 12=Operation failed
4	User-exit work area	Six-fullword work area provided by the compiler for use by the user-exit module. Sixth word: for use by MSGEXIT
5	Not used	(Used by the other exits)
6	Message exit data	Three-halfword area (on a halfword boundary). <ul style="list-style-type: none"> • First halfword: the message number of the message to be customized • Second halfword: for a diagnostic message, the default severity; for a FIPS message, the FIPS category as a numeric code • Third halfword: the user-requested severity for the message (-1 to indicate suppression)
7	<i>str5</i>	First halfword (on a halfword boundary): the length of the string, followed by the string
8	Not used	(Used only by LIBEXIT)
9	Not used	(Used only by LIBEXIT)

Table 124. **MSGEXIT parameters** (continued)

Parameter number	Parameter item	Description of item
10	Not used	(Used only by LIBEXIT)

[“Example: MSGEXIT user exit” on page 823](#)

Related tasks

[“Using the user-exit work area” on page 811](#)

[“Calling from exit modules” on page 812](#)

[“Customizing compiler-message severities” on page 821](#)

[Using the EXIT compiler option with CICS, SQL and SQLIMS statements](#)

Customizing compiler-message severities

To change the severities of compiler messages or suppress compiler messages (including FIPS messages), do the steps described below.

1. Code and compile a COBOL program named ERRMSG. The program needs only a PROGRAM-ID paragraph, as described in the related task.
2. Review the ERRMSG listing, which contains a complete list of compiler messages with their message numbers, severities, and message text.
3. Decide which messages you want to customize.

To understand the customizations that are possible, see the related reference about customizable compiler-message severities.

4. Code a MSGEXIT module to implement the customizations. Refer to the sample MSGEXIT that is included with the COBOL compiler in the SIGYSAMP dataset called IGYMSGXT. For details, see [“Example: MSGEXIT user exit” on page 823](#).
 - a. Verify that the operation-code parameter indicates message-severity customization.
 - b. Check the two input values in the message-exit-data parameter: the message number; and the default severity for a diagnostic message or the FIPS category for a FIPS message.

The FIPS category is expressed as numeric code. For details, see the related reference about customizable compiler-message severities.
 - c. For a message that you want to customize, set the user-requested severity in the message-exit-data parameter to indicate either:
 - A new message severity, by coding severity 0, 4, 8, or 12
 - Message suppression, by coding severity -1
 - d. Set the return code to one of the following values:
 - 0, to indicate that the message was not customized
 - 4, to indicate that the message was found and customized
 - 12, to indicate that the operation failed and that compilation should be terminated
5. Compile and link your MSGEXIT module.
6. Add the data set that contains your MSGEXIT module to the compiler concatenation by using a STEPLIB or JOBLIB DD statement.
7. Recompile program ERRMSG, but use compiler option EXIT (MSGEXIT (*msgmod*)), where *msgmod* is the name of your MSGEXIT module.
8. Review the listing and check for:
 - Updated message severities

- Suppressed messages (indicated by XX in place of the severity)
- Unsupported severity changes or unsupported message suppression (indicated by a severity-U diagnostic message, and compiler termination with return code 16)

Related tasks

[“Generating a list of compiler messages” on page 281](#)

Related references

[“Severity codes for](#)

[compiler diagnostic messages” on page 282](#)

[“Customizable compiler-message severities” on page 822](#)

[“Effect](#)

[of message customization on compilation return code” on page 823](#)

[“Error handling for exit modules” on page 828](#)

Customizable compiler-message severities

To customize compiler-message severities, you need to understand the possible severities of compiler diagnostic messages, the levels or categories of FIPS messages, and the permitted customizations of message severities.

The possible severity codes for compiler diagnostic messages are described in the related reference about severity codes.

The eight categories of FIPS (FLAGSTD) messages are shown in the following table. The category of any given FIPS message is passed as a numeric code to the MSGEXIT module. Those numeric codes are shown in the second column.

FIPS level or category	Numeric code	Description
D	81	Debug module level 1
E	82	Extension (IBM)
H	83	High level
I	84	Intermediate level
N	85	Segmentation module level 1
O	86	Obsolete elements
Q	87	High-level and obsolete elements
S	88	Segmentation module level 2

FIPS messages have an implied severity of zero (severity I).

Permitted message-severity customizations:

You can change the severity of a compiler message in the following ways:

- Severity-I and severity-W compiler diagnostic messages, and FIPS messages, can be changed to have any severity from I through S.

Assigning a severity to a FIPS message converts the FIPS message to a diagnostic message of the assigned severity.

As examples, you can:

- Lower an optimizer warning to severity I.
- Disallow REDEFINING a smaller item with a larger item by raising the severity of message 1154.

- Disallow complex OCCURS DEPENDING ON by changing FIPS message 8235 from a category-E FIPS message to a severity-S compiler diagnostic message.
- Severity-E messages can be raised to severity S, but not lowered to severity I or W, because an error condition has occurred in the program.
- Severity-S and severity-U messages cannot be changed to have a different severity.

You can request suppression of compiler messages as follows:

- I, W, and FIPS messages can be suppressed.
- E and S messages cannot be suppressed.

Related references

[“Severity codes for compiler diagnostic messages” on page 282](#)

[“FLAGSTD” on page 360](#)

[“Effect of message customization on compilation return code” on page 823](#)

Effect of message customization on compilation return code

If you use a MSGEXIT module, the final return code from the compilation of a program could be affected as described below.

If you change the severity of a message, the return code from the compilation might also be changed. For example, if a compilation produces one diagnostic message, and it is a severity-E message, the compilation return code would normally be 8. But if the MSGEXIT module changes the severity of that message to severity S, then the return code from compilation would be 12.

If you suppress a message, the return code from the compilation is no longer affected by the severity of that message. For example, if a compilation produces one diagnostic message, and it is a severity-W message, the compilation return code would normally be 4. But if the MSGEXIT module suppresses that message, then the return code from compilation would be 0.

Related tasks

[“Customizing compiler-message severities” on page 821](#)

Related references

[“Severity codes for compiler diagnostic messages” on page 282](#)

Example: MSGEXIT user exit

The following example shows a MSGEXIT user-exit module that changes message severities and suppresses messages. This example is included with Enterprise COBOL 6 in the SIGYSAMP dataset, in member IGYMSGXT.

For helpful tips about using a message-exit module, see the comments within the code.

```

*****
* IGYMSGXT - Sample COBOL program for MSGEXIT *
*****
*
* IBM Enterprise COBOL for z/OS *
* Version 6 Release 4 Modification 0 *
*
* LICENSED MATERIALS - PROPERTY OF IBM. *
*
* 5655-EC6 COPYRIGHT IBM CORP. 2022 *
* ALL RIGHTS RESERVED *
*
* US GOVERNMENT USERS RESTRICTED RIGHTS - USE, *
* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA *
* ADP SCHEDULE CONTRACT WITH IBM CORP. *
*
*****

```

```

*****
* Function: This is a SAMPLE user exit for the MSGEXIT *
* suboption of the EXIT compiler option. This exit *
* can be used to customize the severity of or *
* suppress compiler diagnostic messages and FIPS *
* messages. This example program includes several *
* sample customizations to show how customizations *
* are done. If you do not want the sample *
* customizations then either delete the unwanted *
* lines of code or comment them out with a comment *
* indicator in column 7 (*). *
* ----- *
* USAGE NOTE: To use this user exit program, make the *
* link-edited load module available to your *
* compilers that will use the MSGEXIT suboption of *
* the EXIT compiler option. Also, the name should *
* be changed, since IBM recommends that you avoid *
* having programs with names that start with IGY. *
* Sample steps to take: *
* 1) Make your customizations *
* 2) Change program name (E.G. MYEXIT) *
* 3) Compile and link into a dataset *
* 4) Include that dataset in your compile *
* JCL concatenation for the compile step. *
* If you link into USER.COBOLLIB: *
* //COBOL.STEPLIB DD DSNAME=SYS1.SIGYCOMP,DISP=SHR *
* // DD DSNAME=USER.COBOLLIB,DISP=SHR *
* 5) Finally, compile your programs with the *
* EXIT compiler option, EG: *
* EXIT(MSGEXIT(MYEXIT)) *
* COMPILE NOTE: Compile this program with NOEXIT. *
*****
Id Division.
Program-Id. IGYMSGXT.
Data Division.

Working-Storage Section.

*****
* Local variables. *
*****

77 EXIT-TYPEN PIC 9(4).
77 EXIT-DEFAULT-SEV-FIPS PIC X.

*****
* Definition of the User-Exit Parameter List, which is *
* passed from the COBOL compiler to the user-exit module. *
*****

Linkage Section.
01 EXIT-TYPE PIC 9(4) COMP.
01 EXIT-OPERATION PIC 9(4) COMP.
01 EXIT-RETURNCODE PIC 9(9) COMP.
01 EXIT-WORK-AREA.
02 EXIT-WORK-AREA-PTR OCCURS 6 POINTER.
01 EXIT-DUMMY POINTER.
01 EXIT-MESSAGE-PARMS.
02 EXIT-MESSAGE-NUM PIC 9(4) COMP.
02 EXIT-DEFAULT-SEV PIC 9(4) COMP.
02 EXIT-USER-SEV PIC S9(4) COMP.
01 EXIT-STRING.
02 EXIT-STR-LEN PIC 9(4) COMP.
02 EXIT-STR-TXT PIC X(64).

*****
*****
* Begin PROCEDURE DIVISION *
* *
* Check parameters and perform the operation requested. *

```

```

*
*****
Procedure Division Using EXIT-TYPE          EXIT-OPERATION
                          EXIT-RETURNCODE  EXIT-WORK-AREA
                          EXIT-DUMMY        EXIT-MESSAGE-PARMS
                          EXIT-STRING       EXIT-DUMMY
                          EXIT-DUMMY       EXIT-DUMMY.

Compute EXIT-RETURNCODE = 0

Evaluate TRUE

*****
* Handle a bad invocation of this exit by the compiler.
* This could happen if this routine was used for one of the
* other EXITS, such as INEXIT, PRTEXIT or LIBEXIT.
*****
When EXIT-TYPE Not = 6
  Move EXIT-TYPE to EXIT-TYPEN
  Display '**** Invalid exit routine identifier'
  Display '**** EXIT TYPE = ' EXIT-TYPE
  Compute EXIT-RETURNCODE = 16

*****
* Handle the OPEN call to this exit by the compiler
* Display the exit string (str5 in syntax diagram) from
* the EXIT(MSGEXIT('str5',mod5)) option specification.
*****
When EXIT-OPERATION = 0
* Display 'Opening MSGEXIT'
* If EXIT-STR-LEN Not Zero Then
*   Display ' str5 len = ' EXIT-STR-LEN
*   Display ' str5 = ' EXIT-STR-TXT(1:EXIT-STR-LEN)
* End-If
Continue

*****
* Handle the CLOSE call to this exit by the compiler
*****
When EXIT-OPERATION = 1
* Display 'Closing MSGEXIT'
Goback

*****
* Handle the customize message severity call to this exit
* Display information about every customized severity.
*****
When EXIT-OPERATION = 5
* Display 'MSGEXIT called with MSGSEV'
  If EXIT-MESSAGE-NUM < 8000 Then
    Perform Error-Messages-Severity
  Else
    Perform FIPS-Messages-Severity
  End-If

* If EXIT-RETURNCODE = 4 Then
*   Display '>>> Customizing message ' EXIT-MESSAGE-NUM
*   ' with new severity ' EXIT-USER-SEV ' <<<<'
*   If EXIT-MESSAGE-NUM > 8000 Then
*     Display 'FIPS sev = ' EXIT-DEFAULT-SEV-FIPS '<<<<'
*   End-If
* End-If

*****
* Handle a bad invocation of this exit by the compiler.
* The compiler should not invoke this exit with EXIT-TYPE = 6
* and an opcode other than 0, 1, or 5. This should not happen
* and IBM service should be contacted if it does.
*****
When Other
  Display '**** Invalid MSGEXIT routine operation '
  Display '**** EXIT OPCODE = ' EXIT-OPERATION
  Compute EXIT-RETURNCODE = 16

End-Evaluate

Goback.

*****
* ERROR MESSAGE PROCESSOR
*

```

```

*****
Error-Messages-Severity.

*   Assume message severity will be customized...
    Compute EXIT-RETURNCODE = 4

    Evaluate EXIT-MESSAGE-NUM

*****
*   Change severity of message 1154(W) to 12 ('S')      *
*   This is the case of redefining a large item        *
*   with a smaller item, IBM Req # MR0904063236       *
*****
    When(1154)
        Compute EXIT-USER-SEV = 12

*****
*   Modify the severity of RULES messages to enforce coding *
*   standards or highlight coding that you want to avoid.  *
*   Here are the message numbers and what they flag:      *
*   1158 RULES(NOOMITODOMIN)   Missing min idx in ODO table def*
*   1348 RULES(NOEVENPACK)     Even digit packed-decimal items *
*   1353 RULES(NOSLACKBYTES)   Slack bytes within records      *
*   1379 RULES(NOSLACKBYTES)   Slack bytes between records     *
*   2159 RULES(NOENDPERIOD)    Cond stmt terminated by period  *
*   2262 RULES(NOUNREFALL)     Unref'd items (source/copybook) *
*   2262 RULES(NOUNREFSOURCE)  Unref'd items (source only)   *
*   2224 RULES(NOLAXPERF)      Ineff. type for PERFORM VARYING *
*   2246 RULES(NOLAXPERF)      Ineff. type for subscript      *
*   2247 RULES(NOLAXPERF)      Compiler option NOAWO in effect *
*   2248 RULES(NOLAXPERF)      Option ARITH(EXTEND) in effect *
*   2249 RULES(NOLAXPERF)      Option NOBLOCK0 in effect      *
*   2250 RULES(NOLAXPERF)      Option NOFASTSRT in effect     *
*   2251 RULES(NOLAXPERF)      Option NUMPROC(NOPFD) in effect*
*   2252 RULES(NOLAXPERF)      Option OPTIMIZE(0) in effect   *
*   2253 RULES(NOLAXPERF)      Option SSRANGE in effect       *
*   2254 RULES(NOLAXPERF)      Option THREAD in effect        *
*   2255 RULES(NOLAXPERF)      Option TRUNC(STD) in effect     *
*   2256 RULES(NOLAXPERF)      Option TRUNC(BIN) in effect    *
*   3084 RULES(NOLAXPERF)     Ineff. type for arith sender   *
*   3123 RULES(NOLAXPERF)     Lots of padding in alph MOVE    *
*
*****
    When(1158)      *> Disallow omitting ODO table min
        Compute EXIT-USER-SEV = 12
    When(1348)      *> Disallow even-digit Comp-3
        Compute EXIT-USER-SEV = 12
    When(1353) When(1379) *> Disallow slack bytes
        Compute EXIT-USER-SEV = 12
    When(2159)      *> Disallow period-termination
        Compute EXIT-USER-SEV = 12 *> of conditional stmts
    When(2262)      *> Disallow unref'd data items
        Compute EXIT-USER-SEV = 12
*   Highlight poorly performing COBOL features
    When(2224)      *> Ineff. type for PERFORM VARYING
    When(2246)      *> Ineff. type for subscript
    When(2247)      *> Compiler option NOAWO in effect
    When(2248)      *> Option ARITH(EXTEND) in effect
    When(2249)      *> Option NOBLOCK0 in effect
    When(2250)      *> Option NOFASTSRT in effect
    When(2251)      *> Option NUMPROC(NOPFD) in effect
    When(2252)      *> Option OPTIMIZE(0) in effect
    When(2253)      *> Option SSRANGE in effect
    When(2254)      *> Option THREAD in effect
    When(2255)      *> Option TRUNC(STD) in effect
    When(2256)      *> Option TRUNC(BIN) in effect
    When(3084)      *> Ineff. type for arith sender
    When(3123)      *> Lots of padding in alph MOVE
        Compute EXIT-USER-SEV = 8

*****
*   Change severity of messages 3178(I) to highlight File *
*   Definitions that could lead to wrong-length read conditions. *
*   Message 3178 is issued when the length of the shortest *
*   record description is less than the FROM integer in the *
*   RECORD IS VARYING clause, and when the length of the *
*   longest record description is greater than the TO integer *
*   in the RECORD IS VARYING clause. *
*****
    When(3178)
        Compute EXIT-USER-SEV = 8

```

```

*****
* Change severity of messages 3188(W) and 3189(W)
* to 12 ('S'). This is to force a fix for all
* SEARCH ALL cases that might behave differently
* between COBOL compilers previous to Enterprise
* COBOL release 3.4 and later compilers such as
* Enterprise COBOL 4.2.
* Another way to handle this migration is to analyze all of
* the warnings you get and then change them to I-level when
* the analysis is complete.
*****
      When(3188) When(3189)
        Compute EXIT-USER-SEV = 12

*****
* Change severity of message 4019 to suppress this message.
* Message 4019 says 'Compiler option "OPTFILE" was specified, "
* but file SYSOPTF could not be opened. No options from a
* "SYSOPTF" file were used.
* This customization would allow users to specify OPTFILE
* as a default option and it would only have an affect
* if there is a SYSOPTF file present
*****
      When(4019)
        Compute EXIT-USER-SEV = -1

*****
* Change severity of 'optimization' messages to suppress them
* so that compilation Return Code can be zero (RC=0)
* 7300: The code from lines &2 in program '&1' can never
* be executed and was therefore discarded.
* 7301: A zero base was raised to a zero power in a numeric
* literal exponentiation. The result was set to 1.
* 7302: A zero base was raised to a negative power in a numeric
* literal exponentiation. The result was set to 0.
* 7304: An exception "&1" occurred while processing numeric
* literals. The result of the operation was set to zero.
* 7307: This statement may cause a program exception at execution
* time.
* 7309: There may be a loop from the "PERFORM" statement at "
* "PERFORM (line &1)" to itself.
* 7312: Procedure starting at 'proc-name' (line 'num') was
* copied one or more times to be inlined for PERFORM
* statements. xxx total bytes were generated from
* copying that procedure.
*****
      When(7300) When(7301) When(7302) When(7304)
      When(7307) When(7309) When(7312)
        Compute EXIT-USER-SEV = -1    *> Suppress the messages

*****
* Change severity of message 7311(W) to 12 ('S'). This is the *
* case of INITCHECK messages about uninitialized data items. *
*****
      When(7311)
        Compute EXIT-USER-SEV = 12

*****
* Message severity Not customized
*****
      When Other
        Compute EXIT-RETURNCODE = 0

      End-Evaluate
      .

*****
* FIPS MESSAGE PROCESSOR
*****
      Fips-Messages-Severity.

* Assume message severity will be customized...
      Compute EXIT-RETURNCODE = 4

* Convert numeric FIPS(FLAGSTD) 'category' to character
* See the Programming Guide for description of FIPS category

      EVALUATE EXIT-DEFAULT-SEV
        When 81
          MOVE 'D' To EXIT-DEFAULT-SEV-FIPS
        When 82
          MOVE 'E' To EXIT-DEFAULT-SEV-FIPS
        When 83

```

```

        MOVE 'H' To EXIT-DEFAULT-SEV-FIPS
    When 84
        MOVE 'I' To EXIT-DEFAULT-SEV-FIPS
    When 85
        MOVE 'N' To EXIT-DEFAULT-SEV-FIPS
    When 86
        MOVE 'O' To EXIT-DEFAULT-SEV-FIPS
    When 87
        MOVE 'Q' To EXIT-DEFAULT-SEV-FIPS
    When 88
        MOVE 'S' To EXIT-DEFAULT-SEV-FIPS
    When Other
        Continue
End-Evaluate

*****
* Example of using FIPS category to force coding
* restrictions. This is not a recommendation!
* Change severity of all OBSOLETE item FIPS
* messages to 'S'
*****
* If EXIT-DEFAULT-SEV-FIPS = '0' Then
*   Display '>>> Default customizing FIPS category '
*   EXIT-DEFAULT-SEV-FIPS ' msg ' EXIT-MESSAGE-NUM '<<<<'
*   Compute EXIT-USER-SEV = 12
* End-If

Evaluate EXIT-MESSAGE-NUM
*****
* Change severity of message 8062(0) to 8 ('E')
* 8062 = GO TO without proc name
*****
When(8062)
    Compute EXIT-USER-SEV = 8

*****
* Change severity of message 8193(E) to 0('I')
* 8193 = GOBACK
*****
When(8193)
    Compute EXIT-USER-SEV = 0

*****
* Change severity of message 8235(E) to 8 (Error)
* to disallow Complex Occurs Depending On
* 8235 = Complex Occurs Depending On
*****
When(8235)
    Compute EXIT-USER-SEV = 08

*****
* Change severity of message 8270(0) to -1 (Suppress)
* 8270 = SERVICE LABEL
*****
When(8270)
    Compute EXIT-USER-SEV = -1

*****
* Message severity Not customized
*****
When Other
*   For the default set '0' to 'S' case...
*   If EXIT-USER-SEV = 12 Then
*       Compute EXIT-RETURNCODE = 4
*   Else
*       Compute EXIT-RETURNCODE = 0
*   End-If

End-Evaluate

END PROGRAM IGYMSGXT.

```

Error handling for exit modules

The conditions described below can occur during processing of the user exits.

Exit load failure:

Message IGYSI5207-U is written to the operator if a LOAD request for any of the user exits fails:

An error occurred while attempting to load user exit *exit-name*.

Exit open failure:

Message IGYSI5208-U is written to the operator if an OPEN request for any of the user exits fails:

An error occurred while attempting to open user exit *exit-name*.

PRTEXIT PUT failure:

- Message IGYSI5203-U is written to the listing:

A PUT request to the PRTEXIT user exit failed with return code *nn*.

- Message IGYSI5217-U is written to the operator:

An error occurred in PRTEXIT user exit *exit-name*. Compiler terminated.

SYSIN GET failures:

The following messages might be written to the listing:

- IGYSI5204-U:

The record address was not set by the *exit-name* user exit.

- IGYSI5205-U:

A GET request from the INEXIT user exit failed with return code *nn*.

- IGYSI5206-U:

The record length was not set by the *exit-name* user exit.

ADEXIT PUT failure:

- Message IGYSI5225-U is written to the operator:

An error occurred in ADEXIT user exit *exit-name*. Compiler terminated.

- Message IGYSI5226-U is written to the listing:

A PUT request to the ADEXIT user exit failed with return code *nn*.

MSGEXIT failures:

Customization failure: Message IGYPP5293-U is written to the listing if an unsupported severity change or unsupported message suppression is attempted:

MSGEXIT user exit *exit-name* specified a message severity customization that is not supported. The message number, default severity, and user-specified severity were: *mm*, *ds*, *us*. Change MSGEXIT user exit *exit-name* to correct this error.

General failure: Message IGYPP5064-U is written to the listing if the MSGEXIT module sets the return code to a nonzero value other than 4:

A call to the MSGEXIT user exit routine *exit-name* failed with return code *nn*.

In the MSGEXIT messages, the two characters *PP* indicate the phase of the compiler that issued the message that resulted in a call to the MSGEXIT module.

Related tasks

[“Customizing compiler-message severities” on page 821](#)

Using the EXIT compiler option with CICS, SQL and SQLIMS statements

When you compile using suboptions of the EXIT compiler option, and your program contains EXEC CICS, EXEC SQL, or EXEC SQLIMS statements, the actions that you can take in the exit modules depend on whether you use the separate CICS translator and Db2 precompiler, or the integrated CICS translator and Db2 coprocessor.

If the program contains EXEC SQLIMS statements, the actions that you can take in the exit modules are the actions that are listed for the integrated translator.

The following table shows which actions you can take in the exit modules depending on whether you use the integrated or separate translators.

Compile with suboption	Translated with integrated or separate CICS and Db2 translators?	Possible actions	Comments
INEXIT	Integrated	Can process EXEC CICS, EXEC SQL, and EXEC SQLIMS statements in the INEXIT module	The INEXIT module does not get control of the COBOL statements that are generated for the EXEC statements.
	Separate	Can process the COBOL statements that are generated for the EXEC statements in the INEXIT module	You can change the generated statements in the INEXIT module, but doing so is not supported by IBM.
LIBEXIT	Integrated	Can process in the LIBEXIT module the statements that are brought in by the EXEC SQL INCLUDE and EXEC SQLIMS INCLUDE statements. Can process EXEC CICS source statements in the LIBEXIT module.	EXEC SQL INCLUDE and EXEC SQLIMS INCLUDE statements are processed like COBOL COPY statements.
	Separate	Can process the COBOL statements that are generated for the EXEC CICS statements in the LIBEXIT module	You can process the input statements that are brought in by the EXEC SQL INCLUDE and SQLIMS INCLUDE statements only by using the INEXIT suboption.
PRTEXT	Integrated	Can process the EXEC CICS, EXEC SQL, and EXEC SQLIMS source statements from the SOURCE listing in the PRTEXT module	The PRTEXT module does not have access to the COBOL statements that are generated.
	Separate	Can process the COBOL SOURCE listing statements that are generated for the EXEC statements in the PRTEXT module	

Table 126. **Actions possible in exit modules for CICS, SQL and SQLIMS statements** (continued)

Compile with suboption	Translated with integrated or separate CICS and Db2 translators?	Possible actions	Comments
ADEXIT	Integrated	Can process the EXEC CICS, EXEC SQL, and EXEC SQLIMS source statements in the ADEXIT module	The ADEXIT module does not have access to the COBOL statements that are generated.
	Separate	Can process the COBOL SYSADATA statements that are generated for the EXEC statements in the ADEXIT module	
MSGEXIT	Integrated	Can process CICS and Db2 messages in the MSGEXIT module	
	Separate	Cannot process CICS and Db2 messages in the MSGEXIT module	Messages from CICS are shown in the separate CICS translator listing; messages from Db2 are shown in the Db2 precompiler listing.

Related concepts

[“Integrated CICS translator” on page 488](#)

[“Db2 coprocessor” on page 493](#)

Related tasks

[“Compiling with the CICS option” on page 487](#)

[“Compiling with the SQL option” on page 497](#)

Related references

[“Processing of INEXIT” on page 812](#)

[“Processing of LIBEXIT” on page 813](#)

[“Processing of PRTEXTIT” on page 816](#)

[“Processing of ADEXIT” on page 818](#)

[“Processing of MSGEXIT” on page 819](#)

Appendix F. JNI.cpy copybook

This listing shows the JNI.cpy copybook, which you can use to access the Java Native Interface (JNI) services from your COBOL programs.

JNI.cpy contains sample COBOL data definitions that correspond to the Java JNI types, and contains JNINativeInterface, the JNI environment structure that contains function pointers for accessing the JNI callable services.

JNI.cpy is in the z/OS UNIX file system in the include subdirectory of the COBOL install directory (typically /usr/lpp/cobol/include). JNI.cpy is analogous to the header file jni.h that C programmers use to access the JNI.

```
*****
* COBOL declarations for Java native method interoperation      *
* *                                                             *
* To use the Java Native Interface callable services from a    *
* COBOL program:                                              *
* 1) Use a COPY statement to include this file into the        *
* the Linkage Section of the program, e.g.                    *
* Linkage Section.                                           *
* Copy JNI                                                    *
* 2) Code the following statements at the beginning of the     *
* Procedure Division:                                         *
* Set address of JNIEnv to JNIEnvPtr                          *
* Set address of JNINativeInterface to JNIEnv                  *
*****
*
* Sample JNI type definitions in COBOL
*
*01 jboolean1 pic X.
* 88 jboolean1-true value X'01' through X'FF'.
* 88 jboolean1-false value X'00'.
*
*01 jbyte1 pic X.
*
*01 jchar1 pic N usage national.
*
*01 jshort1 pic s9(4) comp-5.
*01 jint1 pic s9(9) comp-5.
*01 jlong1 pic s9(18) comp-5.
*
*01 jfloat1 comp-1.
*01 jdouble1 comp-2.
*
*01 jobject1 object reference.
*01 jclass1 object reference.
*01 jstring1 object reference jstring.
*01 jarray1 object reference jarray.
*
*01 jbooleanArray1 object reference jbooleanArray.
*01 jbyteArray1 object reference jbyteArray.
*01 jcharArray1 object reference jcharArray.
*01 jshortArray1 object reference jshortArray.
*01 jintArray1 object reference jintArray.
*01 jlongArray1 object reference jlongArray.
*01 floatArray1 object reference floatArray.
*01 jdoubleArray1 object reference jdoubleArray.
*01 jobjectArray1 object reference jobjectArray.

01 JNIenv pointer.

* JNI Native Method Interface - environment structure.
01 JNINativeInterface.
02 pointer.
02 pointer.
02 pointer.
02 pointer.
02 GetVersion function-pointer.
02 DefineClass function-pointer.
02 FindClass function-pointer.
02 FromReflectedMethod function-pointer.
02 FromReflectedField function-pointer.
```

02 ToReflectedMethod	function-pointer.
02 GetSuperclass	function-pointer.
02 IsAssignableFrom	function-pointer.
02 ToReflectedField	function-pointer.
02 Throw	function-pointer.
02 ThrowNew	function-pointer.
02 ExceptionOccurred	function-pointer.
02 ExceptionDescribe	function-pointer.
02 ExceptionClear	function-pointer.
02 FatalError	function-pointer.
02 PushLocalFrame	function-pointer.
02 PopLocalFrame	function-pointer.
02 NewGlobalRef	function-pointer.
02 DeleteGlobalRef	function-pointer.
02 DeleteLocalRef	function-pointer.
02 IsSameObject	function-pointer.
02 NewLocalRef	function-pointer.
02 EnsureLocalCapacity	function-pointer.
02 AllocObject	function-pointer.
02 NewObject	function-pointer.
02 NewObjectV	function-pointer.
02 NewObjectA	function-pointer.
02 GetObjectClass	function-pointer.
02 IsInstanceOf	function-pointer.
02 GetMethodID	function-pointer.
02 CallObjectMethod	function-pointer.
02 CallObjectMethodV	function-pointer.
02 CallObjectMethodA	function-pointer.
02 CallBooleanMethod	function-pointer.
02 CallBooleanMethodV	function-pointer.
02 CallBooleanMethodA	function-pointer.
02 CallByteMethod	function-pointer.
02 CallByteMethodV	function-pointer.
02 CallByteMethodA	function-pointer.
02 CallCharMethod	function-pointer.
02 CallCharMethodV	function-pointer.
02 CallCharMethodA	function-pointer.
02 CallShortMethod	function-pointer.
02 CallShortMethodV	function-pointer.
02 CallShortMethodA	function-pointer.
02 CallIntMethod	function-pointer.
02 CallIntMethodV	function-pointer.
02 CallIntMethodA	function-pointer.
02 CallLongMethod	function-pointer.
02 CallLongMethodV	function-pointer.
02 CallLongMethodA	function-pointer.
02 CallFloatMethod	function-pointer.
02 CallFloatMethodV	function-pointer.
02 CallFloatMethodA	function-pointer.
02 CallDoubleMethod	function-pointer.
02 CallDoubleMethodV	function-pointer.
02 CallDoubleMethodA	function-pointer.
02 CallVoidMethod	function-pointer.
02 CallVoidMethodV	function-pointer.
02 CallVoidMethodA	function-pointer.
02 CallNonvirtualObjectMethod	function-pointer.
02 CallNonvirtualObjectMethodV	function-pointer.
02 CallNonvirtualObjectMethodA	function-pointer.
02 CallNonvirtualBooleanMethod	function-pointer.
02 CallNonvirtualBooleanMethodV	function-pointer.
02 CallNonvirtualBooleanMethodA	function-pointer.
02 CallNonvirtualByteMethod	function-pointer.
02 CallNonvirtualByteMethodV	function-pointer.
02 CallNonvirtualByteMethodA	function-pointer.
02 CallNonvirtualCharMethod	function-pointer.
02 CallNonvirtualCharMethodV	function-pointer.
02 CallNonvirtualCharMethodA	function-pointer.
02 CallNonvirtualShortMethod	function-pointer.
02 CallNonvirtualShortMethodV	function-pointer.
02 CallNonvirtualShortMethodA	function-pointer.
02 CallNonvirtualIntMethod	function-pointer.
02 CallNonvirtualIntMethodV	function-pointer.
02 CallNonvirtualIntMethodA	function-pointer.
02 CallNonvirtualLongMethod	function-pointer.
02 CallNonvirtualLongMethodV	function-pointer.
02 CallNonvirtualLongMethodA	function-pointer.
02 CallNonvirtualFloatMethod	function-pointer.
02 CallNonvirtualFloatMethodV	function-pointer.
02 CallNonvirtualFloatMethodA	function-pointer.
02 CallNonvirtualDoubleMethod	function-pointer.
02 CallNonvirtualDoubleMethodV	function-pointer.
02 CallNonvirtualDoubleMethodA	function-pointer.

02 CallNonvirtualVoidMethod	function-pointer.
02 CallNonvirtualVoidMethodV	function-pointer.
02 CallNonvirtualVoidMethodA	function-pointer.
02 GetFieldID	function-pointer.
02 GetObjectField	function-pointer.
02 GetBooleanField	function-pointer.
02 GetByteField	function-pointer.
02 GetCharField	function-pointer.
02 GetShortField	function-pointer.
02 GetIntField	function-pointer.
02 GetLongField	function-pointer.
02 GetFloatField	function-pointer.
02 GetDoubleField	function-pointer.
02 SetObjectField	function-pointer.
02 SetBooleanField	function-pointer.
02 SetByteField	function-pointer.
02 SetCharField	function-pointer.
02 SetShortField	function-pointer.
02 SetIntField	function-pointer.
02 SetLongField	function-pointer.
02 SetFloatField	function-pointer.
02 SetDoubleField	function-pointer.
02 GetStaticMethodID	function-pointer.
02 CallStaticObjectMethod	function-pointer.
02 CallStaticObjectMethodV	function-pointer.
02 CallStaticObjectMethodA	function-pointer.
02 CallStaticBooleanMethod	function-pointer.
02 CallStaticBooleanMethodV	function-pointer.
02 CallStaticBooleanMethodA	function-pointer.
02 CallStaticByteMethod	function-pointer.
02 CallStaticByteMethodV	function-pointer.
02 CallStaticByteMethodA	function-pointer.
02 CallStaticCharMethod	function-pointer.
02 CallStaticCharMethodV	function-pointer.
02 CallStaticCharMethodA	function-pointer.
02 CallStaticShortMethod	function-pointer.
02 CallStaticShortMethodV	function-pointer.
02 CallStaticShortMethodA	function-pointer.
02 CallStaticIntMethod	function-pointer.
02 CallStaticIntMethodV	function-pointer.
02 CallStaticIntMethodA	function-pointer.
02 CallStaticLongMethod	function-pointer.
02 CallStaticLongMethodV	function-pointer.
02 CallStaticLongMethodA	function-pointer.
02 CallStaticFloatMethod	function-pointer.
02 CallStaticFloatMethodV	function-pointer.
02 CallStaticFloatMethodA	function-pointer.
02 CallStaticDoubleMethod	function-pointer.
02 CallStaticDoubleMethodV	function-pointer.
02 CallStaticDoubleMethodA	function-pointer.
02 CallStaticVoidMethod	function-pointer.
02 CallStaticVoidMethodV	function-pointer.
02 CallStaticVoidMethodA	function-pointer.
02 GetStaticFieldID	function-pointer.
02 GetStaticObjectField	function-pointer.
02 GetStaticBooleanField	function-pointer.
02 GetStaticByteField	function-pointer.
02 GetStaticCharField	function-pointer.
02 GetStaticShortField	function-pointer.
02 GetStaticIntField	function-pointer.
02 GetStaticLongField	function-pointer.
02 GetStaticFloatField	function-pointer.
02 GetStaticDoubleField	function-pointer.
02 SetStaticObjectField	function-pointer.
02 SetStaticBooleanField	function-pointer.
02 SetStaticByteField	function-pointer.
02 SetStaticCharField	function-pointer.
02 SetStaticShortField	function-pointer.
02 SetStaticIntField	function-pointer.
02 SetStaticLongField	function-pointer.
02 SetStaticFloatField	function-pointer.
02 SetStaticDoubleField	function-pointer.
02 NewString	function-pointer.
02 GetStringLength	function-pointer.
02 GetStringChars	function-pointer.
02 ReleaseStringChars	function-pointer.
02 NewStringUTF	function-pointer.
02 GetStringUTFLength	function-pointer.
02 GetStringUTFChars	function-pointer.
02 ReleaseStringUTFChars	function-pointer.
02 GetArrayLength	function-pointer.
02 NewObjectArray	function-pointer.

02	GetObjectArrayElement	function-pointer.
02	SetObjectArrayElement	function-pointer.
02	NewBooleanArray	function-pointer.
02	NewByteArray	function-pointer.
02	NewCharArray	function-pointer.
02	NewShortArray	function-pointer.
02	NewIntArray	function-pointer.
02	NewLongArray	function-pointer.
02	NewFloatArray	function-pointer.
02	NewDoubleArray	function-pointer.
02	GetBooleanArrayElements	function-pointer.
02	GetByteArrayElements	function-pointer.
02	GetCharArrayElements	function-pointer.
02	GetShortArrayElements	function-pointer.
02	GetIntArrayElements	function-pointer.
02	GetLongArrayElements	function-pointer.
02	GetFloatArrayElements	function-pointer.
02	GetDoubleArrayElements	function-pointer.
02	ReleaseBooleanArrayElements	function-pointer.
02	ReleaseByteArrayElements	function-pointer.
02	ReleaseCharArrayElements	function-pointer.
02	ReleaseShortArrayElements	function-pointer.
02	ReleaseIntArrayElements	function-pointer.
02	ReleaseLongArrayElements	function-pointer.
02	ReleaseFloatArrayElements	function-pointer.
02	ReleaseDoubleArrayElements	function-pointer.
02	GetBooleanArrayRegion	function-pointer.
02	GetByteArrayRegion	function-pointer.
02	GetCharArrayRegion	function-pointer.
02	GetShortArrayRegion	function-pointer.
02	GetIntArrayRegion	function-pointer.
02	GetLongArrayRegion	function-pointer.
02	GetFloatArrayRegion	function-pointer.
02	GetDoubleArrayRegion	function-pointer.
02	SetBooleanArrayRegion	function-pointer.
02	SetByteArrayRegion	function-pointer.
02	SetCharArrayRegion	function-pointer.
02	SetShortArrayRegion	function-pointer.
02	SetIntArrayRegion	function-pointer.
02	SetLongArrayRegion	function-pointer.
02	SetFloatArrayRegion	function-pointer.
02	SetDoubleArrayRegion	function-pointer.
02	RegisterNatives	function-pointer.
02	UnregisterNatives	function-pointer.
02	MonitorEnter	function-pointer.
02	MonitorExit	function-pointer.
02	GetJavaVM	function-pointer.
02	GetStringRegion	function-pointer.
02	GetStringUTFRegion	function-pointer.
02	GetPrimitiveArrayCritical	function-pointer.
02	ReleasePrimitiveArrayCritical	function-pointer.
02	GetStringCritical	function-pointer.
02	ReleaseStringCritical	function-pointer.
02	NewWeakGlobalRef	function-pointer.
02	DeleteWeakGlobalRef	function-pointer.
02	ExceptionCheck	function-pointer.
02	NewDirectByteBuffer	function-pointer.
02	GetDirectBufferAddress	function-pointer.
02	GetDirectBufferCapacity	function-pointer.
02	GetObjectRefType	function-pointer.
02	GetModule	function-pointer.

Related tasks

[“Compiling OO applications under z/OS UNIX” on page 291](#)

[“Accessing JNI services” on page 707](#)

Appendix G. COBOL SYSADATA file contents

When you use the ADATA compiler option, the compiler produces a file, the SYSADATA file, that contains additional program data. You can use this file instead of the compiler listing to extract information about the program. For example, you can extract information about the program for symbolic debugging tools or cross-reference tools.

[“Example: SYSADATA” on page 839](#)

Related references

[“ADATA” on page 335](#)

[“Compiler options that affect the SYSADATA file” on page 837](#)

[“SYSADATA record types” on page 838](#)

[“SYSADATA record descriptions” on page 840](#)

Compiler options that affect the SYSADATA file

Several compiler options could affect the contents of the SYSADATA file.

COMPILE

NOCOMPILE (W|E|S) might stop compilation prematurely, resulting in the loss of specific messages.

EXIT

INEXIT prohibits identification of the compilation source file.

LANGUAGE

LANGUAGE controls the message text (Uppercase English, Mixed-Case English, or Japanese).

Selection of Japanese could result in DBCS characters written to Error Identification records.

NUM

NUM causes the compiler to use the contents of columns 1-6 in the source records for line numbering, rather than using generated sequence numbers. Any invalid (nonnumeric) or out-of-sequence numbers are replaced with a number one higher than that of the previous record.

TEST

TEST causes additional object text records to be created that also affect the contents of the SYSADATA file.

The SYSADATA fields shown in the following table contain line numbers whose contents differ depending on the NUM|NONUM setting.

Type	Field	Record
0020	AE_LINE	External Symbol record
0030	ATOK_LINE	Token record
0032	AF_STMT	Source Error record
0038	AS_STMT	Source record
0039	AS_REP_EXP_SLIN	COPY REPLACING record
0039	AS_REP_EXP_ELIN	COPY REPLACING record
0042	ASY_STMT	Symbol record
0044	AX_DEFN	Symbol Cross Reference record
0044	AX_STMT	Symbol Cross Reference record
0046	AN_STMT	Nested Program record

The Type 0038 Source record contains two fields that relate to line numbers and record numbers:

- AS_STMT contains the compiler line number in both the NUM and NONUM cases.
- AS_CUR_REC# contains the physical source record number.

These two fields can always be used to correlate the compiler line numbers, used in all the above fields, with physical source record numbers.

The remaining compiler options have no direct effect on the SYSADATA file, but might trigger generation of additional error messages associated with the specific option, such as FLAGSTD or SSRANGE.

[“Example: SYSADATA” on page 839](#)

Related references

[“SYSADATA record types” on page 838](#)

[“COMPILE” on page 345](#)

[“EXIT” on page 357](#)

[“LANGUAGE” on page 370](#)

[“NUMBER” on page 378](#)

[“TEST” on page 404](#)

SYSADATA record types

The SYSADATA file contains records classified into different record types. Each type of record provides information about the COBOL program being compiled.

Each record consists of two parts:

- A 12-byte header section, which has the same structure for all record types, and contains the record code that identifies the type of record
- A variable-length data section, which varies by record type

Record type	What it does
“Job identification record: X'0000” on page 843	Provides information about the environment used to process the source data
“ADATA identification record: X'0001” on page 844	Provides common information about the records in the SYSADATA file
“Compilation unit start end record: X'0002” on page 844	Marks the beginning and ending of compilation units in a source file
“Options record: X'0010” on page 845	Describes the compiler options used for the compilation
“External symbol record: X'0020” on page 857	Describes all external names in the program, definitions, and references
“Parse tree record: X'0024” on page 858	Defines a node in the parse tree of the program
“Token record: X'0030” on page 885	Defines a source token
“Source error record: X'0032” on page 913	Describes errors in source program statements
“Source record: X'0038” on page 913	Describes a single source line
“COPY REPLACING record: X'0039” on page 914	Describes an instance of text replacement as a result of a match of COPY . . .REPLACING <i>operand-1</i> with text in the copybook
“Symbol record: X'0042” on page 915	Describes a single symbol defined in the program. There is one symbol record for each symbol defined in the program.

Table 127. **SYSADATA record types** (continued)

Record type	What it does
“Symbol cross-reference record: X'0044” on page 927	Describes references to a single symbol
“Nested program record: X'0046” on page 929	Describes the name and nesting level of a program
“Library record: X'0060” on page 929	Describes the library files and members used from each library
“Statistics record: X'0090” on page 930	Describes the statistics about the compilation
“EVENTS record: X'0120” on page 930	EVENTS records provide compatibility with COBOL/370. The record format is identical with that in COBOL/370, with the addition of the standard ADATA header at the beginning of the record and a field indicating the length of the EVENTS record data.

Example: SYSADATA

The following sample shows part of the listing of a COBOL program. If this COBOL program were compiled with the ADATA option, the records produced in the associated data file would be in the sequence shown in the table below.

```

000001      IDENTIFICATION DIVISION.                AD000020
000002      PROGRAM-ID. AD04202.                    AD000030
000003      ENVIRONMENT DIVISION.                  AD000040
000004      DATA DIVISION.                        AD000050
000005      WORKING-STORAGE SECTION.              AD000060
000006      77  COMP3-FLD2  pic  S9(3)v9.          AD000070
000007      PROCEDURE DIVISION.                   AD000080
000008      STOP  RUN.

```

Type	Description
X'0120'	EVENTS Timestamp record
X'0120'	EVENTS Processor record
X'0120'	EVENTS File-ID record
X'0120'	EVENTS Program record
X'0001'	ADATA Identification record
X'0000'	Job Identification record
X'0010'	Options record
X'0038'	Source record for statement 1
X'0038'	Source record for statement 2
X'0038'	Source record for statement 3
X'0038'	Source record for statement 4
X'0038'	Source record for statement 5
X'0038'	Source record for statement 6
X'0038'	Source record for statement 7
X'0038'	Source record for statement 8

Type	Description
X'0020'	External Symbol record for AD04202
X'0044'	Symbol Cross Reference record for STOP
X'0044'	Symbol Cross Reference record for COMP3-FLD2
X'0044'	Symbol Cross Reference record for AD04202
X'0042'	Symbol record for AD04202
X'0042'	Symbol record for COMP3-FLD2
X'0090'	Statistics record
X'0120'	EVENTS FileEnd record

Related references

[“SYSADATA record descriptions” on page 840](#)

SYSADATA record descriptions

The formats of the records written to the associated data file are shown in the Related references below.

In the fields described in each of the record types, these symbols occur:

C

Indicates character (EBCDIC or ASCII) data

H

Indicates 2-byte binary integer data

F

Indicates 4-byte binary integer data

A

Indicates 4-byte binary integer address and offset data

X

Indicates hexadecimal (bit) data or 1-byte binary integer data

No boundary alignments are implied by any data type, and the implied lengths above might be changed by the presence of a length indicator (*Ln*). All integer data is in big-endian or little-endian format depending on the indicator bit in the header flag byte. *Big-endian* format means that bit 0 is always the most significant bit and bit *n* is the least significant bit. *Little-endian* refers to "byte-reversed" integers as seen on Intel processors.

All undefined fields and unused values are reserved.

Related references

[“Common header section” on page 841](#)

[“Job identification record: X'0000” on page 843](#)

[“ADATA identification record: X'0001” on page 844](#)

[“Compilation unit start | end record: X'0002” on page 844](#)

[“Options record: X'0010” on page 845](#)

[“External symbol record: X'0020” on page 857](#)

[“Parse tree record: X'0024” on page 858](#)

[“Token record: X'0030” on page 885](#)

[“Source error record: X'0032” on page 913](#)

[“Source record: X'0038” on page 913](#)

[“COPY REPLACING record: X'0039” on page 914](#)

[“Symbol record: X'0042” on page 915](#)

[“Symbol cross-reference record: X'0044” on page 927](#)

[“Nested program record: X'0046” on page 929](#)

“Library record: X'0060” on page 929
“Statistics record: X'0090” on page 930
“EVENTS record: X'0120” on page 930

Common header section

The table below shows the format of the header section that is common for all record types. For MVS and VSE, each record is preceded by a 4-byte RDW (record-descriptor word) that is normally used only by access methods and stripped off by download utilities.

Field	Size	Description
Language code	XL1	16 High Level Assembler 17 COBOL on all platforms 40 PL/I on supported platforms

Table 128. **SYSADATA common header section** (continued)

Field	Size	Description
Record type	HL2	The record type, which can be any of the following ones: X'0000' Job Identification record ¹ X'0001' ADATA Identification record X'0002' Compilation unit start/end record X'0010' Options record ¹ X'0020' External Symbol record X'0024' Parse Tree record X'0030' Token record X'0032' Source Error record X'0038' Source record X'0039' COPY REPLACING record X'0042' Symbol record X'0044' Symbol Cross-Reference record X'0046' Nested Program record X'0060' Library record X'0090' Statistics record ¹ X'0120' EVENTS record
Associated data architecture level	XL1	3 Definition level for the header structure
Flag	XL11. ADATA record integers are in little-endian (Intel) format1 This record is continued in the next record 1111 11.. Reserved for future use
Associated data record edition level	XL1	Used to indicate a new format for a specific record type, usually 0
Reserved	CL4	Reserved for future use

Table 128. **SYSADATA common header section** (continued)

Field	Size	Description
Associated data field length	HL2	The length in bytes of the data following the header
1. When a batch compilation (sequence of programs) is run with the ADATA option, there will be multiple Job Identification, Options, and Statistics records for each compilation.		

The mapping of the 12-byte header does not include the area used for the variable-length record-descriptor word required by the access method on MVS and VSE.

Job identification record: X'0000'

The following table shows the contents of the job identification record.

Table 129. **SYSADATA job identification record**

Field	Size	Description
Date	CL8	The date of the compilation in the format YYYYMMDD
Time	CL4	The time of the compilation in the format HHMM
Product number	CL8	The product number of the compiler that produced the associated data file
Product version	CL8	The version number of the product that produced the associated data file, in the form V.R.M
BLD Level	CL8	The build level information (of the form PYYMMDD) of the product that produced the associated data file
System ID	CL24	The system identification of the system on which the compilation was run
Job name	CL8	The MVS job name of the compilation job
Step name	CL8	The MVS step name of the compilation step
Proc step	CL8	The MVS procedure step name of the compilation procedure
Number of input files ¹	HL2	The number of input files recorded in this record. The following group of seven fields will occur <i>n</i> times depending on the value in this field.
...Input file number	HL2	The assigned sequence number of the file
...Input file name length	HL2	The length of the following input file name
...Volume serial number length	HL2	The length of the volume serial number
...Member name length	HL2	The length of the member name
...Input file name	CL(<i>n</i>)	The name of the input file for the compilation
...Volume serial number	CL(<i>n</i>)	The volume serial number of the (first) volume on which the input file resides
...Member name	CL(<i>n</i>)	Where applicable, the name of the member in the input file

Table 129. **SYSADATA job identification record** (continued)

Field	Size	Description
<p>1. Where the number of input files would exceed the record size for the associated data file, the record is continued on the next record. The current number of input files (for that record) is stored in the record, and the record is written to the associated data file. The next record contains the rest of the input files. The count of the number of input files is a count for the current record.</p>		

ADATA identification record: X'0001'

The following table shows the contents of the ADATA identification record.

Table 130. **ADATA identification record**

Field	Size	Description
Time (binary)	XL8	Universal Time (UT) as a binary number of microseconds since midnight Greenwich Mean Time, with the low-order bit representing 1 microsecond. This time can be used as a time-zone-independent time stamp.
CCSID ¹	XL2	Coded Character Set Identifier
Character-set flags	XL1	X'80' EBCDIC (IBM-037) X'40' ASCII (IBM-1252)
Code-page name length	XL2	Length of the code-page name that follows
Code-page name	CL(n)	Name of the code page
<p>1. The appropriate CCS flag will always be set. If the CCSID is set to nonzero, the code-page name length will be zero. If the CCSID is set to zero, the code-page name length will be nonzero and the code-page name will be present.</p>		

Compilation unit start | end record: X'0002'

The following table shows the contents of the compilation unit start|end record.

Table 131. **SYSADATA compilation unit start | end record**

Field	Size	Description
Type	HL2	Compilation unit type, which is one of the following options: X'0000' Start compilation unit X'0001' End compilation unit
Reserved	CL2	Reserved for future use
Reserved	FL4	Reserved for future use

Options record: X'0010'

The following table shows the contents of the options record.

<i>Table 132. SYSADATA options record</i>		
Field	Size	Description
Option byte 0	XL1	<p>1... .. Bit 1 = COPYLOC, Bit 0 = NOCOPYLOC</p> <p>.1... .. Bit 1 = DEFINE, Bit 0 = NODEFINE</p> <p>..1. Bit 1 = SOURCE (HEX), Bit 0 = SOURCE (DEC) or NOSOURCE</p> <p>...1 Bit 1 = INVDATA, Bit 0 = NOINVDATA</p> <p>.... .1... Bit 1 = JAVAIOP, Bit 0 = NOJAVAIOP</p> <p>.... ..1. Bit 1 = CONDCOMP (SKIPSR), Bit 0 = CONDCOMP (NOSKIPSR)</p> <p>.... 11.1 Reserved for future use</p>
Option byte 1	XL1	<p>1... .. Bit 1 = DECK, Bit 0 = NODECK</p> <p>.1... .. Bit 1 = ADATA, Bit 0 = NOADATA</p> <p>..1. Bit 1 = COLLSEQ (EBCDIC), Bit 0 = COLLSEQ (LOCALE BINARY) (AIX only)</p> <p>...1 Bit 1 = SEPOBJ, Bit 0 = NOSEPOBJ (AIX only)</p> <p>.... 1... Bit 1 = NAME, Bit 0 = NONAME</p> <p>.... .1.. Bit 1 = OBJECT, Bit 0 = NOOBJECT</p> <p>.... ..1. Bit 1 = SQL, Bit 0 = NOSQL</p> <p>.... ...1 Bit 1 = CICS, Bit 0 = NOCICS</p>

Table 132. **SYSADATA options record** (continued)

Field	Size	Description
Option byte 2	XL1	<p>1... .. Bit 1 = OFFSET, Bit 0 = NOOFFSET</p> <p>.1... .. Bit 1 = MAP, Bit 0 = NOMAP</p> <p>..1. Bit 1 = LIST, Bit 0 = NOLIST</p> <p>...1 Bit 1 = DBCSXREF, Bit 0 = NODBCSXREF</p> <p>.... 1... Bit 1 = XREF (SHORT), Bit 0 = not XREF (SHORT). This flag should be used in combination with the flag at bit 7. XREF (FULL) is indicated by this flag being off and the flag at bit 7 being on.</p> <p>.... .1.. Bit 1 = SOURCE, Bit 0 = NOSOURCE</p> <p>.... ..1. Bit 1 = VBREF, Bit 0 = NOVBREF</p> <p>.... ...1 Bit 1 = XREF, Bit 0 = not XREF. See also flag at bit 4 above.</p>
Option byte 3	XL1	<p>1... .. Bit 1 = FLAG imbedded diagnostics level specified (a value y is specified as in FLAG(x,y))</p> <p>.1... .. Bit 1 = FLAGSTD, Bit 0 = NOFLAGSTD</p> <p>..1. Bit 1 = NUM, Bit 0 = NONUM</p> <p>...1 Bit 1 = SEQUENCE, Bit 0 = NOSEQUENCE</p> <p>.... 1... Bit 1 = SOSI, Bit 0 = NOSOSI (AIX only)</p> <p>.... .1.. Bit 1 = NSYMBOL (NATIONAL), Bit 0 = NSYMBOL (DBCS)</p> <p>.... ..1. Bit 1 = PROFILE, Bit 0 = NOPROFILE (AIX only)</p> <p>.... ...1 Bit 1 = WORD, Bit 0 = NOWORD</p>

Table 132. **SYSADATA options record** (continued)

Field	Size	Description
Option byte 4	XL1	<p>1... .. Bit 1 = ADV, Bit 0 = NOADV</p> <p>.1... .. Bit 1 = APOST, Bit 0 = QUOTE</p> <p>..1. Bit 1 = DYNAM, Bit 0 = NODYNAM</p> <p>...1 Bit 1 = AWO, Bit 0 = NOAWO</p> <p>.... 1... Bit 1 = RMODE specified, Bit 0 = RMODE (AUTO)</p> <p>.... .1.. Bit 1 = RENT, Bit 0 = NORENT</p> <p>.... ..1. Bit 1 = RES: this flag will always be set on for COBOL.</p> <p>.... ...1 Bit 1 = RMODE (24), Bit 0 = RMODE (ANY)</p>
Option byte 5	XL1	<p>1... .. Bit 1 = SQLCCSID, Bit 0 = NOSQLCCSID</p> <p>.1... .. Bit 1 = OPT (1 2), Bit 0 = OPT (0)</p> <p>..1. Bit 1 = SQLIMS, Bit 0 = NOSQLIMS</p> <p>...1 Bit 1 = DBCS, Bit 0 = NODBCS</p> <p>.... 1... Bit 1 = AFP (VOLATILE), Bit 0 = AFP (NOVOLATILE)</p> <p>.... .1.. Bit 1 = SSRANGE, Bit 0 = NOSSRANGE</p> <p>.... ..1. Bit 1 = TEST, Bit 0 = NOTEST</p> <p>.... ...1 Bit 1 = PROBE, Bit 0 = NOPROBE (Windows only)</p>

Table 132. **SYSADATA options record** (continued)

Field	Size	Description
Option byte 6	XL1	<p>1... .. Bit 1 = SRCFORMAT (EXTEND), Bit 0 = SRCFORMAT (COMPAT)</p> <p>..1. Bit 1 = NUMPROC (PFD), Bit 0 = NUMPROC (NOPFD)</p> <p>...1 Bit 1 = NUMCLS (ALT), Bit 0 = NUMCLS (PRIM)</p> <p>.... .1.. Bit 1 = BINARY (S390), Bit 0 = BINARY (NATIVE) (AIX only)</p> <p>.... ..1. Bit 1 = TRUNC (STD), Bit 0 = TRUNC (OPT)</p> <p>.... ...1 Bit 1 = ZWB, Bit 0 = NOZWB</p> <p>.1.. 1... Reserved for future use</p>
Option byte 7	XL1	<p>1... .. Bit 1 = ALLOWCBL, Bit 0 = NOALLOWCBL</p> <p>.1.. Bit 1 = TERM, Bit 0 = NOTERM</p> <p>..1. Bit 1 = DUMP, Bit 0 = NODUMP</p> <p>.... ..1. Bit 1 = CURRENCY, Bit 0 = NOCURRENCY</p> <p>...1 11.1 Reserved for future use</p>
Option byte 8	XL1	<p>1... .. Bit 1 = RULES, Bit 0 = NORULES</p> <p>.1.. Bit 1 = OPTFILE, Bit 0 = not OPTFILE</p> <p>..1. Bit 1 = ADDR (64), Bit 0 = ADDR (32) (AIX only) Bit 1 = LP (64), Bit 0 = LP (32) (z/OS only)</p> <p>.... 1... Bit 1 = BLOCK0, Bit 0 = NOBLOCK0</p> <p>.... ..1. Bit 1 = DISPSIGN (SEP), Bit 0 = DISPSIGN (COMPAT)</p> <p>.... ...1 Bit 1 = STGOPT, Bit 0 = NOSTGOPT</p> <p>...1 .1.. Reserved for future use</p>

Table 132. **SYSADATA options record** (continued)

Field	Size	Description
Option byte 9	XL1	<p>1... .. Bit 1 = DATA (24), Bit 0 = DATA (31)</p> <p>.1... .. Bit 1 = FASTSRT, Bit 0 = NOFASTSRT</p> <p>.... .1.. Bit 1 = THREAD, Bit 0 = NOTHREAD</p> <p>...1 1.11 Reserved for future use</p>
Option byte A	XL1	<p>1... .. Bit 1 = HGPR (PRESERVE), Bit 0 = HGPR (NOPRESERVE)</p> <p>.1... .. Bit 1 = XMLPARSE (XMLSS), Bit 0 = XMLPARSE (COMPAT)</p> <p>..1. Bit 1 = MAP (DEC), Bit 0 = MAP (HEX)</p> <p>....1 Bit 1 = SPOPT, Bit 0 = NOSPOPT</p> <p>.... .1.. Bit 1 = SUPPRESS, Bit 0 = NOSUPPRESS</p> <p>.... ..1. Bit 1 = VSAMOPENFS (SUCC), BIT 0 = VSAMOPENFS (COMPAT)</p> <p>.... 1..1 Reserved for future use</p>
Option byte B	XL1	<p>1111 1111 Reserved for future use</p>
Option byte C	XL1	<p>1... .. Bit 1 = NCOLLSEQ (LOCALE) (AIX only)</p> <p>.1... .. Reserved for future use</p> <p>..1. Bit 1 = INTDATE (LILIAN), Bit 0 = INTDATE (ANSI)</p> <p>....1 Bit 1 = NCOLLSEQ (BINARY) (AIX only)</p> <p>.... 1... Bit 1 = CHAR (EBCDIC), Bit 0 = CHAR (NATIVE) (AIX only)</p> <p>.... .1.. Bit 1 = FLOAT (HEX), Bit 0 = FLOAT (NATIVE) (AIX only)</p> <p>.... ..1. Bit 1 = COLLSEQ (BINARY) (AIX only)</p> <p>.... ...1 Bit 1 = COLLSEQ (LOCALE) (AIX only)</p>

Table 132. **SYSADATA options record** (continued)

Field	Size	Description
Option byte D	XL1	<p>1... .. Bit 1 = DLL, Bit 0 = NODLL</p> <p>.1.. Bit 1 = EXPORTALL, Bit 0 = NOEXPORTALL</p> <p>..1. Bit 1 = CODEPAGE</p> <p>...1 Bit 1 = SOURCEFORMAT (EXTEND), Bit 0 = SOURCEFORMAT (COMPAT) (AIX only)</p> <p>.... 1... Bit 1 = INITCHECK, Bit 0 = NOINITCHECK</p> <p>.... .1.. Bit 1 = INLINE, Bit 0 = NOINLINE</p> <p>.... ..1. Bit 1 = WSCLEAR, Bit 0 = NOWSCLEAR (AIX only)</p> <p>.... ...1 Bit 1 = BEOPT, Bit 0 = NOBEOPT (AIX only)</p>
Option byte E	XL1	<p>1..... Bit 1 = VLR (COMPAT), Bit 0 = VLR (STANDARD)</p> <p>.1.. Bit 1 = DIAGTRUNC, Bit 0 = NODIAGTRUNC</p> <p>.... .1.. Bit 1 = LSTFILE (UTF-8), Bit 0 = LSTFILE (LOCALE) (AIX only)</p> <p>.... ..1. Bit 1 = MDECK, Bit 0 = NOMDECK</p> <p>.... ...1 Bit 1 = MDECK (NOCOMPILE)</p> <p>..11 1... Reserved for future use</p>

Table 132. **SYSADATA options record** (continued)

Field	Size	Description
Option byte F	XL1	<p>1... .. Bit = INITIAL, Bit 0 = NOINITIAL</p> <p>.1... .. Bit 1 = COPYRIGHT, Bit 0 = NOCOPYRIGHT</p> <p>..1. Bit 1 = QUALIFY (EXTEND), Bit 0 = QUALIFY (COMPAT)</p> <p>...1 Bit 1 = SERVICE, Bit 0 = NOSERVICE</p> <p>.... 1... Bit 1 = ZONEDATA (MIG)</p> <p>.... .1.. Bit 1 = ZONEDATA (NOPFD)</p> <p>.... ..1. Bit 1 = NUMCHECK (ZON PAC BIN ABD MSG), Bit 0 = NONUMCHECK</p> <p>.... ...1 Bit 1 = PARMCHECK (ABD MSG), Bit 0 = NOPARMCHECK</p>
Option byte G	XL1	<p>1... .. Bit 1 = NUMCHECK (ZON), Bit 0 = NUMCHECK (NOZON)</p> <p>.1... .. Bit 1 = NUMCHECK (PAC), Bit 0 = NUMCHECK (NOPAC)</p> <p>..1. Bit 1 = NUMCHECK (BIN), Bit 0 = NUMCHECK (NOBIN)</p> <p>...1 Bit 1 = NUMCHECK (MSG), Bit 0 = NUMCHECK (ABD)</p> <p>.... 1... Bit 1 = NUMCHECK (ZON (NOALPHNUM)), Bit 0 = NUMCHECK (ZON (ALPHNUM))</p> <p>.... .1.. Bit 1 = NUMCHECK (ZON (LAX)), Bit 0 = NUMCHECK (ZON (STRICT))</p> <p>.... ..1. Bit 1 = NUMCHECK (BIN (NOTRUNCBIN)), Bit 0 = NUMCHECK (BIN (TRUNCBIN))</p> <p>.... ...1 Reserved for future use</p>
Option byte H	XL1	<p>1... .. Bit 1 = PARMCHECK (ABD), Bit 0 = PARMCHECK (MSG) (if PARMCHECK in effect)</p> <p>.111 1111 Reserved for future use</p>

Table 132. **SYSADATA options record** (continued)

Field	Size	Description
Option byte I	XL1	<p>1... .. Bit 1 = RULES (ENDPERIOD), Bit 0 = RULES (NOENDPERIOD)</p> <p>.1.. Bit 1 = RULES (EVENPACK), Bit 0 = RULES (NOEVENPACK)</p> <p>..1. Bit 1 = RULES (LAXPERF), Bit 0 = RULES (NOLAXPERF)</p> <p>...1 Bit 1 = RULES (SLACKBYTES), Bit 0 = RULES (NOSLACKBYTES)</p> <p>.... 1... Bit 1 = RULES (OMITODOMIN), Bit 0 = RULES (NOOMITODOMIN)</p> <p>.... .1.. Bit 1 = RULES (UNREFALL), Bit 0 = RULES (NOUNREFALL)</p> <p>.... ..1. Bit 1 = RULES (UNREFSRC), Bit 0 = RULES (NOUNREFSRC)</p> <p>.... ...1 Bit 1 = RULES (LAXREDEF), Bit 0 = RULES (NOLAXREDEF)</p>
Option byte J	XL1	<p>1... .. Bit 1 = SSRANGE (ZLEN, x), Bit 0 = SSRANGE (NOZLEN, x)</p> <p>.1.. Bit 1 = SSRANGE (x, ABD), Bit 0 = SSRANGE (x, MSG)</p> <p>..11 1111 Reserved for future use</p>
Option byte K	XL1	<p>1... .. Bit 1 = INVDATA (FORCENUMCMP, x)</p> <p>.1.. .1.. Bit 1 = INVDATA (NOCLEANSIGN, x)</p>
Option byte L	XL1	<p>1... .. Bit 1 = JAVAIOP (OUTPATH, x, x)</p> <p>..1. Bit 1 = JAVAIOP (x, x, JAVA64)</p>
Flag level	XL1	<p>X'00' Flag(I)</p> <p>X'04' Flag(W)</p> <p>X'08' Flag(E)</p> <p>X'0C' Flag(S)</p> <p>X'10' Flag(U)</p> <p>X'FF' No flag</p>

Table 132. **SYSADATA options record** (continued)

Field	Size	Description
Imbedded diagnostic level	XL1	X'00' Flag(I) X'04' Flag(W) X'08' Flag(E) X'0C' Flag(S) X'10' Flag(U) X'FF' Noflag
FLAGSTD (FIPS) specification	XL1	1... .. Minimum .1.. .. Intermediate ..1. High ...1 IBM extensions 1... Level-1 segmentation1.. Level-2 segmentation1. Debugging1 Obsolete
Reserved for flagging	XL1	1111 1111 Reserved for future use
Compiler mode	XL1	X'00' Unconditional Nocompile, Nocompile(I) X'04' Nocompile(W) X'08' Nocompile(E) X'0C' Nocompile(S) X'FF' Compile
Space value	CL1	

Table 132. **SYSADATA options record** (continued)

Field	Size	Description
Data for 3-valued options	XL1	1... .. NAME (ALIAS) specified .1... .. Reserved for future use ..1. TRUNC (BIN) specified ...1 PARMCHECK (ABD) (if PARMCHECK in effect) 1... INITCHECK (STRICT) specified111 Reserved for future use
TEST suboptions	XL1	1... .. TEST (EJPD) .1... .. TEST (SOURCE) ..1. TEST (SEPARATE) ...1 TEST NOTEST (DWARF) 1... TEST (SEPARATE (DSNAME))111 Reserved for future use
OUTDD name length	HL2	Length of OUTDD name
RWT ID Length	HL2	Length of Reserved Word Table identifier
BLD LEVEL	CL8	Product build level information (of the form PYYMMDD)
PGMNAME suboptions	XL1	1... .. Bit 1 = PGMNAME (COMPAT) .1... .. Bit 1 = PGMNAME (LONGUPPER) ..1. Bit 1 = PGMNAME (LONGMIXED) ...1 1111 Reserved for future use
Entry interface suboptions	XL1	1... .. Bit 1 = EntryInterface(System) (Windows only) .1... .. Bit 1 = EntryInterface(OptLink) (Windows only) ..11 1111 Reserved for future use

Table 132. **SYSADATA options record** (continued)

Field	Size	Description
CALLINTERFACE suboptions	XL1	1... .. Bit 1 = CALLINTERFACE(DLL) .1... .. Bit 1 = CALLINTERFACE(DYNAMIC) ..11 1111 Reserved for future use
ARITH suboption	XL1	1... .. Bit 1 = ARITH (COMPAT) .1... .. Bit 1 = ARITH (EXTEND) ..11 1111 Reserved for future use
DBCS Req	FL4	DBCS XREF storage requirement
DBCS ORDPGM length	HL2	Length of name of DBCS Ordering Program
DBCS ENCTBL length	HL2	Length of name of DBCS Encode Table
DBCS ORD TYPE	CL2	DBCS Ordering type
Reserved	CL5	Reserved for future use
Optimize level	XL1	Optimization level $0 \leq n \leq 2$
Converted SO	CL1	Converted SO hexadecimal value
Converted SI	CL1	Converted SI hexadecimal value
Language ID	CL2	This field holds the two-character abbreviation (one of <i>EN</i> , <i>UE</i> , <i>JA</i> , or <i>JP</i>) from the LANGUAGE option.
INEXIT name length	HL2	Length of SYSIN user-exit name
PRTEXIT name length	HL2	Length of SYSPRINT user-exit name
LIBEXIT name length	HL2	Length of Libraryuser-exit name
ADEXIT name length	HL2	Length of ADATA user-exit name
CURROPT	CL5	CURRENCY option value
ARCH	XL1	ARCH level number
TUNE	XL1	TUNE level number
Reserved	CL1	Reserved for future use
CODEPAGE	HL2	CODEPAGE CCSID option value
Reserved	CL50	Reserved for future use
LINECNT	HL2	LINECOUNT value
Reserved	CL2	Reserved for future use
BUFSIZE	FL4	BUFSIZE option value
Reserved	FL4	Reserved for future use

Table 132. **SYSADATA options record** (continued)

Field	Size	Description
Phase residence bits byte 1	XL1	<p>1... .. Bit 1 = IGYCLIBR in user region</p> <p>.1... .. Bit 1 = IGYCSCAN in user region</p> <p>..1. Bit 1 = IGYCDSCN in user region</p> <p>...1 Bit 1 = IGYCGROU in user region</p> <p>.... 1... Bit 1 = IGYCPSCN in user region</p> <p>.... .1.. Bit 1 = IGYCPANA in user region</p> <p>.... ..1. Bit 1 = IGYCFGEN in user region</p> <p>.... ...1 Bit 1 = IGYCPGEN in user region</p>
Phase residence bits byte 2	XL1	<p>.1... .. Bit 1 = IGYCLSTR in user region</p> <p>..1. Bit 1 = IGYCXREF in user region</p> <p>...1 Bit 1 = IGYCDMAP in user region</p> <p>.... ..1. Bit 1 = IGYCDIAG in user region</p> <p>.... ...1 Bit 1 = IGYCDGEN in user region</p> <p>1... 11.. Reserved for future use</p>
Phase residence bits bytes 3 and 4	XL2	Reserved
Reserved	CL4	Reserved for future use
OUTDD name	CL(n)	OUTDD name
RWT	CL(n)	Reserved word table identifier
DBCS ORDPGM	CL(n)	DBCS Ordering program name
DBCS ENCTBL	CL(n)	DBCS Encode table name
INEXIT name	CL(n)	SYSIN user-exit name
PRTEXIT name	CL(n)	SYSPRINT user-exit name
LIBEXIT name	CL(n)	Library user-exit name
ADEXIT name	CL(n)	ADATA user-exit name

External symbol record: X'0020'

The following table shows the contents of the external symbol record.

Field	Size	Description
Section type	XL1	X'00' PROGRAM- ID name (main entry point name) X'01' ENTRY name (secondary entry point name) X'02' External reference (referenced external entry point) X'04' Not applicable for COBOL X'05' Not applicable for COBOL X'06' Not applicable for COBOL X'0A' Not applicable for COBOL X'12' Internal reference (referenced internal subprogram) X'C0' External class-name (OO COBOL class definition) X'C1' METHOD- ID name (OO COBOL method definition) X'C6' Method reference (OO COBOL method reference) X'FF' Not applicable for COBOL Types X'12', X'C0', X'C1' and X'C6' are for COBOL only.
Flags	XL1	Not applicable for COBOL
Reserved	HL2	Reserved for future use
Symbol-ID	FL4	Symbol-ID of program that contains the reference (only for types x'02' and x'12')
Line number	FL4	Line number of statement that contains the reference (only for types x'02' and x'12')
Section length	FL4	Not applicable for COBOL
LD ID	FL4	Not applicable for COBOL
Reserved	CL8	Reserved for future use
External name length	HL2	Number of characters in the external name
Alias name length	HL2	Not applicable for COBOL
External name	CL(n)	The external name
Alias section name	CL(n)	Not applicable for COBOL

Parse tree record: X'0024'

The following table shows the contents of the parse tree record.

<i>Table 134. SYSADATA parse tree record</i>		
Field	Size	Description
Node number	FL4	The node number generated by the compiler, starting at 1
Node type	HL2	The type of the node: 001 Program 002 Class 003 Method
		101 IDENTIFICATION DIVISION 102 ENVIRONMENT DIVISION 103 DATA DIVISION 104 PROCEDURE DIVISION 105 End Program/Method/Class
		201 Declaratives body 202 Nondeclaratives body
		301 Section 302 Procedure section
		401 Paragraph 402 Procedure paragraph

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		<p>501 Sentence</p> <p>502 File definition</p> <p>503 Sort file definition</p> <p>504 Program-name</p> <p>505 Program attribute</p> <p>508 ENVIRONMENT DIVISION clause</p> <p>509 CLASS attribute</p> <p>510 METHOD attribute</p> <p>511 USE statement</p>
		<p>601 Statement</p> <p>602 Data description clause</p> <p>603 Data entry</p> <p>604 File description clause</p> <p>605 Data entry name</p> <p>606 Data entry level</p> <p>607 EXEC entry</p>

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		701 EVALUATE subject phrase
		702 EVALUATE WHEN phrase
		703 EVALUATE WHEN OTHER phrase
		704 SEARCH WHEN phrase
		705 INSPECT CONVERTING phrase
		706 INSPECT REPLACING phrase
		707 INSPECT TALLYING phrase
		708 PERFORM UNTIL phrase
		709 PERFORM VARYING phrase
		710 PERFORM AFTER phrase
		711 Statement block
		712 Scope terminator
		713 INITIALIZE REPLACING phrase
		714 EXEC CICS Command
		715 INITIALIZE WITH FILLER
		716 INITIALIZE TO VALUE
		717 INITIALIZE TO DEFAULT
		718 ALLOCATE INITIALIZED
		719 ALLOCATE LOC
		720 DATA DIVISION phrase

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		<p>801 Phrase</p> <p>802 ON phrase</p> <p>803 NOT phrase</p> <p>804 THEN phrase</p> <p>805 ELSE phrase</p> <p>806 Condition</p> <p>807 Expression</p> <p>808 Relative indexing</p> <p>809 EXEC CICS Option</p> <p>810 Reserved word</p> <p>811 INITIALIZE REPLACING category</p>

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		901 Section or paragraph name
		902 Identifier
		903 Alphabet-name
		904 Class-name
		905 Condition-name
		906 File-name
		907 Index-name
		908 Mnemonic-name
		910 Symbolic-character
		911 Literal
		912 Function identifier
		913 Data-name
		914 Special register
		915 Procedure reference
		916 Arithmetic operator
		917 All procedures
		918 INITIALIZE literal (no tokens)
		919 ALL literal or figcon
		920 Keyword class test name
		921 Reserved word at identifier level
		922 Unary operator
		923 Relational operator

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		1001 Subscript 1002 Reference modification
Node subtype	HL2	The subtype of the node. For Section type: 0001 CONFIGURATION Section 0002 INPUT-OUTPUT Section 0003 FILE SECTION 0004 WORKING-STORAGE SECTION 0005 LINKAGE SECTION 0006 LOCAL-STORAGE SECTION 0007 REPOSITORY Section

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		<p>For Paragraph type:</p> <p>0001 PROGRAM-ID paragraph</p> <p>0002 AUTHOR paragraph</p> <p>0003 INSTALLATION paragraph</p> <p>0004 DATE-WRITTEN paragraph</p> <p>0005 SECURITY paragraph</p> <p>0006 SOURCE-COMPUTER paragraph</p> <p>0007 OBJECT-COMPUTER paragraph</p> <p>0008 SPECIAL-NAMES paragraph</p> <p>0009 FILE-CONTROL paragraph</p> <p>0010 I-O-CONTROL paragraph</p> <p>0011 DATE-COMPILED paragraph</p> <p>0012 CLASS-ID paragraph</p> <p>0013 METHOD-ID paragraph</p> <p>0014 REPOSITORY paragraph</p>

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		<p>For ENVIRONMENT DIVISION clause type:</p> <p>0001 WITH DEBUGGING MODE</p> <p>0002 MEMORY-SIZE</p> <p>0003 SEGMENT-LIMIT</p> <p>0004 CURRENCY-SIGN</p> <p>0005 DECIMAL POINT</p> <p>0006 PROGRAM COLLATING SEQUENCE</p> <p>0007 ALPHABET</p> <p>0008 SYMBOLIC-CHARACTER</p> <p>0009 CLASS</p> <p>0010 ENVIRONMENT NAME</p> <p>0011 SELECT</p> <p>0012 XML-SCHEMA</p>

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		<p>For data description clause type:</p> <p>0001 BLANK WHEN ZERO</p> <p>0002 DATA-NAME OR FILLER</p> <p>0003 JUSTIFIED</p> <p>0004 OCCURS</p> <p>0005 PICTURE</p> <p>0006 REDEFINES</p> <p>0007 RENAMES</p> <p>0008 SIGN</p> <p>0009 SYNCHRONIZED</p> <p>0010 USAGE</p> <p>0011 VALUE</p> <p>0012 VOLATILE</p> <p>0013 DYNAMIC LENGTH</p> <p>0023 GLOBAL</p> <p>0024 EXTERNAL</p>

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		<p>For file description clause type:</p> <p>0001 FILE STATUS</p> <p>0002 ORGANIZATION</p> <p>0003 ACCESS MODE</p> <p>0004 RECORD KEY</p> <p>0005 ASSIGN</p> <p>0006 RELATIVE KEY</p> <p>0007 PASSWORD</p> <p>0008 PROCESSING MODE</p> <p>0009 RECORD DELIMITER</p> <p>0010 PADDING CHARACTER</p> <p>0011 BLOCK CONTAINS</p> <p>0012 RECORD CONTAINS</p> <p>0013 LABEL RECORDS</p>

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		<p>0014 VALUE OF</p> <p>0015 DATA RECORDS</p> <p>0016 LINAGE</p> <p>0017 ALTERNATE KEY</p> <p>0018 LINES AT TOP</p> <p>0019 LINES AT BOTTOM</p> <p>0020 CODE-SET</p> <p>0021 RECORDING MODE</p> <p>0022 RESERVE</p> <p>0023 GLOBAL</p> <p>0024 EXTERNAL</p> <p>0025 LOCK</p>

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		For Statement type: 0002 NEXT SENTENCE 0003 ACCEPT 0004 ADD 0005 ALTER 0006 CALL 0007 CANCEL 0008 CLOSE 0009 COMPUTE 0010 CONTINUE 0011 DELETE 0012 DISPLAY 0013 DIVIDE (INTO) 0113 DIVIDE (BY) 0014 ENTER 0015 ENTRY 0016 EVALUATE 0017 EXIT 0018 GO 0019 GOBACK 0020 IF 0021 INITIALIZE 0022 INSPECT

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		<p>0023 INVOKE</p> <p>0024 MERGE</p> <p>0025 MOVE</p> <p>0026 MULTIPLY</p> <p>0027 OPEN</p> <p>0028 PERFORM</p> <p>0029 READ</p> <p>0030 READY</p> <p>0031 RELEASE</p> <p>0032 RESET</p> <p>0033 RETURN</p> <p>0034 REWRITE</p> <p>0035 SEARCH</p> <p>0036 SERVICE</p> <p>0037 SET</p>

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		<p>0038 SORT</p> <p>0039 START</p> <p>0040 STOP</p> <p>0041 STRING</p> <p>0042 SUBTRACT</p> <p>0043 UNSTRING</p> <p>0044 EXEC SQL</p> <p>0144 EXEC CICS</p> <p>0045 WRITE</p> <p>0046 XML</p> <p>0047 ALLOCATE</p> <p>0048 FREE</p> <p>0049 JSON</p>

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		<p>For Phrase type:</p> <p>0001 INTO</p> <p>0002 DELIMITED</p> <p>0003 INITIALIZE. . .REPLACING</p> <p>0004 INSPECT. . .ALL</p> <p>0005 INSPECT. . .LEADING</p> <p>0006 SET. . .TO</p> <p>0007 SET. . .UP</p> <p>0008 SET. . .DOWN</p> <p>0009 PERFORM. . .TIMES</p> <p>0010 DIVIDE. . .REMAINDER</p> <p>0011 INSPECT. . .FIRST</p> <p>0012 SEARCH. . .VARYING</p> <p>0013 MORE-LABELS</p> <p>0014 SEARCH ALL</p> <p>0015 SEARCH. . .AT END</p> <p>0016 SEARCH. . .TEST INDEX</p> <p>0017 GLOBAL</p> <p>0018 LABEL</p>

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		<p>0019 DEBUGGING</p> <p>0020 SEQUENCE</p> <p>0021 Reserved for future use</p> <p>0022 Reserved for future use</p> <p>0023 Reserved for future use</p> <p>0024 TALLYING</p> <p>0025 Reserved for future use</p> <p>0026 ON SIZE ERROR</p> <p>0027 ON OVERFLOW</p> <p>0028 ON ERROR</p> <p>0029 AT END</p> <p>0030 INVALID KEY</p> <p>0031 END-OF-PAGE</p> <p>0032 USING</p>

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		<p>0033 BEFORE</p> <p>0034 AFTER</p> <p>0035 EXCEPTION</p> <p>0036 CORRESPONDING</p> <p>0037 Reserved for future use</p> <p>0038 RETURNING</p> <p>0039 GIVING</p> <p>0040 THROUGH</p> <p>0041 KEY</p> <p>0042 DELIMITER</p> <p>0043 POINTER</p> <p>0044 COUNT</p> <p>0045 METHOD</p> <p>0046 PROGRAM</p> <p>0047 INPUT</p> <p>0048 OUTPUT</p> <p>0049 I-O</p> <p>0050 EXTEND</p> <p>0051 RELOAD</p> <p>0052 ASCENDING</p> <p>0053 DESCENDING</p>

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		0054 DUPLICATES 0055 NATIVE (USAGE) 0056 INDEXED 0057 FROM 0058 FOOTING 0059 LINES AT BOTTOM 0060 LINES AT TOP 0061 XML ENCODING 0062 XML GENERATE XML-DECLARATION 0063 XML GENERATE ATTRIBUTES 0064 XML GENERATE NAMESPACE 0065 XML PARSE PROCESSING 0066 XML PARSE VALIDATING 0067 XML GENERATE NAME 0068 XML GENERATE TYPE 0069 XML GENERATE SUPPRESS 0070 BYTE-LENGTH 0071 LIMIT

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		<p>For Function identifier type:</p> <p>0001 COS</p> <p>0002 LOG</p> <p>0003 MAX</p> <p>0004 MIN</p> <p>0005 MOD</p> <p>0006 ORD</p> <p>0007 REM</p> <p>0008 SIN</p> <p>0009 SUM</p> <p>0010 TAN</p> <p>0011 ACOS</p> <p>0012 ASIN</p> <p>0013 ATAN</p> <p>0014 CHAR</p> <p>0015 MEAN</p> <p>0016 SQRT</p> <p>0017 LOG10</p> <p>0018 RANGE</p>

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		<p>0019 LENGTH</p> <p>0020 MEDIAN</p> <p>0021 NUMVAL</p> <p>0022 RANDOM</p> <p>0023 ANNUITY</p> <p>0024 INTEGER</p> <p>0025 ORD-MAX</p> <p>0026 ORD-MIN</p> <p>0027 REVERSE</p> <p>0028 MIDRANGE</p> <p>0029 NUMVAL-C</p> <p>0030 VARIANCE</p> <p>0031 FACTORIAL</p> <p>0032 LOWER-CASE</p>

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		0033 UPPER-CASE 0034 CURRENT-DATE 0035 INTEGER-PART 0036 PRESENT-VALUE 0037 WHEN-COMPILED 0038 DAY-OF-INTEGER 0039 INTEGER-OF-DAY 0040 DATE-OF-INTEGER 0041 INTEGER-OF-DATE 0042 STANDARD-DEVIATION 0043 YEAR-TO-YYYY 0044 DAY-TO-YYYYDDD 0045 DATE-TO-YYYYMMDD 0049 DISPLAY-OF 0050 NATIONAL-OF 0051 UPOS 0052 UVALID 0053 UWIDTH 0054 ULENGTH 0055 USUBSTR

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		0056 USUPPLEMENTARY
		0057 HEX-OF
		0058 BIT-OF
		0059 E
		0060 TRIM
		0061 PI
		0062 ABS
		0063 BYTE-LENGTH
		0064 EXP
		0065 EXP10
		0066 BIT-TO-CHAR
		0067 NUMVAL-F
		0068 HEX-TO-CHAR
		0069 SIGN
		0070 TEST-NUMVAL
		0071 TEST-NUMVAL-C
		0072 TEST-NUMVAL-F

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		<p>0073 UUID4</p> <p>0074 FORMATTED-DATE</p> <p>0075 FORMATTED-TIME</p> <p>0076 TEST-DAY-YYYYDDD</p> <p>0077 COMBINED-DATETIME</p> <p>0078 FORMATTED-DATETIME</p> <p>0079 TEST-DATE-YYYYMMDD</p> <p>0080 SECONDS-PAST-MIDNIGHT</p> <p>0081 FORMATTED-CURRENT-DATE</p> <p>0082 TEST-FORMATTED-DATETIME</p> <p>0083 INTEGER-OF-FORMATTED-DATE</p> <p>0084 SECONDS-FROM-FORMATTED-TIME</p>
		<p>For Special Register type:</p> <p>0001 ADDRESS OF</p> <p>0002 LENGTH OF</p>

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		For Keyword Class Test Name type: 0001 ALPHABETIC 0002 ALPHABETIC-LOWER 0003 ALPHABETIC-UPPER 0004 DBCS 0005 KANJI 0006 NUMERIC 0007 NEGATIVE 0008 POSITIVE 0009 ZERO
		For Reserved Word type: 0001 TRUE 0002 FALSE 0003 ANY 0004 THRU
		For Identifier, Data-name, Index-name, Condition-name or Mnemonic-name type: 0001 REFERENCED 0002 CHANGED 0003 REFERENCED & CHANGED

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		For Initialize literal type: 0001 ALPHABETIC 0002 ALPHANUMERIC 0003 NUMERIC 0004 ALPHANUMERIC-EDITED 0005 NUMERIC-EDITED 0006 DBCS/EGCS 0007 NATIONAL 0008 NATIONAL-EDITED 0009 UTF-8
		For Procedure-name type: 0001 SECTION 0002 PARAGRAPH

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		<p>For Reserved word at identifier level type:</p> <p>0001 ROUNDED</p> <p>0002 TRUE</p> <p>0003 ON</p> <p>0004 OFF</p> <p>0005 SIZE</p> <p>0006 DATE</p> <p>0007 DAY</p> <p>0008 DAY-OF-WEEK</p> <p>0009 TIME</p> <p>0010 WHEN-COMPILED</p> <p>0011 PAGE</p> <p>0012 DATE YYYYMMDD</p> <p>0013 DAY YYYYDDD</p> <p>0014 Attribute</p> <p>0015 Element</p> <p>0016 Content</p> <p>0017 Numeric</p> <p>0018 Nonnumeric</p> <p>0019 Every</p> <p>0020 When</p>

Table 134. *SYSADATA parse tree record (continued)*

Field	Size	Description
		For Arithmetic Operator type: 0001 PLUS 0002 MINUS 0003 TIMES 0004 DIVIDE 0005 DIVIDE REMAINDER 0006 EXPONENTIATE 0007 NEGATE
		For Relational Operator type: 0008 LESS 0009 LESS OR EQUAL 0010 EQUAL 0011 NOT EQUAL 0012 GREATER 0013 GREATER OR EQUAL 0014 AND 0015 OR 0016 CLASS CONDITION 0017 NOT CLASS CONDITION
Parent node number	FL4	The node number of the parent of the node
Left sibling node number	FL4	The node number of the left sibling of the node, if any. If none, the value is zero.

Table 134. **SYSADATA parse tree record** (continued)

Field	Size	Description
Symbol ID	FL4	The Symbol ID of the node, if it is a user-name of one of the following types: <ul style="list-style-type: none"> • Data entry • Identifier • File-name • Index-name • Procedure-name • Condition-name • Mnemonic-name This value corresponds to the Symbol ID in a Symbol (Type 42) record, except for procedure-names where it corresponds to the Paragraph ID. For all other node types this value is zero.
Section Symbol ID	FL4	The Symbol ID of the section containing the node, if it is a qualified paragraph-name reference. This value corresponds to the Section ID in a Symbol (Type 42) record. For all other node types this value is zero.
First token number	FL4	The number of the first token associated with the node
Last token number	FL4	The number of the last token associated with the node
Reserved	FL4	Reserved for future use
Flags	CL1	Information about the node: <p>X'80' Reserved</p> <p>X'40' Generated node, no tokens</p>
Reserved	CL3	Reserved for future use

Token record: X'0030'

The compiler does not generate token records for any lines that are treated as comment lines, which include, but are not limited to, items in the following list.

- Comment lines, which are source lines that have an asterisk (*) or a slash (/) in column 7
- The following compiler-directing statements:
 - *CBL (*CONTROL)
 - BASIS
 - COPY
 - DELETE
 - EJECT
 - INSERT
 - REPLACE
 - SKIP1

- SKIP2
- SKIP3
- TITLE
- Debugging lines, which are source lines that have a D in column 7, if WITH DEBUGGING MODE is not specified

<i>Table 135. SYSADATA token record</i>		
Field	Size	Description
Token number	FL4	The token number within the source file generated by the compiler, starting at 1. Any copybooks have already been included in the source.

Table 135. **SYSADATA token record** (continued)

Field	Size	Description
Token code	HL2	<p>The type of token (user-name, literal, reserved word, and so forth). For reserved words, the compiler reserved-word table values are used. For PICTURE strings, the special code 0000 is used. For each piece (other than the last) of a continued token, the special code 3333 is used. Otherwise, the following codes are used:</p> <p>0001 ACCEPT</p> <p>0002 ADD</p> <p>0003 ALTER</p> <p>0004 CALL</p> <p>0005 CANCEL</p> <p>0007 CLOSE</p> <p>0009 COMPUTE</p> <p>0011 DELETE</p> <p>0013 DISPLAY</p> <p>0014 DIVIDE</p> <p>0017 READY</p> <p>0018 END-PERFORM</p> <p>0019 ENTER</p> <p>0020 ENTRY</p> <p>0021 EXIT</p>

Table 135. *SYSADATA token record* (continued)

Field	Size	Description
		<p>0022 EXEC EXECUTE</p> <p>0023 GO</p> <p>0024 IF</p> <p>0025 INITIALIZE</p> <p>0026 INVOKE</p> <p>0027 INSPECT</p> <p>0028 MERGE</p> <p>0029 MOVE</p>

Table 135. **SYSADATA token record** (continued)

Field	Size	Description
		0030 MULTIPLY
		0031 OPEN
		0032 PERFORM
		0033 READ
		0035 RELEASE
		0036 RETURN
		0037 REWRITE
		0038 SEARCH
		0040 SET
		0041 SORT
		0042 START
		0043 STOP
		0044 STRING
		0045 SUBTRACT
		0048 UNSTRING
		0049 USE
		0050 WRITE

Table 135. *SYSADATA token record* (continued)

Field	Size	Description
		0051 CONTINUE
		0052 END-ADD
		0053 END-CALL
		0054 END-COMPUTE
		0055 END-DELETE
		0056 END-DIVIDE
		0057 END-EVALUATE
		0058 END-IF
		0059 END-MULTIPLY
		0060 END-READ
		0061 END-RETURN
		0062 END-REWRITE
		0063 END-SEARCH
		0064 END-START
		0065 END-STRING
		0066 END-SUBTRACT
		0067 END-UNSTRING
		0068 END-WRITE
		0069 GOBACK

Table 135. **SYSADATA token record** (continued)

Field	Size	Description
		0070 EVALUATE 0071 RESET 0072 SERVICE 0073 END-INVOKE 0074 END-EXEC 0075 XML 0076 END-XML 0077 ALLOCATE 0078 FREE 0079 JSON 0080 END-JSON 0099 FOREIGN-VERB 0101 DATA-NAME 0105 DASHED-NUM 0106 DECIMAL 0107 DIV-SIGN

Table 135. **SYSADATA token record** (continued)

Field	Size	Description
		0108 EQ
		0109 EXPONENTIATION
		0110 GT
		0111 INTEGER
		0112 LT
		0113 LPAREN
		0114 MINUS-SIGN
		0115 MULT-SIGN
		0116 NONUMLIT
		0117 PERIOD
		0118 PLUS-SIGN
		0121 RPAREN
		0122 SIGNED-INTEGER
		0123 QUID
		0124 COLON
		0125 IEOF
		0126 EGCS-LIT
		0127 COMMA-SPACE
		0128 SEMICOLON-SPACE
		0129 PROCEDURE-NAME
		0130 FLT-POINT-LIT
		0131 Language Environment

Table 135. *SYSADATA token record (continued)*

Field	Size	Description
		0132 GE
		0133 IDREF
		0134 EXPREF
		0136 CICS
		0137 NEW
		0138 NATIONAL-LIT
		0200 ADDRESS
		0201 ADVANCING
		0202 AFTER
		0203 ALL
		0204 ALPHABETIC
		0205 ALPHANUMERIC
		0206 ANY
		0207 AND
		0208 ALPHANUMERIC-EDITED
		0209 BEFORE

Table 135. **SYSADATA** token record (continued)

Field	Size	Description
		0210 BEGINNING
		0211 FUNCTION
		0212 CONTENT
		0213 CORR CORRESPONDING
		0214 DAY
		0215 DATE
		0216 DEBUG-CONTENTS
		0217 DEBUG-ITEM
		0218 DEBUG-LINE
		0219 DEBUG-NAME
		0220 DEBUG-SUB-1
		0221 DEBUG-SUB-2
		0222 DEBUG-SUB-3
		0223 DELIMITED
		0224 DELIMITER
		0225 DOWN

Table 135. **SYSADATA token record** (continued)

Field	Size	Description
		0226 NUMERIC-EDITED
		0227 XML-EVENT
		0228 END-OF-PAGE EOP
		0229 EQUAL
		0230 ERROR
		0231 XML-NTEXT
		0232 EXCEPTION
		0233 EXTEND
		0234 FIRST
		0235 FROM
		0236 GIVING
		0237 GREATER
		0238 I-O
		0239 IN
		0240 INITIAL
		0241 INTO
		0242 INVALID
		0243 SQL
		0244 LESS
		0245 LINAGE-COUNTER
		0246 XML-TEXT
		0247 LOCK

Table 135. *SYSADATA token record* (continued)

Field	Size	Description
		<p>0248 GENERATE</p> <p>0249 NEGATIVE</p> <p>0250 NEXT</p> <p>0251 NO</p> <p>0252 NOT</p> <p>0253 NUMERIC</p> <p>0254 KANJI</p> <p>0255 OR</p> <p>0256 OTHER</p> <p>0257 OVERFLOW</p> <p>0258 PAGE</p> <p>0259 CONVERTING</p>

Table 135. **SYSADATA token record** (continued)

Field	Size	Description
		0260 POINTER
		0261 POSITIVE
		0262 DBCS
		0263 PROCEDURES
		0264 PROCEED
		0265 REFERENCES
		0266 DAY-OF-WEEK
		0267 REMAINDER
		0268 REMOVAL
		0269 REPLACING
		0270 REVERSED
		0271 REWIND
		0272 ROUNDED
		0273 RUN
		0274 SENTENCE
		0275 STANDARD
		0276 RETURN-CODE SORT-CORE-SIZE SORT-FILE-SIZE SORT-MESSAGE SORT-MODE-SIZE SORT-RETURN TALLY XML-CODE

Table 135. *SYSADATA token record* (continued)

Field	Size	Description
		<p>0277 TALLYING</p> <p>0278 SUM</p> <p>0279 TEST</p> <p>0280 THAN</p> <p>0281 UNTIL</p> <p>0282 UP</p> <p>0283 UPON</p> <p>0284 VARYING</p> <p>0285 RELOAD</p> <p>0286 TRUE</p>

Table 135. **SYSADATA token record** (continued)

Field	Size	Description
		0287 THEN
		0288 RETURNING
		0289 ELSE
		0290 SELF
		0291 SUPER
		0292 WHEN-COMPILED
		0293 ENDING
		0294 FALSE
		0295 REFERENCE
		0296 NATIONAL-EDITED
		0297 COM-REG
		0298 ALPHABETIC-LOWER
		0299 ALPHABETIC-UPPER
		0301 REDEFINES
		0302 OCCURS
		0303 SYNC SYNCHRONIZED
		0304 MORE-LABELS
		0305 JUST JUSTIFIED

Table 135. **SYSADATA** token record (continued)

Field	Size	Description
		<p>0306 SHIFT-IN</p> <p>0307 BLANK</p> <p>0308 VALUE</p> <p>0309 COMP COMPUTATIONAL</p> <p>0310 COMP-1 COMPUTATIONAL-1</p> <p>0311 COMP-3 COMPUTATIONAL-3</p> <p>0312 COMP-2 COMPUTATIONAL-2</p> <p>0313 COMP-4 COMPUTATIONAL-4</p> <p>0314 DISPLAY-1</p> <p>0315 SHIFT-OUT</p>

Table 135. **SYSADATA token record** (continued)

Field	Size	Description
		0316 INDEX
		0317 USAGE
		0318 SIGN
		0319 LEADING
		0320 SEPARATE
		0321 INDEXED
		0322 LEFT
		0323 RIGHT
		0324 PIC PICTURE
		0325 VALUES
		0326 GLOBAL
		0327 EXTERNAL
		0328 BINARY
		0329 PACKED-DECIMAL
		0330 EGCS
		0331 PROCEDURE-POINTER
		0332 COMP-5 COMPUTATIONAL-5

Table 135. **SYSADATA token record** (continued)

Field	Size	Description
		<p>0333 FUNCTION-POINTER</p> <p>0334 TYPE</p> <p>0335 JNIENVPTR</p> <p>0336 NATIONAL</p> <p>0337 GROUP-USAGE</p> <p>0342 VOLATILE</p> <p>0401 HIGH-VALUE HIGH-VALUES</p> <p>0402 LOW-VALUE LOW-VALUES</p> <p>0403 QUOTE QUOTES</p> <p>0404 SPACE SPACES</p> <p>0405 ZERO</p>

Table 135. **SYSADATA token record** (continued)

Field	Size	Description
		0406 ZEROES ZEROS 0407 NULL NULLS 0501 BLOCK 0502 BOTTOM 0505 CHARACTER 0506 CODE 0507 CODE-SET 0514 FILLER 0516 FOOTING 0520 LABEL 0521 LENGTH 0524 LINAGE 0526 OMITTED 0531 RENAMES 0543 TOP

Table 135. **SYSADATA** token record (continued)

Field	Size	Description
		<p>0545 TRAILING</p> <p>0549 RECORDING</p> <p>0500 POINTER-32</p> <p>0551 BYTE-LENGTH</p> <p>0555 LIMIT</p> <p>0601 INHERITS</p> <p>0603 RECURSIVE</p> <p>0701 ACCESS</p> <p>0702 ALSO</p> <p>0703 ALTERNATE</p> <p>0704 AREA AREAS</p> <p>0705 ASSIGN</p> <p>0707 COLLATING</p> <p>0708 COMMA</p> <p>0709 CURRENCY</p> <p>0710 CLASS</p> <p>0711 DECIMAL-POINT</p> <p>0712 DUPLICATES</p> <p>0713 DYNAMIC</p> <p>0714 EVERY</p>

Table 135. **SYSADATA token record** (continued)

Field	Size	Description
		<p>0716 MEMORY</p> <p>0717 MODE</p> <p>0718 MODULES</p> <p>0719 MULTIPLE</p> <p>0720 NATIVE</p> <p>0721 OFF</p> <p>0722 OPTIONAL</p> <p>0723 ORGANIZATION</p> <p>0724 POSITION</p> <p>0725 PROGRAM</p> <p>0726 RANDOM</p> <p>0727 RELATIVE</p> <p>0728 RERUN</p> <p>0729 RESERVE</p> <p>0730 SAME</p> <p>0731 SEGMENT-LIMIT</p> <p>0732 SELECT</p>

Table 135. *SYSADATA token record* (continued)

Field	Size	Description
		0733 SEQUENCE
		0734 SEQUENTIAL
		0736 SORT-MERGE
		0737 STANDARD-1
		0738 TAPE
		0739 WORDS
		0740 PROCESSING
		0741 APPLY
		0742 WRITE-ONLY
		0743 COMMON
		0744 ALPHABET
		0745 PADDING
		0746 SYMBOLIC
		0747 STANDARD-2
		0748 OVERRIDE
		0750 PASSWORD
		0751 XML-SCHEMA

Table 135. *SYSADATA token record (continued)*

Field	Size	Description
		<p>0801 ARE IS</p> <p>0802 ASCENDING</p> <p>0803 AT</p> <p>0804 BY</p> <p>0805 CHARACTERS</p> <p>0806 CONTAINS</p> <p>0808 COUNT</p> <p>0809 DEBUGGING</p> <p>0810 DEPENDING</p> <p>0811 DESCENDING</p> <p>0812 DIVISION</p> <p>0814 FOR</p> <p>0815 ORDER</p> <p>0816 INPUT</p> <p>0817 REPLACE</p> <p>0818 KEY</p>

Table 135. **SYSADATA** token record (continued)

Field	Size	Description
		<p>0819 LINE LINES</p> <p>0820 XML-INFORMATION</p> <p>0821 OF</p> <p>0822 ON</p> <p>0823 OUTPUT</p> <p>0825 RECORD</p> <p>0826 RECORDS</p> <p>0827 REEL</p> <p>0828 SECTION</p> <p>0829 SIZE</p> <p>0830 STATUS</p> <p>0831 THROUGH THRU</p> <p>0832 TIME</p> <p>0833 TIMES</p> <p>0834 TO</p> <p>0836 UNIT</p>

Table 135. **SYSADATA token record** (continued)

Field	Size	Description
		<p>0837 USING</p> <p>0838 WHEN</p> <p>0839 WITH</p> <p>0840 SQLIMS</p> <p>0841 DEFAULT</p> <p>0901 PROCEDURE</p> <p>0902 DECLARATIVES</p> <p>0903 END</p> <p>1001 DATA</p> <p>1002 FILE</p> <p>1003 FD</p> <p>1004 SD</p> <p>1005 WORKING-STORAGE</p> <p>1006 LOCAL-STORAGE</p>

Table 135. **SYSADATA** token record (continued)

Field	Size	Description
		1007 LINKAGE
		1101 ENVIRONMENT
		1102 CONFIGURATION
		1103 SOURCE-COMPUTER
		1104 OBJECT-COMPUTER
		1105 SPECIAL-NAMES
		1106 REPOSITORY
		1107 INPUT-OUTPUT
		1108 FILE-CONTROL
		1109 I-O-CONTROL
		1201 ID IDENTIFICATION
		1202 PROGRAM-ID
		1203 AUTHOR
		1204 INSTALLATION
		1205 DATE-WRITTEN
		1206 DATE-COMPILED
		1207 SECURITY
		1208 CLASS-ID
		1209 METHOD-ID
		1210 METHOD
		1211 FACTORY

Table 135. **SYSADATA token record** (continued)

Field	Size	Description
		1212 OBJECT
		2020 TRACE
		2046 SUPPRESS
		3000 DATADEF
		3001 F-NAME
		3002 UPSI-SWITCH
		3003 CONDNAME
		3004 CONDVAR
		3005 BLOB
		3006 CLOB
		3007 DBCLOB
		3008 BLOB-LOCATOR
		3009 CLOB-LOCATOR
		3010 DBCLOB-LOCATOR
		3011 BLOB-FILE
		3012 CLOB-FILE
		3013 DBCLOB-FILE
		3014 DFHRESP

Table 135. **SYSADATA token record** (continued)

Field	Size	Description
		<p>5001 PARSE</p> <p>5002 AUTOMATIC</p> <p>5003 PREVIOUS</p> <p>5004 ENCODING</p> <p>5005 NAMESPACE</p> <p>5006 NAMESPACE-PREFIX</p> <p>5007 XML-DECLARATION</p> <p>5008 ATTRIBUTES</p> <p>5009 VALIDATING</p> <p>5010 UNBOUNDED</p> <p>5011 ATTRIBUTE</p> <p>5012 ELEMENT</p> <p>5013 NONNUMERIC</p> <p>5014 NAME</p> <p>5015 CYCLE</p> <p>5016 PARAGRAPH</p> <p>5020 AS</p> <p>5021 INITIALIZED</p> <p>9999 COBOL</p>
Token length	HL2	The length of the token
Token column	FL4	The starting column number of the token in the source listing
Token line	FL4	The line number of the token in the source listing

Field	Size	Description
Flags	CL1	Information about the token: X'80' Token is continued X'40' Last piece of continued token Note that for PICTURE strings, even if the source token is continued, there will be only one Token record generated. It will have a token code of 0000, the token column and line of the first piece, the length of the complete string, no continuation flags set, and the token text of the complete string.
Reserved	CL7	Reserved for future use
Token text	CL(<i>n</i>)	The actual token string

Source error record: X'0032'

The following table shows the contents of the source error record.

Field	Size	Description
Line number	FL4	The line number of the line in error
Error identifier	CL16	The error message identifier (left-justified and padded with blanks)
Error severity	HL2	The severity of the error
Error message length	HL2	The length of the error message text
Line position	XL1	The line position indicator provided in FIPS messages
Reserved	CL7	Reserved for future use
Error message	CL(<i>n</i>)	The error message text

Source record: X'0038'

The following table shows the contents of the source record.

Field	Size	Description
Line number	FL4	The listing line number of the source record
Input record number	FL4	The input source record number in the current input file
Primary file number	HL2	The input file's assigned sequence number if this record is from the primary input file. (Refer to the Input file <i>n</i> field in the Job identification record).
Library file number	HL2	The library input file's assigned sequence number if this record is from a COPY BASIS input file. (Refer to the Member File ID <i>n</i> field in the Library record.)

Field	Size	Description
Reserved	CL8	Reserved for future use
Parent record number	FL4	The parent source record number. This will be the record number of the COPY BASIS statement.
Parent primary file number	HL2	The parent file's assigned sequence number if the parent of this record is from the primary input file. (Refer to the Input file <i>n</i> field in the Job Identification Record.)
Parent library assigned file number	HL2	The parent library file's assigned sequence number if this record's parent is from a COPY BASIS input file. (Refer to the COPY/BASIS Member File ID <i>n</i> field in the Library record.)
Reserved	CL8	Reserved for future use
Length of source record	HL2	The length of the actual source record following
Reserved	CL10	Reserved for future use
Source record	CL(<i>n</i>)	

COPY REPLACING record: X'0039'

One COPY REPLACING type record will be emitted each time a REPLACING action takes place. That is, whenever *operand-1* of the REPLACING phrase is matched with text in the copybook, a COPY REPLACING TEXT record will be written.

The following table shows the contents of the COPY REPLACING record.

Field	Size	Description
Starting line number of replaced string	FL4	The listing line number of the start of the text that resulted from REPLACING
Starting column number of replaced string	FL4	The listing column number of the start of the text that resulted from REPLACING
Ending line number of replaced string	FL4	The listing line number of the end of the text that resulted from REPLACING
Ending column number of replaced string	FL4	The listing column number of the end of the text that resulted from REPLACING
Starting line number of original string	FL4	The source file line number of the start of the text that was changed by REPLACING
Starting column number of original string	FL4	The source file column number of the start of the text that was changed by REPLACING
Ending line number of original string	FL4	The source file line number of the end of the text that was changed by REPLACING
Ending column number of original string	FL4	The source file column number of the end of the text that was changed by REPLACING

Symbol record: X'0042'

The following table shows the contents of the symbol record.

<i>Table 139. SYSADATA symbol record</i>		
Field	Size	Description
Symbol ID	FL4	Unique ID of symbol
Line number	FL4	The listing line number of the source record in which the symbol is defined or declared
Level	XL1	True level-number of symbol (or relative level-number of a data item within a structure). For COBOL, this can be in the range 01-49, 66 (for RENAME items), 77, or 88 (for condition items).
Qualification indicator	XL1	<p>X'00' Unique name; no qualification needed.</p> <p>X'01' This data item needs qualification. The name is not unique within the program. This field applies only when this data item is <i>not</i> the level-01 name.</p>
Symbol type	XL1	<p>X'68' Class-name (Class-ID)</p> <p>X'58' Method-name</p> <p>X'40' Data-name</p> <p>X'20' Procedure-name</p> <p>X'10' Mnemonic-name</p> <p>X'08' Program-name</p> <p>X'81' Reserved</p> <p>The following ORed are into the above types, when applicable:</p> <p>X'04' External</p> <p>X'02' Global</p>

Table 139. **SYSADATA symbol record** (continued)

Field	Size	Description
Symbol attribute	XL1	<p>X'01' Numeric</p> <p>X'02' Elementary character of one of these classes:</p> <ul style="list-style-type: none"> • Alphabetic • Alphanumeric • DBCS • National • UTF-8 <p>X'03' Group</p> <p>X'04' Pointer</p> <p>X'05' Index data item</p> <p>X'06' Index-name</p> <p>X'07' Condition</p> <p>X'0F' File</p> <p>X'10' Sort file</p> <p>X'17' Class-name (repository)</p> <p>X'18' Object reference</p> <p>X'19' Currency-sign symbol</p> <p>X'1A' XML schema name</p>

Table 139. **SYSADATA symbol record** (continued)

Field	Size	Description
Clauses	XL1	<p>Clauses specified in symbol definition.</p> <p>For symbols that have a symbol attribute of Numeric (X'01'), Elementary character (X'02'), Group (X'03'), Pointer (X'04'), Index data item (X'05'), or Object reference (X'18'):</p> <p>1... .. Value</p> <p>.1.. .. Indexed</p> <p>..1. Redefines</p> <p>...1 Renames</p> <p>.... 1... Occurs</p> <p>.... .1.. Has Occurs keys</p> <p>.... ..1. Occurs Depending On</p> <p>.... ...1 Occurs in parent</p> <p>For file types:</p> <p>1... .. Select</p> <p>.1.. .. Assign</p> <p>..1. Rerun</p> <p>...1 Same area</p> <p>.... 1... Same record area</p> <p>.... .1.. Recording mode</p> <p>.... ..1. Reserved</p> <p>.... ...1 Record</p>

Table 139. *SYSADATA symbol record (continued)*

Field	Size	Description
		<p>For mnemonic-name symbols:</p> <p>01 CSP</p> <p>02 C01</p> <p>03 C02</p> <p>04 C03</p> <p>05 C04</p> <p>06 C05</p> <p>07 C06</p> <p>08 C07</p> <p>09 C08</p> <p>10 C09</p> <p>11 C10</p> <p>12 C11</p> <p>13 C12</p> <p>14 S01</p> <p>15 S02</p> <p>16 S03</p> <p>17 S04</p> <p>18 S05</p> <p>19 CONSOLE</p>

Table 139. **SYSADATA** symbol record (continued)

Field	Size	Description
		<p>20 SYSIN SYSIPT</p> <p>22 SYSOUT SYSLST SYSLIST</p> <p>24 SYSPUNCH SYSPCH</p> <p>26 UPSI-0</p> <p>27 UPSI-1</p> <p>28 UPSI-2</p> <p>29 UPSI-3</p> <p>30 UPSI-4</p> <p>31 UPSI-5</p> <p>32 UPSI-6</p> <p>33 UPSI-7</p> <p>34 AFP-5A</p>
Data flags 1	XL1	<p>For file types, and for symbols that have a symbol attribute of Numeric (X'01'), Elementary character (X'02'), Group (X'03'), Pointer (X'04'), Index data item (X'05'), or Object reference (X'18'):</p> <p>1... .. Redefined</p> <p>.1.. Renamed</p> <p>..1. Synchronized</p> <p>...1 Implicitly redefined</p> <p>.... 1... Volatile</p> <p>.... .1.. Implicit redefines</p> <p>.... ..1. FILLER</p> <p>.... ...1 Level 77</p>

Table 139. **SYSADATA symbol record** (continued)

Field	Size	Description
Data flags 2	XL1	<p>For symbols that have a symbol attribute of Numeric (X'01'):</p> <p>1... .. Binary</p> <p>.1.. .. External floating point (of USAGE DISPLAY or USAGE NATIONAL)</p> <p>..1. Internal floating point</p> <p>...1 Packed</p> <p>.... 1... External decimal (of USAGE DISPLAY or USAGE NATIONAL)</p> <p>.... .1.. Scaled negative</p> <p>.... ..1. Numeric edited (of USAGE DISPLAY or USAGE NATIONAL)</p> <p>.... ...1 Binary (of USAGE COMP-5)</p> <p>For symbols that have a symbol attribute of Elementary character (X'02') or Group (X'03'):</p> <p>1... .. Alphabetic</p> <p>.1.. .. Alphanumeric</p> <p>..1. Alphanumeric edited</p> <p>...1 Group contains its own ODO object</p> <p>.... 1... DBCS item</p> <p>.... .1.. Group variable length</p> <p>.... ..1. EGCS item</p> <p>.... ...1 EGCS edited</p>

Table 139. **SYSADATA symbol record** (continued)

Field	Size	Description
		<p>For file types:</p> <p>1... .. Object of ODO in record</p> <p>.1... .. Subject of ODO in record</p> <p>..1. Sequential access</p> <p>...1 Random access</p> <p>.... 1... Dynamic access</p> <p>.... .1.. Locate mode</p> <p>.... ..1. Record area</p> <p>.... ...1 Reserved for future use</p>
Data flags 3	XL1	<p>1... .. All records are the same length</p> <p>.1... .. Fixed length</p> <p>..1. Variable length</p> <p>...1 Undefined</p> <p>.... 1... Spanned</p> <p>.... .1.. Blocked</p> <p>.... ..1. Apply write only</p> <p>.... ...1 Same sort merge area</p>

Table 139. **SYSADATA** symbol record (continued)

Field	Size	Description
File organization and attributes	XL1	<p>1... .. Physical sequential (on host, QSAM)</p> <p>.1... .. ASCII</p> <p>..1. Standard label</p> <p>...1 User label</p> <p>.... 1... Sequential organization</p> <p>.... .1.. Indexed organization</p> <p>.... ..1. Relative organization</p> <p>.... ...1 Line sequential</p>
USAGE clause	FL1	<p>X'00' USAGE IS DISPLAY</p> <p>X'01' USAGE IS COMP-1</p> <p>X'02' USAGE IS COMP-2</p> <p>X'03' USAGE IS PACKED-DECIMAL or USAGE IS COMP-3</p> <p>X'04' USAGE IS BINARY, USAGE IS COMP, USAGE IS COMP-4, or USAGE IS COMP-5</p> <p>X'05' USAGE IS DISPLAY-1</p> <p>X'06' USAGE IS POINTER</p> <p>X'07' USAGE IS INDEX</p> <p>X'08' USAGE IS PROCEDURE-POINTER</p> <p>X'09' USAGE IS OBJECT-REFERENCE</p> <p>X'0A' FUNCTION-POINTER</p> <p>X'0B' NATIONAL</p> <p>X'0C' POINTER-32</p> <p>X'0D' UTF-8</p>

Table 139. **SYSADATA symbol record** (continued)

Field	Size	Description
Sign clause	FL1	X'00' No SIGN clause X'01' SIGN IS LEADING X'02' SIGN IS LEADING SEPARATE CHARACTER X'03' SIGN IS TRAILING X'04' SIGN IS TRAILING SEPARATE CHARACTER
Indicators	FL1	X'01' Has JUSTIFIED clause. Right-justified attribute is in effect. X'02' Has BLANK WHEN ZERO clause.
Size	FL4	The size of this data item. The actual number of bytes this item occupies in storage. If a DBCS item, the number is in bytes, not characters. For variable-length items, this field will reflect the maximum size of storage reserved for this item by the compiler. Also known as the "Length attribute."
Precision	FL1	The precision of a fixed or float data item
Scale	FL1	The scale factor of a fixed data item. This is the number of digits to the right of the decimal point.

Table 139. **SYSADATA** symbol record (continued)

Field	Size	Description
Storage type	FL1	<p>00 Not applicable</p> <p>01 Files</p> <p>02 Working-Storage</p> <p>03 Linkage Section</p> <p>05 Special registers</p> <p>07 Indexed by variable</p> <p>10 UPSI switch</p> <p>13 Variably located items</p> <p>14 External data</p> <p>15 Alphanumeric FUNC</p> <p>16 Alphanumeric EVAL</p> <p>17 Object data</p> <p>19 Local-Storage</p> <p>20 Factory data</p> <p>21 XML - TEXT and XML - NTEXT</p>
Data flags 6	XL1	<p>Miscellaneous symbol flags:</p> <p>1... .. Dynamic-length elementary item</p> <p>.1... .. GROUP dynamic-length item</p> <p>..1. Fixed byte-length UTF-8 item</p>

Table 139. **SYSADATA symbol record** (continued)

Field	Size	Description
Data flags 4	XL1	<p>For symbols that have a symbol attribute of Numeric (X'01'):</p> <p>1... .. Numeric national</p> <p>For symbols that have a symbol attribute of Elementary character (X'02'):</p> <p>1... .. National</p> <p>.1... .. National edited</p> <p>For symbols that have a symbol attribute of Group (X'03'):</p> <p>1... .. Group-Usage National</p> <p>.1... .. Unbounded length group</p>
Data flags 5	XL1	<p>OCCURS flags:</p> <p>1... .. UNBOUNDED</p>
Base locator Cell	FL2	Base locator cell number
Symbol Identifier	FL4	Number identifying the symbol
Structure displacement	AL4	Offset of symbol within structure. This offset is set to 0 for variably located items.
Parent displacement	AL4	Byte offset from immediate parent of the item being defined.
Parent ID	FL4	The symbol ID of the immediate parent of the item being defined.
Redefined ID	FL4	The symbol ID of the data item that this item redefines, if applicable.
Start-renamed ID	FL4	If this item is a level-66 item, the symbol ID of the starting COBOL data item that this item renames. If not a level-66 item, this field is set to 0.
End-renamed ID	FL4	If this item is a level-66 item, the symbol ID of the ending COBOL data item that this item renames. If not a level-66 item, this field is set to 0.
Program-name symbol ID	FL4	ID of the program-name of the program or the class-name of the class where this symbol is defined.
OCCURS minimum Paragraph ID	FL4	Minimum value for OCCURS Proc-name ID for a paragraph-name
OCCURS maximum Section ID	FL4	Maximum value for OCCURS Proc-name ID for a section-name
Dimensions	FL4	Number of dimensions

Table 139. **SYSADATA symbol record** (continued)

Field	Size	Description
Case bit vector	XL4	The case of the characters in the symbol name is represented with one bit per character. Each bit has the following meaning: 0 Uppercase 1 Lowercase Bit 0 represents the case of the first character, bit 1 represents the case of the second character, and so forth.
Dynamic-length item limit	FL4	The value of the LIMIT phrase of the DYNAMIC LENGTH clause for dynamic-length items; zero if the item is not dynamic-length or the LIMIT phrase was not specified.
Reserved	CL4	Reserved for future use
Value pairs count	HL2	Count of value pairs
Symbol name length	HL2	Number of characters in the symbol name
Picture data length for data-name or Assignment-name length for file-name	HL2	Number of characters in the picture data; zero if symbol has no associated PICTURE clause. (Length of the PICTURE field.) Length represents the field as it is found in the source input. This length does not represent the expanded field for PICTURE items that contain a replication factor. The maximum COBOL length for a PICTURE string is 50 bytes. Zero in this field indicates no PICTURE specified. Number of characters in the external file-name if this is a file-name. This is the DD name part of the assignment-name. Zero if file-name and ASSIGN USING specified.
Initial Value length for data-name External class-name length for CLASS-ID	HL2	Number of characters in the symbol value; zero if symbol has no initial value Number of characters in the external class-name for CLASS-ID
ODO symbol name ID for data-name ID of ASSIGN data-name if file-name	FL4	If data-name, ID of the ODO symbol name; zero if ODO not specified If file-name, Symbol-ID for ASSIGN USING data-name; zero if ASSIGN TO specified
Keys count	HL2	The number of keys defined
Index count	HL2	Count of Index symbol IDs; zero if none specified
Symbol name	CL(n)	
Picture data string for data-name or Assignment-name for file-name	CL(n)	The PICTURE character string <i>exactly</i> as the user types it in. The character string includes all symbols, parentheses, and replication factor. The external file-name if this is a file-name. This is the DD name part of the assignment-name.
Index ID list	(n)FL4	ID of each index symbol name

Table 139. **SYSADATA symbol record** (continued)

Field	Size	Description
Keys	(n)XL8	This field contains data describing keys specified for an array. The following three fields are repeated as many times as specified in the 'Keys count' field.
...Key Sequence	FL1	Ascending or descending indicator. X'00' DESCENDING X'01' ASCENDING
...Filler	CL3	Reserved
...Key ID	FL4	The symbol ID of the data item that is the key field in the array
Initial Value data for data-name External class-name for CLASS-ID	CL(n)	This field contains the data specified in the INITIAL VALUE clause for this symbol. The following four subfields are repeated according to the count in the 'Value pairs count' field. The total length of the data in this field is contained in the 'Initial value length' field. The external class-name for CLASS-ID.
...1st value length	HL2	Length of first value
...1st value data	CL(n)	1st value. This field contains the literal (or figurative constant) as it is specified in the VALUE clause in the source file. It includes any beginning and ending delimiters, embedded quotation marks, and SHIFT IN and SHIFT OUT characters. If the literal spans multiple lines, the lines are concatenated into one long string. If a figurative constant is specified, this field contains the actual reserved word, not the value associated with that word.
...2nd value length	HL2	Length of second value, zero if not a THRU value pair
...2nd value data	CL(n)	2nd value. This field contains the literal (or figurative constant) as it is specified in the VALUE clause in the source file. It includes any beginning and ending delimiters, embedded quotation marks, and SHIFT IN and SHIFT OUT characters. If the literal spans multiple lines, the lines are concatenated into one long string. If a figurative constant is specified, this field contains the actual reserved word, not the value associated with that word.

Symbol cross-reference record: X'0044'

The following table shows the contents of the symbol cross-reference record.

Table 140. **SYSADATA symbol cross-reference record**

Field	Size	Description
Symbol name length	HL2	Number of characters in the symbol name

Table 140. **SYSADATA symbol cross-reference record** (continued)

Field	Size	Description
Statement definition	FL4	The line number where the symbol is defined or declared For statement XREF only: statement count - total number of references to this statement.
Number of references ¹	HL2	The number of references in this record to the symbol following
Cross-reference type	XL1	X'01' Program X'02' Procedure X'03' statement X'04' Symbol or data-name X'05' Method X'06' Class
Reserved	CL7	Reserved for future use
Symbol name	CL(n)	The symbol. Variable length.
...Reference flag	CL1	For symbol or data-name references: C' ' Blank means reference only C'M' Modification reference flag For Procedure type symbol references: C'A' ALTER (procedure-name) C'D' GO TO (procedure-name) DEPENDING ON C'E' End of range of (PERFORM) through (procedure-name) C'G' GO TO (procedure-name) C'P' PERFORM (procedure-name) C'T' (ALTER) TO PROCEED TO (procedure-name) C'U' Use for debugging (procedure-name)
...Line number	XL4	The line number on which the symbol or statement is referenced

Table 140. **SYSADATA symbol cross-reference record** (continued)

Field	Size	Description
<p>1. The reference flag field and the statement number field occur as many times as the number of references field dictates. For example, if there is a value of 10 in the number of references field, there will be 10 occurrences of the reference flag and statement number pair for data-name, procedure, or program symbols, or 10 occurrences of the statement number for statements.</p> <p>Where the number of references would exceed the record size for the SYSADATA file, the record is continued on the next record. The continuation flag is set in the common header section of the record.</p>		

Nested program record: X'0046'

The following table shows the contents of the nested program record.

Table 141. **SYSADATA nested program record**

Field	Size	Description
Statement definition	FL4	The line number where the symbol is defined or declared
Nesting level	XL1	Program nesting level
Program attributes	XL1	<p>1... .. Initial</p> <p>.1... .. Common</p> <p>..1. PROCEDURE DIVISION using</p> <p>...1 1111 Reserved for future use</p>
Reserved	XL1	Reserved for future use
Program-name length	XL1	Length of the following field
Program-name	CL(n)	The program-name

Library record: X'0060'

The following table shows the contents of the SYSADATA library record.

Table 142. **SYSADATA library record**

Field	Size	Description
Number of members ¹	HL2	Count of the number of COPY/INCLUDE code members described in this record
Library name length	HL2	The length of the library name
Library volume length	HL2	The length of the library volume ID
Concatenation number	XL2	Concatenation number of the library
Library ddname length	HL2	The length of the library ddname
Reserved	CL4	Reserved for future use

Field	Size	Description
Library name	CL(n)	The name of the library from which the COPY/INCLUDE member was retrieved
Library volume	CL(n)	The volume identification of the volume where the library resides
Library ddname	CL(n)	The ddname (or equivalent) used for this library
...COPY/BASIS member file ID ²	HL2	The library file ID of the name following
...COPY/BASIS name length	HL2	The length of the name following
...COPY/BASIS name	CL(n)	The name of the COPY/BASIS member that has been used
<p>1. If 10 COPY members are retrieved from a library, the "Number of members" field will contain 10 and there will be 10 occurrences of the "COPY/BASIS member file ID" field, the "COPY/BASIS name length" field, and the "COPY/BASIS name" field.</p> <p>2. If COPY/BASIS members are retrieved from different libraries, a library record is written to the SYSADATA file for each unique library.</p>		

Statistics record: X'0090'

The following table shows the contents of the statistics record.

Field	Size	Description
Source records	FL4	The number of source records processed
DATA DIVISION statements	FL4	The number of DATA DIVISION statements processed
PROCEDURE DIVISION statements	FL4	The number of PROCEDURE DIVISION statements processed
Compilation number	HL2	Batch compilation number
Error severity	XL1	The highest error message severity
Flags	XL1	1... .. End of Job indicator .1... .. Class definition indicator ..11 1111 Reserved for future use
EOJ severity	XL1	The maximum return code for the compile job
Program-name length	XL1	The length of the program-name
Program-name	CL(n)	Program-name

EVENTS record: X'0120'

Events records are included in the ADATA file to provide compatibility with previous levels of the compiler.

Events records are of the following types:

- Time stamp
- Processor
- File end
- Program
- File ID
- Error

Field	Size	Description
Header	CL12	Standard ADATA record header
Record length	HL2	Length of following EVENTS record data (excluding this halfword)
EVENTS record type TIMESTAMP record	CL12	C'TIMESTAMP'
Blank separator	CL1	
Revision level	XL1	
Blank separator	CL1	
Date	XL8	YYYYMMDD
Hour	XL2	HH
Minutes	XL2	MI
Seconds	XL2	SS

Field	Size	Description
Header	CL12	Standard ADATA record header
Record length	HL2	Length of following EVENTS record data (excluding this halfword)
EVENTS record type PROCESSOR record	CL9	C'PROCESSOR'
Blank separator	CL1	
Revision level	XL1	
Blank separator	CL1	
Output file ID	XL1	
Blank separator	CL1	
Line-class indicator	XL1	

Field	Size	Description
Header	CL12	Standard ADATA record header
Record length	HL2	Length of following EVENTS record data (excluding this halfword)

Table 146. SYSADATA EVENTS FILE END record layout (continued)

Field	Size	Description
EVENTS record type FILE END record	CL7	C'FILEEND'
Blank separator	CL1	
Revision level	XL1	
Blank separator	CL1	
Input file ID	XL1	
Blank separator	CL1	
Expansion indicator	XL1	

Table 147. SYSADATA EVENTS PROGRAM record layout

Field	Size	Description
Header	CL12	Standard ADATA record header
Record length	HL2	Length of following EVENTS record data (excluding this halfword)
EVENTS record type PROGRAM record	CL7	C'PROGRAM'
Blank separator	CL1	
Revision level	XL1	
Blank separator	CL1	
Output file ID	XL1	
Blank separator	CL1	
Program input record number	XL1	

Table 148. SYSADATA EVENTS FILE ID record layout

Field	Size	Description
Header	CL12	Standard ADATA record header
Record length	HL2	Length of following EVENTS record data (excluding this halfword)
EVENTS record type FILE ID record	CL7	C'FILEID'
Blank separator	CL1	
Revision level	XL1	
Blank separator	CL1	
Input source file ID	XL1	File ID of source file
Blank separator	CL1	
Reference indicator	XL1	
Blank separator	CL1	

Table 148. SYSADATA EVENTS FILE ID record layout (continued)

Field	Size	Description
Source file name length	H2	
Blank separator	CL1	
Source file name	CL(n)	

Table 149. SYSADATA EVENTS ERROR record layout

Field	Size	Description
Header	CL12	Standard ADATA record header
Record length	HL2	Length of following EVENTS record data (excluding this halfword)
EVENTS record type ERROR record	CL5	C'ERROR'
Blank separator	CL1	
Revision level	XL1	
Blank separator	CL1	
Input source file ID	XL1	File ID of source file
Blank separator	CL1	
Annot class	XL1	Annot-class message placement
Blank separator	CL1	
Error input record number	XL10	
Blank separator	CL1	
Error start line number	XL10	
Blank separator	CL1	
Error token start number	XL1	Column number of error token start
Blank separator	CL1	
Error end line number	XL10	
Blank separator	CL1	
Error token end number	XL1	Column number of error token end
Blank separator	CL1	
Error message ID number	XL9	
Blank separator	CL1	
Error message severity code	XL1	
Blank separator	CL1	
Error message severity level number	XL2	

Table 149. **SYSADATA EVENTS ERROR** record layout (continued)

Field	Size	Description
Blank separator	CL1	
Error message length	HL3	
Blank separator	CL1	
Error message text	CL(n)	

Appendix H. Using sample programs

The sample programs, which are included on your product tape, demonstrate many language elements and concepts of COBOL.

This information contains the following items:

- Overview of the programs, including program charts for two of the samples
- Format and sample of the input data
- Sample of reports produced
- Information about how to run the programs
- List of the language elements and concepts that are illustrated

Pseudocode and other comments about the programs are included in the program prolog, which you can obtain in a program listing.

There are three sample programs:

- 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 IBM Interactive System Product Facility (ISPF).
- IGYTSALE is an example of using several of the features of the Language Environment callable services.

Related concepts

[“IGYTCARA: batch application” on page 935](#)

[“IGYTCARB: interactive program” on page 939](#)

[“IGYTSALE: nested program application” on page 941](#)

IGYTCARA: batch application

A company that has several local offices wants to establish employee carpools. Application IGYTCARA validates the transaction-file entries (QSAM sequential file processing) and updates a main file (VSAM indexed file processing).

This batch application does two tasks:

- Produces reports of employees who can share rides from the same home location to the same work location
- Updates the carpool data:
 - Adds data for new employees
 - Changes information for participating employees
 - Deletes employee records
 - Lists update requests that are not valid

The following diagram shows the parts of the application and how they are organized:

6. Home address
7. Home phone
8. Work phone
9. Home location code
10. Work location code
11. Driving status code

This 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: 11/24/2003
-----
TRANS | RE- | SHIFT | COMMUTER | HOME ADDRESS | HOME PHONE | HOME LOCATION JUNCTION | STA- | TRANS. ERROR
CODE | TYPE | WORK CODE | NAME | | | WORK PHONE | WORK LOCATION JUNCTION | CODE |
-----
A NEW 1 01 11 ROBERTS AB 1021 CRYSTAL COURT (415) 555-0190 RODNEY/CRYSTAL D
SAN FRANCISCO CA 99901 (415) 555-1387 BAYFAIR PLAZA
-----
A NEW 2 02 12 KAHN DE 789 EMILY LANE (415) 555-1890 COYOTE D
SAN FRANCISCO CA 99921 (415) 555-2589 14TH STREET/166TH AVENUE
-----
P 4 88 99 (000) 555-7890 HOME CODE ' ' NOT FOUND. T
99 ASDFG (123) ASD-FGHJ WORK CODE ' ' NOT FOUND.
-----
TRANSACT. 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 (415) 555-0190 RODNEY/CRYSTAL D
SAN FRANCISCO CA 99901 (415) 555-1387 BAYFAIR PLAZA
NEW 1 01 11 ROBERTS AB 1221 CRYSTAL COURT (415) 555-0190 RODNEY/CRYSTAL D
SAN FRANCISCO CA 99901 (415) 555-1387 BAYFAIR PLAZA
-----
A 2 02 12 KAHN DE 789 EMILY LANE (415) 555-1890 COYOTE D
SAN FRANCISCO CA 99921 (415) 555-2589 14TH STREET/166TH AVENUE
DUPLICATE REC.
-----
D OLD 2 02 12 KAHN DE 789 EMILY LANE (415) 555-1890 COYOTE D
SAN FRANCISCO CA 99921 (415) 555-2589 14TH STREET/166TH AVENUE
-----
D 2 02 12 KAHN DE REC. NOT FOUND
-----
A NEW 2 02 12 KAHN DE 789 EMILY LANE (415) 555-1890 COYOTE D
SAN FRANCISCO CA 99921 (415) 555-2589 14TH STREET/166TH AVENUE
-----
A NEW 1 01 11 BONNICK FD 1025 FIFTH AVENUE (415) 555-9590 RODNEY
SAN FRANCISCO CA 99905 (415) 555-7895 17TH FREEWAY SAN LEANDRO
-----
A NEW 1 01 11 PETERSON SW 435 THIRD AVENUE (415) 555-4690 RODNEY/THIRD AVENUE

```

Preparing to run IGYTCARA

All files required by the IGYTCARA program (IGYTCARA, IGYTCODE, and IGYTRANX) are supplied on the product installation tape. These files are located in the IGY.V6R4M0.SIGYSAMP data set.

Data-set and procedure names might be changed at installation time. 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
- 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 935](#)

Related tasks

[“Running IGYTCARA” on page 938](#)

Related references

[“Input data for IGYTCARA” on page 936](#)

[“Report produced by IGYTCARA” on page 937](#)

[“Language elements and concepts that are illustrated” on page 949](#)

Running IGYTCARA

The following procedure compiles, link-edits, and runs the IGYTCARA program. If you want only to compile or only to compile and link-edit the program, you must change the IGYWCLG cataloged procedure.

To run IGYTCARA under z/OS, use JCL to define a VSAM cluster and compile the program. Insert the information specific to your system and installation (accounting information, volume serial number, unit name, cluster prefix) in the fields that are shown in lowercase letters. These examples use the name IGYTCAR.MASTFILE; you can use another name if you want to.

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.V6R4M0.SIGYSAMP, DISP=SHR
//COBOL.SYSIN DD DSN=IGY.V6R4M0.SIGYSAMP(IGYTCARA), DISP=SHR
//GO.SYSOUT DD SYSOUT=A
//GO.COMMUTR DD DSN=your-prefix.IGYTCAR.MASTFILE, DISP=SHR
//GO.LOCCODE DD DSN=IGY.V6R4M0.SIGYSAMP(IGYTCODE), DISP=SHR
//GO.UPDTRANS DD DSN=IGY.V6R4M0.SIGYSAMP(IGYTRANX), DISP=SHR
//GO.UPDPRINT DD SYSOUT=A, DCB=BLKSIZE=133
//

```

Related tasks

Chapter 10, “Processing VSAM files,” on page 189

Related references

“Compile, link-edit, and run procedure (IGYWCLG)” on page 259

IGYTCARB: interactive program

IGYTCARB contains an interactive program for entering carpool data by using IBM Interactive System Productivity Facility (ISPF) to invoke Dialog Manager and Enterprise COBOL. IGYTCARB creates a file that can 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                                Previous 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 FRANCISCO
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

“Preparing to run IGYTCARB” on page 939

Preparing to run IGYTCARB

Run the IGYTCARB program under Interactive System Productivity Facility (ISPF). All files required by IGYTCARB (IGYTCARB, IGYTRANB, and IGYTPNL) are supplied on the product installation tape in the IGY.V6R4M0.SIGYSAMP data set.

Data-set names and procedure-names might be changed at installation time. Check with your system programmer to verify the names.

Do not change the following options in the CBL statement in the source file for IGYTCARB:

- 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 939](#)

Related tasks

[“Running IGYTCARB” on page 940](#)

Related references

[“Language elements and concepts that are illustrated” on page 949](#)

Running IGYTCARB

The following procedure compiles, link-edits, and runs the IGYTCARB program. If you want only to compile or only to compile and link-edit the program, you must change the procedure.

To run IGYTCARB under z/OS, do 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
%
% 0 +ISPF PARMS - Specify terminal and user parameters +USERID - &ZUSER
% 1 +BROWSE - Display source data or output listings +TIME - &ZTIME
% 2 +EDIT - Create or change source data +TERMINAL - &ZTERM
% 3 +UTILITIES - Perform utility functions +PF KEYS - &ZKEYS
% 4 +FOREGROUND - Invoke language processors in foreground
% 5 +BATCH - Submit to batch for language processing
% 6 +COMMAND - Enter TSO or Workstation commands
% 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)' (2)
 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. Comment sequence line IB2200 and uncomment sequence line IB2210 in IGYTCARB. (The OPEN EXTEND statement is supported under z/OS.)
4. Compile and link-edit IGYTCARB and place the resulting program object in your LOADLIB.
5. Allocate ISPLLIB by 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 by using the following command:

```
ALLOCATE FILE(UPDTRANS) DA('IGY.V6R4M0.SIGYSAMP(IGYTRANB)') SHR REUSE
```

7. Allocate ISPLLIB by 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 by using your modified panel.

Related references

ISPF Dialog Developer's Guide and Reference

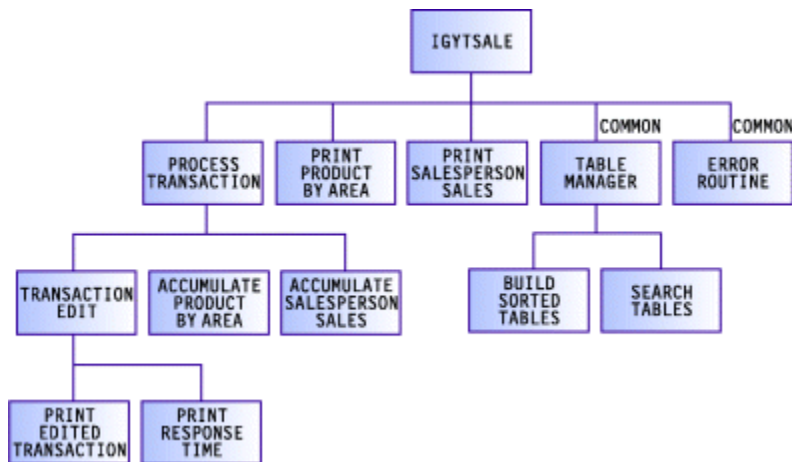
IGYTSALE: nested program application

Application IGYTSALE tracks product sales and sales commissions for a sporting-goods distributor.

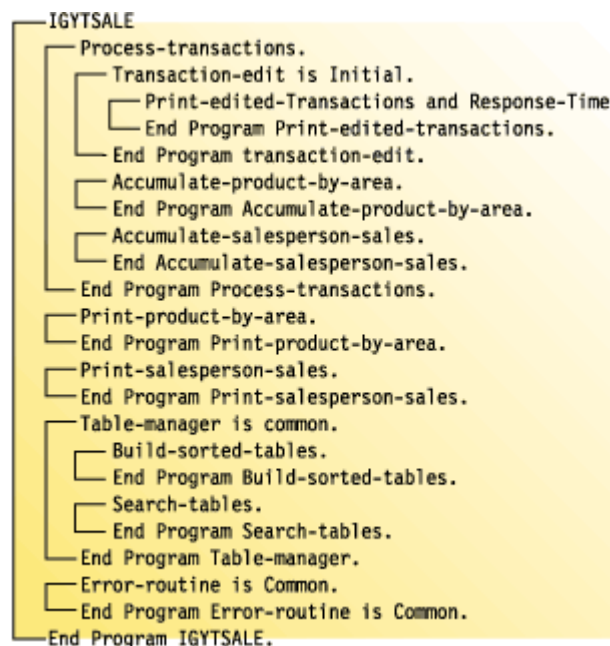
This nested program application does the following tasks:

1. Keeps a record of the product line, customers, and number of salespeople. This data is stored in a file called IGYTABLE.
2. Maintains 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. Processes transactions and report sales by location.
4. Records an individual's sales performance and commission, and prints the results in a report.
5. Reports the sale and shipment dates in local time and UTC (Universal Time Coordinate), and calculates 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

[“Preparing to run IGYTSALE” on page 948](#)

Related references

[“Input data for IGYTSALE” on page 942](#)

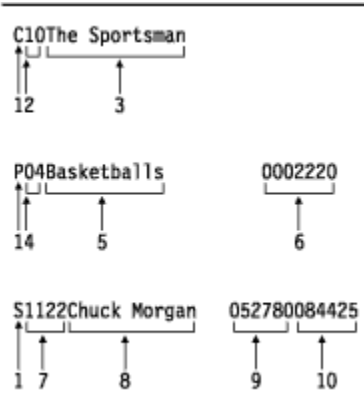
[“Reports produced by IGYTSALE” on page 944](#)

[“Language elements and concepts that are illustrated” on page 949](#)

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

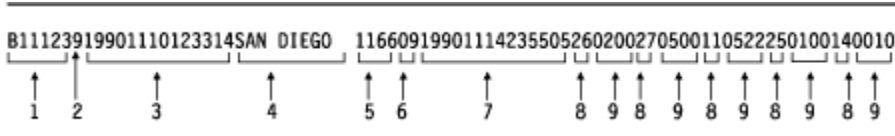
The value of field 1 (C, P, or S) determines the format of the input record. The following sample shows a section of IGYTABLE:

```

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:



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 figures referenced below 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” on page 945](#)

[“Example: IGYTSALE sales analysis by product by area” on page 945](#)

[“Example: IGYTSALE sales and commissions” on page 947](#)

[“Example: IGYTSALE response time from sale to ship” on page 948](#)

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		11/24/2003 03:12		Page: 1
Sales Order	Inv. Items	Sales Time Stamp	Sales Area	Sales Pers	Invalid Edited Transactions Cust. 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 -Salesperson number not numeric -Product code not in product- -Date of ship time stamp is
invalid						
table						
invalid B11117	3	19901201070833	SAN Diego	1165	66 330200132200120522041100	19901203 Error Descriptions -Sales area not in area-table -Salesperson not in sales-per- -Customer code not in customer- -Product code not in product- -Quantity sold not numeric Error Descriptions -Invoiced items is invalid -Product and quantity not -Date of ship time stamp is
table						
table						
table						
B11123	9	19901110123314	SAN DIEGO	1166	09 260200270500110522250100140010	19901114 Error Descriptions -Invoiced items is invalid -Product and quantity not -Date of ship time stamp is
checked						
invalid B11124	2	19901313510000	SAN DIEGO	1166	11 1 0200042200120a22141100	Error Descriptions -Date of sale time stamp is -Product code is invalid -Date of ship time stamp is
invalid						
invalid 133		8119110000	LOS ANGELES	1166	10 040112110210160321251104	Error Descriptions -Sales order number is invalid -Invoiced items is invalid -Date of sale time stamp is -Product and quantity not -Date of ship time stamp is
invalid						
checked						
invalid C11133	4	1990111944		1166	10 040112110210160321251104	Error Descriptions -Date of sale time stamp is -Sales area is missing -Date of ship time stamp is
invalid						
invalid C11138	4	19901117091530	LOS ANGELES	1155	113200102010260321250004	19901119 Error Descriptions -Customer code is invalid Error Descriptions -Invoiced items 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		11/24/2003 03:12		Page: 1
Sales Analysis By Product By Area						
Areas of Sale						
Product Codes	CNTRL COAST	CNTRL VALLEY	LOS ANGELES	NORTH COAST	SAN DIEGO	SF BAY AREA
Totals						Product
====						
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
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

Product Number	26					
Racketball Rackets						
Units Sold	21		862	194	944	31
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

=====						
Total Units Sold	16503	20139	20016	15346	29812	17394 *
Total Sales	\$1,441,929.40	\$968,473.60	\$5,290,487.50	\$128,198.70	\$3,163,713.90	\$3,274,945.70 *

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	11/24/2003	03:12	Page: 1
Sales and Commission Report						
Salesperson:	Billy Jim Bob					
Customers:	Number of Orders	Products Ordered	Total for Order	Discount (if any)	Discount Amount	Commission 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 Orders	Products Ordered	Total for Order	Discount (if any)	Discount Amount	Commission 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 Orders	Products Ordered	Total for Order	Discount (if any)	Discount Amount	Commission Earned
Sports Stop	1	23	\$32.20	2.25%	\$7.22	\$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 Orders	Products Ordered	Total for Order	Discount (if any)	Discount Amount	Commission 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 Orders	Products Ordered	Total for Order	Discount (if any)	Discount Amount	Commission 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 Orders	Products Ordered	Total for Order	Discount (if any)	Discount Amount	Commission 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: Monday   COBOL SPORTS   11/24/2003   03:12   Page: 1
Response Time from USA Sale to European Ship
Prod Code   Units Sold   Sale Date/Time(PST)   Ship Date   Ship Day   Response Time
          YYYMMDD   HHMMSS   YYYMMDD
-----
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

```

Preparing to run IGYTSALE

All files required by the IGYTSALE program (IGYTSALE, IGYTCRC, IGYTPRC, IGYTSRC, IGYTABLE, and IGYTRANA) are on the product installation tape in the IGY.V6R4M0.SIGYSAMP data set.

You can change data-set names and procedure-names at installation time. Check with your system programmer to verify these names.

Do not change these options in the CBL statement in the source file for IGYTSALE:

- NONUMBER
- SEQUENCE
- 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 941](#)

Related tasks

[“Running IGYTSALE” on page 949](#)

Related references

[“Input data for IGYTSALE” on page 942](#)

[“Reports produced by IGYTSALE” on page 944](#)

[“Language elements and concepts that are illustrated” on page 949](#)

Running IGYTSALE

Use the following JCL to compile, link-edit, and run the IGYTSALE program. If you want only to compile or only to compile and link-edit the program, change the IGYWCLG cataloged procedure.

Insert the accounting information for your system or installation in the fields that are shown in lowercase letters.

```
//IGYTSALE JOB (acct-info), 'IGYTSALE',MSGLEVEL=(1,1),TIME=(0,29)
//TEST EXEC IGYWCLG
//COBOL.SYSLIB DD DSN=IGY.V6R4M0.SIGYSAMP,DISP=SHR
//COBOL.SYSIN DD DSN=IGY.V6R4M0.SIGYSAMP(IGYTSALE),DISP=SHR
//GO.SYSOUT DD SYSOUT=A
//GO.IGYTABLE DD DSN=IGY.V6R4M0.SIGYSAMP(IGYTABLE),DISP=SHR
//GO.IGYTRANS DD DSN=IGY.V6R4M0.SIGYSAMP(IGYTRANA),DISP=SHR
//GO.IGYPRINT DD SYSOUT=A,DCB=BLKSIZE=133
//GO.IGYPRT2 DD SYSOUT=A,DCB=BLKSIZE=133
//
```

Language elements and concepts that are illustrated

The sample programs illustrate several COBOL language elements and concepts.

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

Language element or concept	Sequence string
ADD . . . TO	IS4550
AFTER ADVANCING	IS2700
AFTER PAGE	IS2600
ALL	IS4200
ASSIGN	IS1101
AUTHOR	IA0040
CALL	IS0800
Callable services (Language Environment): 1. CEEDATM: format date or time output 2. CEEDCOD: feedback code check 3. CEEGMT0: UTC offset from local time 4. CEELOCT: local date and time 5. CEESECS: convert time stamp to seconds	1. IS0875, IS2575 2. IS0905 3. IS0904 4. IS0850 5. 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
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

Language element or concept	Sequence string
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
Intrinsic functions: 1. CURRENT-DATE 2. MAX 3. MEAN 4. MEDIAN 5. MIN 6. STANDARD-DEVIATION 7. UPPER-CASE 8. VARIANCE 9. WHEN-COMPILED	1. IA9005 2. IA9235 3. IA9215 4. IA9220 5. IA9240 6. IA9230 7. IA9015 8. IA9225 9. IA9000
IS (optional in all clauses)	IS0700
LABEL RECORDS (optional)	IS1150
LINKAGE SECTION	IS4900
Mixing of indexes and subscripts	IS3500

Language element or concept	Sequence string
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
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

Language element or concept	Sequence string
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

Appendix I. Accessing JCL parameters

You can pass a parameter string from JCL to a COBOL program using the PARM= keyword of the EXEC statement. You can access these parameters either by standard COBOL coding, or by calling the CEE3PR2 Language Environment callable service.

Using COBOL coding

You must define the LINKAGE SECTION record (level-01) that is to receive the *user_parameter* data passed by the PARM string, taking into account the halfword parameter list length field that is inserted in front of the string by the system.

The program can test this field length for nonzero to verify that PARM-string data has in fact been passed. For example:

```
LINKAGE SECTION.  
01 PARMDATA.  
    05 STRINGLEN PIC 9(4) USAGE COMP.  
    05 STRINGPARM PIC X(80).  
PROCEDURE DIVISION USING PARMDATA.  
    IF STRINGLEN > 0 . . .
```

For more information, see [“Coding the LINKAGE SECTION” on page 555](#).

Using CEE3PR2

You must define parameters to the CEE3PR2 callable service, without the need to add parameters to your PROCEDURE DIVISION USING statement.

```
WORKING-STORAGE SECTION.  
01 PARMLen PIC S9(9) BINARY.  
01 PARMSTR.  
    02 STR1-LENGTH PIC S9(4) BINARY.  
    02 STR1-STRING.  
        03 STR1-CHAR PIC X  
            OCCURS 0 TO 256 TIMES  
            DEPENDING ON STR1-LENGTH.  
    . . .  
    CALL "CEE3PR2" USING PARMLen, PARMSTR, FC.
```

For more information about the CEE3PR2 callable service, see CEE3PR2 in the [Language Environment Programming Reference](#).

Related tasks

[“Accessing main program parameters under z/OS” on page 564](#)

Appendix J. Accessibility features for Enterprise COBOL for z/OS

Accessibility features assist users who have a disability, such as restricted mobility or limited vision, to use information technology content successfully. The accessibility features in z/OS provide accessibility for Enterprise COBOL for z/OS.

Accessibility features

z/OS includes the following major accessibility features:

- Interfaces that are commonly used by screen readers and screen-magnifier software
- Keyboard-only navigation
- Ability to customize display attributes such as color, contrast, and font size

z/OS uses the latest W3C Standard, [WAI-ARIA 1.0](http://www.w3.org/TR/wai-aria/) (<http://www.w3.org/TR/wai-aria/>), to ensure compliance to [US Section 508](https://www.access-board.gov/ict/) (<https://www.access-board.gov/ict/>) and [Web Content Accessibility Guidelines \(WCAG\) 2.0](http://www.w3.org/TR/WCAG20/) (<http://www.w3.org/TR/WCAG20/>). To take advantage of accessibility features, use the latest release of your screen reader in combination with the latest web browser that is supported by this product.

The Enterprise COBOL for z/OS online product documentation in IBM Knowledge Center is enabled for accessibility. The accessibility features of IBM Knowledge Center are described at <http://www.ibm.com/support/knowledgecenter/en/about/releasenotes.html>.

Keyboard navigation

Users can access z/OS user interfaces by using TSO/E or ISPF.

Users can also access z/OS services by using IBM Developer for z/OS.

For information about accessing these interfaces, see the following publications:

- [z/OS TSO/E Primer](http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/ikj4p120) (<http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/ikj4p120>)
- [z/OS TSO/E User's Guide](http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/ikj4c240/APPENDIX1.3) (<http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/ikj4c240/APPENDIX1.3>)
- [z/OS ISPF User's Guide Volume I](http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/ispzug70) (<http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/ispzug70>)

These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

Interface information

The Enterprise COBOL for z/OS online product documentation is available in IBM Knowledge Center, which is viewable from a standard web browser.

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

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

Assistive technology products work with the user interfaces that are found in z/OS. For specific guidance information, see the documentation for the assistive technology product that you use to access z/OS interfaces.

Related accessibility information

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

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

IBM and accessibility

For more information about the commitment that IBM has to accessibility, see [IBM Accessibility](http://www.ibm.com/able) (www.ibm.com/able).

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who want to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Intellectual Property Dept. for Rational Software
IBM Corporation
5 Technology Park Drive
Westford, MA 01886
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1991, 2024.

PRIVACY POLICY CONSIDERATIONS:

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, or to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at <http://www.ibm.com/privacy> and IBM's Online Privacy Statement at <http://www.ibm.com/privacy/details> in the section entitled "Cookies, Web Beacons and Other Technologies,"

and the "IBM Software Products and Software-as-a-Service Privacy Statement" at <http://www.ibm.com/software/info/product-privacy>.

Trademarks

IBM, the IBM logo, and [ibm.com](http://www.ibm.com)[®] are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other product and service names might be trademarks of IBM or other companies.

Glossary

The terms in this glossary are defined in accordance with their meaning in COBOL. These terms might or might not have the same meaning in other languages.

This glossary includes terms and definitions from the following publications:

- *ANSI INCITS 23-1985, Programming languages - COBOL*, as amended by *ANSI INCITS 23a-1989, Programming Languages - COBOL - Intrinsic Function Module for COBOL*, and *ANSI INCITS 23b-1993, Programming Languages - Correction Amendment for COBOL*
- *ISO 1989:1985, Programming languages - COBOL*, as amended by *ISO/IEC 1989/AMD1:1992, Programming languages - COBOL: Intrinsic function module* and *ISO/IEC 1989/AMD2:1994, Programming languages - Correction and clarification amendment for COBOL*
- *ANSI X3.172-2002, American National Standard Dictionary for Information Systems*
- *INCITS/ISO/IEC 1989-2002, Information technology - Programming languages - COBOL*
- *INCITS/ISO/IEC 1989:2014, Information technology - Programming languages, their environments and system software interfaces - Programming language COBOL*

American National Standard definitions are preceded by an asterisk (*).

A

* abbreviated combined relation condition

The combined condition that results from the explicit omission of a common subject or a common subject and common relational operator in a consecutive sequence of relation conditions.

abend

Abnormal termination of a program.

above the 2 GB bar

Storage located above the so-called 2 GB bar (or boundary). This storage is only addressable by AMODE 64 programs.

above the 16 MB line

Storage located above the so-called 16 MB line (or boundary) but below the 2 GB bar. This storage is not addressable by AMODE 24 programs. Before IBM introduced the MVS/XA architecture in the 1980s, the virtual storage for a program was limited to 16 MB. Programs that have been link-edited as AMODE 24 can address only 16 MB of space, as though they were kept under an imaginary storage line. Since VS COBOL II, a program can have AMODE 31 and can be loaded above the 16 MB line.

* access mode

The manner in which records are to be operated upon within a file.

* actual decimal point

The physical representation, using the decimal point characters period (.) or comma (,), of the decimal point position in a data item.

actual document encoding

For an XML document, one of the following encoding categories that the XML parser determines by examining the first few bytes of the document:

- ASCII
- EBCDIC
- UTF-8
- UTF-16, either big-endian or little-endian
- Other unsupported encoding
- No recognizable encoding

*** 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 position

See *character position*.

alphabetic data item

A data item that is described with a PICTURE character string that contains only the symbol A. An alphabetic data item has USAGE DISPLAY.

*** alphanumeric character**

Any character in the single-byte character set of the computer.

alphanumeric data item

A general reference to a data item that is described implicitly or explicitly as USAGE DISPLAY, and that has category alphanumeric, alphanumeric-edited, or numeric-edited.

alphanumeric-edited data item

A data item that is described by a PICTURE character string that contains at least one instance of the symbol A or X and at least one of the simple insertion symbols B, @, or /. An alphanumeric-edited data item has USAGE DISPLAY.

*** alphanumeric function**

A function whose value is composed of a string of one or more characters from the alphanumeric character set of the computer.

alphanumeric group item

A group item that is defined without a GROUP-USAGE NATIONAL clause. For operations such as INSPECT, STRING, and UNSTRING, an alphanumeric group item is processed as though all its content were described as USAGE DISPLAY regardless of the actual content of the group. For operations that require processing of the elementary items within a group, such as MOVE CORRESPONDING, ADD CORRESPONDING, or INITIALIZE, an alphanumeric group item is processed using group semantics.

alphanumeric literal

A literal that has an opening delimiter from the following set: ' , " , X ' , X " , Z ' , or Z " . The string of characters can include any character in the character set of the computer.

*** alternate record key**

A key, other than the prime record key, whose contents identify a record within an indexed file.

ANSI (American National Standards Institute)

An organization that consists of producers, consumers, and general-interest groups and establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

argument

(1) An identifier, a literal, an arithmetic expression, or a function-identifier that specifies a value to be used in the evaluation of a function. (2) An operand of the USING phrase of a CALL or INVOKE statement, used for passing values to a called program or an invoked method.

*** arithmetic expression**

A numeric literal, an identifier representing a numeric elementary item, such identifiers and literals separated by arithmetic operators, two arithmetic expressions separated by an arithmetic operator, or an arithmetic expression enclosed in parentheses.

*** arithmetic operation**

The process caused by the execution of an arithmetic statement, or the evaluation of an arithmetic expression, that results in a mathematically correct solution to the arguments presented.

*** arithmetic operator**

A single character, or a fixed two-character combination that belongs to the following set:

Character	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

*** arithmetic statement**

A statement that causes an arithmetic operation to be executed. The arithmetic statements are ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT.

array

An aggregate that consists of data objects, each of which can be uniquely referenced by subscripting. An array is roughly analogous to a COBOL table.

*** ascending key**

A key upon the values of which data is ordered, starting with the lowest value of the key up to the highest value of the key, in accordance with the rules for comparing data items.

ASCII

American National Standard Code for Information Interchange. The standard code uses a coded character set that is based on 7-bit coded characters (8 bits including parity check). The standard is used for information interchange between data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters.

IBM has defined an extension to ASCII (characters 128-255).

ASCII DBCS

See *double-byte ASCII*.

assignment-name

A name that identifies the organization of a COBOL file and the name by which it is known to the system.

*** assumed decimal point**

A decimal point position that does not involve the existence of an actual character in a data item. The assumed decimal point has logical meaning but no physical representation.

AT END condition

A condition that is caused during the execution of a READ, RETURN, or SEARCH statement under certain conditions:

- A READ statement runs on a sequentially accessed file when no next logical record exists in the file, or when the number of significant digits in the relative record number is larger than the size of the relative key data item, or when an optional input file is not available.
- A RETURN statement runs when no next logical record exists for the associated sort or merge file.
- A SEARCH statement runs when the search operation terminates without satisfying the condition specified in any of the associated WHEN phrases.

B

basic character set

The basic set of characters used in writing words, character-strings, and separators of the language. The basic character set is implemented in single-byte EBCDIC. The extended character set includes DBCS characters, which can be used in comments, literals, and user-defined words.

Synonymous with *COBOL character set* in the 85 COBOL Standard.

batch compilation

Synonymous with *sequence of programs*.

big-endian

The default format that the mainframe and the AIX workstation use to store binary data and UTF-16 characters. In this format, the least significant byte of a binary data item is at the highest address and the least significant byte of a UTF-16 character is at the highest address. Compare with *little-endian*.

binary item

A numeric data item that is represented in binary notation (on the base 2 numbering system). The decimal equivalent consists of the decimal digits 0 through 9, plus an operational sign. The leftmost bit of the item is the operational sign.

binary search

A dichotomizing search in which, at each step of the search, the set of data elements is divided by two; some appropriate action is taken in the case of an odd number.

*** block**

A physical unit of data that is normally composed of one or more logical records. For mass storage files, a block can contain a portion of a logical record. The size of a block has no direct relationship to the size of the file within which the block is contained or to the size of the logical records that are either contained within the block or that overlap the block. Synonymous with *physical record*.

boolean condition

A boolean condition determines whether a boolean literal is true or false. A boolean condition can only be used in a constant conditional expression.

boolean literal

Can be either B'1', indicating a true value, or B'0', indicating a false value. Boolean literals can only be used in constant conditional expressions.

breakpoint

A place in a computer program, usually specified by an instruction, where external intervention or a monitor program can interrupt the program as it runs.

buffer

A portion of storage that is used to hold input or output data temporarily.

built-in function

See *intrinsic function*.

business method

A method of an enterprise bean that implements the business logic or rules of an application. (Oracle)

byte

A string that consists of a certain number of bits, usually eight, treated as a unit, and representing a character or a control function.

byte order mark (BOM)

A Unicode character that can be used at the start of UTF-16 or UTF-32 text to indicate the byte order of subsequent text; the byte order can be either big-endian or little-endian.

bytecode

Machine-independent code that is generated by the Java compiler and executed by the Java interpreter. (Oracle)

C**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.

canonical decomposition

A way to represent a single precomposed Unicode character using two or more Unicode characters. A canonical decomposition is typically used to separate latin letters with a diacritical mark so that the latin letter and the diacritical mark are represented individually. See *precomposed character* for an example showing a precomposed Unicode character and its canonical decomposition.

case structure

A program-processing logic in which a series of conditions is tested in order to choose between a number of resulting actions.

cataloged procedure

A set of job control statements that are placed in a partitioned data set called the procedure library (SYS1.PROCLIB). You can use cataloged procedures to save time and reduce errors in coding JCL.

CCSID

See *coded character set identifier*.

century window

A 100-year interval within which any two-digit year is unique. Several types of century window are available to COBOL programmers:

- For the windowing intrinsic functions DATE-T0-YYYYMMDD, DAY-T0-YYYYDDD, and YEAR-T0-YYYY, you specify the century window with *argument-2*.
- For Language Environment callable services, you specify the century window in CEESSEN.

*** character**

The basic indivisible unit of the language.

character encoding unit

A unit of data that corresponds to one code point in a coded character set. One or more character encoding units are used to represent a character in a coded character set. Also known as *encoding unit*.

For USAGE NATIONAL, a character encoding unit corresponds to one 2-byte code point of UTF-16.

For USAGE DISPLAY, a character encoding unit corresponds to a byte.

For USAGE DISPLAY-1, a character encoding unit corresponds to a 2-byte code point in the DBCS character set.

character position

The amount of physical storage or presentation space required to hold or present one character. The term applies to any class of character. For specific classes of characters, the following terms apply:

- *Alphanumeric character position*, for characters represented in USAGE DISPLAY
- *DBCS character position*, for DBCS characters represented in USAGE DISPLAY-1
- *National character position*, for characters represented in USAGE NATIONAL; synonymous with *character encoding unit* for UTF-16

character set

A collection of elements that are used to represent textual information, but for which no coded representation is assumed. See also *coded character set*.

character string

A sequence of contiguous characters that form a COBOL word, a literal, a PICTURE character string, or a comment-entry. A character string must be delimited by separators.

checkpoint

A point at which information about the status of a job and the system can be recorded so that the job step can be restarted later.

*** class**

The entity that defines common behavior and implementation for zero, one, or more objects. The objects that share the same implementation are considered to be objects of the same class. Classes can be defined hierarchically, allowing one class to inherit from another.

class (object-oriented)

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.

*** class condition**

The proposition (for which a truth value can be determined) that the content of an item is wholly alphabetic, is wholly numeric, is wholly DBCS, is wholly Kanji, or consists exclusively of the characters that are listed in the definition of a class-name.

*** class definition**

The COBOL source unit that defines a class.

class hierarchy

A tree-like structure that shows relationships among object classes. It places one class at the top and one or more layers of classes below it. Synonymous with *inheritance hierarchy*.

*** class identification entry**

An entry in the CLASS-ID paragraph of the IDENTIFICATION DIVISION; this entry contains clauses that specify the class-name and assign selected attributes to the class definition.

class-name (object-oriented)

The name of an object-oriented COBOL class definition.

*** class-name (of data)**

A user-defined word that is defined in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION; this word assigns a name to the proposition (for which a truth value can be defined) that the content of a data item consists exclusively of the characters that are listed in the definition of the class-name.

class object

The runtime object that represents a class.

*** clause**

An ordered set of consecutive COBOL character strings whose purpose is to specify an attribute of an entry.

client

In object-oriented programming, a program or method that requests services from one or more methods in a class.

COBOL character set

The set of characters used in writing COBOL syntax. The complete COBOL character set consists of these characters:

Character	Meaning
0,1, . . . ,9	Digit
A,B, . . . ,Z	Uppercase letter
a,b, . . . ,z	Lowercase letter
	Space
+	Plus sign
-	Minus sign (hyphen)
*	Asterisk
/	Slant (forward slash)
=	Equal sign
\$	Currency sign
,	Comma
;	Semicolon
.	Period (decimal point, full stop)

Character	Meaning
"	Quotation mark
'	Apostrophe
(Left parenthesis
)	Right parenthesis
>	Greater than
<	Less than
:	Colon
_	Underscore

*** COBOL word**

See *word*.

code page

An assignment of graphic characters and control function meanings to all code points. For example, one code page could assign characters and meanings to 256 code points for 8-bit code, and another code page could assign characters and meanings to 128 code points for 7-bit code. For example, one of the IBM code pages for English on the workstation is IBM-1252 and on the host is IBM-1047. A *coded character set*.

code point

A unique bit pattern that is defined in a coded character set (code page). Graphic symbols and control characters are assigned to code points.

coded character set

A set of unambiguous rules that establish a character set and the relationship between the characters of the set and their coded representation. Examples of coded character sets are the character sets as represented by ASCII or EBCDIC code pages or by the UTF-16 encoding scheme for Unicode.

coded character set identifier (CCSID)

An IBM-defined number in the range 1 to 65,535 that identifies a specific code page.

*** collating sequence**

The sequence in which the characters that are acceptable to a computer are ordered for purposes of sorting, merging, comparing, and for processing indexed files sequentially.

*** column**

A byte position within a print line or within a reference format line. The columns are numbered from 1, by 1, starting at the leftmost position of the line and extending to the rightmost position of the line. A column holds one single-byte character.

*** combined condition**

A condition that is the result of connecting two or more conditions with the AND or the OR logical operator. See also *condition* and *negated combined condition*.

combining characters

A Unicode character used to modify other succeeding or preceding Unicode characters. Combining characters are typically Unicode diacritical mark used to modify latin letters. See *precomposed character* for an example of combining character U+0308 (¨) used with latin letter U+0061 (a).

*** comment-entry**

An entry in the IDENTIFICATION DIVISION that is used for documentation and has no effect on execution.

comment line

A source program line represented by an asterisk (*) in the indicator area of the line or by an asterisk followed by greater-than sign (*>) as the first character string in the program text area (Area A plus Area B), and any characters from the character set of the computer that follow in Area A and Area B of that line. A comment line serves only for documentation. A special form of comment line represented

by a slant (/) in the indicator area of the line and any characters from the character set of the computer in Area A and Area B of that line causes page ejection before the comment is printed.

*** common program**

A program that, despite being directly contained within another program, can be called from any program directly or indirectly contained in that other program.

compilation group

Synonymous with *sequence of programs*.

compilation unit

A unit of COBOL source code that can be separately compiled: a program, class, user-defined function, or prototype definition. Also known as a source unit.

compilation variable

A symbolic name for a particular literal value or the value of a compile-time arithmetic expression as specified by the DEFINE directive or by the DEFINE compiler option.

*** compile**

(1) To translate a program expressed in a high-level language into a program expressed in an intermediate language, assembly language, or a computer language. (2) To prepare a machine-language program from a computer program written in another programming language by making use of the overall logic structure of the program, or generating more than one computer instruction for each symbolic statement, or both, as well as performing the function of an assembler.

*** compile time**

The time at which COBOL source code is translated, by a COBOL compiler, to a COBOL object program.

compile-time arithmetic expression

A subset of arithmetic expressions that are specified in the DEFINE and EVALUATE directives or in a constant conditional expression. The difference between compile-time arithmetic expressions and regular arithmetic expressions is that in a compile-time arithmetic expression:

- The exponentiation operator shall not be specified.
- All operands shall be integer numeric literals or arithmetic expressions in which all operands are integer numeric literals.
- The expression shall be specified in such a way that a division by zero does not occur.

compiler

A program that translates source code written in a higher-level language into machine-language object code.

compiler-directing statement

A statement that causes the compiler to take a specific action during compilation. The standard compiler-directing statements are COPY, REPLACE, and USE.

*** complex condition**

A condition in which one or more logical operators act upon one or more conditions. See also *condition*, *negated simple condition*, and *negated combined condition*.

complex ODO

Certain forms of the OCCURS DEPENDING ON clause:

- Variably located item or group: A data item described by an OCCURS clause with the DEPENDING ON option is followed by a nonsubordinate data item or group. The group can be an alphanumeric group or a national group.
- Variably located table: A data item described by an OCCURS clause with the DEPENDING ON option is followed by a nonsubordinate data item described by an OCCURS clause.
- Table with variable-length elements: A data item described by an OCCURS clause contains a subordinate data item described by an OCCURS clause with the DEPENDING ON option.
- Index name for a table with variable-length elements.
- Element of a table with variable-length elements.

component

(1) A functional grouping of related files. (2) In object-oriented programming, a reusable object or program that performs a specific function and is designed to work with other components and applications. JavaBeans is Oracle's architecture for creating components.

composed form

Representation of a precomposed Unicode character through a canonical decomposition. See *precomposed character* for details.

*** computer-name**

A system-name that identifies the computer where the program is to be compiled or run.

condition (exception)

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 (expression)

A status of data at run time for which a truth value can be determined. Where used in this information in or in reference to "condition" (*condition-1*, *condition-2*, . . .) of a general format, the term refers to a conditional expression that consists of either a simple condition optionally parenthesized or a combined condition (consisting of the syntactically correct combination of simple conditions, logical operators, and parentheses) for which a truth value can be determined. See also *simple condition*, *complex condition*, *negated simple condition*, *combined condition*, and *negated combined condition*.

*** conditional expression**

A simple condition or a complex condition specified in an EVALUATE, IF, PERFORM, or SEARCH statement. See also *simple condition* and *complex condition*.

*** conditional phrase**

A phrase that specifies the action to be taken upon determination of the truth value of a condition that results from the execution of a conditional statement.

*** conditional statement**

A statement that specifies that the truth value of a condition is to be determined and that the subsequent action of the object program depends on this truth value.

*** conditional variable**

A data item one or more values of which has a condition-name assigned to it.

*** condition-name**

A user-defined word that assigns a name to a subset of values that a conditional variable can assume; or a user-defined word assigned to a status of an implementor-defined switch or device.

*** condition-name condition**

The proposition (for which a truth value can be determined) that the value of a conditional variable is a member of the set of values attributed to a condition-name associated with the conditional variable.

*** CONFIGURATION SECTION**

A section of the ENVIRONMENT DIVISION that describes overall specifications of source and object programs and class definitions.

CONSOLE

A COBOL environment-name associated with the operator console.

constant conditional expression

A subset of conditional expressions that may be used in IF directives or WHEN phrases of the EVALUATE directives.

A constant conditional expression shall be one of the following items:

- A relation condition in which both operands are literals or arithmetic expressions that contain only literal terms. The condition shall follow the rules for relation conditions, with the following additions:
 - The operands shall be of the same category. An arithmetic expression is of the category numeric.

- If literals are specified and they are not numeric literals, the relational operator shall be “IS EQUAL TO”, “IS NOT EQUAL TO”, “IS =”, “IS NOT =”, or “IS <>”.

See also *relation condition*.

- A defined condition. See also *defined condition*.
- A boolean condition. See also *boolean condition*.
- A complex condition formed by combining the above forms of simple conditions into complex conditions by using AND, OR, and NOT. Abbreviated combined relation conditions shall not be specified. See also *complex condition*.

contained program

A COBOL program that is nested within another COBOL program.

*** contiguous items**

Items that are described by consecutive entries in the DATA DIVISION, and that bear a definite hierarchic relationship to each other.

copybook

A file or library member that contains a sequence of code that is included in the source program at compile time using the COPY statement. The file can be created by the user, supplied by COBOL, or supplied by another product. Synonymous with *copy file*.

*** counter**

A data item used for storing numbers or number representations in a manner that permits these numbers to be increased or decreased by the value of another number, or to be changed or reset to zero or to an arbitrary positive or negative value.

cross-reference listing

The portion of the compiler listing that contains information on where files, fields, and indicators are defined, referenced, and modified in a program.

currency-sign value

A character string that identifies the monetary units stored in a numeric-edited item. Typical examples are \$, USD, and EUR. A currency-sign value can be defined by either the CURRENCY compiler option or the CURRENCY SIGN clause in the SPECIAL -NAMES paragraph of the ENVIRONMENT DIVISION. If the CURRENCY SIGN clause is not specified and the NOCURRENCY compiler option is in effect, the dollar sign (\$) is used as the default currency-sign value. See also *currency symbol*.

currency symbol

A character used in a PICTURE clause to indicate the position of a currency sign value in a numeric-edited item. A currency symbol can be defined by either the CURRENCY compiler option or the CURRENCY SIGN clause in the SPECIAL -NAMES paragraph of the ENVIRONMENT DIVISION. If the CURRENCY SIGN clause is not specified and the NOCURRENCY compiler option is in effect, the dollar sign (\$) is used as the default currency sign value and currency symbol. Multiple currency symbols and currency sign values can be defined. See also *currency sign value*.

*** current record**

In file processing, the record that is available in the record area associated with a file.

*** current volume pointer**

A conceptual entity that points to the current volume of a sequential file.

D

*** data clause**

A clause, appearing in a data description entry in the DATA DIVISION of a COBOL program, that provides information describing a particular attribute of a data item.

*** data description entry**

An entry in the DATA DIVISION of a COBOL program that is composed of a level-number followed by a data-name, if required, and then followed by a set of data clauses, as required.

DATA DIVISION

The division of a COBOL program or method that describes the data to be processed by the program or method: the files to be used and the records contained within them; internal WORKING-STORAGE

records that will be needed; data to be made available in more than one program in the COBOL run unit.

*** data item**

A unit of data (excluding literals) defined by a COBOL program or by the rules for function evaluation.

data set

Synonym for *file*.

*** 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.

DBCS

See *double-byte character set (DBCS)*.

DBCS character

Any character defined in IBM's double-byte character set.

DBCS character position

See *character position*.

DBCS data item

A data item that is described by a PICTURE character string that contains at least one symbol G, or, when the NSYMBOL (DBCS) compiler option is in effect, at least one symbol N. A DBCS data item has USAGE DISPLAY-1.

*** debugging line**

Any line with a D in the indicator area of the line.

*** debugging section**

A section that contains a USE FOR DEBUGGING statement.

*** declarative sentence**

A compiler-directing sentence that consists of a single USE statement terminated by the separator period.

*** declaratives**

A set of one or more special-purpose sections, written at the beginning of the PROCEDURE DIVISION, the first of which is preceded by the key word DECLARATIVE and the last of which is followed by the key words END DECLARATIVES. A declarative is composed of a section header, followed by a USE compiler-directing sentence, followed by a set of zero, one, or more associated paragraphs.

*** de-edit**

The logical removal of all editing characters from a numeric-edited data item in order to determine the unedited numeric value of the item.

defined condition

A compile-time condition that tests whether a compilation variable is defined. Defined conditions are specified in IF directives or WHEN phrases of the EVALUATE directives.

*** delimited scope statement**

Any statement that includes its explicit scope terminator.

*** delimiter**

A character or a sequence of contiguous characters that identify the end of a string of characters and separate that string of characters from the following string of characters. A delimiter is not part of the string of characters that it delimits.

dependent region

In IMS, the MVS virtual storage region that contains message-driven programs, batch programs, or online utilities.

*** descending key**

A key upon the values of which data is ordered starting with the highest value of key down to the lowest value of key, in accordance with the rules for comparing data items.

digit

Any of the numerals from 0 through 9. In COBOL, the term is not used to refer to any other symbol.

*** digit position**

The amount of physical storage required to store a single digit. This amount can vary depending on the usage specified in the data description entry that defines the data item.

*** direct access**

The facility to obtain data from storage devices or to enter data into a storage device in such a way that the process depends only on the location of that data and not on a reference to data previously accessed.

display floating-point data item

A data item that is described implicitly or explicitly as USAGE DISPLAY and that has a PICTURE character string that describes an external floating-point data item.

*** division**

A collection of zero, one, or more sections or paragraphs, called the division body, that are formed and combined in accordance with a specific set of rules. Each division consists of the division header and the related division body. There are four divisions in a COBOL program: Identification, Environment, Data, and Procedure.

*** division header**

A combination of words followed by a separator period that indicates the beginning of a division. The division headers are:

```
IDENTIFICATION DIVISION.  
ENVIRONMENT DIVISION.  
DATA DIVISION.  
PROCEDURE DIVISION.
```

DLL

See *dynamic link library (DLL)*.

DLL application

An application that references imported programs, functions, or variables.

DLL linkage

A CALL in a program that has been compiled with the DLL and NODYNAM options; the CALL resolves to an exported name in a separate module, or to an INVOKE of a method that is defined in a separate module.

do construct

In structured programming, a DO statement is used to group a number of statements in a procedure. In COBOL, an inline PERFORM statement functions in the same way.

do-until

In structured programming, a do-until loop will be executed at least once, and until a given condition is true. In COBOL, a TEST AFTER phrase used with the PERFORM statement functions in the same way.

do-while

In structured programming, a do-while loop will be executed if, and while, a given condition is true. In COBOL, a TEST BEFORE phrase used with the PERFORM statement functions in the same way.

document type declaration

An XML element that contains or points to markup declarations that provide a grammar for a class of documents. This grammar is known as a document type definition, or DTD.

document type definition (DTD)

The grammar for a class of XML documents. See *document type declaration*.

double-byte ASCII

An IBM character set that includes DBCS and single-byte ASCII characters. (Also known as ASCII DBCS.)

double-byte EBCDIC

An IBM character set that includes DBCS and single-byte EBCDIC characters. (Also known as EBCDIC DBCS.)

double-byte character set (DBCS)

A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets. Because each character requires 2 bytes, entering, displaying, and printing DBCS characters requires hardware and supporting software that are DBCS-capable.

DWARF

DWARF was developed by the UNIX International Programming Languages Special Interest Group (SIG). It is designed to meet the symbolic, source-level debugging needs of different languages in a unified fashion by supplying language-independent debugging information. A DWARF file contains debugging data organized into different elements. For more information, see [*DWARF program information*](#) in the *DWARF/ELF Extensions Library Reference*.

*** dynamic access**

An access mode in which specific logical records can be obtained from or placed into a mass storage file in a nonsequential manner and obtained from a file in a sequential manner during the scope of the same OPEN statement.

dynamic CALL

A CALL *literal* statement in a program that has been compiled with the DYNAM option and the NODLL option, or a CALL *identifier* statement in a program that has been compiled with the NODLL option.

dynamic-length

An adjective describing an item whose logical length might change at runtime.

dynamic-length elementary item

An elementary data item whose data declaration entry contains the DYNAMIC LENGTH clause.

dynamic-length group

A group item that contains a subordinate dynamic-length elementary item.

dynamic link library (DLL)

A file that contains executable code and data that are bound to a program at load time or run time, rather than during linking. Several applications can share the code and data in a DLL simultaneously. Although a DLL is not part of the executable file for a program, it can be required for an executable 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). A DSA is allocated upon invocation of a program or function and persists for the duration of the invocation instance. DSAs are generally allocated within stack segments managed by Language Environment.

E*** EBCDIC (Extended Binary-Coded Decimal Interchange Code)**

A coded character set based on 8-bit coded characters.

EBCDIC character

Any one of the symbols included in the EBCDIC (Extended Binary-Coded-Decimal Interchange Code) set.

EBCDIC DBCS

See *double-byte EBCDIC*.

edited data item

A data item that has been modified by suppressing zeros or inserting editing characters or both.

*** editing character**

A single character or a fixed two-character combination belonging to the following set:

Character	Meaning
	Space
0	Zero
+	Plus
-	Minus
CR	Credit
DB	Debit
Z	Zero suppress
*	Check protect
\$	Currency sign
,	Comma (decimal point)
.	Period (decimal point)
/	Slant (forward slash)

EGCS

See *extended graphic character set (EGCS)*.

EJB

See *Enterprise JavaBeans*.

EJB container

A container that implements the EJB component contract of the J2EE architecture. This contract specifies a runtime environment for enterprise beans that includes security, concurrency, life cycle management, transaction, deployment, and other services. An EJB container is provided by an EJB or J2EE server. (Oracle)

EJB server

Software that provides services to an EJB container. An EJB server can host one or more EJB containers. (Oracle)

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 Language Environment, an enclave is analogous to a run unit. An enclave can create other enclaves by using LINK and by using the system() function in C.

encoding unit

See *character encoding unit*.

end class marker

A combination of words, followed by a separator period, that indicates the end of a COBOL class definition. The end class marker is:

```
END CLASS class-name.
```

end method marker

A combination of words, followed by a separator period, that indicates the end of a COBOL method definition. The end method marker is:

```
END METHOD method-name.
```

*** end of PROCEDURE DIVISION**

The physical position of a COBOL source program after which no further procedures appear.

*** end program marker**

A combination of words, followed by a separator period, that indicates the end of a COBOL source program. The end program marker is:

```
END PROGRAM program-name.
```

enterprise bean

A component that implements a business task and resides in an EJB container. (Oracle)

Enterprise JavaBeans

A component architecture defined by Oracle for the development and deployment of object-oriented, distributed, enterprise-level applications.

*** entry**

Any descriptive set of consecutive clauses terminated by a separator period and written in the IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, or DATA DIVISION of a COBOL program.

*** environment clause**

A clause that appears as part of an ENVIRONMENT DIVISION entry.

ENVIRONMENT DIVISION

One of the four main component parts of a COBOL program, class definition, or method definition. The ENVIRONMENT DIVISION describes the computers where the source program is compiled and those where the object program is run. It provides a linkage between the logical concept of files and their records, and the physical aspects of the devices on which files are stored.

environment-name

A name, specified by IBM, that identifies system logical units, printer and card punch control characters, report codes, program switches or all of these. When an environment-name is associated with a mnemonic-name in the ENVIRONMENT DIVISION, the mnemonic-name can be substituted in any format in which such substitution is valid.

environment variable

Any of a number of variables that define some aspect of the computing environment, and are accessible to programs that operate in that environment. Environment variables can affect the behavior of programs that are sensitive to the environment in which they operate.

escape sequence

A sequence of characters that are used to represent certain special characters within string literals and character literals.

Escape sequences consist of two or more characters, the first of which is the backslash (\) character, which is called the "escape character"; the remaining characters determine the interpretation of the escape sequence. For example, \n is an escape sequence that denotes a newline character.

Escape sequences are used in programming languages such as C, C++, Java, or Python. COBOL does not have the concept of "escape sequence" or "escape character". To handle special characters within COBOL literals, see *Basic alphanumeric literals* and *DBCS literals* in the *Enterprise COBOL for z/OS Language Reference*.

execution time

See *run time*.

execution-time environment

See *runtime environment*.

*** 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.

extended graphic character set (EGCS)

A graphic character set, such as a kanji character set, that requires two bytes to identify each graphic character. It is refined and replaced by *double-byte character set (DBCS)*.

Extensible Markup Language

See *XML*.

extensions

COBOL syntax and semantics supported by IBM compilers in addition to those described in the 85 COBOL Standard.

external code page

For XML documents, the value specified by the CODEPAGE compiler option.

*** external data**

The data that is described in a program as external data items and external file connectors.

*** external data item**

A data item that is described as part of an external record in one or more programs of a run unit and that can be referenced from any program in which it is described.

*** external data record**

A logical record that is described in one or more programs of a run unit and whose constituent data items can be referenced from any program in which they are described.

external decimal data item

See *zoned decimal data item* and *national decimal data item*.

*** external file connector**

A file connector that is accessible to one or more object programs in the run unit.

external floating-point data item

See *display floating-point data item* and *national floating-point data item*.

external program

The outermost program. A program that is not nested.

*** external switch**

A hardware or software device, defined and named by the implementor, which is used to indicate that one of two alternate states exists.

F

factory data

Data that is allocated once for a class and shared by all instances of the class. Factory data is declared in the WORKING-STORAGE SECTION of the DATA DIVISION in the FACTORY paragraph of the class definition, and is equivalent to Java private static data.

factory method

A method that is supported by a class independently of an object instance. Factory methods are declared in the FACTORY paragraph of the class definition, and are equivalent to Java public static methods. They are typically used to customize the creation of objects.

*** figurative constant**

A compiler-generated value referenced through the use of certain reserved words.

*** file**

A collection of logical records.

*** file attribute conflict condition**

An unsuccessful attempt has been made to execute an input-output operation on a file and the file attributes, as specified for that file in the program, do not match the fixed attributes for that file.

*** file clause**

A clause that appears as part of any of the following DATA DIVISION entries: file description entry (FD entry) and sort-merge file description entry (SD entry).

*** file connector**

A storage area that contains information about a file and is used as the linkage between a file-name and a physical file and between a file-name and its associated record area.

File-Control

The name of an ENVIRONMENT DIVISION paragraph in which the data files for a given source program are declared.

file control block

Block containing the addresses of I/O routines, information about how they were opened and closed, and a pointer to the file information block.

*** file control entry**

A SELECT clause and all its subordinate clauses that declare the relevant physical attributes of a file.

FILE-CONTROL paragraph

A paragraph in the ENVIRONMENT DIVISION in which the data files for a given source unit are declared.

*** file description entry**

An entry in the FILE SECTION of the DATA DIVISION that is composed of the level indicator FD, followed by a file-name, and then followed by a set of file clauses as required.

*** file-name**

A user-defined word that names a file connector described in a file description entry or a sort-merge file description entry within the FILE SECTION of the DATA DIVISION.

*** file organization**

The permanent logical file structure established at the time that a file is created.

file position indicator

A conceptual entity that contains the value of the current key within the key of reference for an indexed file, or the record number of the current record for a sequential file, or the relative record number of the current record for a relative file, or indicates that no next logical record exists, or that an optional input file is not available, or that the AT END condition already exists, or that no valid next record has been established.

*** FILE SECTION**

The section of the DATA DIVISION that contains file description entries and sort-merge file description entries together with their associated record descriptions.

file system

The collection of files that conform to a specific set of data-record and file-description protocols, and a set of programs that manage these files.

*** fixed file attributes**

Information about a file that is established when a file is created and that cannot subsequently be changed during the existence of the file. These attributes include the organization of the file (sequential, relative, or indexed), the prime record key, the alternate record keys, the code set, the minimum and maximum record size, the record type (fixed or variable), the collating sequence of the keys for indexed files, the blocking factor, the padding character, and the record delimiter.

*** fixed-length record**

A record associated with a file whose file description or sort-merge description entry requires that all records contain the same number of bytes.

fixed-point item

A numeric data item defined with a PICTURE clause that specifies the location of an optional sign, the number of digits it contains, and the location of an optional decimal point. The format can be either binary, packed decimal, or external decimal.

floating comment indicators (*>)

A floating comment indicator indicates a comment line if it is the first character string in the program-text area (Area A plus Area B), or indicates an inline comment if it is after one or more character strings in the program-text area.

floating point

A format for representing numbers in which a real number is represented by a pair of distinct numerals. In a floating-point representation, the real number is the product of the fixed-point part (the first numeral) and a value obtained by raising the implicit floating-point base to a power denoted by the exponent (the second numeral). For example, a floating-point representation of the number 0.0001234 is 0.1234 -3, where 0.1234 is the mantissa and -3 is the exponent.

floating-point data item

A numeric data item that contains a fraction and an exponent. Its value is obtained by multiplying the fraction by the base of the numeric data item raised to the power that the exponent specifies.

*** format**

A specific arrangement of a set of data.

*** function**

A temporary data item whose value is determined at the time the function is referenced during the execution of a statement.

*** function-identifier**

A syntactically correct combination of character strings and separators that references a function. The data item represented by a function is uniquely identified by a function-name with its arguments, if any. A function-identifier can include a reference-modifier. A function-identifier that references an alphanumeric function can be specified anywhere in the general formats that an identifier can be specified, subject to certain restrictions. A function-identifier that references an integer or numeric function can be referenced anywhere in the general formats that an arithmetic expression can be specified.

function-name

A word that names the mechanism whose invocation, along with required arguments, determines the value of a function.

function-pointer data item

A data item in which a pointer to an entry point can be stored. A data item defined with the USAGE IS FUNCTION-POINTER clause contains the address of a function entry point. Typically used to communicate with C and Java programs.

G**garbage collection**

The automatic freeing by the Java runtime system of the memory for objects that are no longer referenced.

*** 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.

global reference

A reference to an object that is outside the scope of a method.

group item

(1) A data item that is composed of subordinate data items. See *alphanumeric group item* and *national group item*. (2) When not qualified explicitly or by context as a national group or an alphanumeric group, the term refers to groups in general.

grouping separator

A character used to separate units of digits in numbers for ease of reading. The default is the character comma.

H**header label**

(1) A data-set label that precedes the data records in a unit of recording media. (2) Synonym for *beginning-of-file label*.

hide (a method)

To redefine (in a subclass) a factory or static method defined with the same method-name in a parent class. Thus, the method in the subclass *hides* the method in the parent class.

*** high-order end**

The leftmost character of a string of characters.

hiperspace

In a z/OS environment, a range of up to 2 GB of contiguous virtual storage addresses that a program can use as a buffer.

I**IBM COBOL extension**

COBOL syntax and semantics supported by IBM compilers in addition to those described in the 85 COBOL 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, class, or method. 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.

IGZCBSN

The bootstrap routine for COBOL/370 1.1. It must be link-edited with any module that contains a COBOL/370 1.1 program.

IGZCBSO

The bootstrap routine for COBOL for MVS & VM 1.2, COBOL for OS/390 & VM and Enterprise COBOL. It must be link-edited with any module that contains a COBOL for MVS & VM 1.2, COBOL for OS/390 & VM or Enterprise COBOL program.

IGZEBST

The bootstrap routine for VS COBOL II. It must be link-edited with any module that contains a VS COBOL II program.

ILC

InterLanguage Communication. Interlanguage communication is defined as programs that call or are called by other high-level languages. Assembler is not considered a high-level language; thus, calls to and from assembler programs are not considered ILC.

*** 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.

IMS

Information Management System, IBM licensed product. IMS supports hierarchical databases, data communication, translation processing, and database backout and recovery.

*** index**

A computer storage area or register, the content of which represents the identification of a particular element in a table.

*** index data item**

A data item in which the values associated with an index-name can be stored in a form specified by the implementor.

indexed data-name

An identifier that is composed of a data-name, followed by one or more index-names enclosed in parentheses.

*** indexed file**

A file with indexed organization.

*** indexed organization**

The permanent logical file structure in which each record is identified by the value of one or more keys within that record.

indexing

Synonymous with *subscripting* using index-names.

*** index-name**

A user-defined word that names an index associated with a specific table.

inheritance

A mechanism for using the implementation of a class as the basis for another class. By definition, the inheriting class conforms to the inherited classes. Enterprise COBOL does not support *multiple inheritance*; a subclass has exactly one immediate superclass.

inheritance hierarchy

See *class hierarchy*.

*** initial program**

A program that is placed into an initial state every time the program is called in a run unit.

*** initial state**

The state of a program when it is first called in a run unit.

inline

In a program, instructions that are executed sequentially, without branching to routines, subroutines, or other programs.

inline comments

An inline comment is identified by a floating comment indicator (*>) preceded by one or more character-strings in the program-text area, and can be written on any line of a compilation group. All characters that follow the floating comment indicator up to the end of area B are comment text.

*** input file**

A file that is opened in the input mode.

*** input mode**

The state of a file after execution of an OPEN statement, with the INPUT phrase specified, for that file and before the execution of a CLOSE statement, without the REEL or UNIT phrase for that file.

*** input-output file**

A file that is opened in the I-O mode.

*** INPUT-OUTPUT SECTION**

The section of the ENVIRONMENT DIVISION that names the files and the external media required by an object program or method and that provides information required for transmission and handling of data at run time.

*** input-output statement**

A statement that causes files to be processed by performing operations on individual records or on the file as a unit. The input-output statements are ACCEPT (with the identifier phrase), CLOSE, DELETE, DISPLAY, OPEN, READ, REWRITE, SET (with the TO ON or TO OFF phrase), START, and WRITE.

*** input procedure**

A set of statements, to which control is given during the execution of a format 1 SORT statement, for the purpose of controlling the release of specified records to be sorted.

instance data

Data that defines the state of an object. The instance data introduced by a class is defined in the WORKING-STORAGE SECTION of the DATA DIVISION in the OBJECT paragraph of the class definition. The state of an object also includes the state of the instance variables introduced by classes that are inherited by the current class. A separate copy of the instance data is created for each object instance.

*** integer**

(1) A numeric literal that does not include any digit positions to the right of the decimal point. (2) A numeric data item defined in the DATA DIVISION that does not include any digit positions to the right of the decimal point. (3) A numeric function whose definition provides that all digits to the right of the decimal point are zero in the returned value for any possible evaluation of the function.

integer function

A function whose category is numeric and whose definition does not include any digit positions to the right of the decimal point.

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.

interlanguage communication (ILC)

The ability of routines written in different programming languages to communicate. ILC support lets you readily build applications from component routines written in a variety of languages.

intermediate result

An intermediate field that contains the results of a succession of arithmetic operations.

*** internal data**

The data that is described in a program and excludes all external data items and external file connectors. Items described in the LINKAGE SECTION of a program are treated as internal data.

*** internal data item**

A data item that is described in one program in a run unit. An internal data item can have a global name.

internal decimal data item

A data item that is described as USAGE PACKED-DECIMAL or USAGE COMP-3, and that has a PICTURE character string that defines the item as numeric (a valid combination of symbols 9, S, P, or V). Synonymous with *packed-decimal data item*.

*** internal file connector**

A file connector that is accessible to only one object program in the run unit.

internal floating-point data item

A data item that is described as USAGE COMP-1 or USAGE COMP-2. COMP-1 defines a single-precision floating-point data item. COMP-2 defines a double-precision floating-point data item. There is no PICTURE clause associated with an internal floating-point data item.

*** intrarecord data structure**

The entire collection of groups and elementary data items from a logical record that a contiguous subset of the data description entries defines. These data description entries include all entries

whose level-number is greater than the level-number of the first data description entry describing the intra-record data structure.

intrinsic function

A predefined function, such as a commonly used arithmetic function, called by a built-in function reference.

*** invalid key condition**

A condition, at run time, caused when a specific value of the key associated with an indexed or relative file is determined to be not valid.

*** I-O-CONTROL**

The name of an ENVIRONMENT DIVISION paragraph in which object program requirements for rerun points, sharing of same areas by several data files, and multiple file storage on a single input-output device are specified.

*** I-O-CONTROL entry**

An entry in the I-O-CONTROL paragraph of the ENVIRONMENT DIVISION; this entry contains clauses that provide information required for the transmission and handling of data on named files during the execution of a program.

*** I-O mode**

The state of a file after execution of an OPEN statement, with the I-O phrase specified, for that file and before the execution of a CLOSE statement without the REEL or UNIT phrase for that file.

*** I-O status**

A conceptual entity that contains the two-character value indicating the resulting status of an input-output operation. This value is made available to the program through the use of the FILE STATUS clause in the file control entry for the file.

is-a

A relationship that characterizes classes and subclasses in an inheritance hierarchy. Subclasses that have an is-a relationship to a class inherit from that class.

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.

J

J2EE

See *Java 2 Platform, Enterprise Edition (J2EE)*.

Java 2 Platform, Enterprise Edition (J2EE)

An environment for developing and deploying enterprise applications, defined by Oracle. The J2EE platform consists of a set of services, application programming interfaces (APIs), and protocols that provide the functionality for developing multitiered, Web-based applications. (Oracle)

Java Batch Launcher and Toolkit for z/OS (JZOS)

A set of tools that helps you develop z/OS Java applications that run in a traditional batch environment, and that access z/OS system services.

Java batch-processing program (JBP)

An IMS batch-processing program that has access to online databases and output message queues. JBPs run online, but like programs in a batch environment, they are started with JCL or in a TSO session.

Java batch-processing region

An IMS dependent region in which only Java batch-processing programs are scheduled.

Java Database Connectivity (JDBC)

A specification from Oracle that defines an API that enables Java programs to access databases.

Java message-processing program (JMP)

A Java application program that is driven by transactions and has access to online IMS databases and message queues.

Java message-processing region

An IMS dependent region in which only Java message-processing programs are scheduled.

Java Native Interface (JNI)

A programming interface that lets Java code that runs inside a Java virtual machine (JVM) interoperate with applications and libraries written in other programming languages.

Java virtual machine (JVM)

A software implementation of a central processing unit that runs compiled Java programs.

JavaBeans

A portable, platform-independent, reusable component model. (Oracle)

JBP

See *Java batch-processing program (JBP)*.

JDBC

See *Java Database Connectivity (JDBC)*.

JMP

See *Java message-processing program (JMP)*.

job control language (JCL)

A control language used to identify a job to an operating system and to describe the job's requirements.

JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format.

JVM

See *Java virtual machine (JVM)*.

JZOS

See *Java Batch Launcher and Toolkit for z/OS*.

K**K**

When referring to storage capacity, two to the tenth power; 1024 in decimal notation.

*** key**

A data item that identifies the location of a record, or a set of data items that serve to identify the ordering of data.

*** key of reference**

The key, either prime or alternate, currently being used to access records within an indexed file.

*** keyword**

A context-sensitive word or a reserved word whose presence is required when the format in which the word appears is used in a source unit.

kilobyte (KB)

One kilobyte equals 1024 bytes.

L*** language-name**

A system-name that specifies a particular programming language.

Language Environment

Short form of z/OS Language Environment. A set of architectural constructs and interfaces that provides a common runtime environment and runtime services for C, C++, COBOL, FORTRAN and PL/I applications. It is required for programs compiled by Language Environment-conforming compilers and for Java applications.

Language Environment-conforming

A characteristic of compiler products (such as Enterprise COBOL, COBOL for OS/390 & VM, COBOL for MVS & VM, C/C++ for MVS & VM, PL/I for MVS & VM) that produce object code conforming to the Language Environment conventions.

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, inline comments, the separator space, or the separator pseudo-text delimiter in a COBOL library.

Lilian date

The number of days since the beginning of the Gregorian calendar. Day one is Friday, October 15, 1582. The Lilian date format is named in honor of Luigi Lilio, the creator of the Gregorian calendar.

*** lineage-counter**

A special register whose value points to the current position within the page body.

link

(1) The combination of the link connection (the transmission medium) and two link stations, one at each end of the link connection. A link can be shared among multiple links in a multipoint or token-ring configuration. (2) To interconnect items of data or portions of one or more computer programs; for example, linking object programs by a linkage-editor to produce an executable file.

LINKAGE SECTION

The section in the DATA DIVISION of the called program or invoked method that describes data items available from the calling program or invoking method. Both the calling program or invoking method and the called program or invoked method can refer to these data items.

linker

A term that refers to either the z/OS binder (linkage-editor).

literal

A character string whose value is specified either by the ordered set of characters comprising the string or by the use of a figurative constant.

little-endian

The default format that Intel processors use to store binary data and UTF-16 characters. In this format, the most significant byte of a binary data item is at the highest address and the most significant byte of a UTF-16 character is at the highest address. Compare with *big-endian*.

local reference

A reference to an object that is within the scope of your method.

locale

A set of attributes for a program execution environment that indicates culturally sensitive considerations, such as character code page, collating sequence, date and time format, monetary value representation, numeric value representation, or language.

*** LOCAL-STORAGE SECTION**

The section of the DATA DIVISION that defines storage that is allocated and freed on a per-invocation basis, depending on the value assigned in the VALUE clauses.

*** logical operator**

One of the reserved words AND, OR, or NOT. In the formation of a condition, either AND, or OR, or both can be used as logical connectives. NOT can be used for logical negation.

*** logical record**

The most inclusive data item. The level-number for a record is 01. A record can be either an elementary item or a group of items. Synonymous with *record*.

*** low-order end**

The rightmost character of a string of characters.

M

main program

In a hierarchy of programs and subroutines, the first program that receives control when the programs are run within a process.

makefile

A text file that contains a list of the files for your application. The make utility uses this file to update the target files with the latest changes.

*** mass storage**

A storage medium in which data can be organized and maintained in both a sequential manner and a nonsequential manner.

*** mass storage device**

A device that has a large storage capacity, such as a magnetic disk.

*** mass storage file**

A collection of records that is stored in a mass storage medium.

*** megabyte (MB)**

One megabyte equals 1,048,576 bytes.

*** merge file**

A collection of records to be merged by a MERGE statement. The merge file is created and can be used only by the merge function.

message-processing program (MPP)

An IMS application program that is driven by transactions and has access to online IMS databases and message queues.

message queue

The data set on which messages are queued before being processed by an application program or sent to a terminal.

method

Procedural code that defines an operation supported by an object and that is executed by an INVOKE statement on that object.

*** method definition**

The COBOL source code that defines a method.

*** method identification entry**

An entry in the METHOD-ID paragraph of the IDENTIFICATION DIVISION; this entry contains a clause that specifies the method-name.

method invocation

A communication from one object to another that requests the receiving object to execute a method.

method-name

The name of an object-oriented operation. When used to invoke the method, the name can be an alphanumeric or national literal or a category alphanumeric or category national data item. When used in the METHOD-ID paragraph to define the method, the name must be an alphanumeric or national literal.

method hiding

See *hide*.

method overloading

See *overload*.

method overriding

See *override*.

*** 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 program object.

MPP

See *message-processing program (MPP)*.

multitasking

A mode of operation that provides for the concurrent, or interleaved, execution of two or more tasks.

multithreading

Concurrent operation of more than one path of execution within a computer. Synonymous with *multiprocessing*.

N**name**

A word (composed of not more than 30 characters) that defines a COBOL operand.

namespace

See *XML namespace*.

national character

(1) A UTF-16 character in a USAGE NATIONAL data item or national literal. (2) Any character represented in UTF-16.

national character data

A general reference to data represented in UTF-16.

national character position

See *character position*.

national data

See *national character data*.

national data item

A data item of category national, national-edited, or numeric-edited of USAGE NATIONAL.

national decimal data item

An external decimal data item that is described implicitly or explicitly as USAGE NATIONAL and that contains a valid combination of PICTURE symbols 9, S, P, and V.

national-edited data item

A data item that is described by a PICTURE character string that contains at least one instance of the symbol N and at least one of the simple insertion symbols B, Ø, or /. A national-edited data item has USAGE NATIONAL.

national floating-point data item

An external floating-point data item that is described implicitly or explicitly as USAGE NATIONAL and that has a PICTURE character string that describes a floating-point data item.

national group item

A group item that is explicitly or implicitly described with a GROUP-USAGE NATIONAL clause. A national group item is processed as though it were defined as an elementary data item of category national for operations such as INSPECT, STRING, and UNSTRING. This processing ensures correct padding and truncation of national characters, as contrasted with defining USAGE NATIONAL data items within an alphanumeric group item. For operations that require processing of the elementary

items within a group, such as MOVE CORRESPONDING, ADD CORRESPONDING, and INITIALIZE, a national group is processed using group semantics.

*** native character set**

The implementor-defined character set associated with the computer specified in the OBJECT-COMPUTER paragraph.

*** native collating sequence**

The implementor-defined collating sequence associated with the computer specified in the OBJECT-COMPUTER paragraph.

native method

A Java method with an implementation that is written in another programming language, such as COBOL.

*** negated combined condition**

The NOT logical operator immediately followed by a parenthesized combined condition. See also *condition* and *combined condition*.

*** negated simple condition**

The NOT logical operator immediately followed by a simple condition. See also *condition* and *simple condition*.

nested program

A program that is directly contained within another program.

*** next executable sentence**

The next sentence to which control will be transferred after execution of the current statement is complete.

*** next executable statement**

The next statement to which control will be transferred after execution of the current statement is complete.

*** next record**

The record that logically follows the current record of a file.

*** noncontiguous items**

Elementary data items in the WORKING-STORAGE SECTION and LINKAGE SECTION that bear no hierarchic relationship to other data items.

*** noncontiguous items**

Elementary data items in the WORKING-STORAGE and LINKAGE SECTIONS that bear no hierarchic relationship to other data items.

*** nonnumeric item**

A data item whose description permits its content to be composed of any combination of characters taken from the computer's character set. Certain categories of nonnumeric items may be formed from more restricted character sets.

null

A figurative constant that is used to assign, to pointer data items, the value of an address that is not valid. NULLS can be used wherever NULL can be used.

*** numeric character**

A character that belongs to the following set of digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

numeric data item

(1) A data item whose description restricts its content to a value represented by characters chosen from the digits 0 through 9. If signed, the item can also contain a +, -, or other representation of an operational sign. (2) A data item of category numeric, internal floating-point, or external floating-point. A numeric data item can have USAGE DISPLAY, NATIONAL, PACKED-DECIMAL, BINARY, COMP, COMP-1, COMP-2, COMP-3, COMP-4, or COMP-5.

numeric-edited data item

A data item that contains numeric data in a form suitable for use in printed output. The data item can consist of external decimal digits from 0 through 9, the decimal separator, commas, the currency sign,

sign control characters, and other editing characters. A numeric-edited item can be represented in either USAGE DISPLAY or USAGE NATIONAL.

*** numeric function**

A function whose class and category are numeric but that for some possible evaluation does not satisfy the requirements of integer functions.

*** numeric item**

A data item whose description restricts its content to a value represented by characters chosen from the digits from '0' through '9'; if signed, the item may 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

An entity that has state (its data values) and operations (its methods). An object is a way to encapsulate state and behavior. Each object in the class is said to be an instance of the class.

object code

Output from a compiler or assembler that is itself executable machine code or is suitable for processing to produce executable machine code.

*** OBJECT-COMPUTER**

The name of an ENVIRONMENT DIVISION paragraph in which the computer environment, where the object program is run, is described.

*** object computer entry**

An entry in the OBJECT-COMPUTER paragraph of the ENVIRONMENT DIVISION; this entry contains clauses that describe the computer environment in which the object program is to be executed.

object deck

A portion of an object program suitable as input to a linkage-editor. Synonymous with *object module* and *text deck*.

object instance

A single object, of possibly many, instantiated from the specifications in the object paragraph of a COBOL class definition. An object instance has a copy of all the data described in its class definition and all inherited data. The methods associated with an object instance includes the methods defined in its class definition and all inherited methods.

An object instance can be an instance of a Java class.

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 or class definition. Where there is no danger of ambiguity, the word *program* can be used in place of *object program*.

object reference

A value that identifies an instance of a class. If the class is not specified, the object reference is universal and can apply to instances of any class.

*** object time**

The time at which an object program is executed. Synonymous with *run time*.

*** obsolete element**

A COBOL language element in the 85 COBOL Standard that was deleted from the 2002 COBOL Standard.

ODO object

In the example below, X is the object of the OCCURS DEPENDING ON clause (ODO object).

```
WORKING-STORAGE SECTION.  
01 TABLE-1.  
   05 X PIC S9.  
   05 Y OCCURS 3 TIMES  
       DEPENDING ON X PIC X.
```

The value of the ODO object determines how many of the ODO subject appear in the table.

ODO subject

In the example above, Y is the subject of the OCCURS DEPENDING ON clause (ODO subject). The number of Y ODO subjects that appear in the table depends on the value of X.

*** open mode**

The state of a file after execution of an OPEN statement for that file and before the execution of a CLOSE statement without the REEL or UNIT phrase for that file. The particular open mode is specified in the OPEN statement as either INPUT, OUTPUT, I-O, or EXTEND.

*** operand**

(1) The general definition of operand is "the component that is operated upon." (2) For the purposes of this document, any lowercase word (or words) that appears in a statement or entry format can be considered to be an operand and, as such, is an implied reference to the data indicated by the operand.

operation

A service that can be requested of an object.

*** operational sign**

An algebraic sign that is associated with a numeric data item or a numeric literal, to indicate whether its value is positive or negative.

optional file

A file that is declared as being not necessarily available each time the object program is run.

*** optional word**

A reserved word that is included in a specific format only to improve the readability of the language. Its presence is optional to the user when the format in which the word appears is used in a source unit.

*** output file**

A file that is opened in either output mode or extend mode.

*** output mode**

The state of a file after execution of an OPEN statement, with the OUTPUT or EXTEND phrase specified, for that file and before the execution of a CLOSE statement without the REEL or UNIT phrase for that file.

*** output procedure**

A set of statements to which control is given during execution of a format 1 SORT statement after the sort function is completed, or during execution of a MERGE statement after the merge function reaches a point at which it can select the next record in merged order when requested.

overflow condition

A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage.

overload

To define a method with the same name as another method that is available in the same class, but with a different signature. See also *signature*.

override

To redefine an instance method (inherited from a parent class) in a subclass.

P**package**

A group of related Java classes, which can be imported individually or as a whole.

packed-decimal data item

See *internal decimal data item*.

padding character

An alphanumeric or national character that is used to fill the unused character positions in a physical record.

page

A vertical division of output data that represents a physical separation of the data. The separation is based on internal logical requirements or external characteristics of the output medium or both.

*** page body**

That part of the logical page in which lines can be written or spaced or both.

*** paragraph**

In the PROCEDURE DIVISION, a paragraph-name followed by a separator period and by zero, one, or more sentences. In the IDENTIFICATION DIVISION and ENVIRONMENT DIVISION, a paragraph header followed by zero, one, or more entries.

*** paragraph header**

A reserved word, followed by the separator period, that indicates the beginning of a paragraph in the IDENTIFICATION DIVISION and ENVIRONMENT DIVISION. The permissible paragraph headers in the IDENTIFICATION DIVISION are:

```
PROGRAM-ID. (Program IDENTIFICATION
DIVISION)
CLASS-ID. (Class IDENTIFICATION DIVISION)
METHOD-ID. (Method IDENTIFICATION
DIVISION)
AUTHOR.
INSTALLATION.
DATE-WRITTEN.
DATE-COMPILED.
SECURITY.
```

The permissible paragraph headers in the ENVIRONMENT DIVISION are:

```
SOURCE-COMPUTER.
OBJECT-COMPUTER.
SPECIAL-NAMES.
REPOSITORY. (Program or Class
CONFIGURATION SECTION)
FILE-CONTROL.
I-O-CONTROL.
```

*** paragraph-name**

A user-defined word that identifies and begins a paragraph in the PROCEDURE DIVISION.

parameter

(1) Data passed between a calling program and a called program. (2) A data element in the USING phrase of a method invocation. Arguments provide additional information that the invoked method can use to perform the requested operation.

Persistent Reusable JVM

A JVM that can be serially reused for transaction processing by resetting the JVM between transactions. The reset phase restores the JVM to a known initialization state.

*** phrase**

An ordered set of one or more consecutive COBOL character strings that form a portion of a COBOL procedural statement or of a COBOL clause.

*** physical record**

See *block*.

pointer data item

A data item in which address values can be stored. Data items are explicitly defined as pointers with the `USAGE IS POINTER` clause. `ADDRESS OF` special registers are implicitly defined as pointer data items. Pointer data items can be compared for equality or moved to other pointer data items.

port

(1) To modify a computer program to enable it to run on a different platform. (2) In the Internet suite of protocols, a specific logical connector between the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP) and a higher-level protocol or application. A port is identified by a port number.

portability

The ability to transfer an application program from one application platform to another with relatively few changes to the source program.

precomposed character

A single Unicode character that can be represented using two or more Unicode characters through a canonical decomposition. A precomposed character does not have the same physical representation as its composed character form. For example, Unicode character U+00E4 (ä) is a precomposed character that can be represented as a combination of Unicode characters U+0061 + U+0308 (a - latin small letter a + combining diaeresis). A precomposed character is typically used to represent a latin letter with a diacritical mark or some other combining character.

preinitialization

The initialization of the COBOL runtime environment in preparation for multiple calls from programs, especially non-COBOL programs. The environment is not terminated until an explicit termination.

*** prime record key**

A key whose contents uniquely identify a record within an indexed file.

*** priority-number**

A user-defined word that classifies sections in the `PROCEDURE DIVISION` for purposes of segmentation. Segment numbers can contain only the characters 0 through 9. A segment number can be expressed as either one or two digits.

private

As applied to factory data or instance data, accessible only by methods of the class that defines the data.

*** procedure**

A paragraph or group of logically successive paragraphs, or a section or group of logically successive sections, within the `PROCEDURE DIVISION`.

*** procedure branching statement**

A statement that causes the explicit transfer of control to a statement other than the next executable statement in the sequence in which the statements are written in the source code. The procedure branching statements are: `ALTER`, `CALL`, `EXIT`, `EXIT PROGRAM`, `GO TO`, `MERGE` (with the `OUTPUT PROCEDURE` phrase), `PERFORM` and `SORT` (with the `INPUT PROCEDURE` or `OUTPUT PROCEDURE` phrase), `XML PARSE`.

PROCEDURE DIVISION

The COBOL division that contains instructions for solving a problem.

procedure integration

One of the functions of the COBOL optimizer is to simplify calls to performed procedures or contained programs.

`PERFORM` procedure integration is the process whereby a `PERFORM` statement is replaced by its performed procedures. Contained program procedure integration is the process where a call to a contained program is replaced by the program code.

*** procedure-name**

A user-defined word that is used to name a paragraph or section in the `PROCEDURE DIVISION`. It consists of a paragraph-name (which can be qualified) or a section-name.

procedure pointer

A data item in which a pointer to an entry point can be stored. A data item defined with the USAGE IS PROCEDURE-POINTER clause contains the address of a procedure entry point.

procedure-pointer data item

A data item in which a pointer to an entry point can be stored. A data item defined with the USAGE IS PROCEDURE-POINTER clause contains the address of a procedure entry point. Typically used to communicate with COBOL and Language Environment programs.

process

The course of events that occurs during the execution of all or part of a program. Multiple processes can run concurrently, and programs that run within a process can share resources.

program

(1) A sequence of instructions suitable for processing by a computer. Processing may include the use of a compiler to prepare the program for execution, as well as a runtime environment to execute it. (2) A logical assembly of one or more interrelated modules. Multiple copies of the same program can be run in different processes.

program-name

In the IDENTIFICATION DIVISION and the end program marker, a user-defined word or an alphanumeric literal that identifies a COBOL source program.

*** program identification entry**

In the PROGRAM-ID paragraph of the IDENTIFICATION DIVISION, an entry that contains clauses that specify the program-name and assign selected program attributes to the program.

program-name

In the IDENTIFICATION DIVISION and the end program marker, a user-defined word or alphanumeric literal that identifies a COBOL source program.

project

The complete set of data and actions that are required to build a target, such as a dynamic link library (DLL) or other executable (EXE).

*** pseudo-text**

A sequence of text words, comment lines, inline comments, or the separator space in a source program or COBOL library bounded by, but not including, pseudo-text delimiters.

*** pseudo-text delimiter**

Two contiguous equal sign characters (==) used to delimit pseudo-text.

*** punctuation character**

A character that belongs to the following set:

Character	Meaning
,	Comma
;	Semicolon
:	Colon
.	Period (full stop)
"	Quotation mark
(Left parenthesis
)	Right parenthesis
	Space
=	Equal sign

Q

QSAM (Queued Sequential Access Method)

An extended version of the basic sequential access method (BSAM). When this method is used, a queue is formed of input data blocks that are awaiting processing or of output data blocks that have been processed and are awaiting transfer to auxiliary storage or to an output device.

*** 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.

recording mode

The format of the logical records in a file. Recording mode can be F (fixed length), V (variable length), S (spanned), or U (undefined).

recursion

A program calling itself or being directly or indirectly called by one of its called programs.

recursively capable

A program is recursively capable (can be called recursively) if the RECURSIVE attribute is on the PROGRAM-ID statement.

reel

A discrete portion of a storage medium, the dimensions of which are determined by each implementor that contains part of a file, all of a file, or any number of files. Synonymous with *unit* and *volume*.

reentrant

The attribute of a program or routine that lets more than one user share a single copy of a program object.

*** reference format**

A format that provides a standard method for describing COBOL source programs.

reference modification

A method of defining a new category alphanumeric, category DBCS, or category national data item by specifying the leftmost character and length relative to the leftmost character position of a USAGE DISPLAY, DISPLAY-1, or NATIONAL data item.

*** reference-modifier**

A syntactically correct combination of character strings and separators that defines a unique data item. It includes a delimiting left parenthesis separator, the leftmost character position, a colon separator, optionally a length, and a delimiting right parenthesis separator.

*** relation**

See *relational operator* or *relation condition*.

*** relation character**

A character that belongs to the following set:

Character	Meaning
>	Greater than
<	Less than
=	Equal to

*** relation condition**

The proposition (for which a truth value can be determined) that the value of an arithmetic expression, data item, alphanumeric literal, or index-name has a specific relationship to the value of another arithmetic expression, data item, alphanumeric literal, or index name. See also *relational operator*.

*** relational operator**

A reserved word, a relation character, a group of consecutive reserved words, or a group of consecutive reserved words and relation characters used in the construction of a relation condition. The permissible operators and their meanings are:

Character	Meaning
IS GREATER THAN	Greater than
IS >	Greater than
IS NOT GREATER THAN	Not greater than
IS NOT >	Not greater than
IS LESS THAN	Less than
IS <	Less than
IS NOT LESS THAN	Not less than
IS NOT <	Not less than
IS EQUAL TO	Equal to
IS =	Equal to
IS NOT EQUAL TO	Not equal to
IS NOT =	Not equal to
IS GREATER THAN OR EQUAL TO	Greater than or equal to
IS >=	Greater than or equal to

Character

IS LESS THAN OR EQUAL TO

IS <=

Meaning

Less than or equal to

Less than or equal to

*** relative file**

A file with relative organization.

*** relative key**

A key whose contents identify a logical record in a relative file.

*** relative organization**

The permanent logical file structure in which each record is uniquely identified by an integer value greater than zero, which specifies the logical ordinal position of the record in the file.

*** relative record number**

The ordinal number of a record in a file whose organization is relative. This number is treated as a numeric literal that is an integer.

*** reserved word**

A COBOL word that is specified in the list of words that can be used in a COBOL source program, but that must not appear in the program as a user-defined word or system-name.

*** resource**

A facility or service, controlled by the operating system, that an executing program can use.

*** resultant identifier**

A user-defined data item that is to contain the result of an arithmetic operation.

reusable environment

A reusable environment is created when you establish an assembler program as the main program by using either the old COBOL interfaces for preinitialization (RTEREUS runtime option), or the Language Environment interface, CEEPIPI.

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*.**runtime environment**

The environment in which a COBOL program executes.

*** run unit**

A stand-alone object program, or several object programs, that interact by means of COBOL CALL or INVOKE statements and function at run time as an entity.

A run unit is also called an enclave in Language Environment terminology.

S**SBCS**See *single-byte character set (SBCS)*.**scope terminator**

A COBOL reserved word that marks the end of certain PROCEDURE DIVISION statements. It can be either explicit (END-ADD, for example) or implicit (separator period).

*** section**

A set of zero, one, or more paragraphs or entities, called a section body, the first of which is preceded by a section header. Each section consists of the section header and the related section body.

*** section header**

A combination of words followed by a separator period that indicates the beginning of a section in any of these divisions: ENVIRONMENT, DATA, or PROCEDURE. In the ENVIRONMENT DIVISION and

DATA DIVISION, a section header is composed of reserved words followed by a separator period. The permissible section headers in the ENVIRONMENT DIVISION are:

```
CONFIGURATION SECTION.  
INPUT-OUTPUT SECTION.
```

The permissible section headers in the DATA DIVISION are:

```
FILE SECTION.  
WORKING-STORAGE SECTION.  
LOCAL-STORAGE SECTION.  
LINKAGE SECTION.
```

In the PROCEDURE DIVISION, a section header is composed of a section-name, followed by the reserved word SECTION, followed by a separator period.

*** section-name**

A user-defined word that names a section in the PROCEDURE DIVISION.

segmentation

A feature of Enterprise COBOL that is based on the 85 COBOL Standard segmentation module. The segmentation feature uses priority-numbers in section headers to assign sections to fixed segments or independent segments. Segment classification affects whether procedures contained in a segment receive control in initial state or last-used state.

selection structure

A program processing logic in which one or another series of statements is executed, depending on whether a condition is true or false.

*** sentence**

A sequence of one or more statements, the last of which is terminated by a separator period.

*** separately compiled program**

A program that, together with its contained programs, is compiled separately from all other programs.

*** separator**

A character or two or more 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 of programs

A sequence of separate COBOL programs in a single source file that can be input to the compiler.

A sequence of programs is also called a *batch compilation* or a *compilation group*.

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.

session bean

In EJB, an enterprise bean that is created by a client and that usually exists only for the duration of a single client/server session. (Oracle)

77-level-description-entry

A data description entry that describes a noncontiguous data item that has level-number 77.

*** sign condition**

The proposition (for which a truth value can be determined) that the algebraic value of a data item or an arithmetic expression is either less than, greater than, or equal to zero.

signature

(1) The name of an operation and its parameters. (2) The name of a method and the number and types of its formal parameters.

*** simple condition**

Any single condition chosen from this set:

- Relation condition
- Class condition
- Condition-name condition
- Switch-status condition
- Sign condition

See also *condition* and *negated simple condition*.

single-byte character set (SBCS)

A set of characters in which each character is represented by a single byte. See also *ASCII* and *EBCDIC (Extended Binary-Coded Decimal Interchange Code)*.

slack bytes (within records)

Bytes inserted by the compiler between data items to ensure correct alignment of some elementary data items. Slack bytes contain no meaningful data. The SYNCHRONIZED clause instructs the compiler to insert slack bytes when they are needed for proper alignment.

slack bytes (between records)

Bytes inserted by the programmer between blocked logical records of a file, to ensure correct alignment of some elementary data items. In some cases, slack bytes between records improve performance for records processed in a buffer.

*** sort file**

A collection of records to be sorted by a format 1 SORT statement. The sort file is created and can be used by the sort function only.

*** sort-merge file description entry**

An entry in the FILE SECTION of the DATA DIVISION that is composed of the level indicator SD, followed by a file-name, and then followed by a set of file clauses as required.

*** SOURCE-COMPUTER**

The name of an ENVIRONMENT DIVISION paragraph in which the computer environment, where the source program is compiled, is described.

*** source computer entry**

An entry in the SOURCE-COMPUTER paragraph of the ENVIRONMENT DIVISION; this entry contains clauses that describe the computer environment in which the source program is to be compiled.

*** source item**

An identifier designated by a SOURCE clause that provides the value of a printable item.

source program

Although a source program can be represented by other forms and symbols, in this document the term always refers to a syntactically correct set of COBOL statements. A COBOL source program

commences with the IDENTIFICATION DIVISION or a COPY statement and terminates with the end program marker, if specified, or with the absence of additional source program lines.

source unit

A unit of COBOL source code that can be separately compiled: a program or a class definition. Also known as a *compilation unit*.

special character

A character that belongs to the following set:

Character	Meaning
+	Plus sign
-	Minus sign (hyphen)
*	Asterisk
/	Slant (forward slash)
=	Equal sign
\$	Currency sign
,	Comma
;	Semicolon
.	Period (decimal point, full stop)
"	Quotation mark
'	Apostrophe
(Left parenthesis
)	Right parenthesis
>	Greater than
<	Less than
:	Colon
_	Underscore

SPECIAL - NAMES

The name of an ENVIRONMENT DIVISION paragraph in which environment-names are related to user-specified mnemonic-names.

*** special names entry**

An entry in the SPECIAL - NAMES paragraph of the ENVIRONMENT DIVISION; this entry provides means for specifying the currency sign; choosing the decimal point; specifying symbolic characters; relating implementor-names to user-specified mnemonic-names; relating alphabet-names to character sets or collating sequences; and relating class-names to sets of characters.

*** special registers**

Certain compiler-generated storage areas whose primary use is to store information produced in conjunction with the use of a specific COBOL feature.

*** 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.

structured programming

A technique for organizing and coding a computer program in which the program comprises a hierarchy of segments, each segment having a single entry point and a single exit point. Control is passed downward through the structure without unconditional branches to higher levels of the hierarchy.

*** subclass**

A class that inherits from another class. When two classes in an inheritance relationship are considered together, the subclass is the inheritor or inheriting class; the superclass is the inheritee or inherited class.

*** subject of entry**

An operand or reserved word that appears immediately following the level indicator or the level-number in a DATA DIVISION entry.

*** subprogram**

See *called program*.

*** subscript**

An occurrence number that is represented by either an integer, a data-name optionally followed by an integer with the operator + or -, or an index-name optionally followed by an integer with the operator + or -, that identifies a particular element in a table. A subscript can be the word ALL when the subscripted identifier is used as a function argument for a function allowing a variable number of arguments.

*** subscripted data-name**

An identifier that is composed of a data-name followed by one or more subscripts enclosed in parentheses.

substitution character

A character that is used in a conversion from a source code page to a target code page to represent a character that is not defined in the target code page.

*** superclass**

A class that is inherited by another class. See also *subclass*.

surrogate pair

In the UTF-16 format of Unicode, a pair of encoding units that together represents a single Unicode graphic character. The first unit of the pair is called a *high surrogate* and the second a *low surrogate*. The code value of a high surrogate is in the range X'D800' through X'DBFF'. The code value of a low surrogate is in the range X'DC00' through X'DFFF'. Surrogate pairs provide for more characters than the 65,536 characters that fit in the Unicode 16-bit coded character set.

switch-status condition

The proposition (for which a truth value can be determined) that an UPSI switch, capable of being set to an on or off status, has been set to a specific status.

*** symbolic-character**

A user-defined word that specifies a user-defined figurative constant.

syntax

(1) The relationship among characters or groups of characters, independent of their meanings or the manner of their interpretation and use. (2) The structure of expressions in a language. (3) The rules governing the structure of a language. (4) The relationship among symbols. (5) The rules for the construction of a statement.

*** system-name**

A COBOL word that is used to communicate with the operating environment.

T*** table**

A set of logically consecutive items of data that are defined in the DATA DIVISION by means of the OCCURS clause.

*** table element**

A data item that belongs to the set of repeated items comprising a table.

text 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 pseudo-text that is any of the following characters:

- A separator, except for space; a pseudo-text delimiter; and the opening and closing delimiters for alphanumeric literals. The right parenthesis and left parenthesis characters, regardless of context within the library, source program, or pseudo-text, are always considered text words.
- A literal including, in the case of alphanumeric literals, the opening quotation mark and the closing quotation mark that bound the literal.
- Any other sequence of contiguous COBOL characters except comment lines and the word COPY bounded by separators that are neither a separator nor a literal.

thread

A stream of computer instructions (initiated by an application within a process) that is in control of a process.

token

In the COBOL editor, a unit of meaning in a program. A token can contain data, a language keyword, an identifier, or other part of the language syntax.

top-down design

The design of a computer program using a hierarchic structure in which related functions are performed at each level of the structure.

top-down development

See *structured programming*.

trailer-label

(1) A 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.

unbounded table

A table with OCCURS *integer-1* to UNBOUNDED instead of specifying *integer-2* as the upper bound.

Unicode

A universal character encoding standard that supports the interchange, processing, and display of text that is written in any of the languages of the modern world. There are multiple encoding schemes to represent Unicode, including UTF-8, UTF-16, and UTF-32. Enterprise COBOL supports Unicode using UTF-16 in big-endian format as the representation for the national data type.

Uniform Resource Identifier (URI)

A sequence of characters that uniquely names a resource; in Enterprise COBOL, the identifier of a namespace. URI syntax is defined by the document [*Uniform Resource Identifier \(URI\): Generic Syntax*](#).

unit

A module of direct access, the dimensions of which are determined by IBM.

universal object reference

A data-name that can refer to an object of any class.

unrestricted storage

In AMODE 31, unrestricted storage is below the 2 GB bar and can be above or below the 16 MB line.

In AMODE 64, unrestricted storage encompasses all the storage available to your program, both above and below the 2 GB bar.

*** unsuccessful execution**

The attempted execution of a statement that does not result in the execution of all the operations specified by that statement. The unsuccessful execution of a statement does not affect any data referenced by that statement, but can affect status indicators.

UPSI switch

A program switch that performs the functions of a hardware switch. Eight are provided: UPSI-0 through UPSI-7.

URI

See *Uniform Resource Identifier (URI)*.

*** user-defined word**

A COBOL word that must be supplied by the user to satisfy the format of a clause or statement.

V*** variable**

A data item whose value can be changed by execution of the object program. A variable used in an arithmetic expression must be a numeric elementary item.

variable-length item

A group item that contains a table described with the *DEPENDING* phrase of the *OCCURS* clause.

*** variable-length record**

A record associated with a file whose file description or sort-merge description entry permits records to contain a varying number of character positions.

*** variable-occurrence data item**

A variable-occurrence data item is a table element that is repeated a variable number of times. Such an item must contain an *OCCURS DEPENDING ON* clause in its data description entry or be subordinate to such an item.

*** variably located group**

A group item following, and not subordinate to, a variable-length table in the same record. The group item can be an alphanumeric group or a national group.

*** variably located item**

A data item following, and not subordinate to, a variable-length table in the same record.

*** verb**

A word that expresses an action to be taken by a COBOL compiler or object program.

volume

A module of external storage. For tape devices it is a reel; for direct-access devices it is a unit.

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.

VSAM file system

A file system that supports COBOL sequential, relative, and indexed organizations.

W**web service**

A modular application that performs specific tasks and is accessible through open protocols like HTTP and SOAP.

white space

Characters that introduce space into a document. They are:

- Space
- Horizontal tabulation
- Carriage return
- Line feed
- Next line

as named in the Unicode Standard.

*** word**

A character string of not more than 30 characters that forms a user-defined word, a system-name, a reserved word, or a function-name.

*** WORKING-STORAGE SECTION**

The section of the DATA DIVISION that describes WORKING-STORAGE data items, composed either of noncontiguous items or WORKING-STORAGE records or of both.

workstation

A generic term for computers, including personal computers, 3270 terminals, intelligent workstations, and UNIX terminals. Often a workstation is connected to a mainframe or to a network.

wrapper

An object that provides an interface between object-oriented code and procedure-oriented code. Using wrappers lets programs be reused and accessed by other systems.

X**x**

The symbol in a PICTURE clause that can hold any character in the character set of the computer.

XML

Extensible Markup Language. A standard metalanguage for defining markup languages that was derived from and is a subset of SGML. XML omits the more complex and less-used parts of SGML and makes it much easier to write applications to handle document types, author and manage structured information, and transmit and share structured information across diverse computing systems. The use of XML does not require the robust applications and processing that is necessary for SGML. XML is developed under the auspices of the World Wide Web Consortium (W3C).

XML data

Data that is organized into a hierarchical structure with XML elements. The data definitions are defined in XML element type declarations.

XML declaration

XML text that specifies characteristics of the XML document such as the version of XML being used and the encoding of the document.

XML document

A data object that is well formed as defined by the W3C XML specification.

XML namespace

A mechanism, defined by the W3C XML Namespace specifications, that limits the scope of a collection of element names and attribute names. A uniquely chosen XML namespace ensures the unique identity of an element name or attribute name across multiple XML documents or multiple contexts within an XML document.

XML schema

A mechanism, defined by the W3C, for describing and constraining the structure and content of XML documents. An XML schema, which is itself expressed in XML, effectively defines a class of XML documents of a given type, for example, purchase orders.

Z

z/OS UNIX file system

A collection of files and directories that are organized in a hierarchical structure and can be accessed by using z/OS UNIX.

zoned decimal data item

An external decimal data item that is described implicitly or explicitly as USAGE DISPLAY and that contains a valid combination of PICTURE symbols 9, S, P, and V. The content of a zoned decimal data item is represented in characters 0 through 9, optionally with a sign. If the PICTURE string specifies a sign and the SIGN IS SEPARATE clause is specified, the sign is represented as characters + or -. If SIGN IS SEPARATE is not specified, the sign is one hexadecimal digit that overlays the first 4 bits of the sign position (leading or trailing).

#

85 COBOL Standard

The COBOL language defined by the following standards:

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

2002 COBOL Standard

The COBOL language defined by the following standard:

- *INCITS/ISO/IEC 1989-2002, Information technology - Programming languages - COBOL*

2014 COBOL Standard

The COBOL language defined by the following standard:

- *INCITS/ISO/IEC 1989:2014, Information technology - Programming languages, their environments and system software interfaces - Programming language COBOL*

List of resources

Enterprise COBOL for z/OS publications

COBOL for z/OS publications

You can find the following publications in the [Enterprise COBOL for z/OS library](#):

- *What's New*, SC31-5708-00
- *Customization Guide*, SC27-8712-03
- *Language Reference*, SC27-8713-03
- *Programming Guide*, SC27-8714-03
- *Migration Guide*, GC27-8715-03
- *Performance Tuning Guide*, SC27-9202-02
- *Messages and Codes*, SC27-4648-02
- *Program Directory*, GI13-4526-03
- *Licensed Program Specifications*, GI13-4532-03

Softcopy publications

The following collection kits contain Enterprise COBOL and other product publications. You can find them at <https://www.ibm.com/resources/publications>.

- *z/OS Software Products Collection*
- *z/OS and Software Products DVD Collection*

Support

If you have a problem using Enterprise COBOL for z/OS, see the following site that provides up-to-date support information: <https://www.ibm.com/support/pages/node/6560933>.

Related publications

z/OS library publications

You can find the following publications in the [z/OS library](#).

Run-Time Library Extensions

- *Common Debug Architecture Library Reference*
- *Common Debug Architecture User's Guide*
- *DWARF/ELF Extensions Library Reference*

z/Architecture

- *Principles of Operation*

z/OS DFSMS

- *Access Method Services for Catalogs*
- *Checkpoint/Restart*
- *Macro Instructions for Data Sets*
- *Using Data Sets*

- *Utilities*

z/OS DFSORT

- *Application Programming Guide*
- *Installation and Customization*

z/OS ISPF

- *Dialog Developer's Guide and Reference*
- *User's Guide Vol I*
- *User's Guide Vol II*

z/OS Language Environment

- *Concepts Guide*
- *Customization*
- *Debugging Guide*
- *Language Environment Vendor Interfaces*
- *Programming Guide*
- *Programming Reference*
- *Run-Time Messages*
- *Run-Time Application Migration Guide*
- *Writing Interlanguage Communication Applications*

z/OS MVS

- *JCL Reference*
- *JCL User's Guide*
- *Programming: Callable Services for High-Level Languages*
- *Program Management: User's Guide and Reference*
- *System Commands*
- *z/OS Unicode Services User's Guide and Reference*
- *z/OS XML System Services User's Guide and Reference*

z/OS TSO/E

- *Command Reference*
- *Primer*
- *User's Guide*

z/OS UNIX System Services

- *Command Reference*
- *Programming: Assembler Callable Services Reference*
- *User's Guide*

z/OS XL C/C++

- *Programming Guide*
- *Run-Time Library Reference*

CICS Transaction Server for z/OS

You can find the following publications in the [CICS library](#):

- *Developing CICS Applications*

- *API (EXEC CICS) Reference*
- *Developing CICS System Programs*
- *Global User Exit Reference*
- *XPI Reference*
- *Using EXCI with CICS*

COBOL Report Writer Precompiler

- *Programmer's Manual, SC26-4301*
- *Installation and Operation, SC26-4302*

Db2 for z/OS

You can find the following publications in the [Db2 library](#):

- *Application Programming and SQL Guide*
- *Command Reference*
- *SQL Reference*

IBM z/OS Debugger (formerly IBM Debug for z Systems and Debug Tool)

You can find information about IBM z/OS Debugger in the [IBM z/OS Debugger library](#).

IBM Developer for z/OS (formerly IBM Developer for z Systems)

You can find information about IBM Developer for z/OS in the [IBM Developer for z/OS library](#).

Note: IBM Developer for z/OS supersedes IBM Developer for z Systems and Rational Developer for z Systems.

You can find the following publications by searching their publication numbers in the [IBM Publications Center](#).

IMS

- *Application Programming API Reference, SC18-9699*
- *Application Programming Guide, SC18-9698*

WebSphere Application Server for z/OS

- *Applications, SA22-7959*

Softcopy publications for z/OS

The following collection kit contains z/OS and related product publications:

- *z/OS CD Collection Kit, SK3T-4269*

Java

- *IBM SDK for Java - Tools Documentation*, publib.boulder.ibm.com/infocenter/javasdk/tools/index.jsp
- *The Java 2 Enterprise Edition Developer's Guide*, download.oracle.com/javaee/1.2.1/devguide/html/DevGuideTOC.html
- *Java 2 on z/OS*, www.ibm.com/servers/eserver/zseries/software/java/
- *The Java EE 5 Tutorial*, download.oracle.com/javaee/5/tutorial/doc/
- *The Java Language Specification, Third Edition*, by Gosling et al., java.sun.com/docs/books/jls/

- *The Java Native Interface*, download.oracle.com/javase/1.5.0/docs/guide/jni/
- *JDK 5.0 Documentation*, download.oracle.com/javase/1.5.0/docs/

JSON

- JavaScript Object Notation (JSON), www.json.org

Unicode and character representation

- *Unicode*, www.unicode.org/
- *Character Data Representation Architecture Reference and Registry*, SC09-2190

XML

- *Extensible Markup Language (XML)*, www.w3.org/XML/
- *Namespaces in XML 1.0*, www.w3.org/TR/xml-names/
- *Namespaces in XML 1.1*, www.w3.org/TR/xml-names11/
- *XML specification*, www.w3.org/TR/xml/

Index

Special Characters

_BPX_SHAREAS environment variable [517](#)
_CEE_ENVFILE environment variable
description [515](#)
indicating Java settings [296](#)
_CEE_RUNOPTS environment variable
description [515](#)
setting XPLINK [298](#)
specifying runtime options [513](#)
_IGZ_SYSOUT environment variable
setting [515](#)
writing to stdout or stderr [36](#)
-? cob2 option for displaying the compiler manual page [289](#)
-# cob2 option for displaying compile and link steps [289](#)
-b cob2 option
for creating DLLs [286](#)
for passing information to the linker [287](#)
-c cob2 option for compiling but not linking [288](#)
-comprc_ok cob2 option for controlling compiler based on return code [288](#)
-dbrmlib cob2 option for specifying the location to be used for DBRM [288](#)
-e cob2 option for specifying entry point [288](#)
-g cob2 option equivalent to specifying TEST [288](#)
-help cob2 option for displaying the compiler manual page [288](#)
-I cob2 option for searching copybooks [288](#)
-l cob2 option for specifying archive library name [288](#)
-L cob2 option for specifying archive library path [288](#)
-M cob2 option for generating make dependency file [289](#)
-o cob2 option for specifying output file [289](#)
-q cob2 option for specifying compiler options [289](#)
-v cob2 option for displaying and executing compile and link steps [289](#)
! character, hexadecimal values [631](#)
.a suffix with cob2 [289](#)
.adt file [335](#)
.adt suffix with cob2 [290](#)
.cbl suffix with cob2 [289](#)
.dbg suffix with cob2 [290](#)
.dbrm suffix with cob2 [290](#)
.dek suffix with cob2 [290](#)
.lst suffix with cob2 [290](#)
.o suffix with cob2 [289](#), [290](#)
.u suffix with cob2 [290](#)
.x suffix with cob2 [289](#), [290](#)
[character, hexadecimal values [631](#)
] character, hexadecimal values [631](#)
*CBL statement [429](#)
*CONTROL statement [429](#)
character, hexadecimal values [631](#)
| character, hexadecimal values [631](#)

Numerics

16 MB line

16 MB line (*continued*)
CICS programs [484](#)
IMS programs [484](#)
performance options [765](#)
24-bit addressing mode [39](#)
31-bit addressing mode
dynamic call [538](#)
64-bit addressing
no support [39](#)
85 COBOL Standard
checkpoints [743](#)
required compiler options [333](#)
required runtime options [333](#)

A

a suffix with cob2 [289](#)
a.out file from cob2 [290](#)
abends, compile-time [355](#)
ACCEPT statement
assigning input [35](#)
reading from stdin [35](#)
under CICS [485](#)
access JCL parameters
CEE3PR2 [955](#)
coding [955](#)
LINKAGE SECTION [955](#)
access method services
build alternate indexes in advance [211](#)
defining VSAM data sets to z/OS [205](#)
loading a VSAM data set [200](#)
accessibility
of Enterprise COBOL for z/OS
[957](#)
using z/OS [957](#)
accessibility features for this product [957](#)
ADATA compiler option [335](#)
adding records
to line-sequential files [218](#)
to QSAM files [178](#)
to VSAM files [202](#)
ADDRESS OF special register
use in CALL statement [552](#)
addresses
incrementing [557](#)
NULL value [557](#)
passing between programs [557](#)
passing entry-point addresses [546](#)
addressing mode, definition [39](#)
ADEXIT suboption of EXIT option
processing of [818](#)
syntax [358](#)
ADMODE attribute
with multithreading [583](#)
adt suffix with cob2 [290](#)
ADV compiler option [336](#)
AFP compiler option

- AFP compiler option (*continued*)
 - multioption interaction [334](#)
 - performance considerations [764](#)
- AIXBLD runtime option
 - effect on performance [769](#)
- ALL subscript
 - examples [88](#)
 - processing table elements iteratively [87](#)
 - table elements as function arguments [57](#)
- ALL31 runtime option
 - multioption interaction [40](#)
 - OFF for AMODE switching [538](#)
- ALLOCATE command (TSO)
 - compiler data sets [262](#)
 - with z/OS UNIX files [263](#)
- allocation of files
 - description [164](#)
 - line-sequential [216](#)
 - QSAM [180](#)
 - under TSO [262](#)
 - VSAM [208](#)
- ALPHABET clause, establishing collating sequence with [7](#)
- alphabetic data
 - comparing to national [141](#)
 - comparing to UTF-8 [152](#)
 - MOVE statement with [32](#)
- alphanumeric comparison [95](#)
- alphanumeric data
 - comparing
 - effect of ZWB [421](#)
 - to national [141](#)
 - to UTF-8 [152](#)
 - converting
 - to DBCS with IGZCA2D [789](#)
 - to national with MOVE [134](#)
 - to national with NATIONAL-OF [135](#)
 - to UTF-8 with MOVE [150](#)
 - MOVE statement with [32](#)
 - with double-byte characters [789](#)
- alphanumeric group item
 - a group without GROUP-USAGE NATIONAL [25](#)
 - definition [24](#)
- alphanumeric literals
 - conversion of mixed DBCS/EBCDIC [789](#)
 - description [25](#)
 - with DBCS content [159](#)
 - with double-byte characters [789](#)
- alphanumeric-edited data
 - initializing
 - example [29](#)
 - using INITIALIZE [74](#)
 - MOVE statement with [32](#)
- alternate collating sequence
 - choosing [231](#)
 - example [7](#)
- alternate entry point, calling [548](#)
- alternate index
 - creating [206](#)
 - example of [207](#)
 - password for [205](#)
 - path [207](#)
 - performance considerations [212](#)
 - using [193](#)
- ALTERNATE RECORD KEY clause
 - identify alternate indexes [207](#)
 - identifying alternate keys in KSDS files [193](#)
- alternate reserved-word
 - table
 - CICS [490](#)
 - specifying [415](#)
- AMODE
 - and DLLs [571](#)
 - description [39](#)
 - of EXIT modules [812](#)
 - switching
 - ALL31(OFF) [538](#)
 - examples [538](#)
 - overview [538](#)
- AMODE 31/64
 - considerations and restrictions [530](#)
- AMODE 64
 - applications [325](#), [327](#)
 - CALL statement [522](#)
 - compiler limits [522](#)
 - compiler options [522](#)
 - considerations [523](#), [525–528](#)
 - developing programs [521](#)
 - DLL considerations [326](#)
- AMODE 64 programs
 - POINTER data items [521](#)
- AMODE 64 under z/OS UNIX
 - compiling and binding [326](#)
- AMODE COBOL/JAVA
 - considerations and restrictions [528](#)
- AMODE3164 [529](#)
- AMP parameter [208](#)
- ANNUITY intrinsic function [61](#)
- APIs, UNIX and POSIX
 - calling [516](#)
- APOST compiler option [337](#)
- APPLY WRITE-ONLY clause [10](#)
- ARCH compiler option
 - performance considerations [764](#)
- arguments
 - describing in calling program [553](#)
 - from main program
 - accessing in z/OS [564](#)
 - accessing in z/OS UNIX [518](#)
 - passing BY VALUE [554](#)
 - specifying OMITTED [555](#)
 - testing for OMITTED arguments [555](#)
- ARITH compiler option
 - description [339](#)
 - performance considerations [764](#)
- arithmetic
 - COMPUTE statement simpler to code [55](#)
 - error handling [242](#)
 - with intrinsic functions [56](#)
- arithmetic comparisons [63](#)
- arithmetic evaluation
 - conversions and precision [52](#)
 - data format conversion [51](#)
 - examples [62](#), [63](#)
 - fixed-point contrasted with floating-point [62](#)
 - intermediate results [779](#)
 - performance tips [758](#)

- arithmetic evaluation (*continued*)
 - precedence [56](#), [780](#)
 - precision [779](#)
- arithmetic expression
 - as reference modifier [111](#)
 - description of [56](#)
 - in nonarithmetic statement [787](#)
 - in parentheses [56](#)
- arrays
 - COBOL [39](#)
 - Java
 - declaring [712](#)
 - manipulating [714](#)
- ASCII
 - alphabet, QSAM [187](#)
 - code pages supported in XML documents [628](#)
 - converting to EBCDIC [116](#)
 - job control language (JCL) [187](#)
 - record formats, QSAM [187](#)
 - tape files, QSAM [187](#)
- ASCII files
 - CODE-SET clause [13](#)
 - OPTCD= parameter in DCB [13](#)
- assembler
 - expansion of PROCEDURE DIVISION [454](#)
 - programs
 - calls from (in CICS) [485](#)
 - compiling from [264](#)
 - listing of [372](#), [762](#)
 - with multithreading [582](#)
- ASSIGN clause
 - corresponds to ddname [8](#)
 - QSAM files [167](#)
- assigning values [26](#)
- assistive technologies [957](#)
- associated-data file, creating [272](#)
- asynchronous signals with multithreading [583](#)
- AT END (end-of-file) phrase [246](#)
- ATTACH macro [264](#)
- attribute methods [679](#)
- automatic restart [746](#)
- available files
 - QSAM [177](#)
 - VSAM [205](#)
- AWO compiler option
 - APPLY-WRITE ONLY clause performance [10](#)
 - description [340](#)
 - performance considerations [764](#)

B

- Base class
 - equating to java.lang.Object [672](#)
 - using for java.lang.Object [671](#)
- base cluster name [207](#)
- base locator [450](#), [451](#)
- base locator table [471](#)
- basis libraries [270](#)
- BASIS statement [429](#)
- batch compilation
 - description [276](#)
 - LANGUAGE option
 - example [279](#)
- batch compilation (*continued*)
 - precedence of options
 - example [278](#)
 - overview [278](#)
- Bibliography [1007](#)
- big-endian, converting to little-endian [125](#)
- binary data [378](#)
- binary data item
 - general description [48](#)
 - intermediate results [784](#)
 - synonyms [46](#)
 - using efficiently [48](#), [759](#)
- binary digits, converting from [117](#)
- binary digits, converting to [116](#)
- binary search
 - description [86](#)
 - example [86](#)
- binder
 - c89 command [285](#)
 - options needed for DLLs [569](#)
 - recommended for DLLs [569](#)
- binding OO applications
 - example [297](#)
 - using JCL or TSO/E [296](#)
- BIT-OF intrinsic function [116](#)
- BIT-TO-CHAR intrinsic function [117](#)
- BLANK WHEN ZERO clause
 - coded for numeric data [126](#)
 - example with numeric-edited data [45](#)
- BLOCK CONTAINS clause
 - FILE SECTION entry [13](#)
 - no meaning for VSAM files [195](#)
 - QSAM files [169](#), [174](#), [340](#)
- block size
 - ASCII files [188](#)
 - compiler data sets [269](#)
 - QSAM files
 - fixed-length [168](#)
 - record layout [171](#)
 - using DCB [181](#)
 - variable-length [170](#)
 - system-determined
 - compiler data sets [269](#)
 - QSAM files [175](#), [340](#)
- BLOCK0 compiler option
 - description [340](#)
 - performance considerations [765](#)
- blocking factor, definition [169](#)
- blocking QSAM files
 - using BLOCK CONTAINS clause [174](#)
 - using BLOCK0 [340](#)
- blocking records [174](#)
- BPXBATCH utility
 - calling z/OS UNIX programs [514](#)
 - running OO applications [296](#)
- branch, implicit [99](#)
- buffers
 - best use of [10](#)
 - obtaining for QSAM [186](#)
- BUFSIZE compiler option [341](#)
- building and running COBOL/Java interoperable applications
 - AMODE considerations [321](#)

- building and running COBOL/Java interoperable applications (continued)
 - building Java programs [308](#)
 - linking COBOL programs [308](#)
- Building and running non-OO applications
 - Example: building a COBOL/Java interoperable applications [302](#)
- BY CONTENT [551](#)
- BY REFERENCE [551](#)
- BY VALUE
 - description [551](#)
 - restrictions [554](#)
 - valid data types [554](#)
- byte order mark not generated [656](#)
- BYTE-LENGTH intrinsic function
 - using [118](#)
 - with national data [121](#)
- byte-stream files
 - processing with QSAM [186](#)

C

- C/C++ programs
 - with COBOL DLLs [574](#)
 - with multithreading [582](#)
- c89 command for link step [285](#)
- CALL command (TSO) [262](#)
- CALL identifier
 - always dynamic [538](#)
 - dynamic calls [536](#)
 - making from DLLs [570](#)
 - with NODLL [536](#)
 - with NODYNAM [540](#)
- CALL literal
 - dynamic calls [536](#)
 - static calls [536](#)
 - with DYNAM [536](#)
 - with NODLL [536](#)
 - with NODYNAM [536](#), [540](#)
- CALL statement
 - AMODE processing [538](#)
 - BY CONTENT [551](#)
 - BY REFERENCE [551](#)
 - BY VALUE
 - description [551](#)
 - restrictions [554](#)
 - CICS restrictions [485](#)
 - effect of EXIT option on registers [812](#)
 - exception condition [251](#)
 - for error handling [251](#)
 - function-pointer [547](#)
 - handling of program-name in [388](#)
 - Language Environment callable services [775](#)
 - overflow condition [251](#)
 - RETURNING [560](#)
 - to alternate entry points [548](#)
 - USING [554](#)
 - with AMODE 64 support [522](#)
 - with CANCEL [538](#)
 - with DYNAM [356](#)
 - with ON EXCEPTION [251](#)
 - with ON OVERFLOW [19](#), [251](#)
- calls
 - 31-bit addressing mode [538](#)
 - AMODE switching for 24-bit programs [538](#)
 - between COBOL and non-COBOL programs [533](#)
 - between COBOL programs [533](#), [535](#)
 - CICS restrictions [485](#)
 - dynamic
 - example [541](#)
 - making [536](#)
 - performance [540](#)
 - restrictions [536](#)
 - with static calls [540](#)
 - exception condition [251](#)
 - interlanguage [533](#)
 - LINKAGE SECTION [555](#)
 - OMITTED arguments [555](#)
 - overflow condition [251](#)
 - passing arguments [553](#)
 - passing data [551](#)
 - receiving parameters [554](#)
 - recursive [545](#)
 - static
 - example [541](#)
 - making [536](#)
 - performance [540](#)
 - with dynamic calls [540](#)
 - to and from object-oriented programs [545](#)
 - to JNI services [707](#)
 - to Language Environment callable services [775](#)
- CANCEL statement
 - cannot use with DLL linkage [571](#)
 - for subprograms [537](#)
 - handling of program-name in [388](#)
 - with dynamic CALL [537](#)
- case structure, EVALUATE statement for [93](#)
- cataloged procedure
 - JCL for compiling [256](#)
 - to compile (IGYWC) [257](#)
 - to compile and link-edit (IGYWCL) [258](#)
 - to compile, link-edit, run (IGYWCLG) [259](#)
- CBL statement
 - overview [429](#)
 - specifying compiler options [274](#)
- cbl suffix with cob2 [289](#)
- CBLPSHPOP runtime option [491](#)
- CBLQDA
 - DEBUG [327](#)
- CBLQDA runtime option [177](#)
- CCSID
 - conflict in XML documents [635](#), [636](#)
 - definition [125](#)
 - EBCDIC multibyte CCSIDs [344](#)
 - in PARSE statement [614](#)
 - of Db2 string data [498](#)
 - of XML documents [628](#)
 - of XML documents to be parsed [614](#)
 - specifying with CODEPAGE option [343](#)
- chained-list processing
 - example [558](#)
 - overview [557](#)
- changing
 - characters to numbers [115](#)
 - file-name [9](#)
 - title on source listing [5](#)
- CHAR intrinsic function, example [118](#)
- character set, definition [124](#)

- checking for valid data
 - conditional expressions [95](#)
- checkpoint
 - 85 COBOL Standard [743](#)
 - designing [744](#)
 - example of JCL for restart [748](#)
 - messages generated during [746](#)
 - methods [743](#)
 - multiple [743](#), [745](#)
 - overview [743](#)
 - record data set [744](#)
 - restart during DFSORT [237](#)
 - restrictions during sort [744](#)
 - setting [743](#)
 - single
 - disk [745](#)
 - tape [745](#)
 - testing [744](#)
- Chinese GB 18030 data
 - processing [157](#)
- CHKPT keyword [237](#)
- CICS
 - alternate reserved-word table [490](#)
 - calling nested programs [486](#)
 - CICS HANDLE
 - example [492](#)
 - LABEL value [491](#)
 - coding programs to run under
 - calls [485](#)
 - DISPLAY statement [484](#)
 - I/O [484](#)
 - overview [483](#)
 - restrictions [483](#)
 - SORT statement [491](#)
 - command-level interface [483](#)
 - commands and the PROCEDURE DIVISION [483](#)
 - compiling with CICS option [487](#)
 - developing programs for [483](#)
 - DFHCOMMAREA parameter
 - calling nested programs [486](#)
 - calling separately compiled programs [485](#)
 - DFHEIBLK parameter
 - calling nested programs [486](#)
 - calling separately compiled programs [485](#)
 - ECI calls and RETURN-CODE special register [486](#)
 - EXIT compiler option and [830](#)
 - in a multithreaded environment [582](#)
 - integrated translator
 - advantages [488](#)
 - calling nested programs [486](#)
 - compiler options for [487](#)
 - overview [488](#)
 - interlanguage communication under [485](#)
 - macro-level interface [483](#)
 - NODYNAM compiler option [485](#)
 - performance
 - overview [757](#)
 - performance considerations [491](#), [769](#)
 - restrictions
 - 16 MB line [484](#)
 - files [5](#)
 - OO programs [483](#), [667](#)
 - OUTDD compiler option [386](#)
 - parsing with validation using FILE [623](#)
- CICS (*continued*)
 - restrictions (*continued*)
 - separate translator [488](#)
 - sorting [238](#)
 - separate translator
 - calling nested programs [486](#)
 - compiler options for [489](#)
 - restrictions [488](#)
 - using [489](#)
 - sorting under
 - change reserved-word table [491](#)
 - overview [238](#)
 - restrictions [238](#)
 - system date, getting [485](#)
- CICS compiler option
 - description [342](#)
 - enables integrated translator [488](#)
 - multioption interaction [334](#)
 - specifying suboptions [342](#), [488](#)
 - using [487](#)
- CISZ (control interval size), performance considerations [212](#), [769](#)
- cjbuild
 - command reference [321](#)
- CKPT keyword [237](#)
- class
 - defining [670](#)
 - definition of [667](#)
 - factory data [696](#)
 - instance data [673](#)
 - instantiating
 - COBOL [690](#)
 - Java [690](#)
 - name
 - external [672](#), [683](#)
 - in a program [671](#)
 - object, obtaining reference with JNI [708](#)
 - user-defined [8](#)
- class condition
 - testing
 - for DBCS [159](#)
 - for Kanji [159](#)
 - for numeric [54](#)
 - overview [95](#)
 - validating data [435](#)
- CLASSPATH environment variable
 - description [515](#)
 - example of setting [296](#)
 - specifying location of Java classes [293](#)
- client
 - defining [682](#)
 - definition of [682](#)
- CLIST for compiling under TSO [264](#)
- CLOSE statement
 - line-sequential files [217](#)
 - QSAM [176](#)
 - VSAM [196](#)
- closing files
 - line-sequential [219](#)
 - multithreading serialization [580](#)
 - QSAM
 - overview [179](#)
 - with multithreading [179](#)

- closing files (*continued*)
 - VSAM
 - overview [203](#)
 - with multithreading [204](#)
- closing files, automatic
 - line-sequential [219](#)
 - QSAM [179](#)
 - VSAM [203](#)
- cluster, VSAM [205](#)
- cob2 command
 - compiling with
 - examples [286](#)
 - overview [285](#)
 - description [287](#)
 - for compiling OO applications [291](#)
 - for creating DLLs [286](#)
 - for linking OO applications [292](#)
 - input and output [289](#)
 - linking with
 - examples [286](#)
 - overview [285](#)
 - options and syntax [287](#)
- COBJVMINIOPTIONS environment variable
 - description [515](#)
 - specifying JVM options [294](#)
- COBOL
 - and Java
 - binding [296](#)
 - communicating between [707](#)
 - compatibility [299](#)
 - compiling under z/OS UNIX [291](#)
 - compiling using JCL or TSO/E [295](#)
 - linking [292](#)
 - running [293](#), [296](#)
 - structuring applications [703](#)
 - under IMS [509](#)
 - object-oriented
 - binding [296](#)
 - compiling under z/OS UNIX [291](#)
 - compiling using JCL or TSO/E [295](#)
 - linking [292](#)
 - running [293](#)
 - under IMS [509](#)
- COBOL client
 - example [699](#)
 - example of passing object references [687](#)
- COBOL DLL programs, calling [572](#)
- COBOL only
 - runtime options specifications [327](#)
- COBOL runtime
 - options [422](#)
- COBOL terms [23](#)
- COBOL_INSTALL_DIR environment variable [283](#)
- COBOL/Java interoperability
 - Calling static Java methods from COBOL [731](#)
 - Making an existing program callable from Java [729](#)
 - outside of object-oriented COBOL [727](#), [729](#), [731](#)
 - troubleshooting [308](#)
- COBOL/Java interoperability outside of object-oriented COBOL
 - array restrictions [734](#)
 - array semantics [734](#)
 - byte arrays [733](#)
 - group parameters [733](#)
- COBOL/Java interoperability outside of object-oriented COBOL (*continued*)
 - using Java-compatible array types in COBOL [733](#)
- COBOL/Java interoperability outside of the Object-Oriented (OO) COBOL framework
 - AMODE considerations [739](#)
 - COBOL/Java interoperability
 - outside of object-oriented COBOL [735](#)
 - Error handling in Java-interoperable COBOL applications [737](#)
 - COBOL3 translator option [489](#)
 - COBOPT environment variable [283](#)
 - code
 - copy [772](#)
 - optimized [762](#)
 - code page
 - conflict in XML documents [635](#), [636](#)
 - DBCS [344](#)
 - definition [124](#)
 - euro currency support [64](#)
 - hexadecimal values of special characters [631](#)
 - of Db2 string data [498](#)
 - overriding [135](#)
 - specifying [343](#)
 - specifying for alphanumeric XML document [630](#)
 - code point, definition [124](#)
 - CODE-SET clause [13](#)
 - coded character set
 - definition [124](#)
 - in XML documents [628](#)
 - CODEPAGE compiler option
 - DBCS code pages [344](#)
 - description [343](#)
 - for national literals [139](#)
 - items that are not affected [343](#)
 - operations that override [344](#)
 - coding
 - class definition [670](#)
 - clients [682](#)
 - condition tests [96](#)
 - constructor methods [697](#)
 - DATA DIVISION [11](#)
 - decisions [91](#)
 - efficiently [757](#)
 - ENVIRONMENT DIVISION [5](#)
 - errors, avoiding [757](#)
 - EVALUATE statement [93](#)
 - factory definition [695](#)
 - factory methods [697](#)
 - file input/output (overview) [161](#)
 - IDENTIFICATION DIVISION [3](#)
 - IF statement [91](#)
 - input/output overview [163](#)
 - input/output statements
 - for line-sequential files [217](#)
 - for QSAM files [176](#)
 - for VSAM files [196](#)
 - instance methods [674](#), [694](#)
 - interoperable data types with Java [711](#)
 - loops [99](#)
 - OO programs
 - must be reentrant [548](#)
 - overview [667](#)
 - PROCEDURE DIVISION [17](#)
 - programs to run under CICS

- coding (*continued*)
 - programs to run under CICS (*continued*)
 - calls [485](#)
 - DISPLAY statement [484](#)
 - I/O [484](#)
 - must be reentrant [548](#)
 - overview [483](#)
 - restrictions [483](#)
 - SORT statement [491](#)
 - system date, getting [485](#)
 - programs to run under Db2
 - CCSID of string data [498](#)
 - overview [493](#)
 - stored procedures must be reentrant [548](#)
 - programs to run under IMS
 - must be reentrant [548](#)
 - overview [505](#)
 - restrictions [505](#)
 - simplifying [771](#)
 - SQL statements
 - overview [494](#)
 - restriction [494](#)
 - SQLIMS statements
 - overview [506](#)
 - subclasses
 - example [694](#)
 - overview [692](#)
 - tables [67](#)
 - techniques [11](#), [757](#)
 - test conditions [96](#)
- coding user-defined functions [585](#)
- collating sequence
 - alternate
 - choosing [231](#)
 - example [7](#)
 - ASCII [7](#)
 - binary for national keys [229](#)
 - EBCDIC [7](#)
 - HIGH-VALUE [7](#)
 - ISO 7-bit code [7](#)
 - LOW-VALUE [7](#)
 - MERGE [7](#), [231](#)
 - NATIVE [7](#)
 - nonnumeric comparisons [7](#)
 - ordinal position of a character [118](#)
 - SEARCH ALL [7](#)
 - SORT [7](#), [231](#)
 - specifying [7](#)
 - STANDARD-1 [7](#)
 - STANDARD-2 [7](#)
 - symbolic characters in the [8](#)
- COLLATING SEQUENCE phrase
 - does not apply to national keys [229](#)
 - overrides PROGRAM COLLATING SEQUENCE clause [7](#), [231](#)
 - use in SORT or MERGE [231](#)
- columns in tables [67](#)
- comment lines [969](#)
- comments
 - sending xxxv
- COMMON attribute [4](#), [542](#)
- COMP (COMPUTATIONAL) [48](#)
- COMP-1 (COMPUTATIONAL-1)
 - format [49](#)
- COMP-1 (COMPUTATIONAL-1) (*continued*)
 - performance tips [759](#)
- COMP-2 (COMPUTATIONAL-2)
 - format [49](#)
 - performance tips [759](#)
- COMP-3 (COMPUTATIONAL-3) [49](#)
- COMP-4 (COMPUTATIONAL-4) [48](#)
- COMP-5 (COMPUTATIONAL-5) [48](#)
- comparing data items
 - national
 - overview [140](#)
 - to alphabetic, alphanumeric, or DBCS [141](#)
 - to alphanumeric groups [142](#)
 - to numeric [141](#)
 - two operands [140](#)
 - object references [684](#)
- UTF-8
 - overview [151](#)
 - to alphabetic, alphanumeric [152](#)
 - to alphanumeric groups [152](#)
 - two operands [152](#)
 - zoned decimal and alphanumeric, effect of ZWB [421](#)
- compatibility
 - Java and COBOL [299](#)
 - object-oriented syntax [299](#)
- compatibility mode [43](#), [779](#)
- compilation
 - conformance to 85 COBOL Standard [333](#)
 - results [275](#)
 - with z/OS UNIX files [257](#)
- compilation statistics [448](#)
- COMPILE compiler option
 - description [345](#)
 - use NOCOMPILE to find syntax errors [438](#)
- compile-time considerations
 - compiler-directed errors [281](#)
 - display compile and link steps [289](#)
 - dump, generating a [355](#)
 - error messages
 - determining what severity level to produce [359](#)
 - severity levels [282](#)
 - executing compile and link steps after display [289](#)
- compiler
 - calculation of intermediate results [780](#)
 - environment variables under z/OS UNIX [283](#)
 - generating list of error messages [281](#)
 - invoking in the z/OS UNIX
 - shell
 - examples [286](#)
 - overview [285](#)
 - limits
 - DATA DIVISION [11](#)
 - messages
 - choosing severity to be flagged [440](#)
 - customizing [821](#)
 - determining what severity level to produce [359](#)
 - embedding in source listing [440](#)
 - from exit modules [828](#)
 - sending to terminal [271](#)
 - severity levels [282](#), [822](#)
 - return code
 - depends on highest severity [282](#)
 - effect of message customization [823](#)
 - overview [282](#)

- compiler data sets
 - in the z/OS UNIX file system [256](#), [261](#)
 - input and output [267](#)
 - required for compilation [267](#)
 - SYSADATA (ADATA records) [272](#)
 - SYSDEBUG (debug records) [272](#)
 - SYSIN [269](#)
 - SYSJAVA [272](#)
 - SYSLIB (libraries) [270](#)
 - SYSLIN (object code) [271](#)
 - SYSMDECK (library processing) [273](#)
 - SYSOPTF [270](#)
 - SYSOUT (listing) [271](#)
 - SYSPUNCH (object code) [271](#)
 - SYSTEM (messages) [271](#)
 - with cob2 [289](#)
- Compiler directive
 - DATA [528](#)
- compiler limits
 - with AMODE 64 support [522](#)
- compiler listings
 - getting [443](#)
- compiler options
 - 85 COBOL Standard conformance [333](#)
 - abbreviations [329](#)
 - ADATA [335](#)
 - ADV [336](#)
 - AFP
 - performance considerations [764](#)
 - APOST [337](#)
 - ARCH
 - performance considerations [764](#)
 - ARITH
 - description [339](#)
 - performance considerations [764](#)
 - AWO
 - description [340](#)
 - performance considerations [764](#)
 - BLOCK0
 - description [340](#)
 - performance considerations [765](#)
 - BUFSIZE [341](#)
 - CICS [342](#)
 - CODEPAGE [343](#)
 - COMPILE [345](#)
 - CONDCOMP [346](#)
 - conflicting [334](#)
 - COPYLOC [346](#)
 - COPYRIGHT [347](#)
 - CURRENCY [348](#)
 - DATA [349](#)
 - DBCS [350](#)
 - DECK [350](#)
 - DEFINE [350](#)
 - DIAGTRUNC [352](#)
 - DISPSIGN [352](#)
 - DLL [353](#)
 - DUMP [355](#)
 - DYNAM [356](#), [765](#)
 - EXIT [357](#)
 - EXPORTALL [359](#)
 - FASTSRT
 - performance considerations [765](#)

- compiler options (*continued*)
 - FLAG [359](#), [440](#)
 - FLAGSTD [360](#)
 - for AMODE 64 [522](#)
 - for CICS integrated translator [487](#)
 - for CICS separate translator [487](#), [489](#)
 - for debugging
 - overview [437](#)
 - TEST restriction [436](#)
 - THREAD restriction [436](#)
 - HGPR
 - performance considerations [765](#)
 - IMS, recommended for [508](#)
 - in effect [456](#)
 - INITCHECK
 - performance considerations [765](#)
 - INITIAL [364](#)
 - INLINE
 - performance considerations [765](#)
 - INTDATE [365](#)
 - INVDATA [366](#)
 - JAVAIOP [369](#)
 - LANGUAGE
 - description [370](#)
 - example in batch compilation [279](#)
 - LINECOUNT [371](#)
 - LIST [372](#), [443](#)
 - LP [373](#)
 - MAP [374](#), [442](#), [443](#)
 - MAXPCF [375](#)
 - MAXPCF(nnn)
 - performance considerations [766](#)
 - MDECK [376](#)
 - NAME [377](#)
 - NOCOMPILE [438](#)
 - NOFASTSRT [234](#)
 - NSYMBOL [377](#)
 - NUMBER [378](#), [445](#)
 - NUMCHECK
 - performance considerations [766](#)
 - NUMPROC [382](#)
 - NUMPROC(PFD)
 - performance considerations [766](#)
 - NUMPROC(PFD|NOPFD) [53](#)
 - OBJECT [383](#)
 - OFFSET [383](#)
 - on compiler invocation [447](#)
 - OPTFILE [384](#)
 - OPTIMIZE
 - description [385](#)
 - performance considerations [762](#), [766](#)
 - OUTDD [386](#)
 - PARMCHECK
 - performance considerations [766](#)
 - performance considerations [764](#)
 - PGMNAME [388](#)
 - precedence of
 - example [278](#)
 - in batch [278](#)
 - in SYSOPTF data sets [270](#), [384](#)
 - under z/OS [273](#)
 - under z/OS UNIX [284](#)
 - QUALIFY [390](#)
 - QUOTE [337](#)

compiler options (*continued*)
 RENT
 description [391](#)
 performance considerations [766](#)
 RMODE
 description [392](#)
 performance considerations [767](#)
 RULES [393](#)
 SEQUENCE [395](#)
 SERVICE [395](#)
 signature information bytes [456](#)
 SMARTBIN
 performance considerations [767](#)
 SOURCE [397](#), [443](#)
 SPACE [398](#)
 specifying
 using PROCESS (CBL) [274](#)
 specifying under TSO [275](#)
 specifying under z/OS [275](#)
 specifying under z/OS UNIX [284](#)
 specifying with SYSOPTF data set [270](#)
 SQL
 description [398](#)
 using with Db2 [497](#)
 SQLCCSID
 description [399](#)
 effect on CCSID of string data [498](#)
 performance considerations [500](#)
 recommended with Db2 coprocessor [500](#)
 SQLIMS [400](#)
 SSRANGE
 performance considerations [767](#)
 status [447](#)
 STGOPT [402](#)
 SUPPRESS [403](#)
 table of [329](#)
 TERMINAL [403](#)
 TEST
 description [404](#)
 performance considerations [767](#)
 use for debugging [443](#)
 THREAD
 debugging restriction [436](#)
 description [408](#)
 performance considerations [767](#)
 TRUNC
 description [409](#)
 performance considerations [767](#)
 TUNE
 performance considerations [767](#)
 under IMS and CICS [484](#)
 VBREF [412](#), [443](#)
 VLR
 description [412](#)
 VSAMOPENFS [414](#)
 WORD [415](#)
 XMLPARSE [415](#)
 XREF [416](#), [442](#)
 ZONECHECK [417](#)
 ZONEDATA [418](#)
 ZWB [421](#)
 Compiler options
 listing example [467](#)
 compiler output [443](#)
 compiler-directing statements
 description [429](#)
 overview [19](#)
 compiling
 AMODE 64 programs [522](#), [523](#), [525–528](#)
 and binding [325](#)
 batch [276](#)
 control of [273](#)
 data sets for [267](#)
 DLLs [286](#)
 from an assembler program [264](#)
 OO applications
 cob2 command [291](#)
 example [293](#), [297](#)
 under z/OS UNIX [291](#)
 using JCL or TSO/E [295](#)
 under TSO
 example CLIST [264](#)
 overview [262](#)
 under z/OS [255](#)
 under z/OS UNIX [283](#)
 using shell script [290](#)
 using the cob2 command
 examples [286](#)
 overview [285](#)
 with cataloged procedures
 compile [257](#)
 compile and link-edit [258](#)
 compile, link-edit, run [259](#)
 with JCL (job control language) [255](#)
 Compiling and binding AMODE 64 programs
 under z/OS UNIX [326](#)
 Compiling and debugging your program
 Compiling, linking, and running non-OO COBOL/Java
 interoperable applications [301](#)
 compiling and linking in the z/OS UNIX
 shell
 DLLs [286](#)
 examples [286](#)
 OO applications
 cob2 command [292](#)
 example [293](#)
 overview [285](#)
 compiling user-defined functions [586](#)
 Compiling, linking, and running non-OO COBOL/Java
 interoperable applications
 Building and running non-OO applications [301](#)
 Preparing non-OO applications under z/OS UNIX [303](#)
 completion code
 merge [231](#)
 sort [231](#)
 complex OCCURS DEPENDING ON
 basic forms of [81](#)
 complex ODO item [81](#)
 variably located data item [81](#)
 variably located group [81](#)
 computation
 arithmetic data items [759](#)
 of indexes [72](#)
 of subscripts [761](#)
 COMPUTATIONAL (COMP) [48](#)
 COMPUTATIONAL-1 (COMP-1)
 format [49](#)
 performance tips [759](#)

- COMPUTATIONAL-2 (COMP-2)
 - format [49](#)
 - performance tips [759](#)
- COMPUTATIONAL-3 (COMP-3)
 - description [49](#)
- COMPUTATIONAL-4 (COMP-4) [48](#)
- COMPUTATIONAL-5 (COMP-5) [48](#)
- COMPUTE statement
 - assigning arithmetic results [34](#)
 - simpler to code [55](#)
- computer, describing [5](#)
- concatenating data items (STRING) [103](#)
- CONDCOMP compiler option
 - description [346](#)
- condition handling
 - closing QSAM files [179](#)
 - closing VSAM files [203](#)
 - in input or output procedures [227](#)
 - using Language Environment [773](#)
- condition testing [96](#)
- conditional compilation output, example [478](#)
- conditional expression
 - EVALUATE statement [91](#)
 - IF statement [91](#)
 - PERFORM statement [100](#)
- conditional statement
 - overview [18](#)
 - with NOT phrase [19](#)
 - with object references [684](#)
- CONFIGURATION SECTION [5](#)
- conflicting compiler options [334](#)
- conformance requirements
 - 85 COBOL Standard [333](#)
 - example of passing object references in INVOKE [687](#)
 - RETURNING phrase of INVOKE [688](#)
 - USING phrase of INVOKE [686](#)
- considerations
 - for AMODE 64 [523](#), [525–528](#)
- considerations and restrictions
 - for AMODE 31/64 [530](#)
 - for AMODE COBOL/JAVA [528](#)
- Constant area [470](#)
- constants
 - data items [758](#)
 - definition [26](#)
 - figurative, definition [26](#)
- contained program integration [763](#)
- CONTENT-CHARACTERS XML event
 - example [647](#)
 - when parsing segments [626](#)
- continuation
 - entry [236](#)
 - of program [243](#)
 - syntax checking [345](#)
- CONTINUE statement [91](#)
- control
 - in nested programs [542](#)
 - program flow [91](#)
 - transfer [533](#)
- control interval size (CISZ), performance considerations [212](#), [769](#)
- CONTROL statement [429](#)
- converting data items
 - between code pages [116](#)
- converting data items (*continued*)
 - between data formats [51](#)
 - exceptions with national data [136](#)
 - from binary digits [117](#)
 - from hexadecimal digits [117](#)
 - from hexadecimal or binary digits
 - with intrinsic functions [117](#)
 - precision [52](#)
 - reversing order of characters [114](#)
 - to alphanumeric
 - with DISPLAY [36](#)
 - with DISPLAY-OF [135](#)
 - to binary digits [116](#)
 - to Chinese GB 18030 from national [157](#)
 - to hexadecimal digits [116](#)
 - to hexadecimal or binary digits
 - with intrinsic functions [116](#)
 - to integers with INTEGER, INTEGER-PART [112](#)
 - to national
 - from Chinese GB 18030 [157](#)
 - from UTF-8 [153](#)
 - with ACCEPT [35](#)
 - with MOVE [134](#)
 - with NATIONAL-OF [135](#), [151](#)
 - to numbers with NUMVAL, NUMVAL-C [115](#)
 - to uppercase or lowercase
 - with INSPECT [113](#)
 - with intrinsic functions [114](#)
 - to UTF-8
 - with MOVE [150](#)
 - to UTF-8 from national [153](#)
 - with INSPECT [112](#)
 - with intrinsic functions [113](#)
- CONVERTING phrase (INSPECT), example [113](#)
- converting UTF-8 data items
 - to alphanumeric
 - with DISPLAY-OF [151](#)
- coprocessor, Db2
 - CCSID determination of string data [498](#)
 - differences from the precompiler [500](#)
 - enable with SQL compiler option [497](#)
 - overview [493](#)
 - recommended compiler option SQLCCSID [500](#)
 - using SQL INCLUDE with [495](#)
- coprocessor, IMS
 - enable with SQLIMS compiler option [507](#)
 - overview [505](#)
- copy libraries
 - COPY statement [429](#)
 - data set [267](#)
 - example [772](#)
 - search order [430](#)
 - specifying [270](#)
 - SYSLIB [270](#)
 - z/OS UNIX search order [283](#), [288](#)
- COPY statement
 - description [429](#)
 - example [772](#)
 - nested [772](#), [814](#)
 - z/OS considerations [270](#)
 - z/OS UNIX considerations [430](#)
- copybook

- copybook (*continued*)
 - description [429](#)
 - obtaining from user-supplied module [357](#)
 - searching for [288](#), [430](#)
- copybook cross-reference, description [442](#)
- copybooks
 - cross-reference [475](#)
 - using [772](#)
- COPYLOC compiler option [346](#)
- COPYRIGHT compiler option [347](#)
- COUNT IN phrase
 - UNSTRING [105](#)
 - XML GENERATE [656](#)
- counting
 - characters (INSPECT) [112](#)
 - generated XML characters [652](#)
- creating
 - associated-data file [272](#)
 - library-processing output file [273](#)
 - line-sequential files in z/OS [216](#)
 - object code [271](#)
 - objects [689](#)
 - QSAM files, z/OS [180](#), [182](#)
 - SYSJAVA file [272](#)
 - variable-length tables [78](#)
- cross-reference
 - COPY/BASIS [475](#)
 - COPY/BASIS statements [443](#)
 - copybooks [443](#)
 - data and procedure-names [442](#)
 - embedded [443](#)
 - list [416](#)
 - program-name [474](#)
 - special definition symbols [477](#)
 - statement list [412](#)
 - statements [443](#)
 - text-names and data sets [442](#)
- CRP (file position indicator) [198](#), [201](#)
- CURRENCY compiler option [348](#)
- currency signs
 - euro [64](#)
 - hexadecimal literals [64](#)
 - multiple-character [64](#)
 - using [64](#)
- CURRENT-DATE intrinsic function
 - example [60](#)
 - under CICS [485](#)
- customer support [xxxiii](#), [1007](#)

D

- D-format record
 - layout [171](#)
 - requesting [170](#)
- DASD (direct-access storage device) [212](#)
- data
 - concatenating (STRING) [103](#)
 - converting between alphanumeric and DBCS [789](#)
 - efficient execution [757](#)
 - format conversion of [51](#)
 - format, numeric types [46](#)
 - grouping [556](#)
 - incompatible [53](#)
 - naming [12](#)

- data (*continued*)
 - numeric [43](#)
 - passing [551](#)
 - record size [13](#)
 - splitting (UNSTRING) [105](#)
 - validating [53](#)
- DATA [528](#)
- data and procedure-name cross-reference, description [442](#)
- data areas, dynamic [356](#)
- DATA compiler option
 - description [349](#)
 - influencing data location [41](#)
 - multioption interaction [39](#), [334](#)
 - performance considerations [765](#)
 - when passing data [41](#)
- data compression [751](#)
- data definition [450](#)
- data description entry [11](#)
- DATA DIVISION
 - client [683](#)
 - coding [11](#)
 - description [11](#)
 - entries for line-sequential files [215](#)
 - entries for QSAM files [168](#)
 - entries for VSAM files [195](#)
 - factory data [696](#)
 - factory method [697](#)
 - FD entry [11](#)
 - FILE SECTION [11](#)
 - GROUP-USAGE NATIONAL clause [68](#)
 - instance data [673](#), [694](#)
 - instance method [676](#)
 - items present in [456](#)
 - limits [11](#)
 - LINKAGE SECTION [11](#), [16](#)
 - listing [443](#)
 - LOCAL-STORAGE SECTION [11](#)
 - mapping of items [374](#), [443](#)
 - OCCURS clause [67](#)
 - OCCURS DEPENDING ON (ODO) clause [78](#)
 - REDEFINES clause [75](#)
 - restrictions [11](#)
 - signature information bytes [456](#)
 - USAGE clause at the group level [25](#)
 - USAGE IS INDEX clause [72](#)
 - USAGE NATIONAL clause at the group level [133](#), [150](#)
 - WORKING-STORAGE SECTION [11](#)
- data item
 - alphanumeric with double-byte characters [789](#)
 - coding Java types [711](#)
 - common, in subprogram linkage [554](#)
 - concatenating (STRING) [103](#)
 - converting characters (INSPECT) [112](#)
 - converting characters to numbers [115](#)
 - converting from hexadecimal or binary digits [117](#)
 - converting to hexadecimal or binary digits [116](#)
 - converting to uppercase or lowercase [114](#)
 - converting with intrinsic functions [113](#)
 - counting characters (INSPECT) [112](#)
 - DBCS [789](#)
 - elementary, definition [24](#)
 - evaluating with intrinsic functions [117](#)
 - finding the smallest or largest item [118](#)
 - group, definition [24](#)

- data item (*continued*)
 - index, referring to table elements with [70](#)
 - initializing, examples of [27](#)
 - map [275](#)
 - numeric [43](#)
 - reference modification [109](#)
 - referring to a substring [109](#)
 - replacing characters (INSPECT) [112](#)
 - reversing characters [114](#)
 - splitting (UNSTRING) [105](#)
 - unused [385](#), [450](#)
 - variably located [81](#)
- data manipulation
 - character data [103](#)
 - DBCS data [789](#)
- DATA RECORDS clause [13](#)
- data set
 - alternate data-set names [264](#)
 - checkpoint record [744](#)
 - compiler-option [270](#)
 - defining with environment variable [164](#)
 - example of checkpoint/restart [748](#)
 - file, same meaning as [5](#)
 - JAVAERR [296](#)
 - JAVAIN [296](#)
 - JVAOUT [296](#)
 - names, alternate [265](#)
 - output [271](#)
 - source code [269](#)
 - SYSADATA [272](#)
 - SYSDEBUG [272](#)
 - SYSIN [269](#)
 - SYSJAVA [272](#)
 - SYSLIB [270](#)
 - SYSLIN [271](#)
 - SYSMDECK [273](#)
 - SYSOPTF [270](#)
 - SYSPRINT [271](#)
 - SYSPUNCH [271](#)
 - SYSTEM [271](#)
- data sets used for compiling [267](#)
- data-definition attribute codes [450](#)
- data-name
 - cross-reference [473](#)
 - cross-reference list [276](#)
 - in MAP listing [450](#)
 - OMITTED [13](#)
 - password for VSAM files [204](#)
- date and time operations
 - Language Environment callable services [774](#)
- date operations
 - finding date of compilation [121](#)
- DATE-COMPILED paragraph [3](#)
- DATE-OF-INTEGGER intrinsic function [60](#)
- Db2
 - coding considerations [493](#)
 - coprocessor
 - CCSID determination of string data [498](#)
 - database request module (DBRM) [493](#), [497](#)
 - differences from the precompiler [500](#)
 - enable with SQL compiler option [497](#)
 - overview [493](#)
 - recommended compiler option SQLCCSID [500](#)
 - using SQL INCLUDE with [495](#)
 - Db2 (*continued*)
 - DYNAM compiler option with TSO or IMS [502](#)
 - NODYNAM compiler option with CICS or CAF [502](#)
 - precompiler
 - differences from the coprocessor [500](#)
 - recommended compiler option NOSQLCCSID [500](#)
 - specifying code page for host variables [495](#)
 - using [494](#)
 - SQL compiler option [497](#)
 - SQL statements
 - CCSID determination [498](#)
 - coding [494](#)
 - overview [493](#)
 - return codes [497](#)
 - SQL DECLARE [495](#)
 - SQL INCLUDE [495](#)
 - using binary data in [497](#)
 - using character data in [495](#)
 - using national decimal data [496](#)
 - SQLCCSID compiler option [498](#)
 - Db2 precompiler
 - using [494](#)
 - DBCS comparison [95](#)
 - DBCS compiler option
 - description [350](#)
 - for Java interoperability [291](#), [295](#)
 - for OO COBOL [291](#), [295](#)
 - multioption interaction [335](#)
 - DBCS data
 - comparing
 - to national [141](#)
 - converting
 - to alphanumeric with IGZCD2A [791](#)
 - to and from alphanumeric [789](#)
 - to national, overview [160](#)
 - declaring [158](#)
 - encoding and storage [139](#)
 - literals
 - description [25](#)
 - maximum length [159](#)
 - using [159](#)
 - MOVE statement with [32](#)
 - notation for [789](#)
 - testing for [159](#)
 - dbg suffix with cob2 [290](#)
 - DBRM data set
 - defining [497](#)
 - description [493](#)
 - dbrm suffix with cob2 [290](#)
 - DBRMLIB DD statement [493](#), [497](#)
 - DCB [176](#)
 - DD control statement
 - allocating line-sequential files [216](#)
 - AMP parameter [208](#)
 - ASCII tape files [187](#)
 - creating QSAM files [180](#), [182](#)
 - DBRMLIB [497](#)
 - DCB overrides data-set label [181](#)
 - define file [8](#)
 - defining merge data sets [227](#)
 - defining sort data sets [227](#)
 - JAVAERR [296](#)
 - JAVAIN [296](#)
 - JVAOUT [296](#)

- DD control statement (*continued*)
 - RLS parameter [209](#)
 - SYSADATA [272](#)
 - SYSDEBUG [272](#)
 - SYSIN [269](#)
 - SYSJAVA [272](#)
 - SYSLIB [270](#)
 - SYSLIN [271](#)
 - SYSMDECK [273](#)
 - SYSOPTF [270](#)
 - SYSPRINT [271](#)
 - SYPUNCH [271](#)
- ddname definition [8](#)
- deadlock in I/O error declarative [246](#)
- debugging
 - and performance [406](#)
 - compiler options for
 - overview [437](#)
 - TEST restriction [436](#)
 - THREAD restriction [436](#)
 - defining data set [272](#)
 - overview [433](#)
 - runtime options for [436](#)
 - using COBOL language features [434](#)
 - using the debugger [443](#)
- debugging, language features
 - class test [435](#)
 - debugging lines [436](#)
 - debugging statements [436](#)
 - declaratives [436](#)
 - DISPLAY statements [434](#)
 - file status keys [435](#)
 - INITIALIZE statements [435](#)
 - scope terminators [434](#)
 - SET statements [435](#)
 - WITH DEBUGGING MODE clause [436](#)
- DECK compiler option [350](#)
- declarative procedures
 - EXCEPTION/ERROR
 - with multithreading [246](#)
 - USE FOR DEBUGGING [436](#)
- deferred restart [747](#)
- DEFINE compiler option [350](#)
- defining
 - debug data set [272](#)
 - files, overview [8](#), [161](#)
 - libraries [270](#)
 - line-sequential files to z/OS [216](#)
 - QSAM files
 - to z/OS [180](#), [182](#)
 - sort or merge files under z/OS
 - [227](#)
 - VSAM files
 - to z/OS [205](#)
- dek suffix with cob2 [290](#)
- DELETE statement
 - compiler-directing [431](#)
 - multithreading serialization [580](#)
 - VSAM, coding [196](#)
- deleting records from VSAM file [203](#)
- delimited scope statement
 - description of [19](#)
 - nested [20](#)
- DEPENDING ON clause [170](#), [196](#)
- depth in tables [69](#)
- developing
 - AMODE 64 programs [521](#)
- device
 - classes [267](#)
 - requirements [267](#)
- DFHCOMMAREA parameter
 - calling nested CICS programs [486](#)
 - calling separately compiled CICS programs [485](#)
- DFHEIBLK parameter
 - calling nested CICS programs [486](#)
 - calling separately compiled CICS programs [485](#)
- DFSORT
 - defining data sets for [227](#)
 - error message for RETURN statement [226](#)
- diagnostics, program [447](#)
- DIAGTRUNC compiler option [352](#)
- direct-access
 - direct indexing [72](#)
 - file organization [162](#)
 - storage device (DASD) [212](#)
- directories
 - adding a path to [288](#)
- disability [957](#)
- disabling COBOL runtime options report option
 - DISABLEUOPTREPORT [424](#)
- DISPLAY (USAGE IS)
 - encoding and storage [139](#)
 - external decimal [47](#)
 - floating point [48](#)
- display floating-point data (USAGE DISPLAY) [47](#)
- DISPLAY statement
 - directing output [386](#)
 - displaying data values [35](#)
 - displaying on the system logical output device [36](#)
 - interaction with OUTDD [36](#)
 - suppressing line spacing [37](#)
 - under CICS [484](#)
 - using in debugging [434](#)
 - writing to stdout or stderr [36](#)
- DISPLAY-1 (USAGE IS)
 - encoding and storage [139](#)
- DISPLAY-OF intrinsic function
 - example with Chinese data [157](#)
 - example with Greek data [136](#)
 - example with UTF-8 data [153](#)
 - using [135](#), [151](#)
 - with XML documents [629](#)
- DISPSIGN compiler option [352](#)
- DLL
 - considerations [326](#)
- DLL compiler option
 - description [353](#)
 - for Java interoperability [291](#), [295](#)
 - for OO COBOL [291](#), [295](#)
 - multioption interaction [334](#)
- DLL igzjava.x
 - binding with
 - example [297](#)
 - preparing OO applications [296](#)
 - linking with
 - example [293](#)
 - preparing OO applications [292](#)
- DLL libjvm.x

- DLL libjvm.x (*continued*)
 - binding with
 - example [297](#)
 - preparing OO applications [296](#)
 - linking with
 - example [293](#)
 - preparing OO applications [292](#)
 - with EBCDIC services [718](#)
- DLLs (see dynamic link libraries) [567](#)
- do loop [100](#)
- do-until [100](#)
- do-while [100](#)
- documentation of program [5](#)
- DSA memory map [454](#)
- dump
 - requesting [241](#)
 - with DUMP compiler option [275](#)
- DUMP compiler option
 - description [355](#)
 - output [275](#)
- DYNAM compiler option
 - description [356](#)
 - multioption interaction [334](#)
 - performance considerations [765](#)
 - under Db2 with TSO or IMS [502](#)
 - with dynamic calls [536](#)
- dynamic calls
 - example [541](#)
 - making [536](#)
 - performance [540](#)
 - restrictions [536](#)
 - using with DLL linkage [571](#)
 - when to use [537](#)
 - with static calls [540](#)
- dynamic data areas, allocating storage [41](#)
- dynamic file allocation
 - order of allocation [165](#)
 - using CBLQDA [177](#)
 - using environment variables
 - line-sequential files [216](#)
 - QSAM files [180](#)
 - VSAM files [208](#)
- dynamic link libraries
 - about [567](#)
 - binder options for DLLs [569](#)
 - compiler options required [286](#)
 - compiling [568](#)
 - creating
 - from the z/OS UNIX shell [286](#)
 - overview [567](#)
 - creating for OO [292](#)
 - for Java interoperability [292](#)
 - in OO COBOL applications [575](#)
 - linking [569](#)
 - programs with DLL support must be reentrant [548](#)
 - search order for in z/OS UNIX file system [571](#)
 - using CALL identifier with [570](#)
 - using with C/C++ programs [574](#)
 - using with dynamic calls [571](#)
 - using with Java interoperability [293](#)
 - using with OO [293](#)

E

- E-level error message [282](#), [440](#)
- EBCDIC
 - code pages supported in XML documents [628](#)
 - converting to ASCII [116](#)
 - JNI services [717](#)
 - multibyte CCSIDs supported for DBCS [344](#)
- ECI calls and RETURN-CODE special register [486](#)
- efficiency of coding [757](#)
- EGCS [978](#)
- EJECT statement [431](#)
- embedded cross-reference
 - description [443](#)
 - example [476](#)
- embedded error messages [440](#)
- embedded MAP summary [442](#), [451](#)
- enclave [533](#)
- encoding
 - conflicts in XML documents [635](#), [636](#)
 - controlling in generated XML output [656](#)
 - description [139](#)
 - language characters [124](#)
 - of XML documents [628](#), [629](#)
 - of XML documents to be parsed [614](#)
 - specifying for alphanumeric XML document [630](#)
 - specifying with CODEPAGE option [343](#)
- encoding declaration
 - preferable to omit [630](#)
 - specifying [631](#)
- end-of-file (AT END phrase) [246](#)
- END-OF-INPUT XML event
 - example [647](#)
 - when parsing segments [625](#)
- enhancing XML output
 - example of modifying data definitions [662](#)
 - rationale and techniques [661](#)
- ENTER statement [431](#)
- entry point
 - alternate [548](#)
 - alternate in ENTRY statement [546](#)
 - ENTRY label [548](#)
 - passing entry addresses of [546](#)
 - procedure-pointer data item [546](#)
- ENTRY statement
 - for alternate entry points [546](#)
 - handling of program-name in [388](#)
- ENVAR runtime option [296](#)
- ENVIRONMENT DIVISION
 - class [672](#)
 - client [683](#)
 - collating sequence coding [7](#)
 - CONFIGURATION SECTION [5](#)
 - description [5](#)
 - entries for line-sequential files [215](#)
 - entries for QSAM files [167](#)
 - entries for VSAM files [191](#)
 - INPUT-OUTPUT SECTION [5](#)
 - instance method [676](#)
 - items present in, program initialization code [456](#)
 - signature information bytes [456](#)
 - subclass [693](#)
- environment variables
 - _BPX_SHAREAS [517](#)

environment variables (*continued*)

- _CEE_ENVFILE
 - description [515](#)
 - indicating Java settings [296](#)
- _CEE_RUNOPTS
 - description [515](#)
 - setting XPLINK [298](#)
 - specifying runtime options [513](#)
- _IGZ_SYSOUT [515](#)
- allocating line-sequential files [216](#)
- and copybooks [429](#)
- CLASSPATH
 - description [515](#)
 - example of setting [296](#)
 - specifying location of Java classes [293](#)
- COBJVMINITOPTIONS
 - description [515](#)
 - specifying JVM options [294](#)
- COBOL_INSTALL_DIR [283](#)
- COBOPT [283](#)
- compiler [283](#)
- defining files, example [8](#)
- defining QSAM files [180](#)
- example of setting and accessing [516](#)
- LIBPATH
 - description [515](#)
 - example of setting [296](#)
 - specifying location for COBOL classes [293](#)
- library-name [284](#), [429](#)
- PATH
 - description [515](#)
 - example of setting [296](#)
- runtime [515](#)
- setting and accessing [514](#)
- STEPLIB
 - description [515](#)
 - example [285](#)
- SYSLIB
 - description [283](#)
 - specifying location of JNI.cpy [291](#)
- text-name [284](#), [429](#)
- using to allocate files [164](#)

environment-name [5](#)

ERRMSG, for generating list of error messages [281](#)

error

- arithmetic [242](#)
- compiler options, conflicting [334](#)
- handling [241](#)
- handling for I/O [165](#)
- listing [275](#)
- message table
 - example using indexing [77](#)
 - example using subscripting [76](#)
- processing
 - line-sequential files [219](#)
 - QSAM files [179](#)
 - VSAM files [204](#)
 - XML GENERATE [656](#)
 - XML PARSE [635](#)
- routines for handling [252](#)

error messages

- compiler
 - choosing severity to be flagged [440](#)
 - correcting source [280](#)

error messages (*continued*)

- compiler (*continued*)
 - customizing [821](#)
 - determining what severity level to produce [359](#)
 - embedding in source listing [440](#)
 - format [281](#)
 - from exit modules [828](#)
 - generating a list of [281](#)
 - location in listing [281](#)
 - sending to terminal [271](#)
 - severity levels [282](#), [822](#)
- compiler-directed [281](#)
- ESDS (entry-sequenced data sets)
 - file access mode [194](#)
 - organization [192](#)
- euro currency sign [64](#)
- EVALUATE statement
 - case structure [93](#)
 - coding [93](#)
 - contrasted with nested IFs [94](#)
 - example that tests several conditions [95](#)
 - example with multiple WHEN phrases [94](#)
 - example with THRU phrase [94](#)
 - performance [94](#)
 - structured programming [758](#)
 - testing multiple values, example [97](#), [98](#)
 - use to test multiple conditions [91](#)
- evaluating data item contents
 - class test
 - for numeric [54](#)
 - overview [95](#)
 - INSPECT statement [112](#)
 - intrinsic functions [117](#)
- Example: COBPROD application
 - Example: COBPROD application [309](#)
- exception condition
 - CALL [251](#)
 - XML GENERATE [656](#)
 - XML PARSE [635](#)
- exception handling
 - with Java [708](#)
 - with XML GENERATE [656](#)
 - with XML PARSE [633](#)
- EXCEPTION XML event [635](#)
- EXCEPTION/ERROR declarative
 - description [246](#)
 - file status key [248](#)
 - line-sequential error processing [219](#)
 - QSAM error processing [179](#)
 - VSAM error processing [204](#)
- EXEC control statement, RD parameter of [746](#)
- EXIT compiler option
 - considerations for SQL and CICS statements [830](#)
 - description [357](#)
 - MSGEXIT suboption [819](#)
 - register usage [812](#)
 - user-exit work area [811](#)
 - using [357](#)
- exit modules
 - called for SYSADATA data set [818](#)
 - calling COBOL programs [812](#)
 - error messages generated [828](#)
 - message severity customization [819](#)
 - used in place of library-name [813](#)

- exit modules *(continued)*
 - used in place of SYSLIB [813](#)
 - used in place of SYSPRINT [816](#)
- EXIT PROGRAM statement
 - in subprogram [534](#)
 - with multithreading [534](#)
- explicit scope terminator [20](#)
- exponentiation
 - evaluated in fixed-point arithmetic [782](#)
 - evaluated in floating-point arithmetic [786](#)
 - performance tips [760](#)
- EXPORTALL compiler option
 - description [359](#)
 - DLL considerations [568](#)
 - multioption interaction [334](#)
- extended mode [43](#), [779](#)
- external class-name [672](#), [683](#)
- EXTERNAL clause
 - example for files [561](#)
 - for data items [560](#)
 - for sharing files [12](#), [561](#)
- external data
 - obtaining storage for [42](#)
 - sharing [560](#)
 - storage location of [42](#)
- external decimal data
 - national [47](#)
 - zoned [47](#)
- external file [561](#)
- external floating-point data
 - display [47](#)
 - national [48](#)
- External symbols [472](#)

F

- F-format record
 - layout [169](#)
 - requesting [168](#)
- factoring expressions [758](#)
- factory data
 - defining [696](#)
 - definition of [667](#)
 - making it accessible [696](#)
 - private [696](#)
- factory definition, coding [695](#)
- factory methods
 - defining [697](#)
 - definition of [667](#)
 - hiding [698](#)
 - invoking [698](#)
 - using to wrap procedural programs [703](#)
- FACTORY paragraph
 - factory data [696](#)
 - factory methods [697](#)
- factory section, defining [695](#)
- FASTSORT compiler option
 - description [359](#)
 - improving sort performance [232](#), [765](#)
 - information message [232](#)
 - requirements
 - JCL [232](#)
 - QSAM [233](#)
 - sort input and output files [233](#)

- FASTSORT compiler option *(continued)*
 - requirements *(continued)*
 - VSAM [234](#)
- FD (file description) entry [12](#)
- feedback
 - sending xxxv
- figurative constants
 - definition [26](#)
 - HIGH-VALUE restriction [132](#), [148](#)
 - national-character [132](#)
 - UTF-8-character [147](#)
- file access mode
 - choosing [162](#)
 - dynamic [194](#)
 - example [195](#), [425](#)
 - for indexed files (KSDS) [194](#)
 - for relative files (RRDS) [194](#)
 - for sequential files (ESDS) [194](#)
 - performance considerations [211](#)
 - random [194](#)
 - sequential [194](#)
 - summary table of [191](#)
- file allocation [164](#)
- file availability
 - QSAM files under z/OS [177](#)
 - VSAM files under z/OS [205](#)
- file description (FD) entry [12](#)
- file organization
 - choosing [162](#)
 - comparison of ESDS, KSDS, RRDS [190](#)
 - indexed [161](#), [192](#)
 - line-sequential [215](#)
 - overview [161](#)
 - QSAM [167](#)
 - relative [162](#)
 - relative-record [193](#)
 - sequential [161](#), [192](#)
 - VSAM [190](#)
- file position indicator (CRP) [198](#), [201](#)
- FILE SECTION
 - BLOCK CONTAINS clause [13](#)
 - CODE-SET clause [13](#)
 - DATA RECORDS clause [13](#)
 - description [11](#)
 - EXTERNAL clause [12](#)
 - FD entry [12](#)
 - GLOBAL clause [12](#)
 - LABEL RECORDS clause [13](#)
 - LINAGE clause [13](#)
 - OMITTED [13](#)
 - RECORD CONTAINS clause [13](#)
 - record description [11](#)
 - RECORD IS VARYING [13](#)
 - RECORDING MODE clause [14](#)
 - VALUE OF [13](#)
- FILE STATUS clause
 - description [165](#)
 - example [251](#)
 - line-sequential error processing [219](#)
 - NOFASTSORT error processing [234](#)
 - QSAM error processing [179](#)
 - using [247](#)

FILE STATUS clause (*continued*)
 VSAM error processing [204](#)
 with VSAM status code [248](#)

file status code
 02 [201](#)
 30 [200](#)
 37 [176](#)
 39 [177](#), [183](#), [186](#)
 49 [203](#)
 90 [175](#), [179](#), [204](#)
 92 [203](#), [516](#)

file status key
 05 [198](#)
 35 [198](#)
 39 [199](#)
 checking for I/O errors [247](#)
 checking for successful OPEN [247](#), [248](#)
 error handling [435](#)
 set for error handling [165](#)
 used with VSAM status code [248](#)
 VSAM, importance of in [204](#)

FILE-CONTROL paragraph
 example of entries [6](#)
 relation to FD entries [8](#)

files
 associating program files to external files [5](#)
 attributes [183](#)
 available
 QSAM [177](#)
 VSAM [205](#)
 changing name [9](#)
 CICS, restrictions under [5](#)
 COBOL coding
 DATA DIVISION entries [168](#), [195](#), [215](#)
 ENVIRONMENT DIVISION entries [167](#), [191](#), [215](#)
 input/output statements [176](#), [196](#), [217](#)
 overview [163](#)
 data sets, same meaning as [5](#)
 defining to operating system [8](#)
 describing [11](#)
 external [561](#)
 identifying to z/OS [180](#), [182](#), [205](#)
 line-sequential, allocating [216](#)
 multithreaded processing
 example [581](#)
 recommended organization [581](#)
 recommended usage patterns [581](#)
 serialization [580](#)
 optional
 QSAM [177](#)
 VSAM [199](#)
 overview [162](#)
 processing
 line-sequential [215](#)
 QSAM [167](#)
 VSAM [189](#)
 with multithreading [580](#)
 sort performance
 FASTSRT [232](#)
 variable-length files [228](#)
 storage of file-definition records [580](#)
 unavailable
 QSAM [177](#)
 VSAM [205](#)

files (*continued*)
 usage explanation [9](#)

FIPS messages
 categories [822](#)
 FLAGSTD compiler option [360](#)

fixed-length records
 QSAM
 layout [169](#)
 requesting [168](#)
 VSAM
 defining [195](#)
 RRDS [190](#)

fixed-point arithmetic
 comparisons [63](#)
 evaluation [62](#)
 example evaluations [63](#)
 exponentiation [782](#)

fixed-point data
 binary [48](#)
 conversions and precision [52](#)
 conversions between fixed- and floating-point [52](#)
 external decimal [47](#)
 intermediate results [781](#)
 packed-decimal [49](#)
 planning use of [758](#)

FLAG compiler option
 compiler output [441](#)
 description [359](#)
 using [440](#)

flags and switches [96](#)

FLAGSTD compiler option
 multioption interaction [335](#)

floating comment indicators (*>) [980](#)

floating-point arithmetic
 comparisons [63](#)
 evaluation [62](#)
 example evaluations [63](#)
 exponentiation [786](#)

floating-point data
 conversions and precision [52](#)
 conversions between fixed- and floating-point [52](#)
 external [47](#)
 intermediate results [786](#)
 internal
 format [49](#)
 performance tips [759](#)
 planning use of [758](#)

format of record
 fixed-length
 defining for VSAM [195](#)
 layout of QSAM [169](#)
 requesting for QSAM [168](#)
 for QSAM ASCII tape [187](#)
 format D
 layout [171](#)
 requesting [170](#)
 format F
 layout [169](#)
 requesting [168](#)
 format S
 layout [173](#)
 overview [172](#)
 requesting [171](#)
 format U

format of record (*continued*)

format U (*continued*)

layout [174](#)

requesting [173](#)

format V

layout [171](#)

requesting [170](#)

spanned

layout [173](#)

overview [172](#)

requesting [171](#)

undefined

layout [174](#)

requesting [173](#)

variable-length

defining for VSAM [196](#)

layout of QSAM [171](#)

requesting for QSAM [170](#)

formatted dump [241](#)

freeing object instances [691](#)

function prototype

best practices [590](#)

external name [589](#)

language standard considerations [591](#)

using [589](#)

function-pointer data item

addressing JNI services [833](#)

CALL statement [547](#)

calling COBOL [547](#)

calling DLL program

example [573](#)

calling Language Environment services [547](#)

definition [546](#)

SET function-pointer [546](#)

with DLLs [572](#)

G

garbage collection [691](#)

GB 18030 data

converting to or from national [157](#)

processing [157](#)

generating JSON null values [609](#)

generating XML output

example [657](#)

overview [651](#)

get and set methods [679](#)

GETMAIN, saving address of [811](#)

GLOBAL clause for files [12](#), [16](#)

global names [545](#)

Glossary [963](#)

GOBACK statement

in main program [534](#)

in subprogram [534](#)

with multithreading [534](#)

group item

cannot subordinate alphanumeric group within national group [137](#)

comparing to national data [142](#)

comparing with UTF-8 data [152](#)

definition [24](#)

for defining tables [67](#)

group move contrasted with elementary move [33](#), [138](#)

initializing

group item (*continued*)

initializing (*continued*)

using a VALUE clause [75](#)

using INITIALIZE [31](#), [73](#)

MOVE statement with [33](#)

passing as an argument [556](#)

treated as a group item

example with INITIALIZE [73](#)

in INITIALIZE [31](#)

variably located [81](#)

group move contrasted with elementary move [33](#), [138](#)

GROUP-USAGE NATIONAL clause

communicating with Java [712](#)

defining a national group [137](#)

defining tables [68](#)

example of declaring a national group [24](#)

initializing a national group [31](#)

grouping data to pass as an argument [556](#)

H

handling JSON null values [602](#)

header on listing [5](#)

HEAP runtime option

influencing data location [41](#)

multioption interaction [40](#)

HEX-OF intrinsic function [116](#)

HEX-TO-CHAR intrinsic function [117](#)

hexadecimal digits, converting from [117](#)

hexadecimal digits, converting to [116](#)

hexadecimal literals

as currency sign [64](#)

national

description [25](#)

using [126](#)

HGPR compiler option

multioption interaction [334](#)

performance considerations [765](#)

hiding factory methods [698](#)

hierarchy of compiler options

in batch [278](#)

in SYSOPTF data sets [384](#)

under z/OS [273](#)

under z/OS UNIX [284](#)

I

I-level message [282](#), [440](#)

IDENTIFICATION DIVISION

class [671](#)

CLASS-ID paragraph [671](#),

[693](#)

client [682](#)

coding [3](#)

DATE-COMPILED paragraph [3](#)

errors [3](#)

listing header example [5](#)

method [675](#)

PROGRAM-ID paragraph [3](#)

required paragraphs [3](#)

subclass [693](#)

TITLE statement [5](#)

IF statement

- IF statement (*continued*)
 - coding [91](#)
 - nested [92](#)
 - use EVALUATE instead for multiple conditions [92](#)
 - with null branch [91](#)
- IGZCA2D service routine [789](#)
- IGZCD2A service routine [791](#)
- igzcjava.x
 - binding with
 - example [297](#)
 - preparing OO applications [296](#)
 - linking with
 - example [293](#)
 - preparing OO applications [292](#)
- IGZEOPT module
 - with multithreading [583](#)
- IGZETUN module
 - with multithreading [583](#)
- IGZSRTCD data set [236](#)
- IGZUOPT
 - runtime options
 - JCL samples [425](#)
- IGZUOPT and IGZQUOPT runtime option [425](#)
- imperative statement, list [18](#)
- implicit scope terminator [20](#)
- IMS
 - COBOL-Java interoperability
 - accessing databases [511](#)
 - calling COBOL method from Java [509](#)
 - calling Java method from COBOL [510](#)
 - messages [511](#)
 - restriction on EXEC SQL [511](#)
 - STOP RUN [510](#)
 - synchronizing transactions [511](#)
 - using the AIB [511](#)
 - coding programs under
 - overview [505](#)
 - restrictions [5](#), [505](#)
 - compiling and linking for [508](#)
 - coprocessor
 - overview [505](#)
 - performance considerations [769](#)
 - SQLIMS compiler option [507](#)
 - SQLIMS statements
 - return codes [507](#)
 - SQLIMS INCLUDE [506](#)
 - using character data in [506](#)
 - using EXEC SQL under IMS [511](#)
- IMS SQL
 - coprocessor [506](#)
- incrementing addresses [557](#)
- index
 - assigning a value to [72](#)
 - computation of element displacement, example [70](#)
 - creating with OCCURS INDEXED BY clause [72](#)
 - definition [70](#)
 - incrementing or decrementing [72](#)
 - initializing [72](#)
 - key, detecting faulty [250](#)
 - range checking [439](#)
 - referencing other tables with [72](#)
- index data item
 - cannot use as subscript or index [72](#)
 - creating with USAGE IS INDEX clause [72](#)
- indexed file organization
 - description [161](#)
 - specifying [192](#)
- indexing
 - computation of element displacement, example [70](#)
 - definition [70](#)
 - example [77](#)
 - preferred to subscripting [760](#)
 - tables [72](#)
- INEXIT suboption of EXIT option
 - processing of [812](#)
 - syntax [358](#)
- inheritance hierarchy, definition of [668](#)
- INITCHECK compiler option
 - description [363](#)
 - invalid COBOL data [439](#)
- INITIAL attribute
 - effect on subprograms [536](#)
 - use of dynamic call and CANCEL instead [538](#)
- INITIAL clause
 - effect on main program [535](#)
 - effect on nested programs [4](#)
 - setting programs to initial state [4](#)
- INITIAL compiler option
 - description [364](#)
- Initial heap storage map [469](#)
- INITIALIZE statement
 - examples [27](#)
 - loading group values [31](#)
 - loading national group values [31](#)
 - loading table values [73](#)
 - REPLACING phrase [73](#)
 - using for debugging [435](#)
- initializing
 - a group item
 - using a VALUE clause [75](#)
 - using INITIALIZE [31](#), [73](#)
 - a national group item
 - using a VALUE clause [75](#)
 - using INITIALIZE [31](#), [74](#)
 - a structure using INITIALIZE [31](#)
 - a table
 - all occurrences of an element [76](#)
 - at the group level [75](#)
 - each item individually [75](#)
 - using INITIALIZE [73](#)
 - using PERFORM VARYING [101](#)
 - examples [27](#)
 - instance data [689](#)
 - variable-length group [80](#)
- inline comments [982](#)
- INLINE compiler option
 - description [365](#)
- inline PERFORM
 - example [100](#)
 - overview [99](#)
- input
 - coding for CICS [484](#)
 - coding for line-sequential files [217](#)
 - coding for QSAM files [176](#)
 - coding for VSAM files [196](#)
 - from files [161](#)
 - to compiler, under z/OS [267](#)
- input procedure

- input procedure (*continued*)
 - [coding 224](#)
 - [example 230](#)
 - [FASTSRT option not effective 233](#)
 - [requires RELEASE or RELEASE FROM 224](#)
 - [restrictions 227](#)
- INPUT-OUTPUT SECTION [5](#)
- input/output
 - [checking for errors 247](#)
 - [coding overview 163](#)
 - [controlling with FASTSRT option 359](#)
 - [logic flow after error 243](#)
 - [overview 161](#)
 - [processing errors](#)
 - [line-sequential files 219](#)
 - [QSAM files 179, 243](#)
 - [VSAM files 204, 243](#)
- input/output coding
 - [AT END \(end-of-file\) phrase 246](#)
 - [checking for successful operation 247](#)
 - [checking VSAM status codes 248](#)
 - [detecting faulty index key 250](#)
 - [error handling techniques 243](#)
 - [EXCEPTION/ERROR declaratives 246](#)
- INSERT statement [431](#)
- INSPECT statement
 - [avoid with UTF-8 data 632](#)
 - [examples 112](#)
 - [using 112](#)
- inspecting data (INSPECT) [112](#)
- instance
 - [creating 689](#)
 - [definition of 667](#)
 - [deleting 691](#)
- instance data
 - [defining 673, 694](#)
 - [definition of 667](#)
 - [initializing 689](#)
 - [making it accessible 679](#)
 - [private 673](#)
- instance methods
 - [defining 674, 694](#)
 - [definition of 667](#)
 - [invoking overridden 689](#)
 - [overloading 678](#)
 - [overriding 678](#)
- INTDATE compiler option
 - [description 365](#)
 - [effect on calendar starting date 59](#)
- INTEGER intrinsic function, example [112](#)
- INTEGER-OF-DATE intrinsic function [60](#)
- INTEGER-PART intrinsic function [112](#)
- integrated CICS translator
 - [advantages 488](#)
 - [compiler options for 487](#)
 - [overview 488](#)
- interactive program, example [939](#)
- Interactive System Productivity Facility (ISPF) [939](#)
- interlanguage communication
 - [and PL/I tasking 582](#)
 - [between COBOL and Java 707](#)
 - [IMS applications 510](#)
 - [subprograms 533](#)
 - [under CICS 485](#)
- interlanguage communication (*continued*)
 - [with multithreading 582](#)
- intermediate results [779](#)
- internal floating-point data (COMP-1, COMP-2) [49](#)
- interoperable data types with Java [711](#)
- interrupts [743](#)
- intrinsic functions
 - [as reference modifiers 111](#)
 - [converting alphanumeric data items with 113](#)
 - [converting national data items with 113](#)
 - [evaluating data items 117](#)
 - [example of](#)
 - [ANNUITY 61](#)
 - [BIT-OF 117](#)
 - [BIT-TO-CHAR 117](#)
 - [CHAR 118](#)
 - [CURRENT-DATE 60](#)
 - [DISPLAY-OF 136](#)
 - [HEX-OF 116](#)
 - [HEX-TO-CHAR 117](#)
 - [INTEGER 112](#)
 - [INTEGER-OF-DATE 60](#)
 - [LENGTH 60, 119, 120](#)
 - [LOG 61](#)
 - [LOWER-CASE 114](#)
 - [MAX 60, 88, 118, 119](#)
 - [MEAN 61](#)
 - [MEDIAN 61, 88](#)
 - [MIN 111](#)
 - [NATIONAL-OF 136](#)
 - [NUMVAL 115](#)
 - [NUMVAL-C 60, 115](#)
 - [ORD 118](#)
 - [ORD-MAX 88, 119](#)
 - [PRESENT-VALUE 61](#)
 - [RANGE 61, 88](#)
 - [REM 61](#)
 - [REVERSE 114](#)
 - [SQRT 61](#)
 - [SUM 88](#)
 - [UPPER-CASE 114](#)
 - [WHEN-COMPILED 121](#)
 - [example of Unicode functions 155](#)
 - [finding date of compilation 121](#)
 - [finding largest or smallest item 118](#)
 - [finding length of data items 120](#)
 - [intermediate results 784, 787](#)
 - [introduction to 38](#)
 - [nesting 38](#)
 - [numeric functions](#)
 - [differences from Language Environment callable services 58](#)
 - [equivalent Language Environment callable services 58](#)
 - [examples of 57](#)
 - [integer, floating-point, mixed 56](#)
 - [nested 57](#)
 - [special registers as arguments 57](#)
 - [table elements as arguments 57](#)
 - [uses for 56](#)
 - [processing table elements 87](#)
 - [UTF-8 153](#)
- INVALID KEY phrase
 - [description 250](#)

- INVALID KEY phrase (*continued*)
 - example [251](#)
- INVDATA compiler option
 - description [366](#)
- INVOKE statement
 - RETURNING phrase [688](#)
 - USING phrase [686](#)
 - using to create objects [689](#)
 - using to invoke methods [685](#)
 - with ON EXCEPTION [686](#), [698](#)
 - with PROCEDURE DIVISION RETURNING [560](#)
- invoking
 - COBOL programs under z/OS [564](#)
 - COBOL programs under z/OS UNIX [513](#)
 - factory or static methods [698](#)
 - instance methods [685](#)
 - Language Environment callable services [775](#)
- invoking user-defined functions [586](#)
- ISAM data set, analogous to VSAM KSDS data set [189](#)
- ISPF (Interactive System Productivity Facility) [939](#)

J

- J2EE client
 - example [719](#)
 - running [294](#)
- Java
 - and COBOL
 - binding [296](#)
 - communicating between [707](#)
 - compatibility [299](#)
 - compiling under z/OS UNIX [291](#)
 - compiling using JCL or TSO/E [295](#)
 - linking [292](#)
 - running [293](#), [296](#)
 - structuring applications [703](#)
 - array classes [711](#)
 - arrays
 - declaring [712](#)
 - example [715](#)
 - manipulating [714](#)
 - boolean array [712](#)
 - boolean type [712](#)
 - byte array [712](#)
 - byte type [712](#)
 - char array [712](#)
 - char type [712](#)
 - class types [712](#)
 - double array [713](#)
 - double type [712](#)
 - example
 - exception handling [709](#)
 - J2EE client [719](#)
 - processing an integer array [715](#)
 - exception
 - catching [708](#)
 - example [709](#)
 - handling [708](#)
 - throwing [708](#)
 - float array [713](#)
 - float type [712](#)
 - global references
 - JNI services for [710](#)
 - managing [709](#)

- Java (*continued*)
 - global references (*continued*)
 - object [709](#)
 - passing [710](#)
 - int array [712](#)
 - int type [712](#)
 - interoperability [707](#)
 - interoperable data types, coding [711](#)
 - invoking from a batch COBOL program [722](#)
 - jstring class [711](#)
 - local references
 - deleting [710](#)
 - freeing [710](#)
 - JNI services for [710](#)
 - managing [709](#)
 - object [709](#)
 - passing [710](#)
 - per multithreading [710](#)
 - saving [710](#)
 - long array [712](#)
 - long type [712](#)
 - methods
 - access control [711](#)
 - object array [713](#)
 - running with COBOL
 - under z/OS UNIX [293](#)
 - using JCL or TSO/E [296](#)
 - XPLINK linkage [298](#)
 - sharing data with [711](#)
 - short array [712](#)
 - short type [712](#)
 - string array [713](#)
 - strings
 - declaring [712](#)
 - manipulating [716](#)
- Java virtual machine
 - exceptions [708](#)
 - initializing [294](#)
 - object references [709](#)
- java.lang.Object
 - referring to as Base [671](#)
- javac command
 - compiling Java class definitions [291](#)
 - recompile for Java 6 or later [299](#)
- JAVAERR data set [296](#)
- JAVAIN data set [296](#)
- JAVAIOP compiler option
 - description [369](#)
- JAVAOUT data set [296](#)
- JCL
 - ASCII tape files [187](#)
 - cataloged procedures [256](#)
 - example of checkpoint/restart [748](#)
 - FASTSORT requirement [232](#)
 - for compiling [255](#)
 - for compiling in the z/OS UNIX file system [257](#)
 - for line-sequential files [216](#)
 - for merge [227](#)
 - for OO applications
 - example [297](#)
 - for QSAM files [181](#)
 - for sort [227](#)
 - for VSAM data sets [208](#)

- JNI
 - accessing services [707](#)
 - comparing object references [684](#)
 - converting local references to global [690](#)
 - EBCDIC services [717](#)
 - environment structure
 - addressability for [707](#)
 - exception handling services [708](#)
 - Java array services [714](#)
 - Java string services [716](#)
 - obtaining class object reference [708](#)
 - restrictions when using [708](#)
 - Unicode services [716](#)
 - UTF-8 services [719](#)
- JNI.cpy
 - for compiling [291](#)
 - for JNINativeInterface [707](#)
 - listing [833](#)
- JNIEnvPtr special register
 - use for JNI callable services [707](#)
- JNINativeInterface
 - environment structure [707](#)
 - JNI.cpy [707](#)
- JOB control statement, RD parameter of [746](#)
- job resubmission [748](#)
- job stream [533](#)
- JSON document
 - generating
 - null values [609](#)
 - overview [607](#)
 - parsing
 - description [597](#)
 - example [603](#)
 - null values [602](#)
 - overview [597](#)
- JSON exception codes
 - for generating [805](#)
 - for parsing [805](#)
- JSON generation
 - overview [607](#)
- JSON input
 - parsing
 - example [603](#)
 - overview [597](#)
- JSON output
 - generating
 - overview [607](#)
- JSON parse
 - example [603](#)
 - overview [597](#)
- JSON-CODE special register
 - exception codes for generating [805](#)
 - exception codes for parsing [805](#)
- JSON-STATUS special register
 - nonexception reason codes for parsing [805](#)
- jstring Java class [711](#)
- JZOS
 - example [722](#)
 - Java Batch Launcher and Toolkit for z/OS [725](#)

K

- Kanji comparison [95](#)

- Kanji data, testing for [159](#)
- keyboard navigation [957](#)
- keys
 - alternate in KSDS file [193](#)
 - for binary search [86](#)
 - for merging
 - defining [229](#)
 - overview [222](#)
 - for sorting
 - defining [229](#)
 - overview [222](#)
 - permissible data types
 - in MERGE statement [229](#)
 - in OCCURS clause [68](#)
 - in SORT statement [229](#)
 - prime in KSDS file [192](#)
 - relative-record [193](#)
 - to specify order of table elements [68](#)
- keyword [985](#)
- KSDS (key-sequenced data sets)
 - file access mode [194](#)
 - organization [192](#)

L

- LABEL RECORDS clause
 - FILE SECTION entry [13](#)
- LANGUAGE compiler option
 - description [370](#)
- Language Environment callable services
 - condition handling [773](#)
 - corresponding math intrinsic functions [58](#)
 - date and time computations [774](#)
 - differences from intrinsic functions [58](#)
 - dynamic storage services [773](#)
 - example of using [776](#)
 - feedback code [775](#)
 - for date and time [59](#)
 - for mathematics [58](#)
 - invoking with CALL [775](#)
 - mathematics [774](#)
 - message handling [774](#)
 - national language support [774](#)
 - omitted feedback code [776](#)
 - overview [773](#)
 - return code [776](#)
 - RETURN-CODE special register [776](#)
 - sample list of [774](#)
 - types of [773](#)
- large block interface (LBI) [175](#)
- largest or smallest item, finding [118](#)
- last-used state
 - subprograms with EXIT PROGRAM or GOBACK [535](#)
 - subprograms without INITIAL attribute [536](#)
- LBI (large block interface) [175](#)
- LENGTH intrinsic function
 - compared with LENGTH OF special register [121](#)
 - example [60](#), [120](#)
 - using [118](#)
 - variable-length results [119](#)
 - with national data [120](#)
- length of data items, finding [120](#)
- LENGTH OF special register
 - passing [552](#)

LENGTH OF special register (*continued*)
using [121](#)

level-88 item

- conditional expressions [95](#)
- setting switches off, example [99](#)
- setting switches on, example [98](#)
- switches and flags [96](#)
- testing multiple values, example [97](#)
- testing single values, example [97](#)

level-number [450](#)

LIBEXIT suboption of EXIT option

- processing of [813](#)
- syntax [358](#)

libjvm.x

- binding with
 - example [297](#)
 - preparing OO applications [296](#)
- linking with
 - example [293](#)
 - preparing OO applications [292](#)
- with EBCDIC services [718](#)

LIBPATH environment variable

- description [515](#)
- example of setting [296](#)
- specifying location for COBOL classes [293](#)

library

- BASIS [270](#)
- COPY [270](#)
- defining [270](#)
- directory entry [264](#)
- specifying path for [429](#)

library-name

- alternative if not specified [288](#)
- cross-reference to data-set names [475](#)
- when not used [813](#)

library-name environment variable [284](#)

limits of the compiler

- DATA DIVISION [11](#)
- user data [11](#)

line number [448](#), [479](#)

line-sequential files

- adding records to [218](#)
- blocking [12](#)
- closing [219](#)
- closing to prevent reopening [217](#)
- control characters in [216](#)
- DATA DIVISION entries [215](#)
- ENVIRONMENT DIVISION entries [215](#)
- input/output error processing [219](#)
- input/output statements for [217](#)
- national data not supported [218](#)
- opening [217](#)
- organization [215](#)
- processing [215](#)
- reading from [217](#)
- reading records from [218](#)
- under z/OS
 - allocating [216](#)
 - creating [216](#)
 - job control language (JCL) [216](#)
 - writing to [217](#)

LINECOUNT compiler option [371](#)

LINK macro [264](#)

link-editing user-defined functions [587](#)

LINKAGE SECTION

- coding [555](#)
 - for describing parameters [554](#)
 - with recursive calls [16](#)
 - with the THREAD option [16](#)
- linked-list processing, example [558](#)
- linking in the z/OS UNIX shell
- c89 command [285](#)
 - passing information to cob2 [287](#)
 - using the cob2 command
 - DLLs [286](#)
 - examples [286](#)
 - overview [285](#)

linking OO applications

- cob2 command [292](#)
- under z/OS UNIX
 - example [293](#)
 - overview [292](#)
- using JCL or TSO/E
 - example [297](#)
 - overview [296](#)

LIST compiler option

- assembler code for source program [454](#)
- base locator table [471](#)
- compiler output [456](#), [465](#)
- conflict with OFFSET option [443](#)
- Constant area section [470](#)
- description [372](#)
- DSA memory map [454](#), [472](#)
- External symbols section [472](#)
- getting output [443](#)
- Initial heap storage map section [469](#)
- MD5 signature example [466](#)
- multioption interaction [335](#)
- reading output [454](#)
- special register table [471](#)
- symbols used in output [453](#)
- Timestamp and version information example [466](#)

List of resources [1007](#)

listing output [443](#)

listings

- assembler expansion of PROCEDURE DIVISION [454](#)
- data and procedure-name cross-reference [442](#)
- embedded error messages [440](#)
- generating a short listing [443](#)
- line numbers, user-supplied [445](#)
- sorted cross-reference of program-names [474](#)
- sorted cross-reference of text-names [475](#)
- terms used in MAP output [452](#)
- text-name cross-reference [442](#)

literals

- alphanumeric
 - description [25](#)
 - with DBCS content [159](#)
- DBCS
 - description [25](#)
 - maximum length [159](#)
 - using [159](#)
- definition [25](#)
- hexadecimal
 - using [126](#)
- national
 - description [25](#)
 - using [126](#)

- literals (*continued*)
 - numeric [25](#)
 - using [25](#)
- little-endian, converting to big-endian [125](#)
- loading a table dynamically [73](#)
- local names [544](#)
- local references, converting to global [690](#)
- LOCAL-STORAGE SECTION
 - client [683](#), [684](#)
 - comparison with WORKING-STORAGE
 - example [14](#)
 - OO client [684](#)
 - overview [14](#)
 - determining location [41](#)
- LOG intrinsic function [61](#)
- logical record
 - description [161](#)
 - fixed-length format
 - defining for VSAM [195](#)
 - requesting for QSAM [168](#)
 - QSAM, definition [168](#)
 - variable-length format
 - defining for VSAM [196](#)
 - layout for QSAM [171](#)
 - requesting for QSAM [170](#)
- loops
 - coding [99](#)
 - conditional [100](#)
 - do [100](#)
 - in a table [101](#)
 - performed an explicit number of times [100](#)
- LOWER-CASE intrinsic function [114](#)
- lowercase, converting to [114](#)
- LP compiler option
 - multioption interaction [334](#)
- lst suffix with cob2 [290](#)

M

- main program
 - accessing parameter list in z/OS
 - example [564](#)
 - overview [564](#)
 - accessing parameter list in z/OS UNIX
 - example [518](#)
 - overview [518](#)
 - and subprograms [533](#)
 - dynamic calls [536](#)
- main storage, allocating to buffers [341](#)
- MAP compiler option
 - data items and relative addresses [275](#)
 - description [374](#)
 - embedded MAP summary [443](#)
 - example [449](#), [454](#)
 - nested program map
 - example [454](#)
 - symbols used in output [453](#)
 - terms used in output [452](#)
 - using [442](#), [443](#)
- mapping of DATA DIVISION items [443](#)
- mathematics
 - mathematics (*continued*)
 - intrinsic functions [57](#), [61](#)
 - Language Environment callable services [58](#), [774](#)
 - MAX intrinsic function
 - example table calculation [88](#)
 - example with functions [60](#)
 - using [118](#)
 - MAXPCF compiler option [375](#)
 - MDECK compiler option
 - description [376](#)
 - MEAN intrinsic function
 - example statistics calculation [61](#)
 - example table calculation [88](#)
 - MEDIAN intrinsic function
 - example statistics calculation [61](#)
 - example table calculation [88](#)
 - memory map
 - DSA [454](#)
 - merge
 - alternate collating sequence [231](#)
 - completion code [231](#)
 - criteria [229](#)
 - data sets needed under z/OS [227](#)
 - DD statements for defining z/OS data sets [227](#)
 - description [221](#)
 - determining success [231](#)
 - diagnostic message [231](#)
 - files, describing [223](#)
 - keys
 - defining [229](#)
 - overview [222](#)
 - pass control statements to [236](#)
 - process [222](#)
 - restrictions [221](#)
 - storage use [236](#)
 - terminating [232](#)
 - work files
 - describing [222](#)
 - MERGE statement
 - ASCENDING|DESCENDING KEY phrase [229](#)
 - COLLATING SEQUENCE phrase [7](#), [231](#)
 - description [228](#)
 - GIVING phrase [228](#)
 - overview [221](#)
 - restrictions [221](#)
 - USING phrase [228](#)
 - message handling, Language Environment callable services [774](#)
 - messages
 - compiler
 - choosing severity to be flagged [440](#)
 - customizing [821](#)
 - determining what severity level to produce [359](#)
 - embedding in source listing [440](#)
 - generating a list of [281](#)
 - sending to terminal [271](#)
 - severity levels [282](#), [822](#)
 - compiler-directed [281](#)
 - from exit modules [828](#)
 - sending to SYSTERM [403](#)
 - METHOD-ID paragraph [675](#)
 - methods
 - constructor [697](#)

methods (*continued*)

- factory [697](#)
- hiding factory [698](#)
- instance [674](#), [694](#)
- invoking [685](#), [698](#)
- invoking superclass [689](#)
- Java access control [711](#)
- obtaining passed arguments [677](#)
- overloading [678](#)
- overriding [678](#), [698](#)
- returning a value from [677](#)
- signature [675](#)
- migration considerations
 - Java and COBOL [299](#)
- MIN intrinsic function
 - example [111](#)
 - using [118](#)
- Mixed AMODE [525](#)
- mixed DBCS/EBCDIC literal
 - alphanumeric to DBCS conversion [789](#)
 - DBCS to alphanumeric conversion [791](#)
- mnemonic-name
 - SPECIAL-NAMES paragraph [5](#)
- MOVE statement
 - assigning arithmetic results [34](#)
 - converting to national data [134](#)
 - converting to UTF-8 data [150](#)
 - CORRESPONDING [33](#)
 - effect of ODO on lengths of sending and receiving items [78](#)
 - group move contrasted with elementary move [33](#), [138](#)
 - with elementary receiving items [32](#)
 - with group receiving items [33](#)
 - with national items [32](#)
- MSGEXIT suboption of EXIT option
 - effect on compilation return code [823](#)
 - example user exit [823](#)
 - message severity levels [822](#)
 - processing of [819](#)
 - syntax [358](#)
- MSGFILE runtime option [386](#)
- multiple currency signs
 - example [65](#)
 - using [64](#)
- multiple inheritance, not permitted [670](#), [692](#)
- multiple thread environment, running in [408](#)
- multithreading
 - AMODE setting [583](#)
 - asynchronous signals [583](#)
 - choosing data section
 - in an OO client [684](#)
 - closing QSAM files [179](#)
 - closing VSAM files [204](#)
 - COBOL programs [577](#)
 - coding file I/O
 - example [581](#)
 - recommended organization [581](#)
 - recommended usage patterns [581](#)
 - serialization [580](#)
 - control transfer [579](#)
 - ending programs [579](#)
 - EXIT PROGRAM statement [534](#)
 - GOBACK statement [534](#)
 - I/O error declaratives [246](#)

multithreading (*continued*)

- IGZEOPT [583](#)
- IGZETUN [583](#)
- interlanguage communication [582](#)
- limitations [582](#)
- nested programs [582](#)
- older compilers [583](#)
- overview [577](#)
- preinitializing [579](#)
- preparing COBOL programs for [577](#)
- recursion [579](#)
- recursive requirement [582](#)
- reentrancy [582](#)
- reentrancy requirement [582](#)
- runtime restrictions [583](#)
- sort and merge restriction [221](#)
- STOP RUN statement [534](#)
- synchronizing access to resources [582](#)
- terminology [577](#)
- THREAD compiler option
 - restrictions with [408](#)
 - when to choose [578](#)
- UPSI switches [583](#)
- with PL/I tasks [582](#)

N

- N delimiter for national or DBCS literals [25](#)
- NAME compiler option
 - description [377](#)
 - using [3](#)
- name declaration
 - searching for [545](#)
- NAMESPACE-DECLARATION XML event [621](#)
- naming
 - files [8](#)
 - programs [3](#)
- NATIONAL (USAGE IS)
 - external decimal [47](#)
 - floating point [48](#)
- national comparison [95](#)
- national data
 - BYTE-LENGTH intrinsic function and [121](#)
 - communicating with Java [712](#)
 - comparing
 - overview [140](#)
 - to alphabetic, alphanumeric, or DBCS [141](#)
 - to alphanumeric groups [142](#)
 - to numeric [141](#)
 - two operands [140](#)
 - concatenating (STRING) [103](#)
 - converting
 - exceptions [136](#)
 - from alphanumeric or DBCS with NATIONAL-OF [135](#)
 - from alphanumeric, DBCS, or integer with MOVE [134](#)
 - from UTF-8 with NATIONAL-OF [151](#)
 - overview [134](#)
 - to alphanumeric with DISPLAY-OF [135](#)
 - to numbers with NUMVAL, NUMVAL-C [115](#)
 - to or from Chinese GB 18030 [157](#)
 - to or from Greek alphanumeric, example [136](#)
 - to or from UTF-8 [153](#)
 - to uppercase or lowercase [114](#)

- national data (*continued*)
 - converting (*continued*)
 - to UTF-8 with DISPLAY-OF [151](#)
 - with INSPECT [112](#)
 - defining [126](#)
 - displaying on output [36](#)
 - encoding in XML documents [629](#)
 - evaluating with intrinsic functions [117](#)
 - external decimal [47](#)
 - external floating-point [48](#)
 - figurative constants [132](#)
 - finding the smallest or largest item [118](#)
 - in conditional expressions [140](#)
 - in generated XML documents [651](#)
 - in keys
 - in MERGE statement [229](#)
 - in OCCURS clause [68](#)
 - in SORT statement [229](#)
 - initializing, example of [29](#)
 - input with ACCEPT [35](#)
 - inspecting (INSPECT) [112](#)
 - LENGTH intrinsic function and [120](#)
 - LENGTH OF special register [121](#)
 - literals
 - using [126](#)
 - MOVE statement with [32](#), [134](#)
 - NSYMBOL compiler option if no USAGE clause [126](#)
 - reference modification of [109](#)
 - reversing characters [114](#)
 - specifying [125](#)
 - splitting (UNSTRING) [105](#)
 - VALUE clause with alphanumeric literal, example [120](#)
- national decimal data (USAGE NATIONAL)
 - defining [132](#)
 - example [43](#)
 - format [47](#)
 - initializing, example of [30](#)
- national floating-point data (USAGE NATIONAL)
 - defining [132](#)
 - definition [48](#)
- national group item
 - advantages over alphanumeric groups [133](#)
 - BYTE-LENGTH intrinsic function and [121](#)
 - can contain only national data [24](#), [137](#)
 - communicating with Java [712](#)
 - contrasted with USAGE NATIONAL group [25](#)
 - defining [137](#)
 - example [24](#)
 - for defining tables [68](#)
 - in generated XML documents [651](#)
 - initializing
 - using a VALUE clause [75](#)
 - using INITIALIZE [31](#), [74](#)
 - LENGTH intrinsic function and [120](#)
 - MOVE statement with [33](#)
 - overview [133](#)
 - passing as an argument [556](#)
 - treated as a group item
 - example with INITIALIZE [138](#)
 - in INITIALIZE [31](#)
 - in MOVE CORRESPONDING [33](#)
 - summary [138](#)
 - treated as an elementary item
 - example with MOVE [33](#)
- national group item (*continued*)
 - treated as an elementary item (*continued*)
 - in most cases [24](#), [133](#)
 - using
 - as an elementary item [138](#)
 - overview [137](#)
 - VALUE clause with alphanumeric literal, example [75](#)
- national language support (NLS)
 - DBCS [158](#)
 - LANGUAGE compiler option [370](#)
 - processing data [123](#)
- national literals
 - description [25](#)
 - using [126](#)
- national-edited data
 - defining [126](#)
 - editing symbols [126](#)
 - initializing
 - example [29](#)
 - using INITIALIZE [74](#)
 - MOVE statement with [32](#)
 - PICTURE clause [126](#)
- NATIONAL-OF intrinsic function
 - example with Chinese data [157](#)
 - example with Greek data [136](#)
 - example with UTF-8 data [153](#)
 - using [135](#), [151](#)
 - with XML documents [629](#)
- nested COPY statement [772](#), [814](#)
- nested delimited scope statements [20](#)
- nested IF statement
 - coding [92](#)
 - CONTINUE statement [91](#)
 - EVALUATE statement preferred [92](#)
 - with null branches [91](#)
- nested intrinsic functions [57](#)
- nested program integration [763](#)
- nested program map
 - description [443](#)
 - example [454](#)
- nested programs
 - calling [542](#)
 - description [543](#)
 - effect of INITIAL clause [4](#)
 - guidelines [542](#)
 - map [443](#), [454](#)
 - scope of names [544](#)
 - transfer of control [542](#)
- nesting level
 - program [448](#), [454](#), [479](#)
 - statement [448](#), [479](#)
- NEXT SENTENCE statement [92](#)
- NOCBLCARD translator option [489](#)
- NOCOMPILE compiler option
 - use to find syntax errors [438](#)
- NODLL compiler option
 - with dynamic calls [536](#)
 - with static calls [536](#)
- NODYNAM compiler option
 - under CICS [485](#)
 - under Db2 with CICS or CAF [502](#)
 - with static and dynamic calls [540](#)
 - with static calls [536](#)
 - with stored procedures [502](#)

NOFASTSRT compiler option [234](#), [237](#)
 NORENT compiler option
 multioption interaction [335](#)
 NOSQLCCSID compiler option recommended for compatibility with Db2 precompiler [500](#)
 Notices [959](#)
 NSYMBOL compiler option
 description [377](#)
 effect on N literals [25](#)
 for DBCS literals [126](#)
 for national data items [126](#)
 for national literals [126](#)
 multioption interaction [335](#)
 null branch [91](#)
 null-terminated strings
 example [108](#)
 handling [556](#)
 manipulating [108](#)
 NUMBER compiler option
 description [378](#)
 for debugging [445](#)
 NUMCHECK compiler option
 invalid COBOL data [439](#)
 NUMCLS installation option, effect on numeric class test [54](#)
 numeric class test
 checking for valid data [54](#)
 effect of NUMPROC, NUMCLS [54](#)
 numeric comparison [95](#)
 numeric data
 binary
 USAGE BINARY [48](#)
 USAGE COMPUTATIONAL (COMP) [48](#)
 USAGE COMPUTATIONAL-4 (COMP-4) [48](#)
 USAGE COMPUTATIONAL-5 (COMP-5) [48](#)
 can compare algebraic values regardless of USAGE [141](#)
 comparing to national [141](#)
 converting
 between fixed- and floating-point [51](#)
 precision [52](#)
 to national with MOVE [134](#)
 to UTF-8 with MOVE [150](#)
 defining [43](#)
 display floating-point (USAGE DISPLAY) [47](#)
 editing symbols [45](#)
 external decimal
 USAGE DISPLAY [47](#)
 USAGE NATIONAL [47](#)
 external floating-point
 USAGE DISPLAY [47](#)
 USAGE NATIONAL [48](#)
 internal floating-point
 USAGE COMPUTATIONAL-1 (COMP-1) [49](#)
 USAGE COMPUTATIONAL-2 (COMP-2) [49](#)
 national decimal (USAGE NATIONAL) [47](#)
 national floating-point (USAGE NATIONAL) [48](#)
 packed-decimal
 sign representation [53](#)
 USAGE COMPUTATIONAL-3 (COMP-3) [49](#)
 USAGE PACKED-DECIMAL [49](#)
 PICTURE clause [43](#), [45](#)
 storage formats [46](#)
 USAGE DISPLAY [43](#)
 USAGE NATIONAL [43](#)
 zoned decimal (USAGE DISPLAY)

numeric data (*continued*)
 zoned decimal (USAGE DISPLAY) (*continued*)
 format [47](#)
 sign representation [53](#)
 numeric intrinsic functions
 differences from Language Environment callable services [58](#)
 equivalent Language Environment callable services [58](#)
 example of
 ANNUITY [61](#)
 CURRENT-DATE [60](#)
 INTEGER [112](#)
 INTEGER-OF-DATE [60](#)
 LENGTH [60](#), [119](#)
 LOG [61](#)
 MAX [60](#), [88](#), [118](#), [119](#)
 MEAN [61](#)
 MEDIAN [61](#), [88](#)
 MIN [111](#)
 NUMVAL [115](#)
 NUMVAL-C [60](#), [115](#)
 ORD [118](#)
 ORD-MAX [88](#)
 PRESENT-VALUE [61](#)
 RANGE [61](#), [88](#)
 REM [61](#)
 SQRT [61](#)
 SUM [88](#)
 integer, floating-point, mixed [56](#)
 nested [57](#)
 special registers as arguments [57](#)
 table elements as arguments [57](#)
 uses for [56](#)
 numeric literals, description [25](#)
 numeric-edited data
 BLANK WHEN ZERO clause
 coding with numeric data [126](#)
 example [45](#)
 defining [126](#)
 editing symbols [45](#)
 initializing
 examples [30](#)
 using INITIALIZE [74](#)
 PICTURE clause [45](#)
 USAGE DISPLAY
 displaying [45](#)
 initializing, example of [30](#)
 USAGE NATIONAL
 displaying [45](#)
 initializing, example of [30](#)
 NUMPROC compiler option
 affected by NUMCLS [54](#)
 description [382](#)
 effect on sign processing [53](#)
 performance considerations [766](#)
 NUMVAL intrinsic function
 description [115](#)
 NUMVAL-C intrinsic function
 description [115](#)
 example [60](#)
 NX delimiter for national literals [25](#)

O

o suffix with cob2 [289](#), [290](#)

object

- creating [689](#)
- definition of [667](#)
- deleting [691](#)

object code

- compilation and listing [275](#)
- creating [271](#)
- generating [345](#)
- producing in 80-column record [350](#)

OBJECT compiler option

- description [383](#)
- multioption interaction [335](#)

object instances, definition of [667](#)

OBJECT paragraph

- instance data [673](#), [694](#)
- instance methods [674](#)

object references

- comparing [684](#)
- converting from local to global [690](#)
- example of passing [687](#)
- setting [685](#)
- typed [684](#)
- universal [684](#)

OBJECT-COMPUTER paragraph [5](#)

object-oriented COBOL

- binding
 - example [297](#)
 - overview [296](#)
- calls to and from OO programs [545](#)
- communicating with Java [711](#)
- compatibility [299](#)
- compiling
 - under z/OS UNIX [291](#)
 - using JCL or TSO/E [295](#)

DLLs in [575](#)

IMS

- accessing databases [511](#)
- calling COBOL method from Java [509](#)
- calling Java method from COBOL [510](#)

linking

- example [293](#)
- overview [292](#)

preparing applications

- under z/OS UNIX [292](#)
- using JCL or TSO/E [296](#)

programs must be reentrant [548](#)

restrictions

- cannot run under CICS [483](#)
- CICS [667](#)
- EXEC CICS statements [667](#)
- EXEC SQL statements [667](#)
- sort and merge [221](#)
- SQL compiler option [667](#)
- SQL statements [494](#)

running

- under z/OS UNIX [293](#)
- using JCL or TSO/E [296](#)
- XPLINK linkage [298](#)

writing OO programs [667](#)

OCCURS clause

- ASCENDING|DESCENDING KEY phrase

OCCURS clause (*continued*)

ASCENDING|DESCENDING KEY phrase (*continued*)

- example [86](#)
- needed for binary search [86](#)
- specify order of table elements [68](#)
- cannot use in a level-01 item [67](#)
- defining tables [67](#)
- for defining table elements [67](#)
- INDEXED BY phrase for creating indexes [72](#)
- nested for creating multidimensional tables [68](#)

OCCURS DEPENDING ON (ODO) clause

- complex [81](#)
- for creating variable-length tables [78](#)
- initializing ODO elements [80](#)
- ODO object [78](#)
- ODO subject [78](#)
- optimization [761](#)
- simple [78](#)
- variable-length records
 - QSAM [170](#)
 - VSAM [196](#)

OCCURS INDEXED BY clause, creating indexes with [72](#)

ODO object [78](#)

ODO subject [78](#)

OFFSET compiler option

- description [383](#)
- multioption interaction [335](#)
- output [477](#)

OMITTED clause, FILE SECTION [13](#)

OMITTED parameters [776](#)

OMITTED phrase for omitting arguments [555](#)

ON EXCEPTION phrase

- INVOKE statement [686](#), [698](#)

OPEN statement

- file availability [177](#), [198](#), [217](#)
- file status key [247](#)
- line-sequential files [217](#)
- multithreading serialization [580](#)
- QSAM files [176](#)
- VSAM files [196](#)

opening files

- line-sequential [217](#)
- multithreading serialization [580](#)
- QSAM [177](#)
- VSAM

empty [199](#)

overview [198](#)

OPTFILE compiler option [384](#)

optimization

- avoid ALTER statement [758](#)
- BINARY data items [759](#)
- consistent data [759](#)
- constant data items [758](#)
- contained program integration [763](#)
- effect of compiler options on [763](#)
- effect on parameter passing [554](#)
- effect on performance [757](#)
- factor expressions [758](#)
- index computations [761](#)
- indexing [760](#)
- nested program integration [763](#)
- OCCURS DEPENDING ON [761](#)
- out-of-line PERFORM [758](#)
- packed-decimal data items [759](#)

- optimization (*continued*)
 - performance implications [761](#)
 - procedure integration [763](#)
 - structured programming [757](#)
 - subscript computations [761](#)
 - subscripting [760](#)
 - table elements [760](#)
 - top-down programming [758](#)
 - unreachable code [763](#)
 - unused data items [385](#), [450](#)
- OPTIMIZE compiler option
 - description [385](#)
 - effect on parameter passing [554](#)
 - performance considerations [762](#), [766](#)
- optimizer
 - overview [762](#)
- optional files
 - QSAM [177](#)
 - VSAM [199](#)
- options
 - runtime [421](#)
- ORD intrinsic function, example [118](#)
- ORD-MAX intrinsic function
 - example table calculation [88](#)
 - using [119](#)
- ORD-MIN intrinsic function [119](#)
- order of evaluation
 - arithmetic operators [56](#), [780](#)
 - compiler options [334](#)
- out-of-line PERFORM [99](#)
- OUTDD compiler option
 - DD not allocated [36](#)
 - description [386](#)
 - interaction with DISPLAY [36](#)
- output
 - coding for CICS [484](#)
 - coding for line-sequential files [217](#)
 - coding for QSAM files [176](#)
 - coding for VSAM files [196](#)
 - data set [271](#)
 - from compiler, under z/OS [268](#)
 - to files [161](#)
- output files with cob2 [289](#)
- output procedure
 - coding [225](#)
 - example [226](#), [230](#)
 - FASTSRT option not effective [233](#)
 - requires RETURN or RETURN INTO [225](#)
 - restrictions [227](#)
- overflow condition
 - CALL [251](#)
 - joining and splitting strings [242](#)
 - UNSTRING [105](#)
- overloading instance methods [678](#)
- overriding
 - factory methods [698](#)
 - instance methods [678](#)

P

- packed decimal data [378](#)
- packed-decimal data item
 - description [49](#)
 - sign representation [53](#)

- packed-decimal data item (*continued*)
 - synonym [46](#)
 - using efficiently [49](#), [759](#)
- page
 - control [178](#)
 - depth [13](#)
- paragraph
 - definition [18](#)
 - grouping [101](#)
- parameters
 - accessing from main program in
 - z/OS
 - example [564](#)
 - overview [564](#)
 - accessing from main program in z/OS
 - UNIX
 - example [518](#)
 - overview [518](#)
 - ADEXIT [818](#)
 - describing in called program [554](#)
 - INEXIT [812](#)
 - LIBEXIT [815](#)
 - MSGEXIT [820](#)
 - PRTEXIT [817](#)
- PARMCHECK compiler option
 - invalid COBOL program [439](#)
- parse data item, definition [614](#)
- parsing JSON documents [597](#)
- parsing XML documents
 - description [613](#)
 - one segment at a time
 - example [646](#)
 - overview [625](#)
 - overview [612](#)
 - UTF-8 [632](#)
 - white space [629](#)
 - with validation
 - example [648](#)
 - overview [622](#)
 - performance considerations [623](#)
 - restrictions [623](#)
 - XML declaration [629](#)
- passing arguments BY CONTENT to user-defined functions [587](#)
- passing data between programs
 - addresses [557](#)
 - arguments in calling program [553](#)
 - BY CONTENT [551](#)
 - BY REFERENCE [551](#)
 - BY VALUE
 - overview [551](#)
 - restrictions [554](#)
 - EXTERNAL data [560](#)
 - JNI services [708](#)
 - OMITTED arguments [555](#)
 - options considerations [41](#)
 - parameters in called program [554](#)
 - RETURN-CODE special register [559](#)
 - with Java [711](#)
- password
 - alternate index [205](#)
 - example [205](#)
 - VSAM files [204](#)
- PASSWORD clause [204](#)

- PATH environment variable
 - description [515](#)
 - example of setting [296](#)
- path name
 - for copybook search [288](#), [429](#)
- PERFORM statement
 - coding loops [99](#)
 - for a table
 - example using indexing [77](#)
 - example using subscripting [76](#)
 - for changing an index [72](#)
 - inline [99](#)
 - out-of-line [99](#)
 - performed an explicit number of times [100](#)
 - TEST AFTER [100](#)
 - TEST BEFORE [100](#)
 - THRU [101](#)
 - TIMES [100](#)
 - UNTIL [100](#)
 - VARYING [101](#)
 - VARYING WITH TEST AFTER [101](#)
 - WITH TEST AFTER . . . UNTIL [100](#)
 - WITH TEST BEFORE . . . UNTIL [100](#)
- performance
 - AIXBLD runtime option [769](#)
 - and debugging [406](#)
 - APPLY WRITE-ONLY clause [10](#)
 - arithmetic evaluations [758](#)
 - arithmetic expressions [759](#)
 - blocking QSAM files [174](#), [340](#)
 - calls [540](#)
 - CBLPShPOP considerations [491](#)
 - CBLPShPOP runtime option [491](#)
 - CICS
 - overview [757](#)
 - CICS coding [769](#)
 - coding for [757](#)
 - coding tables [760](#)
 - compiler option
 - AFP [764](#)
 - ARCH [764](#)
 - ARITH [764](#)
 - AWO [764](#)
 - BLOCK0 [765](#)
 - DYNAM [765](#)
 - FASTSRT [765](#)
 - HGPR [765](#)
 - INITCHECK [765](#)
 - INLINE [765](#)
 - MAXPCF [766](#)
 - NUMCHECK [766](#)
 - NUMPROC [53](#), [766](#)
 - OPTIMIZE [762](#), [766](#)
 - PARM CHECK [766](#)
 - RENT [766](#)
 - RMODE [767](#)
 - SMARTBIN [767](#)
 - SQLCCSID [500](#)
 - SSRANGE [767](#)
 - TEST [767](#)
 - THREAD [409](#), [767](#)
 - TRUNC [409](#), [767](#)
 - TUNE [767](#)
 - consistent data types [759](#)
 - performance (*continued*)
 - data usage [759](#)
 - effect of compiler options on [763](#)
 - effects of buffer size [341](#)
 - exponentiations [760](#)
 - FASTSRT [232](#)
 - IMS environment [508](#), [769](#)
 - OCCURS DEPENDING ON [761](#)
 - optimizer
 - overview [762](#)
 - order of WHEN phrases in EVALUATE [94](#)
 - out-of-line PERFORM compared with inline [99](#)
 - parsing XML documents with validation [623](#)
 - programming style [757](#)
 - sorting with FASTSRT [232](#)
 - striped extended-format QSAM data sets [185](#)
 - table handling [761](#)
 - table searching
 - binary compared with serial [84](#)
 - improving serial search [85](#)
 - tape, QSAM [175](#)
 - tuning [757](#)
 - variable subscript data format [71](#)
 - VSAM files [211](#), [769](#)
 - worksheet [768](#)
 - period as scope terminator [20](#)
 - PGMNAME compiler option
 - COMPAT suboption [388](#)
 - description [388](#)
 - LONGMIXED suboption [389](#)
 - LONGUPPER suboption [389](#)
 - phrase, definition of [18](#)
 - physical block [161](#)
 - physical record [14](#), [161](#)
 - PICTURE clause
 - cannot use for internal floating point [44](#)
 - determining symbol used [348](#)
 - incompatible data [53](#)
 - N for national data [126](#)
 - national-edited data [126](#)
 - numeric data [43](#)
 - numeric-edited data [126](#)
 - Z for zero suppression [45](#)
 - PL/I tasking
 - POSIX runtime option [582](#)
 - with COBOL [582](#)
 - pointer data item
 - description [39](#)
 - incrementing addresses with [557](#)
 - NULL value [557](#)
 - passing addresses [557](#)
 - processing chained lists [557](#)
 - used to process chained list [558](#)
 - porting applications
 - effect of separate sign [44](#)
 - POSIX
 - calling APIs [516](#)
 - threads [582](#)
 - POSIX runtime option
 - effect on DLL search order [571](#)
 - use in OO applications [296](#)
 - precedence
 - arithmetic operators [56](#), [780](#)
 - CICS options [487](#)

precedence (*continued*)

- compiler options
 - in batch [278](#)
 - in SYSOPTF data sets [270](#), [384](#)
 - under z/OS [273](#)
 - under z/OS UNIX [284](#)
- copybook search order [283](#)

preferred sign [53](#)

preinitializing the COBOL environment

- with multithreading [579](#)

PRESENT-VALUE intrinsic function [61](#)

preserving original sequence in a sort [231](#)

priority numbers, segmentation [764](#)

procedure and data-name cross-reference, description [442](#)

PROCEDURE DIVISION

- additional information [456](#)
- client [682](#)
- description [17](#)
- in subprograms [556](#)
- instance method [677](#)
- RETURNING
 - to return a value [17](#)
 - using [560](#)
- s present in [456](#)
- signature information bytes [456](#)
- statements
 - compiler-directing [19](#)
 - conditional [18](#)
 - delimited scope [19](#)
 - imperative [18](#)
- terminology [17](#)
- USING
 - BY VALUE [556](#)
 - to receive parameters [17](#), [554](#)

procedure integration [763](#)

procedure-pointer data item

- calling C/C++ [547](#)
- calling JNI services [547](#)
- definition [546](#)
- entry address for entry point [546](#)
- passing parameters to callable services [546](#)
- SET procedure-pointer [546](#)
- with DLLs [572](#)

process

- definition [577](#)

PROCESS (CBL) statement

- batch compiling [278](#)
- conflicting options in [334](#)
- overview [431](#)
- precedence
 - in batch [278](#)
 - under z/OS [273](#)
 - under z/OS UNIX [284](#)
- specifying compiler options [274](#)

processing

- chained lists
 - example [558](#)
 - overview [557](#)
- tables
 - example using indexing [77](#)
 - example using subscripting [76](#)

processing JSON input [597](#)

producing JSON output [607](#)

producing XML output [651](#)

product support xxxiii, [1007](#)

program

- attribute codes [454](#)
- compiling and linking using cob2
 - DLLs [286](#)
 - examples [286](#)
 - overview [285](#)
- compiling under z/OS [255](#)
- compiling under z/OS UNIX [283](#)
- decisions
 - EVALUATE statement [91](#)
 - IF statement [91](#)
 - loops [100](#)
 - PERFORM statement [100](#)
 - switches and flags [96](#)
- developing for z/OS UNIX [513](#)
- diagnostics [447](#)
- initialization code [465](#)
- limitations [757](#)
- main [533](#)
- nesting level [448](#), [479](#)
- reentrant [548](#)
- restarting [746](#)
- signature information bytes [456](#)
- statistics [448](#)
- structure [3](#)
- subprogram [533](#)

PROGRAM COLLATING SEQUENCE clause

- does not affect national or DBCS operands [7](#)
- establishing collating sequence [7](#)
- overridden by COLLATING SEQUENCE phrase [7](#)
- overrides default collating sequence [231](#)

Program information

- listing example [467](#)

program processing table [485](#)

Program prolog area

- listing example [468](#)

program termination

- actions taken in main and subprogram [534](#)
- statements [534](#)

PROGRAM-ID paragraph

- coding [3](#)
- COMMON attribute [4](#)
- INITIAL clause [4](#)

program-names

- avoid using certain prefixes [3](#)
- cross-reference [474](#)
- handling of case [388](#)
- specifying [3](#)

protecting VSAM files [204](#)

PRTEXIT suboption of EXIT option

- processing of [816](#)
- syntax [358](#)

publications [1007](#)

Q

QSAM files

- adding records to [178](#)
- ASCII tape file [187](#)
- ASSIGN clause [167](#)
- attributes [183](#)
- BLOCK CONTAINS clause [174](#), [340](#)

- QSAM files *(continued)*
 - block size [174, 340](#)
 - blocking enhances performance [174, 340](#)
 - blocking records [174, 186](#)
 - closing [179](#)
 - closing to prevent reopening [177](#)
 - DATA DIVISION entries [168](#)
 - ENVIRONMENT DIVISION entries [167](#)
 - FASTSRT requirements [233](#)
 - input/output error processing [179, 243](#)
 - input/output statements for [176](#)
 - obtaining buffers for [186](#)
 - opening [177](#)
 - processing
 - existing files [184](#)
 - in reverse order [177](#)
 - new files [185](#)
 - overview [167](#)
 - z/OS UNIX files [186](#)
 - replacing records [178](#)
 - retrieving [182](#)
 - striped extended-format [185](#)
 - tape performance [175](#)
 - under z/OS
 - creating files [180, 182](#)
 - DD statement for [180, 182](#)
 - defining [180, 182](#)
 - environment variable for [180](#)
 - file availability [177](#)
 - job control language (JCL) [181](#)
 - updating files [178](#)
 - using same input/output file under FASTSRT [233](#)
 - writing to a printer [178](#)
- QUALIFY compiler option [390](#)
- QUOTE compiler option [337](#)

R

- railroad track diagrams, how to read [xxxix](#)
- random numbers, generating [58](#)
- RANGE intrinsic function
 - example statistics calculation [61](#)
 - example table calculation [88](#)
- RCFs
 - sending [xxxv](#)
- RD parameter of JOB or EXEC statement [746](#)
- READ INTO for format-V VSAM files [196](#)
- READ NEXT statement [196](#)
- READ statement
 - AT END phrase [246](#)
 - line-sequential files [217](#)
 - multithreading serialization [580](#)
 - QSAM [176](#)
 - VSAM [196](#)
- reader comments
 - sending [xxxv](#)
- reading records
 - block size [174](#)
 - from line-sequential files [218](#)
- reading records from VSAM files
 - dynamically [201](#)
 - randomly [201](#)
 - sequentially [201](#)

- reason code from XML parsing [633, 795](#)
- record
 - description [11](#)
 - format
 - fixed-length QSAM [168, 169](#)
 - fixed-length VSAM [195](#)
 - format D [170, 171, 187](#)
 - format F [168, 169, 187](#)
 - format S [171–173](#)
 - format U [173, 174, 187](#)
 - format V [170, 171, 187](#)
 - QSAM ASCII tape [187](#)
 - spanned [171–173](#)
 - undefined [173, 174](#)
 - variable-length QSAM [170, 171](#)
 - variable-length VSAM [196](#)
 - order, effect of organization on [161](#)
- RECORD CONTAINS clause
 - FILE SECTION entry [13](#)
- RECORD KEY clause
 - identifying prime key in KSDS files [192](#)
- RECORDING MODE clause
 - fixed-length records, QSAM [168](#)
 - QSAM files [14](#)
 - specify record format [168](#)
 - variable-length records, QSAM [170, 171](#)
- recursive calls
 - and the LINKAGE SECTION [16](#)
 - coding [545](#)
 - identifying [4](#)
- REDEFINES clause, making a record into a table using [75](#)
- reentrant programs [548](#)
- reference modification
 - example [110](#)
 - expression checking with SSRANGE [401](#)
 - generated XML documents [652](#)
 - intrinsic functions [109](#)
 - national data [109](#)
 - out-of-range values [110](#)
 - tables [71, 109](#)
 - UTF-8 documents [153](#)
- reference modifier
 - arithmetic expression as [111](#)
 - intrinsic function as, example [111](#)
 - variables as [110](#)
- registers used by EXIT compiler option [812](#)
- relation condition [95](#)
- relative file organization [162](#)
- RELEASE FROM statement
 - compared to RELEASE [224](#)
 - example [224](#)
- RELEASE statement
 - compared to RELEASE FROM [224](#)
 - with SORT [224](#)
- REM intrinsic function [61](#)
- RENT compiler option
 - description [391](#)
 - for DLLs [568](#)
 - for IMS [508](#)
 - for Java interoperability [291, 295](#)
 - for OO COBOL [291, 295](#)
 - influencing addressability [41](#)
 - multioption interaction [40, 334](#)
 - performance considerations [766](#)

- RENT compiler option (*continued*)
 - when passing data [41](#)
- REPLACE statement
 - Db2 considerations [501](#)
 - description [431](#)
- replacing
 - data items (INSPECT) [112](#)
 - records in QSAM file [178](#)
 - records in VSAM file [203](#)
 - text, Db2 considerations [501](#)
- REPLACING phrase (INSPECT), example [112](#)
- REPOSITORY paragraph
 - class [672](#)
 - client [683](#)
 - coding [5](#)
 - subclass [693](#)
- representation
 - data [53](#)
 - sign [53](#)
- RERUN clause
 - checkpoint/restart [237](#)
- reserved-word table, CICS
 - alternate
 - overview [490](#)
 - specifying with WORD [415](#)
- residency mode, definition [39](#)
- restart
 - automatic [746](#)
 - automatic or deferred [743](#)
 - deferred [747](#)
 - overview [743](#)
- restarting a program [746](#)
- restrictions
 - CICS
 - 16 MB line [484](#)
 - calls [485](#)
 - coding [483](#)
 - files [5](#)
 - OUTDD compiler option [386](#)
 - parsing with validation using FILE [623](#)
 - separate translator [488](#)
 - sorting [238](#)
 - Db2 coprocessor [498](#)
 - IMS
 - 16 MB line [484](#)
 - coding [5](#), [505](#)
 - IMS SQL coprocessor [507](#)
 - input/output procedures [227](#)
 - OO programs [667](#)
 - SQL compiler option [498](#)
 - SQL statements [494](#)
 - SQLIMS compiler option [507](#)
 - subscripting [71](#)
- resubmitting a job [748](#)
- return code
 - compiler
 - depends on highest severity [282](#)
 - effect of message customization [823](#)
 - overview [282](#)
 - feedback code from Language Environment services [776](#)
 - from CICS ECI [486](#)
 - from Db2 SQL statements [497](#)
 - from XML parsing [633](#), [795](#)
- return code (*continued*)
 - RETURN-CODE special register [560](#), [776](#)
 - VSAM files
 - description [248](#)
 - example [249](#)
 - RLS mode [210](#)
 - when control returns to operating system [560](#)
- RETURN statement
 - required in output procedure [225](#)
 - with INTO phrase [226](#)
- RETURN-CODE special register
 - calls to Language Environment services [776](#)
 - CICS ECI calls [486](#)
 - considerations for Db2 [497](#)
 - not set by INVOKE [686](#)
 - passing data between programs [560](#)
 - sharing return codes between programs [559](#)
 - when control returns to operating system [560](#)
- RETURNING phrase
 - CALL statement [560](#)
 - INVOKE statement [688](#)
 - PROCEDURE DIVISION header [560](#), [677](#)
- REVERSE intrinsic function [114](#)
- reverse order of tape files [177](#)
- reversing characters [114](#)
- REWRITE statement
 - multithreading serialization [580](#)
 - QSAM [176](#)
 - VSAM [196](#)
- RLS parameter [209](#)
- RMODE
 - description [39](#)
 - of EXIT modules [812](#)
- RMODE compiler option
 - description [392](#)
 - influencing addressability [40](#)
 - multioption interaction [40](#)
 - performance considerations [767](#)
 - when passing data [41](#)
- ROUNDED phrase [780](#)
- rows in tables [69](#)
- RRDS (relative-record data sets)
 - file access mode [194](#)
 - fixed-length records [190](#), [194](#)
 - organization [193](#)
 - performance considerations [212](#)
 - simulating variable-length records [194](#)
 - variable-length records [190](#), [194](#)
- RULES compiler option
 - description [393](#)
- run time
 - accessing arguments in
 - z/OS
 - example [564](#)
 - overview [564](#)
 - accessing arguments in z/OS
 - UNIX
 - example [518](#)
 - overview [518](#)
 - changing file-name [9](#)
 - multithreading restrictions [583](#)
 - performance [757](#)
- run unit
 - description [533](#)

- run unit (*continued*)
 - role in multithreading [577](#)
- Running
 - COBOL AMODE 64 applications [327](#)
- running OO applications
 - under z/OS UNIX
 - overview [293](#)
 - XPLINK linkage [298](#)
 - using JCL or TSO/E [296](#)
 - XPLINK linkage [298](#)
- runtime
 - options [421](#)
- runtime option
 - AMODE3164 [529](#)
- runtime options
 - 85 COBOL Standard conformance [333](#)
 - AIXBLD [769](#)
 - ALL31 [538](#)
 - CBLOPTS [564](#)
 - CBLP SHPOP [491](#)
 - DEBUG [436](#)
 - ENVAR [296](#)
 - MSGFILE [386](#)
 - POSIX
 - DLL search order [571](#)
 - use in OO applications [296](#)
 - specifying under z/OS [564](#)
 - specifying under z/OS UNIX [513](#)
 - TRAP
 - closing files in QSAM [179](#)
 - closing files in VSAM [203](#)
 - closing line-sequential files [219](#)
 - ON SIZE ERROR [242](#)
 - XPLINK
 - not recommended as a default [298](#)
 - setting [298](#)

S

- S-format record
 - layout [173](#)
 - overview [172](#)
 - requesting [171](#)
- S-level error message [282](#), [440](#)
- sample JCL
 - sample JCL [314](#)
- sample programs [935](#)
- scope of names
 - global [545](#)
 - local [544](#)
- scope terminator
 - aids in debugging [434](#)
 - explicit [19](#), [20](#)
 - implicit [20](#)
- SD (sort description) entry, example [223](#)
- SEARCH ALL statement
 - binary search [86](#)
 - example [86](#)
 - for changing an index [72](#)
 - table must be ordered [86](#)
- search order
 - DLLs in the z/OS UNIX file system [571](#)
- SEARCH statement

- SEARCH statement (*continued*)
 - example [85](#)
 - for changing an index [72](#)
 - nesting to search more than one level of a table [85](#)
 - serial search [85](#)
- searching
 - for name declarations [545](#)
 - tables
 - binary search [86](#)
 - overview [84](#)
 - performance [84](#)
 - serial search [85](#)
- section
 - declarative [21](#)
 - definition [18](#)
 - grouping [101](#)
 - segmentation [764](#)
- SELECT clause
 - ASSIGN clause [8](#)
 - naming files [8](#)
 - vary input-output file [9](#)
- SELECT OPTIONAL
 - QSAM [177](#)
 - VSAM [199](#)
- SELF [685](#)
- sentence, definition of [18](#)
- separate CICS translator
 - compiler options for [487](#), [489](#)
 - restrictions [488](#)
 - using [489](#)
- separate sign
 - for line-sequential files [218](#)
 - portability [44](#)
 - printing [44](#)
 - required for signed national decimal [44](#)
- SEQUENCE compiler option [395](#)
- sequential file organization [161](#)
- sequential search
 - description [85](#)
 - example [85](#)
- sequential storage device [162](#)
- serial search
 - description [85](#)
 - example [85](#)
- serialization of files with multithreading [580](#)
- SERVICE compiler option [395](#)
- SERVICE LABEL statement [431](#)
- SET condition-name TO TRUE statement
 - example [100](#), [101](#)
 - switches and flags [98](#)
- SET statement
 - for changing an index [72](#)
 - for changing index data items [72](#)
 - for function-pointer data items [545](#)
 - for object references [685](#)
 - for procedure-pointer data items [545](#)
 - for setting a condition, example [98](#)
 - handling of program-name in [388](#)
 - using for debugging [435](#)
- setting
 - index data items [72](#)
 - indexes [72](#)
 - switches and flags [98](#)
- sharing

sharing (*continued*)

data

- between separately compiled programs [560](#)
- coding the LINKAGE SECTION [555](#)
- from another program [16](#)
- in recursive or multithreaded programs [16](#)
- in separately compiled programs [16](#)
- overview [551](#)
- parameter-passing mechanisms [551](#)
- passing arguments to a method [686](#)
- PROCEDURE DIVISION header [556](#)
- RETURN-CODE special register [559](#)
- returning a value from a method [688](#)
- scope of names [544](#)
- with Java [711](#)

files

- scope of names [544](#)
- using EXTERNAL clause [12](#), [561](#)
- using GLOBAL clause [12](#)

short listing, example [445](#)

sign condition

- testing sign of numeric operand [95](#)

SIGN IS SEPARATE clause

- for line-sequential files [218](#)
- portability [44](#)
- printing [44](#)
- required for signed national decimal data [44](#)

sign representation [53](#)

signature

- definition of [675](#)
- must be unique [675](#)

signature information bytes

- compiler options in effect [456](#)
- DATA DIVISION [456](#)
- ENVIRONMENT DIVISION [456](#)
- PROCEDURE DIVISION [456](#)

size of printed page, control [178](#)

skip a block of records [174](#)

SMARTBIN compiler option

- performance considerations [767](#)

softcopy information [xxxiii](#)

sort

- alternate collating sequence [231](#)
- checkpoint/restart [237](#)
- completion code [231](#)
- controlling behavior of [235](#)
- criteria [229](#)
- data sets needed under z/OS [227](#)
- DD statements for defining z/OS data sets [227](#)
- description [221](#)
- determining success [231](#)
- diagnostic message [231](#)
- FASTSRT compiler option
 - improving performance [232](#)
 - requirements [233](#)
 - using same QSAM file for input and output [233](#)
- files, describing [223](#)
- input procedures
 - coding [224](#)
 - example [230](#)
- keys
 - defining [229](#)
 - overview [222](#)
- NOFASTSRT compiler option [234](#)

sort (*continued*)

output procedures

- coding [225](#)
- example [226](#), [230](#)
- pass control statements to [236](#)
- performance

FASTSRT [232](#)

variable-length files [228](#)

preserving original sequence [231](#)

process [222](#)

restrictions [221](#)

restrictions on input/output procedures [227](#)

special registers [235](#)

storage use [236](#)

terminating [232](#)

under CICS [238](#)

variable-length records [227](#)

work files

describing [222](#)

workspace [237](#)

SORT statement

ASCENDING|DESCENDING KEY phrase [229](#)

COLLATING SEQUENCE phrase [7](#), [231](#)

description [228](#)

GIVING phrase [228](#)

overview [221](#)

restrictions [221](#)

restrictions for CICS applications [238](#)

under CICS

- change reserved-word table [491](#)

USING phrase [228](#)

SORT-CONTROL special register [235](#)

SORT-CORE-SIZE special register [235](#)

SORT-FILE-SIZE special register [235](#)

SORT-MESSAGE special register [235](#)

SORT-MODE-SIZE special register [235](#)

SORT-RETURN special register

determining sort or merge success [231](#)

terminating sort or merge [232](#)

SORTCKPT DD statement [237](#)

sorting

tables

overview [87](#)

SOURCE and NUMBER output, example [448](#)

source code

compiler data set [269](#)

line number [448](#), [450](#), [454](#), [479](#)

listing, description [443](#)

program listing [275](#)

SOURCE compiler option

description [397](#)

getting output [443](#)

SOURCE-COMPUTER paragraph [5](#)

SPACE compiler option [398](#)

spanned files [172](#)

spanned record format

description [171](#)

layout [173](#)

requesting [171](#)

special feature specification [5](#)

special register

ADDRESS OF

use in CALL statement [552](#)

- special register (*continued*)
 - arguments in intrinsic functions [57](#)
 - JNIEnvPtr
 - use for JNI callable services [707](#)
 - LENGTH OF [121](#), [552](#)
 - RETURN-CODE [559](#)
 - SORT-RETURN
 - determining sort or merge success [231](#)
 - terminating sort or merge [232](#)
 - using in XML parsing [615](#), [617](#)
 - WHEN-COMPILED [121](#)
 - XML-CODE [615](#), [618](#)
 - XML-EVENT [615](#), [617](#)
 - XML-INFORMATION [615](#)
 - XML-NAMESPACE [616](#), [621](#)
 - XML-NAMESPACE-PREFIX [616](#), [621](#)
 - XML-NNAMESPACE [616](#), [621](#)
 - XML-NNAMESPACE-PREFIX [616](#), [621](#)
 - XML-NTEXT [615](#), [620](#)
 - XML-TEXT [615](#), [620](#)
- special register table [471](#)
- SPECIAL-NAMES paragraph
 - coding [5](#)
 - QSAM files [187](#)
- Specifying
 - COBOL only runtime options [327](#)
 - COBOL runtime options [422](#)
- splitting data items (UNSTRING) [105](#)
- SQL compiler option
 - description [398](#)
 - restrictions
 - compiling in batch [498](#)
 - OO programs [667](#)
 - using [497](#)
- SQL statements
 - CCSID determination [498](#)
 - coding
 - overview [494](#)
 - restriction [494](#)
 - EXIT compiler option and [830](#)
 - overview [493](#)
 - restrictions [494](#)
 - return codes [497](#)
 - SQL DECLARE [495](#)
 - SQL INCLUDE [495](#)
 - use for Db2 services [493](#)
 - using binary data in [497](#)
 - using character data in [495](#)
 - using national decimal data [496](#)
- SQLCA
 - declare for programs that use SQL statements [494](#)
 - declare for programs that use SQLIMS statements [506](#)
 - return codes from Db2 [497](#)
- SQLCCSID compiler option
 - description [399](#)
 - effect on CCSID of string data [498](#)
 - performance considerations [500](#)
 - recommended with Db2 coprocessor [500](#)
- SQLIMS compiler option
 - restrictions
 - compiling in batch [507](#)
 - using [507](#)
- SQLIMS statements
 - coding
 - SQLIMS statements (*continued*)
 - coding (*continued*)
 - overview [506](#)
 - EXIT compiler option and [830](#)
 - SQLIMS INCLUDE [506](#)
 - SQRT intrinsic function [61](#)
 - SSRANGE compiler option
 - description [401](#)
 - performance considerations [767](#)
 - reference modification [110](#)
 - using [439](#)
 - STACK runtime option
 - influencing data location [41](#)
 - multioption interaction [40](#)
 - STANDARD clause, FD entry [13](#)
 - START statement
 - multithreading serialization [580](#)
 - VSAM [196](#)
 - statement
 - compiler-directing [19](#)
 - conditional [18](#)
 - definition [18](#)
 - delimited scope [19](#)
 - explicit scope terminator [20](#)
 - imperative [18](#)
 - implicit scope terminator [20](#)
 - nesting level [448](#), [479](#)
 - statement cross-reference listing
 - description [443](#)
 - statements used in program [443](#)
 - static calls
 - example [541](#)
 - making [536](#)
 - performance [540](#)
 - with dynamic calls [540](#)
 - static data areas, allocating storage [41](#)
 - static data, definition of [667](#)
 - static methods
 - definition of [667](#)
 - invoking [698](#)
 - statistics intrinsic functions [61](#)
 - status code, VSAM files
 - description [248](#)
 - example [249](#)
 - stderr
 - controlling line spacing [37](#)
 - directing with DISPLAY [36](#)
 - setting DISPLAY to [515](#)
 - stdin
 - reading with ACCEPT [35](#)
 - stdout
 - controlling line spacing [37](#)
 - directing with DISPLAY [36](#)
 - setting DISPLAY to [515](#)
 - STEPLIB environment variable
 - description [515](#)
 - example of specifying compiler [285](#)
 - STGOPT compiler option [402](#)
 - STOP RUN statement
 - in main program [534](#)
 - in subprogram [534](#)
 - with multithreading [534](#)
 - storage
 - character data [139](#)

- storage (*continued*)
 - device
 - direct-access [162](#)
 - sequential [162](#)
 - for arguments [553](#)
 - management with Language Environment callable services [773](#)
 - mapping [443](#)
 - use during sort [236](#)
- Storage [469](#)
- stride, table [761](#)
- STRING statement
 - example [104](#)
 - overflow condition [242](#)
 - using [103](#)
 - with DBCS data [789](#)
- strings
 - handling [103](#)
 - Java
 - declaring [712](#)
 - manipulating [716](#)
 - null-terminated [556](#)
 - striped extended-format QSAM file [185](#)
 - structure, initializing using INITIALIZE [31](#)
 - structured programming [758](#)
 - structuring OO applications [703](#)
 - structuring user-defined functions [585](#)
- subclass
 - coding
 - example [694](#)
 - overview [692](#)
 - instance data [694](#)
- subprogram
 - and main program [533](#)
 - definition [551](#)
 - description [533](#)
 - linkage
 - common data items [554](#)
 - PROCEDURE DIVISION in [556](#)
 - termination
 - effects [534](#)
- subscript
 - definition [70](#)
 - literal, example [70](#)
 - range checking [439](#)
 - variable, example [70](#)
- subscripting
 - definition [70](#)
 - example [76](#)
 - literal, example [70](#)
 - reference modification [71](#)
 - relative [71](#)
 - restrictions [71](#)
 - use data-name or literal [71](#)
 - variable, example [70](#)
- substitution character [132](#)
- substrings
 - of table elements [109](#)
 - reference modification of [109](#)
- SUM intrinsic function, example table calculation [88](#)
- SUPER [689](#)
- support xxxiii, [1007](#)
- SUPPRESS compiler option [403](#)
- switch-status condition [95](#)
- switches and flags
 - defining [97](#)
 - description [96](#)
 - resetting [98](#)
 - setting switches off, example [99](#)
 - setting switches on, example [98](#)
 - testing multiple values, example [97](#)
 - testing single values, example [97](#)
- SYMBOLIC CHARACTERS clause [8](#)
- symbolic constant [758](#)
- syntax diagrams, how to read xxxi
- syntax errors
 - finding with NOCOMPILE compiler option [438](#)
- SYSABEND file
 - description [268](#)
- SYSADATA
 - file, creating [272](#)
 - output [335](#)
 - records, exit module [818](#)
- SYSADATA file
 - description [268](#)
 - example [839](#)
 - file contents [837](#)
 - record descriptions [840](#)
 - record types [838](#)
- SYSDEBUG data set
 - defining [272](#)
 - use of [405](#)
- SYSDEBUG file
 - description [268](#)
- SYSIN data set
 - defining [269](#)
 - description [267](#)
- SYSJAVA file
 - defining [272](#)
 - description [268](#)
- SYSLIB data set
 - defining [270](#)
 - description [267](#)
 - when not used [813](#)
- SYSLIB environment variable
 - description [283](#)
 - specifying location of JNI.cpy [291](#)
- SYSLIN data set
 - description [268](#)
- SYSMDECK file
 - defining [273](#)
 - description [268](#)
- SYSMDUMP file
 - description [268](#)
- SYSOPTF data set
 - defining [270](#)
 - description [267](#)
- SYSPRINT data set
 - defining [271](#)
 - description [268](#)
 - when not used [816](#)
- SYSPUNCH data set
 - description [268](#), [271](#)
 - requirements for DECK compiler option [350](#)
- system date
 - under CICS [485](#)
- system dump [241](#)
- system-determined block size

- system-determined block size (*continued*)
 - compiler data sets [269](#)
 - QSAM files [175](#), [340](#)
- system-name [5](#)
- SYSTEM data set
 - defining [271](#)
 - description [268](#)
 - sending messages to [403](#)
- SYSUDUMP file
 - description [268](#)
- SYSUT data set [267](#)

T

table

- assigning values to [74](#)
- columns [67](#)
- compare to array [39](#)
- defining with OCCURS clause [67](#)
- definition [67](#)
- depth [69](#)
- description [39](#)
- dynamically loading [73](#)
- efficient coding [760](#), [761](#)
- elements [67](#)
- identical element specifications [760](#)
- index, definition [70](#)
- initializing
 - all occurrences of an element [76](#)
 - at the group level [75](#)
 - each item individually [75](#)
 - using INITIALIZE [73](#)
 - using PERFORM VARYING [101](#)
- loading values in [73](#)
- looping through [101](#)
- multidimensional [68](#)
- one-dimensional [67](#)
- processing with intrinsic functions [87](#)
- redefining a record as [75](#)
- reference modification [71](#)
- referencing substrings of elements [109](#)
- referencing with indexes, example [70](#)
- referencing with subscripts, example [70](#)
- referring to elements [70](#)
- rows [69](#)
- searching
 - binary [86](#)
 - overview [84](#)
 - performance [84](#)
 - sequential [85](#)
 - serial [85](#)
- sorting
 - overview [87](#)
- stride computation [761](#)
- subscript, definition [70](#)
- three-dimensional [69](#)
- two-dimensional [69](#)
- variable-length
 - creating [78](#)
 - example of loading [80](#)
 - initializing [80](#)
 - preventing overlay in [83](#)
- TALLYING phrase (INSPECT), example [112](#)
- tape files

- tape files (*continued*)
 - performance [175](#)
 - reverse order [177](#)
- TERMINAL compiler option [403](#)
- terminal, sending messages to the [403](#)
- terminating XML parsing [637](#)
- termination [534](#)
- terminology
 - VSAM [189](#)
- terms used in MAP output [452](#)
- test
 - conditions [100](#)
 - data [95](#)
 - numeric operand [95](#)
 - UPSI switch [95](#)
- TEST AFTER [100](#)
- TEST BEFORE [100](#)
- TEST compiler option
 - description [404](#)
 - multioption interaction [335](#)
 - performance considerations [767](#)
 - use for debugging [443](#)
- text-name cross-reference, description [442](#)
- text-name environment variable [284](#)
- THREAD compiler option
 - and the LINKAGE SECTION [16](#)
 - cannot use with nested programs [543](#)
 - description [408](#)
 - for Java interoperability [291](#), [295](#)
 - for OO COBOL [291](#), [295](#)
 - multioption interaction [335](#)
 - performance considerations [767](#)
- threading
 - and preinitialization [579](#)
 - control transfer [579](#)
 - ending programs [579](#)
 - z/OS UNIX considerations [513](#)
- TITLE statement
 - controlling header on listing [5](#)
- top-down programming
 - constructs to avoid [758](#)
- TRACK OVERFLOW option [176](#)
- Trademarks [961](#)
- transferring control
 - between COBOL and non-COBOL programs [533](#)
 - between COBOL programs [535](#), [542](#)
 - called program [533](#)
 - calling program [533](#)
 - main and subprograms [533](#)
 - nested programs [543](#)
- transforming COBOL data to XML
 - example [657](#)
 - overview [651](#)
- TRAP runtime option
 - closing line-sequential files [219](#)
 - closing QSAM files [179](#)
 - closing VSAM files [203](#)
 - ON SIZE ERROR [242](#)
- troubleshooting
 - COBOL/Java interoperability [308](#)
- TRUNC compiler option
 - description [409](#)
 - performance considerations [767](#)

TRUNC compiler option (*continued*)
 suboptions for separate CICS translator [489](#)
 TSO
 ALLOCATE command [262](#)
 CALL command [262](#)
 compiling under
 example CLIST [264](#)
 overview [262](#)
 SYSTERM for compiler messages [271](#)
 TUNE compiler option
 performance considerations [767](#)
 tuning considerations, performance [763](#), [764](#)
 typed object references [684](#)

U

u suffix with cob2 [290](#)
 U-format record
 layout [174](#)
 requesting [173](#)
 U-level error message [282](#), [440](#)
 unavailable files
 QSAM [177](#)
 VSAM [205](#)
 UNBOUNDED groups
 processing [88](#)
 undefined record format
 layout [174](#)
 QSAM [187](#)
 requesting [173](#)
 unfilled tracks [176](#)
 Unicode
 description [124](#)
 encoding and storage [139](#)
 JNI services [716](#)
 processing data [123](#)
 using with Db2 [495](#)
 universal object references [684](#)
 UNIX
 calling APIs [516](#)
 unreachable code [763](#)
 UNSTRING statement
 example [106](#)
 overflow condition [242](#)
 using [105](#)
 with DBCS data [789](#)
 updating VSAM records [201](#)
 UPPER-CASE intrinsic function [114](#)
 uppercase, converting to [114](#)
 UPSI switches with multithreading [583](#)
 USAGE clause
 at the group level [25](#)
 incompatible data [53](#)
 INDEX phrase, creating index data items with [72](#)
 NATIONAL phrase at the group level [133](#)
 OBJECT REFERENCE [684](#)
 UTF-8 phrase at the group level [150](#)
 USE FOR DEBUGGING declaratives
 overview [436](#)
 USE statement [432](#)
 user exit routine
 DFSORT [237](#)
 user-defined condition [95](#)
 user-defined functions

user-defined functions (*continued*)
 coding [585](#)
 compiling [586](#)
 invoking [586](#)
 link-editing [587](#)
 passing arguments BY CONTENT [587](#)
 specifying external names [587](#)
 structuring [585](#)
 using [585](#), [589](#)
 user-exit work area [811](#)
 using function prototype [589](#)
 USING phrase
 INVOKE statement [686](#)
 PROCEDURE DIVISION header [556](#), [677](#)
 using user-defined functions [585](#), [589](#)
 UTF-16
 definition [125](#)
 encoding for national data [125](#)
 UTF-8
 'U' pic symbol [142](#)
 avoid INSPECT [632](#)
 avoid moves that truncate [632](#)
 avoid reference modification with XML documents [153](#)
 Character-data [144](#)
 converting to or from national [153](#)
 data items [142](#)
 Db2 considerations [147](#)
 definition [125](#)
 DYNAMIC LENGTH [142](#)
 dynamic-length [142](#)
 encoding and storage [139](#)
 encoding for ASCII invariant characters [125](#)
 example of generating an XML document [653](#)
 fixed byte-length [142](#)
 fixed character-length [142](#)
 intrinsic functions supported [146](#)
 JNI services [719](#)
 literals [144](#)
 parsing XML documents [632](#)
 PICTURE [142](#)
 processing data items [153](#)
 statements supported [145](#)
 using intrinsic functions [153](#)
 UTF-8 data
 comparing
 overview [151](#)
 to alphabetic, alphanumeric [152](#)
 to alphanumeric groups [152](#)
 two operands [152](#)
 converting
 from alphanumeric or national with MOVE [150](#)
 overview [150](#)
 to national with NATIONAL-OF [151](#)
 figurative constants [147](#)
 in conditional expressions [151](#), [152](#)
 MOVE statement with [150](#)
 using Unicode intrinsic functions [155](#)
 UTF-8 data items
 reference modifiers [148](#)
 UTF-8 data types [142](#)
 UTF-8 group item
 advantages over alphanumeric groups [149](#)
 overview [149](#)
 treated as an elementary item

UTF-8 group item (*continued*)
treated as an elementary item (*continued*)
in most cases [149](#)

V

V-format record
layout [171](#)
requesting [170](#)

validating XML documents
example [648](#)
overview [622](#)
performance considerations [623](#)
restrictions [623](#)

VALUE clause
alphanumeric literal with national data, example [120](#)
alphanumeric literal with national group, example [75](#)
assigning table values
at the group level [75](#)
to each item individually [75](#)
to each occurrence of an element [76](#)
assigning to a variable-length group [80](#)
cannot use for external floating point [48](#)
initializing internal floating-point literals [44](#)
large literals with COMP-5 [49](#)
large, with TRUNC(BIN) [410](#)

VALUE IS NULL [557](#)

VALUE OF clause [13](#)

variable
as reference modifier [110](#)
definition [23](#)

variable-length records
OCCURS DEPENDING ON (ODO) clause [761](#)

QSAM
layout [171](#)
requesting [170](#)

sorting [227](#)

VSAM
defining [196](#)
RRDS [190](#)

variable-length table
assigning values to [80](#)
creating [78](#)
example [79](#)
example of loading [80](#)
preventing overlay in [83](#)

variables, environment
example of setting and accessing [516](#)
library-name [429](#)
runtime [515](#)

variably located data item [81](#)

variably located group [81](#)

VBREF compiler option
description [412](#)
output example [478](#)
using [443](#)

VLR compiler option
description [412](#)

VSAM
dynamic access
option [424](#)

VSAM files
adding records to [202](#)
allocating with environment variable [208](#)

VSAM files (*continued*)
closing [203](#)
coding input/output statements [196](#)
comparison of file organizations [190](#)
creating alternate indexes [206](#)
DATA DIVISION entries [195](#)
deleting records from [203](#)
ENVIRONMENT DIVISION entries [191](#)
error processing [243](#)
extended addressability [212](#)
file position indicator (CRP) [198](#), [201](#)
file status key [204](#)
input/output error processing [204](#)
loading
dynamically or randomly [199](#)
extended format [200](#)
sequentially [199](#)
with access method services [200](#)

opening
empty [199](#)
overview [198](#)
performance considerations [211](#)
processing files [189](#)
protecting with password [204](#)
reading records from [201](#)
record-level sharing (RLS)
error handling [210](#)
overview [209](#)
preventing update problems [210](#)
restrictions [210](#)

replacing records in [203](#)

status codes
description [248](#)
example [249](#)

under z/OS
defining data sets [205](#)
file availability [205](#)
JCL [208](#)
RLS mode [209](#)
updating records [201](#)

VSAM terminology
BDAM data set [189](#)
comparison to non-VSAM terms
[189](#)
ESDS for QSAM [189](#)
KSDS for ISAM [189](#)
RRDS for BDAM [189](#)

VSAMOPENFS compiler option [414](#)

W

W-level message [282](#), [440](#)

WHEN phrase
EVALUATE statement [93](#)
SEARCH ALL statement [86](#)
SEARCH statement [85](#)

WHEN-COMPILED intrinsic function [121](#)

WHEN-COMPILED special register [121](#)

white space in XML documents [629](#)

with AMODE 64 programs
POINTER data items [521](#)

WITH DEBUGGING MODE clause
for debugging lines [436](#)
for debugging statements [436](#)

- WITH POINTER phrase
 - STRING [103](#)
 - UNSTRING [105](#)
- WORD compiler option
 - description [415](#)
 - multioption interaction [335](#)
 - recommended for CICS integrated translator [487](#)
 - recommended for CICS separate translator [489](#)
- work data sets for compiling [267](#)
- WORKING-STORAGE SECTION
 - client [683](#), [684](#)
 - comparison with LOCAL-STORAGE
 - example [14](#)
 - OO client [684](#)
 - overview [14](#)
 - factory data [696](#)
 - instance data [673](#), [694](#)
 - instance method [676](#)
 - multithreading considerations [684](#)
 - storage location for data [349](#)
- workspace
 - use during sort [237](#)
- wrapper, definition of [703](#)
- wrapping procedure-oriented programs [703](#)
- write a block of records [174](#)
- WRITE ADVANCING statement [178](#)
- WRITE statement
 - line-sequential files [217](#)
 - multithreading serialization [580](#)
 - QSAM [176](#)
 - VSAM [196](#)

X

- x suffix with cob2 [289](#), [290](#)
- XML declaration
 - generating [653](#)
 - specifying encoding declaration [631](#)
 - white space cannot precede [629](#)
- XML document
 - accessing [613](#)
 - code pages supported [628](#)
 - controlling the encoding of [656](#)
 - EBCDIC special characters [631](#)
 - encoding [628](#), [629](#)
 - enhancing
 - example of modifying data definitions [662](#)
 - rationale and techniques [661](#)
 - events
 - example [642](#)
 - generating
 - example [657](#)
 - overview [651](#)
 - handling parsing exceptions [633](#)
 - national language [629](#)
 - parser [612](#)
 - parsing
 - description [613](#)
 - example [638](#), [642](#), [646](#)
 - large documents [627](#)
 - one segment at a time [625](#)
 - UTF-8 [632](#)
 - parsing with validation

- XML document (*continued*)
 - parsing with validation (*continued*)
 - example [648](#)
 - overview [622](#)
 - performance considerations [623](#)
 - restrictions [623](#)
 - processing [611](#)
 - specifying encoding if alphanumeric [630](#)
 - white space [629](#)
 - XML declaration [629](#)
- XML event
 - CONTENT-CHARACTERS
 - example [647](#)
 - when parsing segments [626](#)
 - encoding conflicts [635](#), [636](#)
 - END-OF-INPUT
 - example [647](#)
 - when parsing segments [625](#)
 - EXCEPTION [635](#)
 - fatal errors [635](#)
 - NAMESPACE-DECLARATION [621](#)
 - overview [617](#)
 - processing [612](#), [615](#)
 - processing procedure [613](#)
- XML exception codes
 - for generating [803](#)
 - for parsing [795](#)
 - for parsing with XMLPARSE(COMPAT)
 - handleable [797](#)
 - not handleable [800](#)
 - for parsing with XMLPARSE(XMLSS) [795](#)
- XML GENERATE statement
 - COUNT IN [656](#)
 - NAME [654](#)
 - NAMESPACE [653](#)
 - NAMESPACE-PREFIX [653](#)
 - NOT ON EXCEPTION [655](#)
 - ON EXCEPTION [656](#)
 - SUPPRESS [654](#)
 - TYPE [655](#)
 - WITH ATTRIBUTES [652](#)
 - WITH ENCODING [656](#)
 - XML-DECLARATION [653](#)
- XML generation
 - controlling type of XML data [655](#)
 - counting generated characters [652](#)
 - description [651](#)
 - enhancing output
 - example of modifying data definitions [662](#)
 - rationale and techniques [661](#)
 - example [657](#)
 - generating attributes [652](#)
 - generating elements [652](#)
 - handling errors [656](#)
 - ignored data items [652](#)
 - naming attributes or elements [654](#)
 - no byte order mark [656](#)
 - overview [651](#)
 - suppressing generation of specified attributes or elements [654](#)
 - using namespace prefixes [653](#)
 - using namespaces [653](#)
- XML output
 - controlling the encoding of [656](#)

- XML output (*continued*)
 - enhancing
 - example of modifying data definitions [662](#)
 - rationale and techniques [661](#)
 - generating
 - example [657](#)
 - overview [651](#)
- XML PARSE statement
 - NOT ON EXCEPTION [614](#)
 - ON EXCEPTION [614](#)
 - overview [612](#)
 - using [613](#)
- XML parser
 - error handling [635](#)
 - overview [612](#)
- XML parsing
 - control flow with processing procedure [618](#)
 - description [613](#)
 - fatal errors [635](#)
 - handling encoding conflicts [635](#), [636](#)
 - handling exceptions [633](#)
 - one segment at a time
 - example [646](#)
 - overview [625](#)
 - overview [611](#)
 - reason code [633](#), [795](#)
 - return code [633](#), [795](#)
 - special registers [615](#), [617](#)
 - terminating [637](#)
 - with validation
 - example [648](#)
 - overview [622](#)
 - performance considerations [623](#)
 - restrictions [623](#)
- XML processing procedure
 - control flow with parser [618](#)
 - error with EXIT PROGRAM or GOBACK [616](#)
 - example
 - one segment at a time [646](#)
 - parsing with validation [648](#)
 - program for processing XML [638](#)
 - handling encoding conflicts [636](#)
 - handling parsing exceptions [633](#)
 - multiple segments [625](#)
 - restriction on XML PARSE [616](#)
 - setting XML-CODE in [636](#)
 - specifying [613](#)
 - using special registers [615](#), [617](#)
 - writing [615](#)
- XML schemas [624](#)
- XML-CODE special register
 - content [618](#)
 - continuation after nonzero value [636](#)
 - control flow between parser and processing procedure [618](#)
 - description [615](#)
 - exception codes for generating [803](#)
 - exception codes for parsing [795](#)
 - exception codes for parsing with XMLPARSE(COMPAT)
 - encoding conflicts [634](#)
 - handleable [797](#)
 - not handleable [800](#)
 - exception codes for parsing with XMLPARSE(XMLSS) [795](#)
- XML-CODE special register (*continued*)
 - fatal errors [635](#)
 - reason code [633](#), [795](#)
 - return code [633](#), [795](#)
 - setting to -1 [618](#), [637](#)
 - setting to 1 [625](#)
 - subtracting 100,000 from [636](#)
 - terminating parsing [637](#)
 - using in generating [655](#)
 - using in parsing [611](#)
 - with code-page conflicts [635](#)
 - with encoding conflicts [636](#)
 - with generating exceptions [656](#)
 - with parsing exceptions [635](#)
- XML-EVENT special register
 - content [617](#), [637](#)
 - description [615](#)
 - using [611](#), [615](#)
 - with parsing exceptions [635](#)
- XML-INFORMATION special register
 - content [620](#)
 - description [615](#)
- XML-NAMESPACE special register
 - content [621](#)
 - description [616](#)
 - using [611](#)
- XML-NAMESPACE-PREFIX special register
 - content [621](#)
 - description [616](#)
 - using [611](#)
- XML-NNAMESPACE special register
 - content [621](#)
 - description [616](#)
 - using [611](#)
- XML-NNAMESPACE-PREFIX special register
 - content [621](#)
 - description [616](#)
 - using [611](#)
- XML-NTEXT special register
 - content [620](#)
 - description [615](#)
 - using [611](#)
 - with parsing exceptions [635](#)
- XML-TEXT special register
 - content [620](#), [637](#)
 - description [615](#)
 - using [611](#)
 - with parsing exceptions [635](#)
- XMLPARSE compiler option
 - choosing the parser [611](#)
 - description [415](#)
- XPLINK linkage convention in OO applications [298](#)
- XPLINK runtime option
 - not recommended as a default [298](#)
 - setting [298](#)
- XREF compiler option
 - description [416](#)
 - finding copybook data sets [442](#)
 - finding data- and procedure-names [442](#)
 - getting output [443](#)
- XREF output
 - COPY/BASIS cross-references [475](#)
 - data-name cross-references [473](#)

XREF output (*continued*)
program-name cross-references [474](#)

Z

z/OS

accessing main parameters under
example [564](#)
overview [564](#)
compiling under [255](#)
running programs under [564](#)

z/OS Debugger

compiler options for [443](#)
description [433](#)

z/OS UNIX

accessing environment variables
example [516](#)
overview [514](#)
accessing main parameters under
example [518](#)
overview [518](#)
compiler environment variables [283](#)
compiling from script [290](#)
compiling OO applications
example [293](#)
overview [291](#)
compiling under [283](#)
copybook search order [283](#), [288](#), [430](#)
copybooks [430](#)
developing programs [513](#)
execution environments [513](#)
linking OO applications
example [293](#)
overview [292](#)
preparing OO applications
example [293](#)
overview [292](#)
programs must be reentrant [548](#)
restrictions [513](#)
running OO applications
overview [293](#)
XPLINK linkage [298](#)
running programs [513](#)
setting environment variables
example [516](#)
overview [514](#)
sort and merge restriction [221](#)
specifying compiler options [284](#)

z/OS UNIX file system

compiler data sets [257](#)
defining file with environment variable [164](#)
processing files with QSAM [186](#)
reading file with ACCEPT [35](#)
search order for DLLs in [571](#)
writing files with DISPLAY [36](#)

zero suppression

example of BLANK WHEN ZERO clause [45](#)
PICTURE symbol Z [45](#)

zlib

ZONECHECK compiler option [417](#)

zoned decimal data (USAGE DISPLAY)

effect of ZWB on comparison to alphanumeric [421](#)
example [43](#)
format [47](#)

zoned decimal data (USAGE DISPLAY) (*continued*)

sign representation [53](#)

ZONEDATA compiler option [418](#)

ZWB compiler option [421](#)



Product Number: 5655-EC6

SC27-8714-03

