COBOL for AIX

Programming Guide

Version 3.1

SC23-8566-00

COBOL for AIX

Programming Guide

Version 3.1

SC23-8566-00

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 771.

First Edition (January 2009)

This edition applies to Version 3.1 of IBM COBOL for AIX (program number 5724-V62) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

You can order publications online at www.ibm.com/shop/publications/order/, or order by phone or fax. IBM Software Manufacturing Solutions takes publication orders between 8:30 a.m. and 7:00 p.m. Eastern Standard Time (EST). The phone number is (800)879-2755. The fax number is (800)445-9269.

You can also order publications through your IBM representative or the IBM branch office serving your locality.

Contents

Tables . . . xi

Preface . . . xiii

About this document . . . xiii
  How this document will help you . . . xiii
  Abbreviated terms . . . xiii
  How to read syntax diagrams . . . xiv
  How examples are shown . . . xv
  Related information . . . xvi
| Summary of changes . . . xvi
| Version 3.1 (January 2009) . . . xvi
How to send your comments . . . xviii
| Accessibility of COBOL for AIX . . . xix
| Accessibility features . . . xix
| Keyboard navigation . . . xix
| Interface information . . . xix
| Related accessibility information . . . xix
| IBM and accessibility . . . xix

Part 1. Coding your program . . . 1

Chapter 1. Structuring your program . . . 5

Identifying a program . . . 5
  Identifying a program as recursive . . . 6
  Marking a program as callable by containing programs . . . 6
  Setting a program to an initial state . . . 6
  Changing the header of a source listing . . . 6
Describing the computing environment . . . 7
  Example: FILE-CONTROL paragraph . . . 7
  Specifying the collating sequence . . . 8
  Defining symbolic characters . . . 9
  Defining a user-defined class . . . 10
  Identifying files to the operating system . . . 10
Describing the data . . . 11
  Using data in input and output operations . . . 12
  Comparison of WORKING-STORAGE and LOCAL-STORAGE . . . 13
  Using data from another program . . . 15
Processing the data . . . 16
  How logic is divided in the PROCEDURE DIVISION . . . 17
  Declaratives . . . 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 . . . 27
  Examples: initializing data items . . . 28
  Initializing a structure (INITIALIZE) . . . 30
  Assigning values to elementary data items (MOVE) . . . 32
  Assigning values to group data items (MOVE) . 33
  Assigning arithmetic results (MOVE or COMPUTE) . . . 34
  Assigning input from a screen or file (ACCEPT) 34
Displaying values on a screen or in a file (DISPLAY) 35
Using intrinsic functions (built-in functions) . . . 36
Using tables (arrays) and pointers . . . 37

Chapter 3. Working with numbers and arithmetic . . . 39

Defining numeric data . . . 39
Displaying numeric data . . . 41
Controlling how numeric data is stored . . . 42
Formats for numeric data . . . 43
  External decimal (DISPLAY and NATIONAL) items . . . 43
  External floating-point (DISPLAY and NATIONAL) items . . . 44
  Binary (COMP) items . . . 44
  Native binary (COMP-5) items . . . 45
  Packed-decimal (COMP-3) items . . . 45
  Internal floating-point (COMP-1 and COMP-2) items . . . 46
  Examples: numeric data and internal representation . . . 46
Data format conversions . . . 48
  Conversions and precision . . . 49
Sign representation of zoned and packed-decimal data . . . 50
Checking for incompatible data (numeric class test) 51
Performing arithmetic . . . 51
  Using COMPUTE and other arithmetic statements . . . 51
  Using arithmetic expressions . . . 52
  Using numeric intrinsic functions . . . 53
  Examples: numeric intrinsic functions . . . 54
Fixed-point contrasted with floating-point arithmetic 56
  Floating-point evaluations . . . 56
  Fixed-point evaluations . . . 57
  Arithmetic comparisons (relation conditions) . . 57
  Examples: fixed-point and floating-point evaluations . . . 58
Using currency signs . . . 58
  Example: multiple currency signs . . . 59

Chapter 4. Handling tables . . . 61

Defining a table (OCCURS) . . . 61
Nesting tables . . . 63
  Example: subscripting . . . 64
  Example: indexing . . . 64
Referring to an item in a table . . . 64
  Subscripting . . . 65
  Indexing . . . 66

Putting values into a table 67
Loading a table dynamically. 67
Initializing a table (INITIALIZE) 68
Assigning values when you define a table (VALUE) 69
Example: PERFORM and subscripting 71
Example: PERFORM and indexing. 72
Creating variable-length tables (DEPENDING ON) 73
Loading a variable-length table 74
Assigning values to a variable-length table . . . 75
Searching a table 75
Doing a serial search (SEARCH) 76
Doing a binary search (SEARCH ALL) 77
Processing table items using intrinsic functions . . 78
Example: processing tables using intrinsic functions 79

Chapter 5. Selecting and repeating program actions 81

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

Chapter 6. Handling strings. 93

Joining data items (STRING) 93
Example: STRING statement. 94
Splitting data items (UNSTRING) 95
Example: UNSTRING statement 96
Manipulating null-terminated strings 98
Example: null-terminated strings 99
Referring to substrings of data items 99
Reference modifiers 101
Example: arithmetic expressions as reference modifiers 102
Example: intrinsic functions as reference modifiers 102
Tallying and replacing data items (INSPECT). . . 103
Examples: INSPECT statement 103
Converting data items (intrinsic functions) . . . 104
Converting to uppercase or lowercase (UPPER-CASE, LOWER-CASE) 105
Transforming to reverse order (REVERSE) . . . 105
Converting to numbers (NUMVAL, NUMVAL-C) 105
Converting from one code page to another . . 107
Evaluating data items (intrinsic functions) 107
Evaluating single characters for collating sequence 107
Finding the largest or smallest data item . . . 108
Finding the length of data items 110
Finding the date of compilation 111

Chapter 7. Processing files 113

Identifying files. 113
Identifying VSAM files 114
Identifying Encina SFS files. 114
Identifying STL files 115
Identifying RSD files 115
File systems 115
VSAM file system 116
SFS file system 117
STL file system 118
RSD file system 120
| STL and VSAM files that are larger than 2 GB 121
| File-system summary 121
Specifying a file organization and access mode . . 122
File organization and access mode 122
Protecting against errors when opening files . . . 125
Setting up a field for file status 126
Describing the structure of a file in detail 126
Coding input and output statements for files . . . 127
Example: COBOL coding for files. 127
File position indicator 129
Opening a file 129
Reading records from a file. 132
Statements used when writing records to a file 133
Adding records to a file 133
Replacing records in a file 134
Deleting records from a file. 135
PROCEDURE DIVISION statements used to update files 135
Enabling COBOL applications to access SFS files 136
Example: accessing SFS files 137
Improving SFS performance 139

Chapter 8. Sorting and merging files 143

Sort and merge process 143
Describing the sort or merge file 144
Describing the input to sorting or merging . . . 145
Example: describing sort and input files for SORT 145
Coding the input procedure 146
Describing the output from sorting or merging . . 147
Coding the output procedure 147
Restrictions on input and output procedures . . . 148
Requesting the sort or merge 148
Setting sort or merge criteria 149
Choosing alternate collating sequences 150
Example: sorting with input and output procedures 150
Determining whether the sort or merge was successful 151
Sort and merge error numbers. 152
Stopping a sort or merge operation prematurely 154

Chapter 9. Handling errors. 157

Handling errors in joining and splitting strings . . 157
Handling errors in arithmetic operations 158
Example: checking for division by zero 158
Handling errors in input and output operations 158
Using the end-of-file condition (AT END) . . . 160
Coding ERROR declaratives 160
Using file status keys. 161
Using file system status codes 162
Coding INVALID KEY phrases 164

Handling errors when calling programs 165

Part 2. Enabling programs for international environments 167

Chapter 10. Processing data in an international environment 169

COBOL statements and national data 170
Intrinsic functions and national data. 172
Unicode and the encoding of language characters 173
Using national data (Unicode) in COBOL 174
  Defining national data items 174
  Using national literals 175
  Using national-character figurative constants 176
  Defining national numeric data items 177
  National groups 177
  Using national groups 179
  Storage of character data 181
Converting to or from national (Unicode) representation 182
  Converting alphanumeric, DBCS, and integer to national (MOVE) 182
  Converting alphanumeric or DBCS to national (NATIONAL-OF) 183
  Converting national to alphanumeric (DISPLAY-OF) 184
  Overriding the default code page. 184
  Example: converting to and from national data 184
Processing UTF-8 data 185
Processing Chinese GB 18030 data 186
Comparing national (UTF-16) data 186
  Comparing two class national operands . . . 187
  Comparing class national and class numeric operands 188
  Comparing national numeric and other numeric operands 188
  Comparing national and other character-string operands 189
  Comparing national data and alphanumeric-group operands. 189
Coding for use of DBCS support 189
  Declaring DBCS data 190
  Using DBCS literals 190
  Testing for valid DBCS characters 191
  Processing alphanumeric data items that contain DBCS data 192

Chapter 11. Setting the locale 193

The active locale 193
Specifying the code page for character data . . . 194
Using environment variables to specify a locale 195
  Determination of the locale from system settings 196
  Types of messages for which translations are available 197
Locales and code pages that are supported . . . 197
Controlling the collating sequence with a locale 199
  Controlling the alphanumeric collating sequence with a locale. 200
  Controlling the DBCS collating sequence with a locale 201
  Controlling the national collating sequence with a locale 202
  Intrinsic functions that depend on collating sequence 202
Accessing the active locale and code-page values 203
  Example: get and convert a code-page ID . . . 204

Part 3. Compiling, linking, running, and debugging your program . . . 207

Chapter 12. Compiling, linking, and running programs 209

Setting environment variables 209
  Compiler environment variables 210
  Runtime environment variables 211
  Example: setting and accessing environment variables 217
Compiling programs 218
  Compiling from the command line 219
  Compiling using shell scripts 220
  Compiling for SFS files 220
  Specifying compiler options with the PROCESS (CBL) statement 221
  Modifying the default compiler configuration 221
Correcting errors in your source program 224
  Severity codes for compiler error messages . . 225
  Generating a list of compiler error messages . . 225
cob2, cob2_j, and cob2_r options 227
  Options that apply to compiling 227
  Options that apply to linking 228
  Options that apply to both compiling and linking 228
Linking programs 229
  Passing options to the linker 230
  Linker input and output files 232
Correcting errors in linking. 234
Running programs 234
Redistributing COBOL for AIX library components 235

Chapter 13. Compiling, linking, and running OO applications 237

Compiling OO applications. 237
Preparing OO applications 238
Example: compiling and linking a COBOL class definition. 239
Running OO applications 239
  Running OO applications that start with a main method 240
  Running OO applications that start with a COBOL program 240
| Java SDK support 241

Chapter 14. Compiler options 243

| Conflicting compiler options 245
ADATA 246
| ADDR. 246
ARITH 248

BINARY 249
CALLINT 249
CHAR. 250
| CICS 252
COLLSEQ 253
COMPILE 255
CURRENCY. 255
DATEPROC 256
DIAGTRUNC 257
DYNAM 258
ENTRYINT 259
EXIT 259
  Character string formats. 261
  User-exit work area 261
  Parameter list for exit modules 261
  Using INEXIT 262
  Using LIBEXIT 262
  Using PRTEXIT. 263
  Using ADEXIT 264
FLAG 264
FLAGSTD 265
FLOAT 267
LIB 268
LINECOUNT 268
LIST 269
LSTFILE 269
MAP 270
MAXMEM 271
| MDECK 272
NCOLLSEQ 273
NSYMBOL 273
NUMBER 274
OPTIMIZE 275
PGMNAME 276
  PGMNAME(UPPER) 277
  PGMNAME(MIXED) 277
PROFILE 277
QUOTE/APOST 278
SEPOBJ 278
  Batch compilation 279
SEQUENCE 280
SIZE 280
SOSI 281
SOURCE 282
SPACE 283
SPILL 284
SQL 284
SSRANGE 285
TERMINAL 286
TEST 286
THREAD. 287
TRUNC 288
  TRUNC example 1 289
  TRUNC example 2 290
VBREF 290
WSCLEAR 291
XREF 291
YEARWINDOW 292
ZWB 293

Chapter 15. Compiler-directing statements 295

Chapter 16. Runtime options. 301
CHECK 301
DEBUG 302
ERRCOUNT. 302
| FILEMODE 303
FILESYS 303
TRAP 304
UPSI 305

Chapter 17. Debugging 307
Debugging with source language. 307
  Tracing program logic 307
  Finding and handling input-output errors . . . 308
  Validating data 309
  Finding uninitialized data 309
  Generating information about procedures . . . 309
Debugging using compiler options 311
  Finding coding errors 312
  Finding line sequence problems 312
  Checking for valid ranges 312
  Selecting the level of error to be diagnosed . . 313
  Finding program entity definitions and references 315
  Listing data items 316
Using the debugger 316
Getting listings 316
  Example: short listing 318
  Example: SOURCE and NUMBER output . . . 320
  Example: MAP output 320
  Example: XREF output: data-name cross-references. 324
  Example: VBREF compiler output 326
Debugging with messages that have offset information 327
Debugging user exits 328
Debugging assembler routines. 328

Part 4. Accessing databases 329

Chapter 18. Programming for a DB2 environment 331
DB2 coprocessor 331
Coding SQL statements 332
  Using SQL INCLUDE with the DB2 coprocessor 332
  Using binary items in SQL statements 333
  Determining the success of SQL statements . . 333
Starting DB2 before compiling. 333
Compiling with the SQL option 333
  Separating DB2 suboptions 334
  Using package and bind file-names 334

Chapter 19. Developing COBOL programs for CICS 337
Coding COBOL programs to run under CICS . . 337
  Getting the system date under CICS. 338
  Making dynamic calls under CICS 339

Accessing SFS data 340
Calling between COBOL and C/C++ under CICS 341
Compiling and running CICS programs 341
| Integrated CICS translator 342
Debugging CICS programs 342

Chapter 20. Open Database Connectivity (ODBC) 345
Comparison of ODBC and embedded SQL . . . 345
Background 346
Installing and configuring software for ODBC . . 346
Coding ODBC calls from COBOL: overview . . . 346
Using data types appropriate for ODBC . . . 346
Passing pointers as arguments in ODBC calls 347
Accessing function return values in ODBC calls 349
Testing bits in ODBC calls 349
Using COBOL copybooks for ODBC APIs 350
Example: sample program using ODBC copybooks 351
Example: copybook for ODBC procedures . . . 352
Example: copybook for ODBC data definitions 355
ODBC names truncated or abbreviated for COBOL 355
Compiling and linking programs that make ODBC calls 357

Part 5. Using XML and COBOL together 359

Chapter 21. Processing XML input 361
XML parser in COBOL 361
Accessing XML documents 363
Parsing XML documents 363
Writing procedures to process XML 365
| XML-EVENT 366
XML-CODE 366
XML-TEXT and XML-NTEXT 367
Transforming XML text to COBOL data items 367
Example: program for processing XML 368
The encoding of XML documents. 371
XML input document encoding 371
Specifying the code page 372
| EBCDIC code-page-sensitive characters in XML
| markup 373
| Parsing XML documents encoded in UTF-8 . . . 374
Handling XML PARSE exceptions 374
How the XML parser handles errors. 376
Handling conflicts in code pages 376
Terminating XML parsing 377

Chapter 22. Producing XML output 379
Generating XML output 379
Controlling the encoding of generated XML output 383
Handling XML GENERATE exceptions 384
Example: generating XML 385
Program XGFX 385
Program Pretty 386
Output from program XGFX 389
Enhancing XML output 389
Example: enhancing XML output 390
Example: converting hyphens in element or attribute names to underscores 393

Part 6. Developing object-oriented programs 395

Chapter 23. Writing object-oriented programs 397
Example: accounts. 398
Subclasses 399
Defining a class 400
CLASS-ID paragraph for defining a class . . . 402
REPOSITORY paragraph for defining a class 402
WORKING-STORAGE SECTION for defining class instance data. 404
Example: defining a class 405
Defining a class instance method 405
METHOD-ID paragraph for defining a class instance method 406
INPUT-OUTPUT SECTION for defining a class instance method 407
DATA DIVISION for defining a class instance method 407
PROCEDURE DIVISION for defining a class instance method 408
Overriding an instance method 409
Overloading an instance method 410
Coding attribute (get and set) methods 411
Example: defining a method 412
Defining a client 414
REPOSITORY paragraph for defining a client 415
DATA DIVISION for defining a client 416
Comparing and setting object references . . . 417
Invoking methods (INVOKE) 418
Creating and initializing instances of classes . . 422
Freeing instances of classes 424
Example: defining a client 424
Defining a subclass 425
CLASS-ID paragraph for defining a subclass 426
REPOSITORY paragraph for defining a subclass 427
WORKING-STORAGE SECTION for defining subclass instance data 427
Defining a subclass instance method 428
Example: defining a subclass (with methods) 428
Defining a factory section 429
WORKING-STORAGE SECTION for defining factory data 430
Defining a factory method 431
Example: defining a factory (with methods) . . 433
Wrapping procedure-oriented COBOL programs 438
Structuring OO applications 439
Examples: COBOL applications that run using the java command. 439

Chapter 24. Communicating with Java methods 443
Accessing JNI services 443

Handling Java exceptions 445
Managing local and global references 446
Java access controls 447
Sharing data with Java 448
Coding interoperable data types in COBOL and Java 448
Declaring arrays and strings for Java 449
Manipulating Java arrays 450
Manipulating Java strings 453

Part 7. Working with more complex applications 455

Chapter 25. Porting applications between platforms 457

Getting mainframe applications to compile . . . 457
Getting mainframe applications to run: overview 458
Fixing differences caused by data representations 458
Fixing environment differences that affect portability 460
Fixing differences caused by language elements 461
Writing code to run on the mainframe 462
Writing applications that are portable between AIX and Windows 462

Chapter 26. Using subprograms . . . 465

Main programs, subprograms, and calls 465
Ending and reentering main programs or subprograms 465
Calling nested COBOL programs 466
Nested programs 467
Example: structure of nested programs 468
Scope of names. 469
Calling nonnested COBOL programs 470
CALL identifier and CALL literal. 470
Calling between COBOL and C/C++ programs . . 471
Initializing environments 472
Passing data between COBOL and C/C++ . . 472
Collapsing stack frames and terminating run units or processes 473
COBOL and C/C++ data types 473
Example: COBOL program calling C functions 474
Example: C programs that are called by and call COBOL 475
Example: COBOL program called by a C program 476
Example: results of compiling and running examples 476
| Example: COBOL program calling C++ function 477
Making recursive calls 478
Passing return codes 478

Chapter 27. Sharing data 481

Passing data. 481
Describing arguments in the calling program 483
Describing parameters in the called program 483
Testing for OMITTED arguments 484
Coding the LINKAGE SECTION 484
Coding the PROCEDURE DIVISION for passing arguments 485
Grouping data to be passed 485
Handling null-terminated strings 486
Using pointers to process a chained list . . . 486
Using procedure and function pointers 489
Passing return-code information 490
Understanding the RETURN-CODE special register 490
Using PROCEDURE DIVISION RETURNING 490
Specifying CALL . . . RETURNING 490
Sharing data by using the EXTERNAL clause. . . 491
Sharing files between programs (external files) . . 491
Example: using external files 492
Using command-line arguments 495
Example: command-line arguments without -host option 495
Example: command-line arguments with -host option 496

Chapter 28. Building shared libraries 497

Static linking and shared libraries 497
How the linker resolves references to shared libraries 498
Example: creating a sample shared library. . . 498
Example: creating a makefile for the sample shared library 501
Example: calling shared libraries using CALL identifier 501
Example: COBOL program calling COBOL and C shared libraries 503

Chapter 29. Preparing COBOL programs for multithreading 505

Multithreading 505
Working with language elements with multithreading 506
Working with elements that have run-unit scope 507
Working with elements that have program invocation instance scope 507
Scope of COBOL language elements with multithreading 508
Choosing THREAD to support multithreading . . 508
Transferring control to multithreaded programs 509
Ending multithreaded programs 509
Handling COBOL limitations with multithreading 510
Example: using COBOL in a multithreaded environment. 510
Source code for thrcob.c 511
Source code for subd.cbl. 512
Source code for sube.cbl. 512

Chapter 30. Preinitializing the COBOL runtime environment 513

Initializing persistent COBOL environment . . . 513
Terminating preinitialized COBOL environment 514
Example: preinitializing the COBOL environment 515

Chapter 31. Processing two-digit-year dates 519
Millennium language extensions (MLE) 520
Principles and objectives of these extensions . . 520
Resolving date-related logic problems 521
Using a century window 522
Using internal bridging 523
Moving to full field expansion. 525
Using year-first, year-only, and year-last date fields 527
Compatible dates 527
Example: comparing year-first date fields . . . 528
Using other date formats 528
Example: isolating the year 529
Manipulating literals as dates 529
Assumed century window 530
Treatment of nondates 531
Using sign conditions 532
Performing arithmetic on date fields. 533
Allowing for overflow from windowed date fields 533
Specifying the order of evaluation 534
Controlling date processing explicitly 535
Using DATEVAL 535
Using UNDATE 536
Example: DATEVAL 536
Example: UNDATE 536
Analyzing and avoiding date-related diagnostic messages 537
Avoiding problems in processing dates 538
Avoiding problems with packed-decimal fields 539
Moving from expanded to windowed date fields 539

Part 8. Improving performance and productivity 541

Chapter 32. Tuning your program . . . 543
Using an optimal programming style 544
Using structured programming 544
Factoring expressions. 544
Using symbolic constants 545
Grouping constant computations 545
Grouping duplicate computations 545
Choosing efficient data types 546
Choosing efficient computational data items . . 546
Using consistent data types. 546
Making arithmetic expressions efficient. . . . 547
Making exponentiations efficient 547
Handling tables efficiently 547
Optimization of table references 549
Optimizing your code 550
Optimization 551
Choosing compiler features to enhance performance. 552
Performance-related compiler options 552
Evaluating performance 554

Chapter 33. Simplifying coding 557
Eliminating repetitive coding 557
Example: using the COPY statement. 558
Manipulating dates and times 559
Getting feedback from date and time callable services 559
Handling conditions from date and time callable services 560
Example: manipulating dates 560
Example: formatting dates for output 560
Feedback token. 561
Picture character terms and strings 562
Example: date-and-time picture strings 564
Century window 565

Part 9. Appendixes 567

| **Appendix A. Migration and compatibility. 569**
| Object code 569
| Object-oriented syntax, and Java 5 or Java 6 SDKs 569
| Data size limits. 569
| XML generation 570
| SFS files 570
| Migration of 32-bit applications to 64-bit mode . . 570

Appendix B. Summary of differences with host COBOL. 573
Compiler options 573
Data representation 573
Zoned decimal data 574
Packed-decimal data 574
Display floating-point data 574
National data 574
EBCDIC and ASCII data. 574
Code-page determination for data conversion 574
DBCS character strings 575
Environment variables 575
File specification 576
Interlanguage communication (ILC) 576
Input and output 576
Runtime options 576
Source code line size 577
Language elements 577

Appendix C. zSeries host data format considerations. 581
CICS access 581
Date and time callable services 581
Floating-point overflow exceptions 581
DB2 582
Object-oriented syntax for Java interoperability . . 582
Distributed Computing Environment applications 582
File data 582
SORT 582

Appendix D. Intermediate results and arithmetic precision 583
Terminology used for intermediate results. . . . 584
Example: calculation of intermediate results . . . 585
Fixed-point data and intermediate results 585

Addition, subtraction, multiplication, and division 585
Exponentiation 586
Example: exponentiation in fixed-point arithmetic 587
Truncated intermediate results. 588
Binary data and intermediate results 588
Intrinsic functions evaluated in fixed-point arithmetic 588
Integer functions 588
Mixed functions 589
Floating-point data and intermediate results . . . 590
Exponentiations evaluated in floating-point arithmetic 591
Intrinsic functions evaluated in floating-point arithmetic 591
Arithmetic expressions in nonarithmetic statements 591

Appendix E. Complex OCCURS DEPENDING ON 593
Example: complex ODO 593
How length is calculated 594
Setting values of ODO objects 594
Effects of change in ODO object value 594
Preventing index errors when changing ODO object value 595
Preventing overlay when adding elements to a variable table 595

Appendix F. Date and time callable services. 599
CEECBLDY: convert date to COBOL integer format 600
CEEDATE: convert Lilian date to character format 604
CEEDATM: convert seconds to character timestamp 608
CEEDAYS: convert date to Lilian format 611
CEEDYWK: calculate day of week from Lilian date 615
CEEGMT: get current Greenwich Mean Time . . . 617
CEEGMTO: get offset from Greenwich Mean Time to local time 619
CEEISEC: convert integers to seconds 621
CEELOCT: get current local date or time 623
CEEQCEN: query the century window 625
CEESCEN: set the century window 627
CEESECI: convert seconds to integers 629
CEESECS: convert timestamp to seconds 632
CEEUTC: get coordinated universal time 636
IGZEDT4: get current date 636

Appendix G. XML reference material 639
XML PARSE exceptions that allow continuation 639
XML PARSE exceptions that do not allow continuation 643
XML conformance 646
XML GENERATE exceptions 649

Appendix H. JNI.cpy 651

Appendix I. COBOL SYSADATA file contents 657
Existing compiler options that affect the SYSADATA file. 657
SYSADATA record types 658
Example: SYSADATA 659
SYSADATA record descriptions 660
Common header section. 661
Job identification record: X′0000′ 662
ADATA identification record: X′0001′ 663
Compilation unit start|end record: X′0002′ . . . 663
Options record: X′0010′ 664
External symbol record: X′0020′ 673
Parse tree record: X′0024′ 674
Token record: X′0030′. 689
Source error record: X′0032′ 702
Source record: X′0038′ 703
COPY REPLACING record: X′0039′ 704
Symbol record: X′0042′ 704
Symbol cross-reference record: X′0044′ 717
Nested program record: X′0046′ 718
Library record: X′0060′ 719
Statistics record: X′0090′. 719
EVENTS record: X′0120′. 720

Appendix J. Runtime messages . . . 725

Notices 771
Trademarks 773

Glossary 775

List of resources 807
COBOL for AIX 807
Related publications 807

Index 809

Tables

1. FILE SECTION entries 13
2. Assignment to data items in a program 27
3. Ranges in value of COMP-5 data items 45
4. Internal representation of binary numeric items 46
5. Internal representation of native numeric items 47
6. Internal representation of numeric items when CHAR(EBCDIC) and FLOAT(HEX) are in effect 48
7. Order of evaluation of arithmetic operators 52
8. Numeric intrinsic functions 53
9. STL file system return codes 118
10. STL file system adapter open routine return codes 120
11. File-system summary 122
12. File organization and access mode 123
13. Valid COBOL statements for sequential files 130
14. Valid COBOL statements for line-sequential files 131
15. Valid COBOL statements for indexed and relative files 131
16. Statements used when writing records to a file 133
17. PROCEDURE DIVISION statements used to update files 135
18. Sort and merge error numbers 152
19. COBOL statements and national data 170
20. Intrinsic functions and national character data. 172
21. National group items that are processed with group semantics 181
22. Encoding and size of alphanumeric, DBCS, and national data 181
23. Supported locales and code pages 197
24. Intrinsic functions that depend on collating sequence 202
25. TZ environment parameter variables 217
26. Output from the cob2 command 220
27. Examples of compiler-option syntax in a shell script 220
28. Stanza attributes 223
29. Severity codes for compiler error messages 225
30. Common linker options 230
31. Default file-names assumed by the linker 233
32. Commands for compiling and linking a class definition 238
33. java command options for customizing the JVM 240
34. Compiler options 243
35. Mutually exclusive compiler options 245
36. Effect of comparand data type and collating sequence on comparisons 254
37. Parameter list for exit modules 261
38. Runtime options 301
39. Severity levels of compiler messages 313
40. Using compiler options to get listings 317
41. Terms and symbols used in MAP output 322
42. ODBC C types and corresponding COBOL declarations 346
43. ODBC copybooks 350
44. ODBC names truncated or abbreviated for COBOL 355
45. Special registers used by the XML parser 365
46. Hexadecimal values of special characters for various EBCDIC CCSIDs 373
47. Encoding of generated XML if the ENCODING phrase is omitted 383
48. Structure of class definitions 400
49. Structure of instance method definitions 406
50. Structure of COBOL clients 414
51. Conformance of arguments in a COBOL client 419
52. Conformance of the returned data item in a COBOL client 421
53. Structure of factory definitions 430
54. Structure of factory method definitions 431
55. JNI services for local and global references 447
56. Interoperable data types in COBOL and Java 448
57. Interoperable arrays and strings in COBOL and Java 449
58. Noninteroperable array types in COBOL and Java 450
59. JNI array services 451
60. Services that convert between jstring references and national data 453
61. Services that convert between jstring references and UTF-8 data 453
62. ASCII characters contrasted with EBCDIC 459
63. ASCII comparisons contrasted with EBCDIC 459
64. IEEE contrasted with hexadecimal 459
65. COBOL and C/C++ data types 473
66. Methods for passing data in the CALL statement 482
67. Scope of COBOL language elements with multithreading 508
68. Advantages and disadvantages of Year 2000 solutions 522
69. Performance-related compiler options 552
70. Performance-tuning worksheet 555
71. Picture character terms and strings 563
72. Japanese Eras 564
73. Examples of date-and-time picture strings 564
74. Language differences between Enterprise COBOL for z/OS and COBOL for AIX . . . 577
75. Maximum floating-point values 581
76. Date and time callable services 599
77. Date and time intrinsic functions 600
78. CEECBLDY symbolic conditions 602
79. CEEDATE symbolic conditions 605
80. CEEDATM symbolic conditions 608
81. CEEDAYS symbolic conditions 613

82. **CEEDYWK symbolic conditions** 615
83. **CEEGMT symbolic conditions** 618
84. **CEEGMTO symbolic conditions** 620
85. **CEEISEC symbolic conditions**. 622
86. **CEELOCT symbolic conditions** 624
87. **CEEQCEN symbolic conditions** 626
88. **CEESCEN symbolic conditions** 628
89. **CEESECI symbolic conditions**. 630
90. **CEESECS symbolic conditions** 633
91. **XML PARSE exceptions that allow continuation**. 639
92. **XML PARSE exceptions that do not allow continuation**. 644
93. **XML GENERATE exceptions** 649
94. **SYSADATA record types** 658
95. **SYSADATA common header section** 661
96. **SYSADATA job identification record** 662
97. **ADATA identification record** 663
98. **SYSADATA compilation unit start|end record** 663
99. **SYSADATA options record**. 664
100. **SYSADATA external symbol record** 674
101. **SYSADATA parse tree record** 674
102. **SYSADATA token record** 689
103. **SYSADATA source error record** 703
104. **SYSADATA source record** 703
105. **SYSADATA COPY REPLACING record** 704
106. **SYSADATA symbol record**. 704
107. **SYSADATA symbol cross-reference record** 717
108. **SYSADATA nested program record** 718
109. **SYSADATA library record** 719
110. **SYSADATA statistics record** 719
111. **SYSADATA EVENTS TIMESTAMP record layout** 720
112. **SYSADATA EVENTS PROCESSOR record layout** 720
113. **SYSADATA EVENTS FILE END record layout** 721
114. **SYSADATA EVENTS PROGRAM record layout** 721
115. **SYSADATA EVENTS FILE ID record layout** 721
116. **SYSADATA EVENTS ERROR record layout** 722
117. **Runtime messages**. 725

Preface

About this document

Welcome to IBM® COBOL for AIX®, IBM's COBOL compiler and run time for AIX!

There are some differences between host and workstation COBOL. For details on language and system differences between COBOL for AIX and Enterprise COBOL for z/OS®, see Appendix B, "Summary of differences with host COBOL," on page 573.

How this document will help you

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

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

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

Abbreviated terms

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

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

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

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

- ANSI INCITS 23b-1993, Programming Language - Correction Amendment for COBOL

The ISO standards are identical to the American National Standards.

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

How to read syntax diagrams

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

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

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

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

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

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

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

```
►►──required_item───────────────────────────────────────────────────────►◄
```

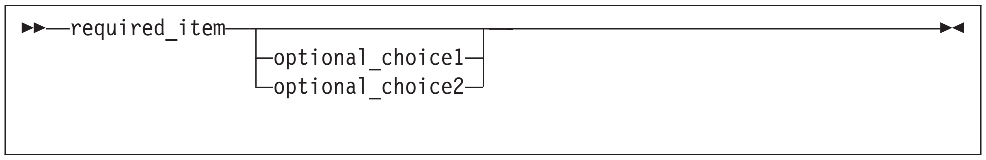
- Optional items appear below the main path.

```
►►──required_item──┬───────────────┬────────────────────────────────────►◄
                   └─optional_item─┘
```

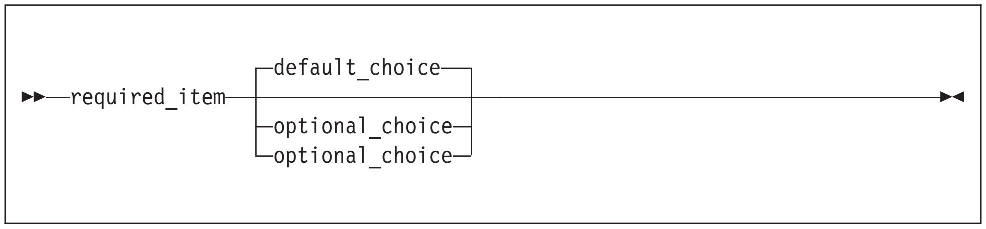
- If you can choose from two or more items, they appear vertically, in a stack. If you must choose one of the items, one item of the stack appears on the main path.

```
►►──required_item──┬─required_choice1─┬─────────────────────────────────►◄
                   └─required_choice2─┘
```

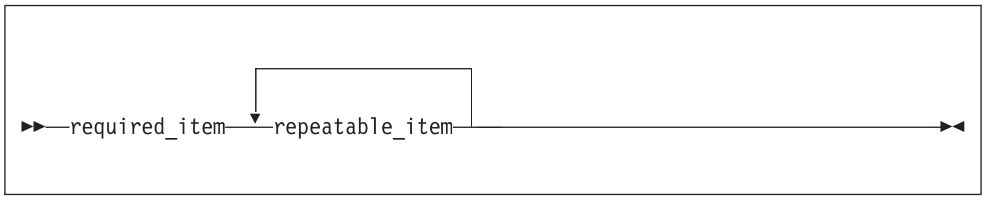
  If choosing one of the items is optional, the entire stack appears below the main path.



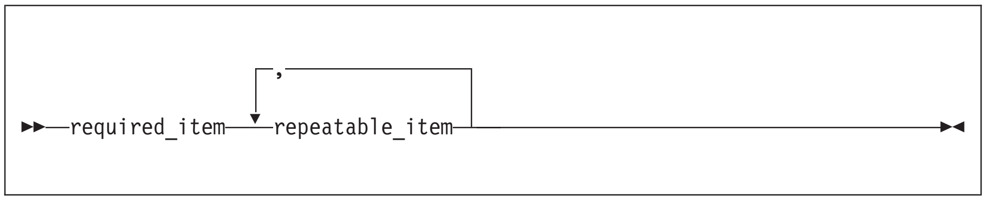
If one of the items is the default, it appears above the main path and the remaining choices are shown below.



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



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



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

How examples are shown

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

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

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

Related information

The information in this Programming Guide is available online in the IBM AIX Compiler Information Center at publib.boulder.ibm.com/infocenter/comphelp/index.jsp. The Information Center also has all the COBOL Language Reference information and the information for using the debugger. You might also want to check the list of resources online or in this document.

Summary of changes

This section lists the key changes that have been made to COBOL for AIX in Version 3. The changes that are described in this document have an associated cross-reference for your convenience. The changes with respect to COBOL for AIX Version 2 (both technical changes and editorial changes to terminology) are marked with a vertical bar (|) in the left margin in the PDF version.

Version 3.1 (January 2009)

- You can now create 64-bit applications. Features have been added in support of 64-bit exploitation:
  - A new compiler option, `ADDR`, lets you indicate whether 32-bit object programs or 64-bit object programs will be generated ("ADDR" on page 246).
  - Several limits on COBOL data-item size have been significantly raised. For example, the maximum size of the `LOCAL-STORAGE`, `WORKING-STORAGE`, and `LINKAGE` sections is essentially unlimited (up to the addressing capacity of the machine) if `ADDR(64)` is in effect.

    This support facilitates programming with large amounts of data, for example:
    - DB2/COBOL applications that use DB2 BLOB and CLOB data types
    - COBOL XML applications that parse or generate large XML documents

    For full details about increased compiler limits, see Compiler limits (*COBOL for AIX Language Reference*).
- Support for files that are larger than 2 GB: STL and VSAM files created by Version 3 programs have by default a new large-format metadata representation (internal control information) that no longer limits them to a 2 GB maximum size. These files can also be processed by Version 2 programs that are run with the Version 3 runtime library ("STL and VSAM files that are larger than 2 GB" on page 121).
- A new runtime option, `FILEMODE`, lets you specify whether newly created STL and VSAM files will have the new large format, or have the small-format metadata representation and 2 GB maximum size that is compatible with STL and VSAM files created by Version 2 programs ("FILEMODE" on page 303).
- Support for national (Unicode UTF-16) data has been enhanced. Several additional kinds of data items can now be described implicitly or explicitly as `USAGE NATIONAL`:
  - External decimal (*national decimal*) items ("Defining numeric data" on page 39)
  - External floating-point (*national floating-point*) items ("Formats for numeric data" on page 43)

- Numeric-edited items ("Displaying numeric data" on page 41)
- National-edited items ("Using national data (Unicode) in COBOL" on page 174)
- Group (*national group*) items, supported by the GROUP-USAGE NATIONAL clause ("Using national groups" on page 179)

- Many COBOL language elements support the new kinds of UTF-16 data, or newly support the processing of national data:
  - Numeric data with USAGE NATIONAL (national decimal and national floating point) can be used in arithmetic operations and in any language constructs that support numeric operands ("Formats for numeric data" on page 43).
  - Edited data with USAGE NATIONAL is supported in the same language constructs as any existing edited type, including editing and de-editing operations associated with moves ("Displaying numeric data" on page 41 and "Using national data (Unicode) in COBOL" on page 174).
  - Group items that contain all national data can be defined with the GROUP-USAGE NATIONAL clause, which results in the group behaving as an elementary item in most language constructs. This support facilitates use of national groups in statements such as STRING, UNSTRING, and INSPECT ("National groups" on page 177).
  - The XML GENERATE statement supports national groups as receiving data items, and national-edited, numeric-edited of USAGE NATIONAL, national decimal, national floating-point, and national group items as sending data items ("Generating XML output" on page 379).
  - The NUMVAL and NUMVAL-C intrinsic functions can take a national literal or national data item as an argument ("Converting to numbers (NUMVAL, NUMVAL-C)" on page 105).

  Using these new national data capabilities, it is now practical to develop COBOL programs that exclusively use Unicode for all application data.
- The XML GENERATE statement has been enhanced (Chapter 22, "Producing XML output," on page 379):
  - You can specify a namespace using the NAMESPACE phrase, and a namespace prefix to be applied to each element using the NAMESPACE-PREFIX phrase.
  - You can specify the encoding of the generated document using the ENCODING phrase ("Controlling the encoding of generated XML output" on page 383).
  - XML documents can now be generated in UTF-8 as well as in UTF-16 or various ASCII or EBCDIC code pages ("The encoding of XML documents" on page 371).
  - The WITH ATTRIBUTES phrase causes eligible elementary items to be expressed as attributes rather than as child elements in the generated XML.
  - The WITH XML-DECLARATION phrase causes an XML declaration to be generated.
- When you use the XML PARSE statement, you can now parse documents that are encoded in UTF-8 ("Parsing XML documents encoded in UTF-8" on page 374) as well as documents that are encoded in UTF-16 or in various ASCII or EBCDIC code pages ("The encoding of XML documents" on page 371).
- Compiler listings now cross-reference copybooks to the library-names and file names from which the copybooks are obtained ("Example: XREF output: COPY/BASIS cross-references" on page 325).
- Support for new features of DB2 V9 is enabled when you use the integrated DB2 coprocessor (SQL compiler option).

- Usability enhancements to COBOL-DB2 applications are available when you use the integrated DB2 coprocessor, for example, an explicitly coded `LOCAL-STORAGE SECTION` or `WORKING-STORAGE SECTION` is no longer required.
- Support for compilation of programs and copybooks that contain CICS statements, without the need for a separate translation step, has been added ("Integrated CICS translator" on page 342).
- A new compiler option, `CICS`, enables integrated CICS translation and specification of CICS options ("CICS" on page 252).
- Debugging has been enhanced to support IBM Debugger for AIX.
- A new callable service, iwzGetSortErrno, makes it possible to obtain the sort or merge error number after each sort or merge operation ("Determining whether the sort or merge was successful" on page 151).
- The `REDEFINES` clause has been enhanced such that for data items that are not level 01, the subject of the entry can be larger than the data item being redefined.
- A new compiler option, `MDECK`, causes the output from library-processing statements to be written to a file ("MDECK" on page 272).
- The literal in a `VALUE` clause for a data item of class national can be alphanumeric ("Initializing a table at the group level" on page 70).

The following terminology changes were also made in this release:

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

How to send your comments

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

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

  IBM Corporation
  Reader Comments
  DTX/E269
  555 Bailey Avenue
  San Jose, CA 95141-1003
  USA
- Use the Online Readers' Comments Form at www.ibm.com/software/awdtools/rcf/.
- Send your comments to the following e-mail address: comments@us.ibm.com.

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

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

Accessibility of COBOL for AIX

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

Accessibility features

The major accessibility features of COBOL for AIX are:

- Interfaces that are commonly used by screen readers
- An accessible command-line interface through a direct attached or networked remote Windows® or Linux® system using AT® and terminal emulation software

The AIX Compiler Information Center and its related publications are accessibility enabled. The accessibility features of the information center are described in the IBM AIX Compiler Information Center at www.ibm.com/software/awdtools/cobol/aix/library.

Keyboard navigation

For information about accessing AIX interfaces, refer to the IBM AIX Compiler Information Center at www.ibm.com/software/awdtools/cobol/aix/library.

Interface information

Assistive technology products work with the user interfaces that are found in AIX. For specific guidance information, consult the documentation for the assistive technology product that you use to access AIX interfaces.

Related accessibility information

The English-language XHTML format of this document that will be provided in the IBM AIX Compiler Information Center at www.ibm.com/software/awdtools/cobol/aix/library is accessible to visually impaired individuals who use a screen reader.

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

IBM and accessibility

See the IBM Human Ability and Accessibility Center at www.ibm.com/able for more information about the commitment that IBM has to accessibility.

Part 1. Coding your program

Chapter 1. Structuring your program 5
Identifying a program 5
  Identifying a program as recursive 6
  Marking a program as callable by containing programs 6
  Setting a program to an initial state. 6
  Changing the header of a source listing 6
Describing the computing environment 7
  Example: FILE-CONTROL paragraph 7
  Specifying the collating sequence 8
    Example: specifying the collating sequence . . 9
  Defining symbolic characters 9
  Defining a user-defined class 10
  Identifying files to the operating system 10
    Varying the input or output file at run time . 11
Describing the data 11
  Using data in input and output operations . . . 12
    FILE SECTION entries. 13
  Comparison of WORKING-STORAGE and LOCAL-STORAGE 13
    Example: storage sections. 14
  Using data from another program 15
    Sharing data in separately compiled programs 15
    Sharing data in nested programs 16
    Sharing data in recursive or multithreaded programs 16
Processing the data 16
  How logic is divided in the PROCEDURE DIVISION 17
    Imperative statements 18
    Conditional statements 18
    Compiler-directing statements 19
    Scope terminators 19
  Declaratives 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 27
  Examples: initializing data items 28
  Initializing a structure (INITIALIZE) 30
  Assigning values to elementary data items (MOVE) 32
  Assigning values to group data items (MOVE) . 33
  Assigning arithmetic results (MOVE or COMPUTE) 34
  Assigning input from a screen or file (ACCEPT) 34
Displaying values on a screen or in a file (DISPLAY) 35
Using intrinsic functions (built-in functions) . . . 36
Using tables (arrays) and pointers 37

Chapter 3. Working with numbers and arithmetic 39
Defining numeric data. 39
Displaying numeric data 41
Controlling how numeric data is stored 42
Formats for numeric data. 43
  External decimal (DISPLAY and NATIONAL) items 43
  External floating-point (DISPLAY and NATIONAL) items 44
  Binary (COMP) items 44
  Native binary (COMP-5) items 45
  Packed-decimal (COMP-3) items 45
  Internal floating-point (COMP-1 and COMP-2) items 46
  Examples: numeric data and internal representation 46
Data format conversions 48
  Conversions and precision 49
    Conversions that lose precision 49
    Conversions that preserve precision 49
    Conversions that result in rounding 50
Sign representation of zoned and packed-decimal data 50
Checking for incompatible data (numeric class test) 51
Performing arithmetic 51
  Using COMPUTE and other arithmetic statements. 51
  Using arithmetic expressions 52
  Using numeric intrinsic functions 53
  Examples: numeric intrinsic functions 54
    General number handling 54
    Date and time 55
    Finance. 55
    Mathematics 56
    Statistics 56
Fixed-point contrasted with floating-point arithmetic 56
  Floating-point evaluations 56
  Fixed-point evaluations 57
  Arithmetic comparisons (relation conditions) . . 57
  Examples: fixed-point and floating-point evaluations 58
Using currency signs 58
  Example: multiple currency signs 59

Chapter 4. Handling tables. 61
Defining a table (OCCURS) 61
Nesting tables 63
  Example: subscripting 64
  Example: indexing 64
Referring to an item in a table 64
  Subscripting 65
  Indexing 66
Putting values into a table 67
  Loading a table dynamically. 67
  Initializing a table (INITIALIZE) 68
  Assigning values when you define a table (VALUE) 69

Initializing each table item individually . . . 69
Initializing a table at the group level 70
Initializing all occurrences of a given table element. 70
Example: PERFORM and subscripting 71
Example: PERFORM and indexing. 72
Creating variable-length tables (DEPENDING ON) 73
Loading a variable-length table 74
Assigning values to a variable-length table . . . 75
Searching a table 75
Doing a serial search (SEARCH) 76
Example: serial search 76
Doing a binary search (SEARCH ALL) 77
Example: binary search 78
Processing table items using intrinsic functions . . 78
Example: processing tables using intrinsic functions 79

Chapter 5. Selecting and repeating program actions 81
Selecting program actions 81
Coding a choice of actions 81
Using nested IF statements 82
Using the EVALUATE statement 83
Coding conditional expressions. 86
Switches and flags 87
Defining switches and flags 87
Example: switches 87
Example: flags 88
Resetting switches and flags 88
Example: set switch on 88
Example: set switch off 89
Repeating program actions 89
Choosing inline or out-of-line PERFORM . . . 90
Example: inline PERFORM statement. . . . 90
Coding a loop 91
Looping through a table 92
Executing multiple paragraphs or sections . . . 92

Chapter 6. Handling strings 93
Joining data items (STRING) 93
Example: STRING statement. 94
STRING results 95
Splitting data items (UNSTRING) 95
Example: UNSTRING statement 96
UNSTRING results 97
Manipulating null-terminated strings 98
Example: null-terminated strings 99
Referring to substrings of data items 99
Reference modifiers 101
Example: arithmetic expressions as reference modifiers 102
Example: intrinsic functions as reference modifiers 102
Tallying and replacing data items (INSPECT). . . 103
Examples: INSPECT statement 103
Converting data items (intrinsic functions) . . . 104
Converting to uppercase or lowercase (UPPER-CASE, LOWER-CASE) 105
Transforming to reverse order (REVERSE) . . . 105
Converting to numbers (NUMVAL, NUMVAL-C) 105
Converting from one code page to another . . 107
Evaluating data items (intrinsic functions) 107
Evaluating single characters for collating sequence 107
Finding the largest or smallest data item . . . 108
Returning variable-length results with alphanumeric or national functions 109
Finding the length of data items 110
Finding the date of compilation 111

Chapter 7. Processing files 113
Identifying files. 113
Identifying VSAM files 114
Identifying Encina SFS files. 114
Identifying STL files 115
Identifying RSD files 115
File systems 115
VSAM file system 116
SFS file system 117
STL file system 118
STL file system return codes 118
RSD file system 120
| STL and VSAM files that are larger than 2 GB 121
| File-system summary 121
Specifying a file organization and access mode . . 122
File organization and access mode 122
Sequential file organization. 123
Line-sequential file organization 124
Indexed file organization 124
Relative file organization 124
Sequential access 125
Random access 125
Dynamic access. 125
Protecting against errors when opening files . . . 125
Setting up a field for file status 126
Describing the structure of a file in detail 126
Coding input and output statements for files. . . 127
Example: COBOL coding for files. 127
File position indicator 129
Opening a file 129
Valid COBOL statements for sequential files 130
Valid COBOL statements for line-sequential files 130
Valid COBOL statements for indexed and relative files 131
Reading records from a file. 132
Statements used when writing records to a file 133
Adding records to a file 133
Replacing records in a file 134
Deleting records from a file. 135
PROCEDURE DIVISION statements used to update files 135
Enabling COBOL applications to access SFS files 136
Example: accessing SFS files 137
COBOL file descriptions. 137
sfsadmin commands 138
export commands 139
Improving SFS performance 139
Enabling client-side caching 139

Reducing the frequency of saving changes 140

Chapter 8. Sorting and merging files. 143
Sort and merge process 143
Describing the sort or merge file 144
Describing the input to sorting or merging . . . 145
Example: describing sort and input files for SORT 145
Coding the input procedure 146
Describing the output from sorting or merging . . 147
Coding the output procedure 147
Restrictions on input and output procedures . . . 148
Requesting the sort or merge 148
Setting sort or merge criteria 149
Choosing alternate collating sequences 150
Example: sorting with input and output procedures 150
Determining whether the sort or merge was successful 151
Sort and merge error numbers. 152
Stopping a sort or merge operation prematurely 154

Chapter 9. Handling errors 157
Handling errors in joining and splitting strings . . 157
Handling errors in arithmetic operations 158
Example: checking for division by zero. . . . 158
Handling errors in input and output operations 158
Using the end-of-file condition (AT END) . . . 160
Coding ERROR declaratives 160
Using file status keys. 161
Example: file status key 162
Using file system status codes 162
Example: checking file system status codes 163
Coding INVALID KEY phrases 164
Example: FILE STATUS and INVALID KEY 165
Handling errors when calling programs 165

Chapter 1. Structuring your program

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

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

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

RELATED TASKS
"Identifying a program"
"Describing the computing environment" on page 7
"Describing the data" on page 11
"Processing the data" on page 16
"Defining a class" on page 400
"Defining a class instance method" on page 405
"Structuring OO applications" on page 439

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.  12/07/2008.
Date-Compiled. 12/14/2008.
```

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

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

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

RELATED TASKS
"Changing the header of a source listing" on page 6
"Identifying a program as recursive" on page 6
"Marking a program as callable by containing programs" on page 6
"Setting a program to an initial state" on page 6

RELATED REFERENCES
Compiler limits (*COBOL for AIX Language Reference*)
Conventions for program-names (*COBOL for AIX Language Reference*)

Identifying a program as recursive

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

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

RELATED TASKS
"Sharing data in recursive or multithreaded programs" on page 16
"Making recursive calls" on page 478

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 467

Setting a program to an initial state

Use the `INITIAL` attribute to specify that whenever a program is called, that program and any nested programs that it contains are to be placed in their initial state.

When a program is in its initial state:
- Data items that have `VALUE` clauses are set to the specified values.
- Changed `GO TO` statements and `PERFORM` statements are in their initial states.
- Non-`EXTERNAL` files are closed.

RELATED TASKS
"Ending and reentering main programs or subprograms" on page 465

Changing the header of a source listing

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

The following example shows these five elements:

```
PP 5724-V62 IBM COBOL for AIX 3.1.0       Date 12/14/2008   Time 15:05:19   Page     1
```

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

RELATED REFERENCES
TITLE statement (*COBOL for AIX Language Reference*)

Describing the computing environment

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

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

Use the FILE-CONTROL and I-O-CONTROL paragraphs of the INPUT-OUTPUT SECTION to:
- Identify and describe the characteristics of the files in the program.
- Associate your files with the corresponding system file-name, directly or indirectly.
- Optionally identify the file system (for example, VSAM or STL file system) that is associated with a file. You can also do this at run time.
- Provide information about how the files are accessed.

"Example: FILE-CONTROL paragraph"

RELATED TASKS
"Specifying the collating sequence" on page 8
"Defining symbolic characters" on page 9
"Defining a user-defined class" on page 10
"Identifying files to the operating system" on page 10

RELATED REFERENCES
Sections and paragraphs (*COBOL for AIX Language Reference*)

Example: FILE-CONTROL paragraph

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

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

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

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

(3) The `ORGANIZATION` clause describes the file's organization. If omitted, the default is `ORGANIZATION IS SEQUENTIAL`.

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

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

RELATED TASKS
"Describing the computing environment" on page 7

Specifying the collating sequence

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

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

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

"Example: specifying the collating sequence" on page 9

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

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

You can also specify a collating sequence that you define.

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

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

RELATED TASKS
"Choosing alternate collating sequences" on page 150
"Controlling the collating sequence with a locale" on page 199
Chapter 11, "Setting the locale," on page 193
"Comparing national (UTF-16) data" on page 186

Example: specifying the collating sequence

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

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

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

RELATED TASKS
"Specifying the collating sequence" on page 8

Defining symbolic characters

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

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

```
SYMBOLIC CHARACTERS PLUS IS 44
```

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

RELATED TASKS
Chapter 11, "Setting the locale," on page 193

Defining a user-defined class

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

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

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

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

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

Identifying files to the operating system

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

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

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

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

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

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

RELATED TASKS
"Varying the input or output file at run time" on page 11
"Identifying files" on page 113

RELATED REFERENCES
"FILESYS" on page 303
ASSIGN clause (*COBOL for AIX Language Reference*)

Varying the input or output file at run time

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

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

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

Example: using different input files:

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

Consider a COBOL program that contains the following SELECT clause:

```
SELECT MASTER ASSIGN TO MASTERA
```

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

```
export MASTERA=/accounts/checking
export MASTERA=/accounts/savings
```

You can use the same program to access either the checking or savings file as the file called MASTER in the program without having to change or recompile the source.

Describing the data

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

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

The COBOL for AIX 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 13

RELATED TASKS
"Using data in input and output operations"
"Using data from another program" on page 15

RELATED REFERENCES
"cob2, cob2_j, and cob2_r options" on page 227
"ADDR" on page 246
Compiler limits (*COBOL for AIX Language Reference*)

Using data in input and output operations

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

Provide the following information about the data:

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

  Data items defined in the `FILE SECTION` are not available to `PROCEDURE DIVISION` statements until the file has been successfully opened.
- In the record description that follows the `FD` entry, describe the records in the file and their fields. The record-name is the object of `WRITE` and `REWRITE` statements.

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

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

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

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

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

RELATED CONCEPTS
"Nested programs" on page 467

RELATED TASKS
"Sharing files between programs (external files)" on page 491

RELATED REFERENCES
"FILE SECTION entries" on page 13

FILE SECTION entries

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

Table 1. **FILE SECTION entries**

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

RELATED REFERENCES
File section (*COBOL for AIX Language Reference*)

Comparison of WORKING-STORAGE and LOCAL-STORAGE

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

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

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

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

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

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

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

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

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

"Example: storage sections"

RELATED TASKS
"Ending and reentering main programs or subprograms" on page 465
Chapter 29, "Preparing COBOL programs for multithreading," on page 505
"WORKING-STORAGE SECTION for defining class instance data" on page 404

RELATED REFERENCES
Working-storage section (*COBOL for AIX Language Reference*)
Local-storage section (*COBOL for AIX Language Reference*)

Example: storage sections

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

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

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

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

The program produces the following output:

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

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

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

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

RELATED CONCEPTS
"Comparison of WORKING-STORAGE and LOCAL-STORAGE" on page 13

Using data from another program

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

RELATED TASKS
"Sharing data in separately compiled programs"
"Sharing data in nested programs" on page 16
"Sharing data in recursive or multithreaded programs" on page 16
"Passing data" on page 481

Sharing data in separately compiled programs

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

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

RELATED TASKS
"Passing data" on page 481

Sharing data in nested programs

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

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

RELATED CONCEPTS
"Nested programs" on page 467

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 505

RELATED TASKS
"Making recursive calls" on page 478

RELATED REFERENCES
"THREAD" on page 287
SET statement (*COBOL for AIX Language Reference*)

Processing the data

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

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

```
PROCEDURE DIVISION.
```

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

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

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

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

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

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

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

```
PROCEDURE DIVISION RETURNING dataname2
```

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

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

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

RELATED CONCEPTS
"How logic is divided in the PROCEDURE DIVISION"

RELATED TASKS
"Eliminating repetitive coding" on page 557

RELATED REFERENCES
The procedure division header (*COBOL for AIX Language Reference*)
The USING phrase (*COBOL for AIX Language Reference*)
CALL statement (*COBOL for AIX Language Reference*)

How logic is divided in the PROCEDURE DIVISION

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

Section
Logical subdivision of your processing logic.

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

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

Paragraph
Subdivision of a section, procedure, or program.

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

A paragraph can be the subject of a statement.

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

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

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

Phrase

A subdivision of a statement.

RELATED CONCEPTS
"Compiler-directing statements" on page 19
"Scope terminators" on page 19
"Imperative statements"
"Conditional statements"
"Declaratives" on page 21

RELATED REFERENCES
PROCEDURE DIVISION structure (*COBOL for AIX Language Reference*)

Imperative statements

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

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

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

RELATED CONCEPTS
"Conditional statements"
"Scope terminators" on page 19

Conditional statements

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

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

You can use a delimited scope statement in these ways:

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

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

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

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

```
        AT END
            SET TRANSACTION-EOF TO TRUE
    END-READ
END-PERFORM
```

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

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

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

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

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

RELATED CONCEPTS
"Imperative statements" on page 18
"Scope terminators"

RELATED TASKS
"Selecting program actions" on page 81

RELATED REFERENCES
Conditional statements (*COBOL for AIX Language Reference*)

Compiler-directing statements

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

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

RELATED REFERENCES
Chapter 15, "Compiler-directing statements," on page 295
Compiler-directing statements (*COBOL for AIX Language Reference*)

Scope terminators

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

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

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

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

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

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

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

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

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

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

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

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

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

RELATED CONCEPTS
"Conditional statements" on page 18
"Imperative statements" on page 18

Declaratives

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

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

RELATED TASKS
"Finding and handling input-output errors" on page 308

RELATED REFERENCES
Declaratives (*COBOL for AIX Language Reference*)

Chapter 2. Using data

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

RELATED TASKS
"Using variables, structures, literals, and constants"
"Assigning values to data items" on page 27
"Displaying values on a screen or in a file (DISPLAY)" on page 35
"Using intrinsic functions (built-in functions)" on page 36
"Using tables (arrays) and pointers" on page 37
Chapter 10, "Processing data in an international environment," on page 169

Using variables, structures, literals, and constants

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

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

RELATED TASKS
"Using variables"
"Using data items and group items" on page 24
"Using literals" on page 25
"Using constants" on page 26
"Using figurative constants" on page 26

RELATED REFERENCES
Classes and categories of data (*COBOL for AIX Language Reference*)

Using variables

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

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

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

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

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 173

RELATED TASKS
"Using national data (Unicode) in COBOL" on page 174

RELATED REFERENCES
"NSYMBOL" on page 273
"Storage of character data" on page 181
PICTURE clause (*COBOL for AIX Language Reference*)

Using data items and group items

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

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

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

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

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

```
Data Division.
File Section.
FD  Customer-File
    Record Contains 90 Characters.
01  Customer-Record        Group-Usage National.
    05  Customer-Name.
        10 Last-Name       Pic n(17).
        10 Filler          Pic n.
        10 Initials        Pic nn.
    05  Part-Order.
        10 Part-Name       Pic n(15).
        10 Part-Color      Pic n(10).
Working-Storage Section.
01  Orig-Customer-Name     Group-Usage National.
    05  Surname            Pic n(17).
    05  Initials           Pic n(3).
01  Inventory-Part-Name    Pic n(15)  Usage National.
. . .
Procedure Division.
    Move Customer-Name to Orig-Customer-Name
    Move Part-Name to Inventory-Part-Name
    . . .
```

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

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 173
"National groups" on page 177

RELATED TASKS
"Using national data (Unicode) in COBOL" on page 174
"Using national groups" on page 179

RELATED REFERENCES
"FILE SECTION entries" on page 13
"Storage of character data" on page 181
Classes and categories of group items (*COBOL for AIX Language Reference*)
PICTURE clause (*COBOL for AIX Language Reference*)
MOVE statement (*COBOL for AIX Language Reference*)
USAGE clause (*COBOL for AIX Language Reference*)

Using literals

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

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

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

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

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

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

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

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

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 173

RELATED TASKS
"Using national literals" on page 175
"Using DBCS literals" on page 190

RELATED REFERENCES
"NSYMBOL" on page 273
Literals (*COBOL for AIX 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 174
"Coding for use of DBCS support" on page 189

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 176
"Coding for use of DBCS support" on page 189

RELATED REFERENCES
Figurative constants (*COBOL for AIX Language Reference*)

Assigning values to data items

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

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

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

"Examples: initializing data items" on page 28

RELATED TASKS
"Initializing a structure (INITIALIZE)" on page 30
"Assigning values to elementary data items (MOVE)" on page 32
"Assigning values to group data items (MOVE)" on page 33
"Assigning input from a screen or file (ACCEPT)" on page 34
"Joining data items (STRING)" on page 93

"Splitting data items (UNSTRING)" on page 95
"Assigning arithmetic results (MOVE or COMPUTE)" on page 34
"Tallying and replacing data items (INSPECT)" on page 103
Chapter 10, "Processing data in an international environment," on page 169

Examples: initializing data items

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

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

Initializing a data item to blanks or zeros:

```
INITIALIZE identifier-1
```

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

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

Initializing an alphanumeric data item:

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

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

Initializing an alphanumeric right-justified data item:

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

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

Initializing an alphanumeric-edited data item:

```
01  ALPHANUM-EDIT-1   PIC XXBX/XXX  VALUE "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]	AB*b*C/DEF[1]	M/*bb*/*bbb*[1]
1. The symbol *b* represents a blank space.		

Initializing a national data item:

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

NATIONAL-3	NATIONAL-1 before	NATIONAL-1 after
00430044[1]	00410042[2]	00430044[1]
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 $ZZ9V  VALUE "$127".
01  NUM-DISP-3        PIC 999V   VALUE 12.
. . .
    INITIALIZE NUM-EDIT-DISP-1
        REPLACING NUMERIC DATA BY NUM-DISP-3
```

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

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

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

NUM-NATL-3	**NUM-EDIT-NATL-1 before**	**NUM-EDIT-NATL-1 after**
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)"
"Initializing a table (INITIALIZE)" on page 68
"Defining numeric data" on page 39

RELATED REFERENCES
"NSYMBOL" on page 273

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`	*b*`000000000000000000000000000`[1]
2	`R0013900004800000000000000000`	*b*`000000000000000000000000000`[1]
3	`S0014100001200000000000000000`	*b*`000000000000000000000000000`[1]
4	`C0013830000000004250000000000`	*b*`000000000000000000000000000`[1]
5	`C0020100000000000000100000000`	*b*`000000000000000000000000000`[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 177

RELATED TASKS
"Initializing a table (INITIALIZE)" on page 68
"Using national groups" on page 179

RELATED REFERENCES
INITIALIZE statement (*COBOL for AIX Language Reference*)

Assigning values to elementary data items (MOVE)

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

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

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

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

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

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

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

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

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

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

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 173

RELATED TASKS
"Assigning values to group data items (MOVE)" on page 33
"Converting to or from national (Unicode) representation" on page 182

RELATED REFERENCES

Classes and categories of data (*COBOL for AIX Language Reference*)
MOVE statement (*COBOL for AIX 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 173
"National groups" on page 177

RELATED TASKS
"Assigning values to elementary data items (MOVE)" on page 32
"Using national groups" on page 179
"Converting to or from national (Unicode) representation" on page 182

RELATED REFERENCES
Classes and categories of group items (*COBOL for AIX Language Reference*)
MOVE statement (*COBOL for AIX Language Reference*)
CORRESPONDING phrase (*COBOL for AIX Language Reference*)

Assigning arithmetic results (MOVE or COMPUTE)

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

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

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

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

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

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

RELATED REFERENCES
"DIAGTRUNC" on page 257
Intrinsic functions (*COBOL for AIX Language Reference*)

Assigning input from a screen or file (ACCEPT)

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

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

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

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

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

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

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

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

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

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

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

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

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 173

RELATED TASKS
"Setting environment variables" on page 209
"Converting alphanumeric or DBCS to national (NATIONAL-OF)" on page 183
"Getting the system date under CICS" on page 338

RELATED REFERENCES
ACCEPT statement (*COBOL for AIX Language Reference*)
SPECIAL-NAMES paragraph (*COBOL for AIX Language Reference*)

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

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

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

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

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

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

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

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

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 173

RELATED TASKS
"Converting national to alphanumeric (DISPLAY-OF)" on page 184
"Coding COBOL programs to run under CICS" on page 337

RELATED REFERENCES
"Runtime environment variables" on page 211
DISPLAY statement (*COBOL for AIX 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). For example, the groups of highlighted words below are function-identifiers:

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

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

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

```
Working-Storage Section.
01  x                Pic 99  value 2.
01  y                Pic 99  value 4.
01  z                Pic 99  value 0.
01  Sqrt             Pic 99  value 0.
```

```
. . .
    Compute Sqrt = 16 ** .5
    Compute z = x + Function Sqrt(y)
    . . .
```

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

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

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

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

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

RELATED TASKS
"Processing table items using intrinsic functions" on page 78
"Converting data items (intrinsic functions)" on page 104
"Evaluating data items (intrinsic functions)" on page 107

Using tables (arrays) and pointers

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

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

You can perform the following operations with pointer data items:
- Pass them between programs by using the CALL . . . BY REFERENCE statement.
- Move them to other pointers by using the SET statement.
- Compare them to other pointers for equality by using a relation condition.
- Initialize them to contain an invalid address by using VALUE IS NULL.

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

RELATED TASKS
"Defining a table (OCCURS)" on page 61
"Using procedure and function pointers" on page 489

RELATED REFERENCES
"ADDR" on page 246

Chapter 3. Working with numbers and arithmetic

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

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

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

RELATED TASKS
"Defining numeric data"
"Displaying numeric data" on page 41
"Controlling how numeric data is stored" on page 42
"Checking for incompatible data (numeric class test)" on page 51
"Performing arithmetic" on page 51
"Using currency signs" on page 58

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 46

RELATED CONCEPTS
"Formats for numeric data" on page 43
Appendix D, "Intermediate results and arithmetic precision," on page 583

RELATED TASKS
"Displaying numeric data" on page 41
"Controlling how numeric data is stored" on page 42
"Performing arithmetic" on page 51
"Defining national numeric data items" on page 177

RELATED REFERENCES
"Sign representation of zoned and packed-decimal data" on page 50
"Storage of character data" on page 181
"ARITH" on page 248
SIGN clause (*COBOL for AIX Language Reference*)

Displaying numeric data

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

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

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

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

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

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

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

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

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

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

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

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

"Examples: numeric data and internal representation" on page 46

RELATED TASKS
"Displaying values on a screen or in a file (DISPLAY)" on page 35
"Controlling how numeric data is stored"
"Defining numeric data" on page 39
"Performing arithmetic" on page 51
"Defining national numeric data items" on page 177
"Converting to or from national (Unicode) representation" on page 182

RELATED REFERENCES
MOVE statement (*COBOL for AIX Language Reference*)
BLANK WHEN ZERO clause (*COBOL for AIX Language Reference*)

Controlling how numeric data is stored

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

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

The numeric data you use in your program will have one of the following formats available with COBOL:
- External decimal (`USAGE DISPLAY` or `USAGE NATIONAL`)
- External floating point (`USAGE DISPLAY` or `USAGE NATIONAL`)
- Internal decimal (`USAGE PACKED-DECIMAL`)
- Binary (`USAGE BINARY`)
- Native binary (`USAGE COMP-5`)
- Internal floating point (`USAGE COMP-1` or `USAGE COMP-2`)

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

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

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

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

"Examples: numeric data and internal representation" on page 46

RELATED CONCEPTS
"Formats for numeric data"
"Data format conversions" on page 48
Appendix D, "Intermediate results and arithmetic precision," on page 583

RELATED TASKS
"Defining numeric data" on page 39
"Displaying numeric data" on page 41
"Performing arithmetic" on page 51

RELATED REFERENCES
"Conversions and precision" on page 49
"Sign representation of zoned and packed-decimal data" on page 50

Formats for numeric data

Several formats are available for numeric data.

External decimal (DISPLAY and NATIONAL) items

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

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

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

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

External floating-point (DISPLAY and NATIONAL) items

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

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

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

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

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

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

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

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

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

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

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

Binary (COMP) items

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

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

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

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

Native binary (COMP-5) items

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

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

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

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

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

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

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

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

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

Packed-decimal (COMP-3) items

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

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

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

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

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 173
Appendix D, "Intermediate results and arithmetic precision," on page 583

RELATED TASKS
"Defining numeric data" on page 39
"Defining national numeric data items" on page 177

RELATED REFERENCES
"Storage of character data" on page 181
"TRUNC" on page 288
"FLOAT" on page 267
Classes and categories of data (*COBOL for AIX Language Reference*)
SIGN clause (*COBOL for AIX Language Reference*)
VALUE clause (*COBOL for AIX Language Reference*)

Examples: numeric data and internal representation

The following tables show the internal representation of numeric items.

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

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

Numeric type	PICTURE and USAGE and optional SIGN clause	Value	Internal representation
Binary	PIC S9999 BINARY PIC S9999 COMP PIC S9999 COMP-4	+ 1234	04 D2
		- 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

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

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

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

Numeric type	PICTURE and USAGE and optional SIGN clause	Value	Internal representation
External decimal	PIC S9999 DISPLAY	+ 1234	31 32 33 34
		- 1234	31 32 33 74
		1234	31 32 33 34
	PIC 9999 DISPLAY	1234	31 32 33 34
	PIC 9999 NATIONAL	1234	00 31 00 32 00 33 00 34
	PIC S9999 DISPLAY SIGN LEADING	+ 1234	31 32 33 34
		- 1234	71 32 33 34
	PIC S9999 DISPLAY SIGN LEADING SEPARATE	+ 1234	2B 31 32 33 34
		- 1234	2D 31 32 33 34
	PIC S9999 DISPLAY SIGN TRAILING SEPARATE	+ 1234	31 32 33 34 2B
		- 1234	31 32 33 34 2D
	PIC S9999 NATIONAL SIGN LEADING SEPARATE	+ 1234	00 2B 00 31 00 32 00 33 00 34
		- 1234	00 2D 00 31 00 32 00 33 00 34
	PIC S9999 NATIONAL SIGN TRAILING SEPARATE	+ 1234	00 31 00 32 00 33 00 34 00 2B
		- 1234	00 31 00 32 00 33 00 34 00 2D
Internal decimal	PIC S9999 PACKED-DECIMAL PIC S9999 COMP-3	+ 1234	01 23 4C
		- 1234	01 23 4D
	PIC 9999 PACKED-DECIMAL PIC 9999 COMP-3	1234	01 23 4C
Internal floating point	COMP-1	+ 1234	44 9A 40 00
		- 1234	C4 9A 40 00
	COMP-2	+ 1234	40 93 48 00 00 00 00 00
		- 1234	C0 93 48 00 00 00 00 00
External floating point	PIC +9(2).9(2)E+99 DISPLAY	+ 12.34E+02	2B 31 32 2E 33 34 45 2B 30 32
		- 12.34E+02	2D 31 32 2E 33 34 45 2B 30 32
	PIC +9(2).9(2)E+99 NATIONAL	+ 12.34E+02	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

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

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

Numeric type	PICTURE and USAGE and optional SIGN clause	Value	Internal representation
External decimal	PIC S9999 DISPLAY	+ 1234	F1 F2 F3 C4
		- 1234	F1 F2 F3 D4
		1234	F1 F2 F3 C4
	PIC 9999 DISPLAY	1234	F1 F2 F3 F4
	PIC 9999 NATIONAL	1234	00 31 00 32 00 33 00 34
	PIC S9999 DISPLAY SIGN LEADING	+ 1234	C1 F2 F3 F4
		- 1234	D1 F2 F3 F4
	PIC S9999 DISPLAY SIGN LEADING SEPARATE	+ 1234	4E F1 F2 F3 F4
		- 1234	60 F1 F2 F3 F4
	PIC S9999 DISPLAY SIGN TRAILING SEPARATE	+ 1234	F1 F2 F3 F4 4E
		- 1234	F1 F2 F3 F4 60
	PIC S9999 NATIONAL SIGN LEADING SEPARATE	+ 1234	00 2B 00 31 00 32 00 33 00 34
		- 1234	00 2D 00 31 00 32 00 33 00 34
	PIC S9999 NATIONAL SIGN TRAILING SEPARATE	+ 1234	00 31 00 32 00 33 00 34 00 2B
		- 1234	00 31 00 32 00 33 00 34 00 2D
Internal decimal	PIC S9999 PACKED-DECIMAL PIC S9999 COMP-3	+ 1234	01 23 4C
		- 1234	01 23 4D
	PIC 9999 PACKED-DECIMAL PIC 9999 COMP-3	1234	01 23 4C
Internal floating point	COMP-1	+ 1234	43 4D 20 00
		- 1234	C3 4D 20 00
	COMP-2	+ 1234	43 4D 20 00 00 00 00 00
		- 1234	C3 4D 20 00 00 00 00 00
External floating point	PIC +9(2).9(2)E+99 DISPLAY	+ 12.34E+02	4E F1 F2 4B F3 F4 C5 4E F0 F2
		- 12.34E+02	60 F1 F2 4B F3 F4 C5 4E F0 F2
	PIC +9(2).9(2)E+99 NATIONAL	+ 12.34E+02	00 2B 00 31 00 32 00 2E 00 33 00 34 00 45 00 2B 00 30 00 32
		- 12.34E+02	00 2D 00 31 00 32 00 2E 00 33 00 34 00 45 00 2B 00 30 00 32

Data format conversions

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

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

Conversions and precision

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

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

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

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

Conversions that lose precision

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

Conversions that preserve precision

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

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

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

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

Conversions that result in rounding

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

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

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

RELATED CONCEPTS
Appendix D, "Intermediate results and arithmetic precision," on page 583

Sign representation of zoned and packed-decimal data

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

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

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

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

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

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

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

Positive:
A, C, E, and F

Negative:
B and D

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

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

RELATED REFERENCES
"ZWB" on page 293
"Data representation" on page 573

Checking for incompatible data (numeric class test)

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

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

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

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

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

Performing arithmetic

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

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

RELATED TASKS
"Using COMPUTE and other arithmetic statements"
"Using arithmetic expressions" on page 52
"Using numeric intrinsic functions" on page 53

Using COMPUTE and other arithmetic statements

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

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

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

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

COMPUTE	Equivalent arithmetic statements
`Compute Increment = Increment + 1`	`Add 1 to Increment`

`Compute Balance =`
`    Balance - Overdraft`
	`Subtract Overdraft from Balance`

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

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

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

RELATED CONCEPTS
"Fixed-point contrasted with floating-point arithmetic" on page 56
Appendix D, "Intermediate results and arithmetic precision," on page 583

RELATED TASKS
"Defining numeric data" on page 39

Using arithmetic expressions

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

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

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

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

Arithmetic operators are evaluated in the following order of precedence:

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

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

Table 7. **Order of evaluation of arithmetic operators** *(continued)*

Operator	Meaning	Order of evaluation
Binary + or -	Addition or subtraction	Last

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

RELATED CONCEPTS
"Fixed-point contrasted with floating-point arithmetic" on page 56
Appendix D, "Intermediate results and arithmetic precision," on page 583

Using numeric intrinsic functions

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

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

Numeric functions are classified into the following categories:

Integer
Those that return an integer

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

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

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

Table 8. **Numeric intrinsic functions**

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

"Examples: numeric intrinsic functions"

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

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

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

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

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

RELATED CONCEPTS
"Fixed-point contrasted with floating-point arithmetic" on page 56
Appendix D, "Intermediate results and arithmetic precision," on page 583

RELATED REFERENCES
"ARITH" on page 248

Examples: numeric intrinsic functions

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

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

General number handling

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

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

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

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

Date and time

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

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

Finance

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

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

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

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

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

Mathematics

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

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

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

Statistics

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

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

RELATED TASKS
"Converting to numbers (NUMVAL, NUMVAL-C)" on page 105

Fixed-point contrasted with floating-point arithmetic

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

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

- General arithmetic

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

- Expressions and functions

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

- Arithmetic comparisons

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

Floating-point evaluations

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

- An operand or result field is floating point.

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

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

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

- An exponent contains decimal places.

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

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

Fixed-point evaluations

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

Arithmetic comparisons (relation conditions)

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

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

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

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

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

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

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

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

RELATED REFERENCES
"Arithmetic expressions in nonarithmetic statements" on page 591

Examples: fixed-point and floating-point evaluations

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

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

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

These statements are evaluated using floating-point arithmetic:

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

These statements are evaluated using fixed-point arithmetic:

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

Using currency signs

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

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

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

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

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

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

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

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

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

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

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

RELATED REFERENCES
"CURRENCY" on page 255
CURRENCY SIGN clause (*COBOL for AIX Language Reference*)

Example: multiple currency signs

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

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EuroSamp.
Environment Division.
Configuration Section.
Special-Names.
    Currency Sign is "CHF "  with Picture Symbol "F"
    Currency Sign is "EUR "  with Picture Symbol "U".
Data Division.
Working-Storage Section.
01  Deposit-in-Euro        Pic S9999V99 Value 8000.00.
01  Deposit-in-CHF         Pic S99999V99.
01  Deposit-Report.
    02  Report-in-Franc    Pic -FFFFF9.99.
    02  Report-in-Euro     Pic -UUUUU9.99.
01  EUR-to-CHF-Conv-Rate   Pic 9V99999  Value 1.53893.
. . .
PROCEDURE DIVISION.
Report-Deposit-in-CHF-and-EUR.
    Move Deposit-in-Euro to Report-in-Euro
    Compute Deposit-in-CHF Rounded
          = Deposit-in-Euro * EUR-to-CHF-Conv-Rate
      On Size Error
        Perform Conversion-Error
      Not On Size Error
        Move Deposit-in-CHF to Report-in-Franc
        Display "Deposit in euro  = " Report-in-Euro
        Display "Deposit in franc = " Report-in-Franc
    End-Compute
    Goback.
Conversion-Error.
           Display "Conversion error from EUR to CHF"
           Display "Euro value: " Report-in-Euro.
```

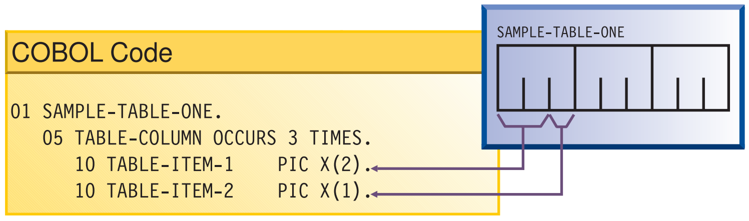
The above example produces the following display output:

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

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

Chapter 4. Handling tables

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



```
01 SAMPLE-TABLE-ONE.
   05 TABLE-COLUMN OCCURS 3 TIMES.
      10 TABLE-ITEM-1    PIC X(2).
      10 TABLE-ITEM-2    PIC X(1).
```

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

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

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

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

RELATED TASKS
"Nesting tables" on page 63
"Defining a table (OCCURS)"
"Referring to an item in a table" on page 64
"Putting values into a table" on page 67
"Creating variable-length tables (DEPENDING ON)" on page 73
"Searching a table" on page 75
"Processing table items using intrinsic functions" on page 78
"Handling tables efficiently" on page 547

Defining a table (OCCURS)

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

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

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

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

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

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

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

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

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

You must code the ASCENDING or DESCENDING KEY phrase of the OCCURS clause to do a binary search (SEARCH ALL) of a table.

"Example: binary search" on page 78

RELATED CONCEPTS
"National groups" on page 177

RELATED TASKS
"Nesting tables" on page 63
"Referring to an item in a table" on page 64
"Putting values into a table" on page 67
"Creating variable-length tables (DEPENDING ON)" on page 73
"Using national groups" on page 179
"Doing a binary search (SEARCH ALL)" on page 77
"Defining numeric data" on page 39

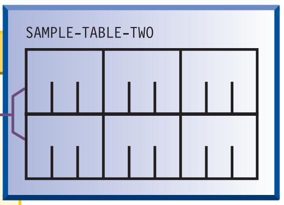
RELATED REFERENCES
OCCURS clause (*COBOL for AIX Language Reference*)
SIGN clause (*COBOL for AIX Language Reference*)
ASCENDING KEY and DESCENDING KEY phrases (*COBOL for AIX Language Reference*)

Nesting tables

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

COBOL Code

```
01 SAMPLE-TABLE-TWO.
   05 TABLE-ROW OCCURS 2 TIMES.
      10 TABLE-COLUMN OCCURS 3 TIMES.
         15 TABLE-ITEM-1   PIC X(2).
         15 TABLE-ITEM-2   PIC X(1).
```

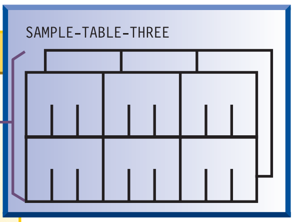


For example, in `SAMPLE-TABLE-TWO` above, `TABLE-ROW` is an element of a one-dimensional table that occurs two times. `TABLE-COLUMN` is an element of a two-dimensional table that occurs three times in each occurrence of `TABLE-ROW`.

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:

COBOL Code

```
01 SAMPLE-TABLE-THREE.
   05 TABLE-DEPTH OCCURS 2 TIMES.
      10 TABLE-ROW OCCURS 2 TIMES.
         15 TABLE-COLUMN OCCURS 3 TIMES.
            20 TABLE-ITEM-1   PIC X(2).
            20 TABLE-ITEM-2   PIC X(1).
```



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

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

"Example: subscripting" on page 64
"Example: indexing" on page 64

RELATED TASKS
"Defining a table (OCCURS)" on page 61
"Referring to an item in a table" on page 64
"Putting values into a table" on page 67
"Creating variable-length tables (DEPENDING ON)" on page 73
"Searching a table" on page 75
"Processing table items using intrinsic functions" on page 78
"Handling tables efficiently" on page 547

RELATED REFERENCES
OCCURS clause (*COBOL for AIX Language Reference*)

Example: subscripting

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

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

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

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

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

RELATED TASKS
"Subscripting" on page 65

Example: indexing

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

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

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

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

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

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

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

This calculation is based on the following element lengths:

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

RELATED TASKS
"Indexing" on page 66

Referring to an item in a table

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

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

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

RELATED TASKS
"Subscripting"
"Indexing" on page 66

Subscripting

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

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

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

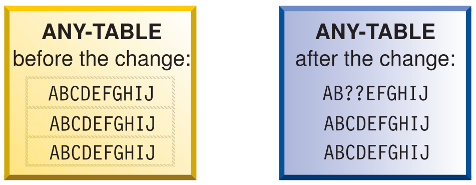
You can increment or decrement a literal or variable subscript by a specified integer amount. For example:

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

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

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

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



"Example: subscripting" on page 64

RELATED TASKS
"Indexing" on page 66

"Putting values into a table" on page 67
"Searching a table" on page 75
"Handling tables efficiently" on page 547

Indexing

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

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

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

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

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

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

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

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

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

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

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

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

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

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

"Example: indexing" on page 64

RELATED TASKS
"Subscripting" on page 65
"Putting values into a table"
"Searching a table" on page 75
"Processing table items using intrinsic functions" on page 78
"Handling tables efficiently" on page 547

RELATED REFERENCES
"ADDR" on page 246
INDEXED BY phrase (*COBOL for AIX Language Reference*)
INDEX phrase (*COBOL for AIX Language Reference*)
SET statement (*COBOL for AIX Language Reference*)

Putting values into a table

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

RELATED TASKS
"Loading a table dynamically"
"Loading a variable-length table" on page 74
"Initializing a table (INITIALIZE)" on page 68
"Assigning values when you define a table (VALUE)" on page 69
"Assigning values to a variable-length table" on page 75

Loading a table dynamically

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

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

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

"Example: PERFORM and subscripting" on page 71
"Example: PERFORM and indexing" on page 72

RELATED REFERENCES
PERFORM with VARYING phrase (*COBOL for AIX Language Reference*)

Initializing a table (INITIALIZE)

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

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

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

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

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

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

```
01  TABLE-ONE.
  02  Trans-out  Occurs 20.
      05  Trans-code      Pic X    Value "R".
      05  Part-number     Pic XX   Value "13".
      05  Trans-quan      Pic 99   Value 10.
      05  Price-fields.
          10  Unit-price   Pic 99V  Value 50.
          10  Discount     Pic 99V  Value 25.
          10  Sales-Price  Pic 999  Value 375.
      . . .
      Initialize TABLE-ONE Replacing Numeric Data By 3
                                Alphanumeric Data By "X"
```

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

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

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

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

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

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

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

"Examples: initializing data items" on page 28

RELATED TASKS
"Initializing a structure (INITIALIZE)" on page 30
"Assigning values when you define a table (VALUE)"
"Assigning values to a variable-length table" on page 75
"Looping through a table" on page 92
"Using data items and group items" on page 24
"Using national groups" on page 179

RELATED REFERENCES
INITIALIZE statement (*COBOL for AIX Language Reference*)

Assigning values when you define a table (VALUE)

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

Set static values in tables in one of these ways:
- Initialize each table item individually.
- Initialize an entire table at the group level.
- Initialize all occurrences of a given table element to the same value.

RELATED TASKS
"Initializing each table item individually"
"Initializing a table at the group level" on page 70
"Initializing all occurrences of a given table element" on page 70
"Initializing a structure (INITIALIZE)" on page 30

Initializing each table item individually

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

Use the following technique, which is shown in the example code below:
1. Declare a record (such as Error-Flag-Table below) that contains the items that are to be in the table.
2. Set the initial value of each item in a VALUE clause.
3. Code a REDEFINES entry to make the record into a table.

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

In the example above, the VALUE clause at the 01 level initializes each of the table items to the same value. Each table item could instead be described with its own VALUE clause to initialize that item to a distinct value.

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

RELATED TASKS
"Initializing a structure (INITIALIZE)" on page 30
"Assigning values to a variable-length table" on page 75

RELATED REFERENCES
REDEFINES clause (*COBOL for AIX Language Reference*)
OCCURS clause (*COBOL for AIX Language Reference*)

Initializing a table at the group level

Code an alphanumeric or national group data item and assign to it, through the VALUE clause, the contents of the whole table. Then, in a subordinate data item, use an OCCURS clause to define the individual table items.

In the following example, the alphanumeric group data item TABLE-ONE uses a VALUE clause that initializes each of the four elements of TABLE-TWO:

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

In the following example, the national group data item Table-OneN uses a VALUE clause that initializes each of the three elements of the subordinate data item Table-TwoN (each of which is implicitly USAGE NATIONAL). Note that you can initialize a national group data item with a VALUE clause that uses an alphanumeric literal, as shown below, or a national literal.

```
01  Table-OneN  Group-Usage National  Value "AB12CD34EF56".
    05  Table-TwoN   Occurs 3 Times   Indexed By MyI.
        10  ElementOneN  Pic nn.
        10  ElementTwoN  Pic 99.
```

After Table-OneN is initialized, ElementOneN(1) contains NX"00410042" (the UTF-16 representation of 'AB'), the national decimal item ElementTwoN(1) contains NX"00310032" (the UTF-16 representation of '12'), and so forth.

RELATED REFERENCES
OCCURS clause (*COBOL for AIX Language Reference*)
GROUP-USAGE clause (*COBOL for AIX Language Reference*)

Initializing all occurrences of a given table element

You can use the VALUE clause in the data description of a table element to initialize all instances of that element to the specified value.

```
01  T2.
    05 T-OBJ                   PIC 9   VALUE 3.
    05 T OCCURS 5 TIMES
           DEPENDING ON T-OBJ.
       10 X                    PIC XX  VALUE "AA".
       10 Y                    PIC 99  VALUE 19.
       10 Z                    PIC XX  VALUE "BB".
```

For example, the code above causes all the X elements (1 through 5) to be initialized to AA, all the Y elements (1 through 5) to be initialized to 19, and all the Z elements (1 through 5) to be initialized to BB. T-OBJ is then set to 3.

RELATED TASKS
"Assigning values to a variable-length table" on page 75

RELATED REFERENCES
OCCURS clause (*COBOL for AIX Language Reference*)

Example: PERFORM and subscripting

This example traverses an error-flag table using subscripting until an error code that has been set is found. If an error code is found, the corresponding error message is moved to a print report field.

```
***********************************************************
***          E R R O R   F L A G   T A B L E          ***
***********************************************************
 01  Error-Flag-Table                   Value Spaces.
   88 No-Errors                         Value Spaces.
     05 Type-Error                      Pic X.
     05 Shift-Error                     Pic X.
     05 Home-Code-Error                 Pic X.
     05 Work-Code-Error                 Pic X.
     05 Name-Error                      Pic X.
     05 Initials-Error                  Pic X.
     05 Duplicate-Error                 Pic X.
     05 Not-Found-Error                 Pic X.
 01  Filler Redefines Error-Flag-Table.
     05 Error-Flag Occurs 8 Times
           Indexed By Flag-Index        Pic X.
 77  Error-on                           Pic X  Value "E".
***********************************************************
***        E R R O R   M E S S A G E   T A B L E      ***
***********************************************************
 01  Error-Message-Table.
     05  Filler                          Pic X(25) Value
          "Transaction Type Invalid".
     05  Filler                          Pic X(25) Value
          "Shift Code Invalid".
     05  Filler                          Pic X(25) Value
          "Home Location Code Inval.".
     05  Filler                          Pic X(25) Value
          "Work Location Code Inval.".
     05  Filler                          Pic X(25) Value
          "Last Name - Blanks".
     05  Filler                          Pic X(25) Value
          "Initials - Blanks".
     05  Filler                          Pic X(25) Value
          "Duplicate Record Found".
     05  Filler                          Pic X(25) Value
          "Commuter Record Not Found".
 01  Filler Redefines Error-Message-Table.
     05  Error-Message Occurs 8 Times
            Indexed By Message-Index     Pic X(25).
 . . .
 PROCEDURE DIVISION.
     . . .
     Perform
          Varying Sub From 1 By 1
          Until No-Errors
        If Error-Flag (Sub) = Error-On
          Move Space To Error-Flag (Sub)
          Move Error-Message (Sub) To Print-Message
          Perform 260-Print-Report
        End-If
     End-Perform
     . . .
```

Example: PERFORM and indexing

This example traverses an error-flag table using indexing until an error code that has been set is found. If an error code is found, the corresponding error message is moved to a print report field.

```
***********************************************************
***          E R R O R   F L A G   T A B L E           ***
***********************************************************
 01  Error-Flag-Table                    Value Spaces.
   88 No-Errors                          Value Spaces.
     05 Type-Error                       Pic X.
     05 Shift-Error                      Pic X.
     05 Home-Code-Error                  Pic X.
     05 Work-Code-Error                  Pic X.
     05 Name-Error                       Pic X.
     05 Initials-Error                   Pic X.
     05 Duplicate-Error                  Pic X.
     05 Not-Found-Error                  Pic X.
 01  Filler Redefines Error-Flag-Table.
     05 Error-Flag Occurs 8 Times
          Indexed By Flag-Index          Pic X.
 77  Error-on                            Pic X  Value "E".
***********************************************************
***        E R R O R   M E S S A G E   T A B L E      ***
***********************************************************
 01  Error-Message-Table.
     05  Filler                           Pic X(25) Value
          "Transaction Type Invalid".
     05  Filler                           Pic X(25) Value
          "Shift Code Invalid".
     05  Filler                           Pic X(25) Value
          "Home Location Code Inval.".
     05  Filler                           Pic X(25) Value
          "Work Location Code Inval.".
     05  Filler                           Pic X(25) Value
          "Last Name - Blanks".
     05  Filler                           Pic X(25) Value
          "Initials - Blanks".
     05  Filler                           Pic X(25) Value
          "Duplicate Record Found".
     05  Filler                           Pic X(25) Value
          "Commuter Record Not Found".
 01  Filler Redefines Error-Message-Table.
     05  Error-Message Occurs 8 Times
            Indexed By Message-Index      Pic X(25).
 . . .
 PROCEDURE DIVISION.
     . . .
     Set Flag-Index To 1
     Perform Until No-Errors
       Search Error-Flag
         When Error-Flag (Flag-Index) = Error-On
           Move Space To Error-Flag (Flag-Index)
           Set Message-Index To Flag-Index
           Move Error-Message (Message-Index) To
             Print-Message
           Perform 260-Print-Report
       End-Search
     End-Perform
     . . .
```

Creating variable-length tables (DEPENDING ON)

If you do not know before run time how many times a table element occurs, define a variable-length table. To do so, use the OCCURS DEPENDING ON (ODO) clause.

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

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

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

- Correct calculation of record lengths

  The length of the variable portions of a group item is the product of the object of the DEPENDING ON phrase and the length of the subject of the OCCURS clause.
- Conformance of the data in the object of the OCCURS DEPENDING ON clause to its PICTURE clause

  If the content of the ODO object does not match its PICTURE clause, the program could terminate abnormally. You must ensure that the ODO object correctly specifies the current number of occurrences of table elements.

The following example shows a group item (REC-1) that contains both the subject and object of the OCCURS DEPENDING ON clause. The way the length of the group item is determined depends on whether it is sending or receiving data.

```
WORKING-STORAGE SECTION.
01  MAIN-AREA.
    03 REC-1.
        05 FIELD-1                        PIC 9.
        05 FIELD-2 OCCURS 1 TO 5 TIMES
           DEPENDING ON FIELD-1           PIC X(05).
01  REC-2.
    03 REC-2-DATA                         PIC X(50).
```

If you want to move REC-1 (the sending item in this case) to REC-2, the length of REC-1 is determined immediately before the move, using the current value in FIELD-1. If the content of FIELD-1 conforms to its PICTURE clause (that is, if FIELD-1 contains a zoned decimal item), the move can proceed based on the actual length of REC-1. Otherwise, the result is unpredictable. You must ensure that the ODO object has the correct value before you initiate the move.

When you do a move to REC-1 (the receiving item in this case), the length of REC-1 is determined using the maximum number of occurrences. In this example, five occurrences of FIELD-2, plus FIELD-1, yields a length of 26 bytes. In this case, you do not need to set the ODO object (FIELD-1) before referencing REC-1 as a receiving item. However, the sending field's ODO object (not shown) must be set to a valid numeric value between 1 and 5 for the ODO object of the receiving field to be validly set by the move.

However, if you do a move to REC-1 (again the receiving item) where REC-1 is followed by a variably located group (a type of *complex ODO*), the actual length of REC-1 is calculated immediately before the move, using the current value of the ODO object (FIELD-1). In the following example, REC-1 and REC-2 are in the same record, but REC-2 is not subordinate to REC-1 and is therefore variably located:

```
01  MAIN-AREA
    03 REC-1.
       05 FIELD-1                        PIC 9.
       05 FIELD-3                        PIC 9.
       05 FIELD-2 OCCURS 1 TO 5 TIMES
            DEPENDING ON FIELD-1         PIC X(05).
```

```
   03 REC-2.
      05 FIELD-4 OCCURS 1 TO 5 TIMES
           DEPENDING ON FIELD-3       PIC X(05).
```

The compiler issues a message that lets you know that the actual length was used. This case requires that you set the value of the ODO object before using the group item as a receiving field.

The following example shows how to define a variable-length table when the ODO object (LOCATION-TABLE-LENGTH below) is outside the group:

```
 DATA DIVISION.
 FILE SECTION.
 FD  LOCATION-FILE.
 01  LOCATION-RECORD.
     05  LOC-CODE              PIC XX.
     05  LOC-DESCRIPTION       PIC X(20).
     05  FILLER                PIC X(58).
 WORKING-STORAGE SECTION.
 01  FLAGS.
     05 LOCATION-EOF-FLAG      PIC X(5) VALUE SPACE.
        88 LOCATION-EOF           VALUE "FALSE".
 01  MISC-VALUES.
     05 LOCATION-TABLE-LENGTH  PIC 9(3) VALUE ZERO.
     05 LOCATION-TABLE-MAX     PIC 9(3) VALUE 100.
*****************************************************************
***              L O C A T I O N   T A B L E              ***
***              FILE CONTAINS LOCATION CODES.            ***
*****************************************************************
 01  LOCATION-TABLE.
     05 LOCATION-CODE OCCURS 1 TO 100 TIMES
          DEPENDING ON LOCATION-TABLE-LENGTH   PIC X(80).
```

RELATED CONCEPTS
Appendix E, "Complex OCCURS DEPENDING ON," on page 593

RELATED TASKS
"Assigning values to a variable-length table" on page 75
"Loading a variable-length table"
"Preventing overlay when adding elements to a variable table" on page 595
"Finding the length of data items" on page 110

RELATED REFERENCES
OCCURS DEPENDING ON clause (*COBOL for AIX Language Reference*)

Loading a variable-length table

You can use a *do-until* structure (a TEST AFTER loop) to control the loading of a variable-length table. For example, after the following code runs, LOCATION-TABLE-LENGTH contains the subscript of the last item in the table.

```
 DATA DIVISION.
 FILE SECTION.
 FD  LOCATION-FILE.
 01  LOCATION-RECORD.
     05  LOC-CODE            PIC XX.
     05  LOC-DESCRIPTION     PIC X(20).
     05  FILLER              PIC X(58).
 . . .
 WORKING-STORAGE SECTION.
 01  FLAGS.
     05 LOCATION-EOF-FLAG    PIC X(5) VALUE SPACE.
        88 LOCATION-EOF              VALUE "YES".
 01  MISC-VALUES.
     05 LOCATION-TABLE-LENGTH   PIC 9(3) VALUE ZERO.
     05 LOCATION-TABLE-MAX      PIC 9(3) VALUE 100.
*****************************************************************
***              L O C A T I O N   T A B L E                  ***
***              FILE CONTAINS LOCATION CODES.                ***
*****************************************************************
 01  LOCATION-TABLE.
     05 LOCATION-CODE OCCURS 1 TO 100 TIMES
          DEPENDING ON LOCATION-TABLE-LENGTH   PIC X(80).
 . . .
 PROCEDURE DIVISION.
     . . .
     Perform Test After
         Varying Location-Table-Length From 1 By 1
           Until Location-EOF
           Or Location-Table-Length = Location-Table-Max
       Move Location-Record To
           Location-Code (Location-Table-Length)
       Read Location-File
           At End Set Location-EOF To True
       End-Read
     End-Perform
```

Assigning values to a variable-length table

You can code a VALUE clause for an alphanumeric or national group item that has a subordinate data item that contains the OCCURS clause with the DEPENDING ON phrase. Each subordinate structure that contains the DEPENDING ON phrase is initialized using the maximum number of occurrences.

If you define the entire table by using the DEPENDING ON phrase, all the elements are initialized using the maximum defined value of the ODO (OCCURS DEPENDING ON) object.

If the ODO object is initialized by a VALUE clause, it is logically initialized after the ODO subject has been initialized.

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

For example, in the code above, the ODO subject Y(1) is initialized to 'A', Y(2) to 'B', . . ., Y(5) to 'E', and finally the ODO object X is initialized to 3. Any subsequent reference to TABLE-THREE (such as in a DISPLAY statement) refers to X and the first three elements, Y(1) through Y(3), of the table.

RELATED TASKS
"Assigning values when you define a table (VALUE)" on page 69

RELATED REFERENCES
OCCURS DEPENDING ON clause (*COBOL for AIX Language Reference*)

Searching a table

COBOL provides two search techniques for tables: *serial* and *binary*.

To do serial searches, use SEARCH and indexing. For variable-length tables, you can use PERFORM with subscripting or indexing.

To do binary searches, use SEARCH ALL and indexing.

A binary search can be considerably more efficient than a serial search. For a serial search, the number of comparisons is of the order of *n*, the number of entries in the table. For a binary search, the number of comparisons is of the order of only the logarithm (base 2) of *n*. A binary search, however, requires that the table items already be sorted.

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

Doing a serial search (SEARCH)

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

The conditions in the WHEN phrase are evaluated in the order in which they appear:
- If none of the conditions is satisfied, the index is increased to correspond to the next table element, and the WHEN conditions are evaluated again.
- If one of the WHEN conditions is satisfied, the search ends. The index remains pointing to the table element that satisfied the condition.
- If the entire table has been searched and no conditions were met, the AT END imperative statement is executed if there is one. If you did not code AT END, control passes to the next statement in the program.

You can reference only one level of a table (a table element) with each SEARCH statement. To search multiple levels of a table, use nested SEARCH statements. Delimit each nested SEARCH statement with END-SEARCH.

Performance: If the found condition comes after some intermediate point in the table, you can speed up the search by using the SET statement to set the index to begin the search after that point. Arranging the table so that the data used most often is at the beginning of the table also enables more efficient serial searching. If the table is large and is presorted, a binary search is more efficient.

"Example: serial search"

RELATED REFERENCES
SEARCH statement (*COBOL for AIX Language Reference*)

Example: serial search

The following example shows how you might find a particular string in the innermost table of a three-dimensional table.

Each dimension of the table has its own index (set to 1, 4, and 1, respectively). The innermost table (TABLE-ENTRY3) has an ascending key.

```
01  TABLE-ONE.
    05 TABLE-ENTRY1 OCCURS 10 TIMES
          INDEXED BY TE1-INDEX.
       10 TABLE-ENTRY2 OCCURS 10 TIMES
             INDEXED BY TE2-INDEX.
          15 TABLE-ENTRY3 OCCURS 5 TIMES
                ASCENDING KEY IS KEY1
                INDEXED BY TE3-INDEX.
             20 KEY1               PIC X(5).
             20 KEY2               PIC X(10).
. . .
PROCEDURE DIVISION.
    . . .
    SET TE1-INDEX TO 1
    SET TE2-INDEX TO 4
    SET TE3-INDEX TO 1
    MOVE "A1234" TO KEY1 (TE1-INDEX, TE2-INDEX, TE3-INDEX + 2)
    MOVE "AAAAAAAA00" TO KEY2 (TE1-INDEX, TE2-INDEX, TE3-INDEX + 2)
    . . .
    SEARCH TABLE-ENTRY3
      AT END
        MOVE 4 TO RETURN-CODE
      WHEN TABLE-ENTRY3(TE1-INDEX, TE2-INDEX, TE3-INDEX)
          = "A1234AAAAAAAA00"
        MOVE 0 TO RETURN-CODE
    END-SEARCH
```

Values after execution:

```
TE1-INDEX = 1
TE2-INDEX = 4
TE3-INDEX points to the TABLE-ENTRY3 item
          that equals "A1234AAAAAAAA00"
RETURN-CODE = 0
```

Doing a binary search (SEARCH ALL)

If you use SEARCH ALL to do a binary search, you do not need to set the index before you begin. The index is always the one that is associated with the first index-name in the OCCURS clause. The index varies during execution to maximize the search efficiency.

| To use the SEARCH ALL statement to search a table, the table must specify the ASCENDING or DESCENDING KEY phrases of the OCCURS clause, or both, and must already be ordered on the key or keys that are specified in the ASCENDING and DESCENDING KEY phrases.

| In the WHEN phrase of the SEARCH ALL statement, you can test any key that is named in the ASCENDING or DESCENDING KEY phrases for the table, but you must test all preceding keys, if any. The test must be an equal-to condition, and the WHEN phrase must specify either a key (subscripted by the first index-name associated with the table) or a condition-name that is associated with the key. The WHEN condition can be a compound condition that is formed from simple conditions that use AND as the only logical connective.

| Each key and its object of comparison must be compatible according to the rules for comparison of data items. Note though that if a key is compared to a national literal or identifier, the key must be a national data item.

"Example: binary search" on page 78

RELATED TASKS
"Defining a table (OCCURS)" on page 61

RELATED REFERENCES
SEARCH statement (*COBOL for AIX Language Reference*)
General relation conditions (*COBOL for AIX Language Reference*)

Example: binary search

The following example shows how you can code a binary search of a table.

Suppose you define a table that contains 90 elements of 40 bytes each, and three keys. The primary and secondary keys (KEY-1 and KEY-2) are in ascending order, but the least significant key (KEY-3) is in descending order:

```
01  TABLE-A.
    05 TABLE-ENTRY OCCURS 90 TIMES
           ASCENDING KEY-1, KEY-2
           DESCENDING KEY-3
           INDEXED BY INDX-1.
       10 PART-1       PIC 99.
       10 KEY-1        PIC 9(5).
       10 PART-2       PIC 9(6).
       10 KEY-2        PIC 9(4).
       10 PART-3       PIC 9(18).
       10 KEY-3        PIC 9(5).
```

You can search this table by using the following statements:

```
SEARCH ALL TABLE-ENTRY
  AT END
    PERFORM NOENTRY
  WHEN KEY-1 (INDX-1) = VALUE-1 AND
       KEY-2 (INDX-1) = VALUE-2 AND
       KEY-3 (INDX-1) = VALUE-3
    MOVE PART-1 (INDX-1) TO OUTPUT-AREA
END-SEARCH
```

If an entry is found in which each of the three keys is equal to the value to which it is compared (VALUE-1, VALUE-2, and VALUE-3, respectively), PART-1 of that entry is moved to OUTPUT-AREA. If no matching key is found in the entries in TABLE-A, the NOENTRY routine is performed.

Processing table items using intrinsic functions

| You can use intrinsic functions to process alphabetic, alphanumeric, national, or
| numeric table items. (You can process DBCS data items only with the NATIONAL-OF
| intrinsic function.) The data descriptions of the table items must be compatible with the requirements for the function arguments.

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

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

You might often need to iteratively process the data in tables. For intrinsic functions that accept multiple arguments, you can use the subscript ALL to reference all the items in the table or in a single dimension of the table. The iteration is handled automatically, which can make your code shorter and simpler.

You can mix scalars and array arguments for functions that accept multiple arguments:

```
Compute Table-Median = Function Median(Arg1 Table-One(ALL))
```

"Example: processing tables using intrinsic functions"

RELATED TASKS
"Using intrinsic functions (built-in functions)" on page 36
"Converting data items (intrinsic functions)" on page 104
"Evaluating data items (intrinsic functions)" on page 107

RELATED REFERENCES
Intrinsic functions (*COBOL for AIX Language Reference*)

Example: processing tables using intrinsic functions

These examples show how you can apply an intrinsic function to some or all of the elements in a table by using the `ALL` subscript.

Assuming that `Table-Two` is a 2 x 3 x 2 array, the following statement adds the values in elements `Table-Two(1,3,1)`, `Table-Two(1,3,2)`, `Table-Two(2,3,1)`, and `Table-Two(2,3,2)`:

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

The following example computes various salary values for all the employees whose salaries are encoded in `Employee-Table`:

```
01  Employee-Table.
    05 Emp-Count      Pic s9(4) usage binary.
    05 Emp-Record     Occurs 1 to 500 times
                        depending on Emp-Count.
       10 Emp-Name    Pic x(20).
       10 Emp-Idme    Pic 9(9).
       10 Emp-Salary  Pic 9(7)v99.
. . .
Procedure Division.
    Compute Max-Salary    = Function Max(Emp-Salary(ALL))
    Compute I             = Function Ord-Max(Emp-Salary(ALL))
    Compute Avg-Salary    = Function Mean(Emp-Salary(ALL))
    Compute Salary-Range  = Function Range(Emp-Salary(ALL))
    Compute Total-Payroll = Function Sum(Emp-Salary(ALL))
```

Chapter 5. Selecting and repeating program actions

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

These controls include the `IF`, `EVALUATE`, and `PERFORM` statements, and the use of switches and flags.

RELATED TASKS
"Selecting program actions"
"Repeating program actions" on page 89

Selecting program actions

You can provide for different program actions depending on the tested value of one or more data items.

The `IF` and `EVALUATE` statements in COBOL test one or more data items by means of a conditional expression.

RELATED TASKS
"Coding a choice of actions"
"Coding conditional expressions" on page 86

RELATED REFERENCES
IF statement (*COBOL for AIX Language Reference*)
EVALUATE statement (*COBOL for AIX Language Reference*)

Coding a choice of actions

Use `IF . . . ELSE` to code a choice between two processing actions. (The word `THEN` is optional.) Use the `EVALUATE` statement to code a choice among three or more possible actions.

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

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

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

Such coding is suitable for simple cases. For complex logic, you probably need to use the `ELSE` clause. For example, suppose you have nested `IF` statements in which there is an action for only one of the processing choices. You could use the `ELSE` clause and code the null branch of the `IF` statement with the `CONTINUE` statement:

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

The `EVALUATE` statement is an expanded form of the `IF` statement that allows you to avoid nesting `IF` statements, a common source of logic errors and debugging problems.

RELATED TASKS
"Using nested IF statements"
"Using the EVALUATE statement" on page 83
"Coding conditional expressions" on page 86

Using nested IF statements

When an `IF` statement contains an `IF` statement as one of its possible branches, the `IF` statements are said to be *nested*. Theoretically, there is no limit to the depth of nested `IF` statements.

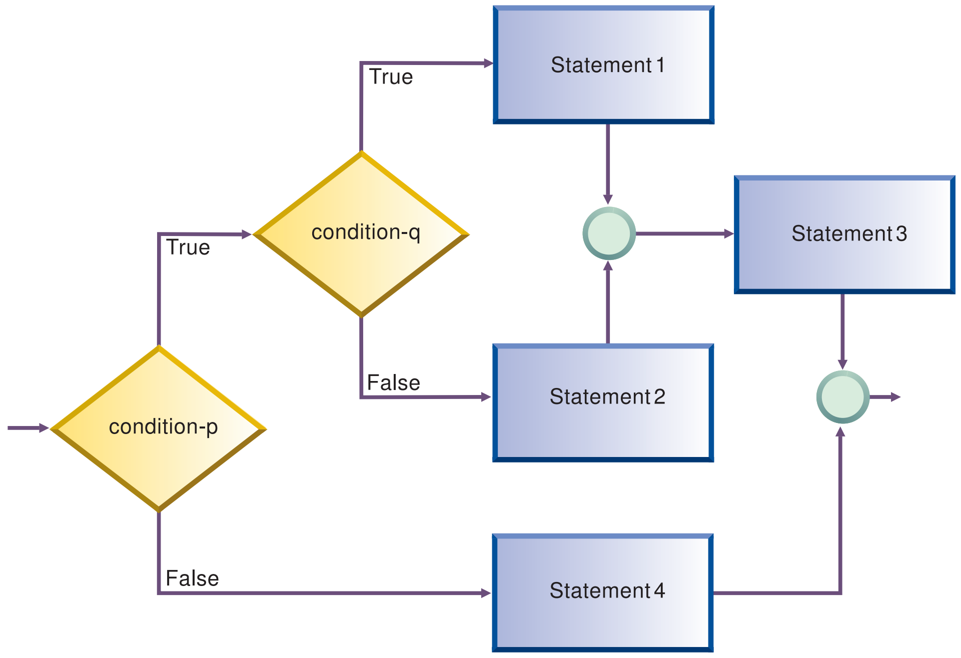
However, use nested `IF` statements sparingly. The logic can be difficult to follow, although explicit scope terminators and indentation help. When a program has to test a variable for more than two values, `EVALUATE` is probably a better choice.

The following pseudocode depicts a nested `IF` statement:

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

In the pseudocode above, an `IF` statement and a sequential structure are nested in one branch of the outer `IF`. In this structure, the `END-IF` that closes the nested `IF` is very important. Use `END-IF` instead of a period, because a period would end the outer `IF` structure also.

The following figure shows the logic structure of the pseudocode above.



RELATED TASKS
"Coding a choice of actions" on page 81

RELATED REFERENCES
Explicit scope terminators (*COBOL for AIX Language Reference*)

Using the EVALUATE statement

You can use the EVALUATE statement instead of a series of nested IF statements to test several conditions and specify a different action for each. Thus you can use the EVALUATE statement to implement a *case structure* or decision table.

You can also use the EVALUATE statement to cause multiple conditions to lead to the same processing, as shown in these examples:

"Example: EVALUATE using THRU phrase" on page 84
"Example: EVALUATE using multiple WHEN phrases" on page 84

In an EVALUATE statement, the operands before the WHEN phrase are referred to as *selection subjects*, and the operands in the WHEN phrase are called the *selection objects*. Selection subjects can be identifiers, literals, conditional expressions, or the word TRUE or FALSE. Selection objects can be identifiers, literals, conditional or arithmetic expressions, or the word TRUE, FALSE, or ANY.

You can separate multiple selection subjects with the ALSO phrase. You can separate multiple selection objects with the ALSO phrase. The number of selection objects within each set of selection objects must be equal to the number of selection subjects, as shown in this example:

"Example: EVALUATE testing several conditions" on page 85

Identifiers, literals, or arithmetic expressions that appear within a selection object must be valid operands for comparison to the corresponding operand in the set of selection subjects. Conditions or the word TRUE or FALSE that appear in a selection object must correspond to a conditional expression or the word TRUE or FALSE in the set of selection subjects. (You can use the word ANY as a selection object to correspond to any type of selection subject.)

The execution of the EVALUATE statement ends when one of the following conditions occurs:

- The statements associated with the selected WHEN phrase are performed.
- The statements associated with the WHEN OTHER phrase are performed.
- No WHEN conditions are satisfied.

WHEN phrases are tested in the order that they appear in the source program. Therefore, you should order these phrases for the best performance. First code the WHEN phrase that contains selection objects that are most likely to be satisfied, then the next most likely, and so on. An exception is the WHEN OTHER phrase, which must come last.

RELATED TASKS
"Coding a choice of actions" on page 81

RELATED REFERENCES
EVALUATE statement (*COBOL for AIX Language Reference*)
General relation conditions (*COBOL for AIX Language Reference*)

Example: EVALUATE using THRU phrase:

This example shows how you can code several conditions in a range of values to lead to the same processing action by coding the THRU phrase. Operands in a THRU phrase must be of the same class.

In this example, CARPOOL-SIZE is the *selection subject*; 1, 2, and 3 THRU 6 are the *selection objects*:

```
EVALUATE CARPOOL-SIZE
  WHEN 1
    MOVE "SINGLE" TO PRINT-CARPOOL-STATUS
  WHEN 2
    MOVE "COUPLE" TO PRINT-CARPOOL-STATUS
  WHEN 3 THRU 6
    MOVE "SMALL GROUP" TO PRINT-CARPOOL STATUS
  WHEN OTHER
    MOVE "BIG GROUP" TO PRINT-CARPOOL STATUS
END-EVALUATE
```

The following nested IF statements represent the same logic:

```
IF CARPOOL-SIZE = 1 THEN
  MOVE "SINGLE" TO PRINT-CARPOOL-STATUS
ELSE
  IF CARPOOL-SIZE = 2 THEN
    MOVE "COUPLE" TO PRINT-CARPOOL-STATUS
  ELSE
    IF CARPOOL-SIZE >= 3 and CARPOOL-SIZE <= 6 THEN
      MOVE "SMALL GROUP" TO PRINT-CARPOOL-STATUS
    ELSE
      MOVE "BIG GROUP" TO PRINT-CARPOOL-STATUS
    END-IF
  END-IF
END-IF
```

Example: EVALUATE using multiple WHEN phrases:

The following example shows that you can code multiple WHEN phrases if several conditions should lead to the same action. Doing so gives you more flexibility than using only the THRU phrase, because the conditions do not have to evaluate to values in a range nor have the same class.

```
EVALUATE MARITAL-CODE
  WHEN "M"
    ADD 2 TO PEOPLE-COUNT
  WHEN "S"
  WHEN "D"
  WHEN "W"
    ADD 1 TO PEOPLE-COUNT
END-EVALUATE
```

The following nested IF statements represent the same logic:

```
IF MARITAL-CODE = "M" THEN
  ADD 2 TO PEOPLE-COUNT
ELSE
  IF MARITAL-CODE = "S" OR
     MARITAL-CODE = "D" OR
     MARITAL-CODE = "W" THEN
       ADD 1 TO PEOPLE-COUNT
  END-IF
END-IF
```

Example: EVALUATE testing several conditions:

This example shows the use of the ALSO phrase to separate two selection subjects (True ALSO True) and to separate the two corresponding selection objects within each set of selection objects (for example, When A + B < 10 Also C = 10).

Both selection objects in a WHEN phrase must satisfy the TRUE, TRUE condition before the associated action is performed. If both objects do not evaluate to TRUE, the next WHEN phrase is processed.

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

Coding conditional expressions

Using the `IF` and `EVALUATE` statements, you can code program actions that will be performed depending on the truth value of a conditional expression.

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

- Relation conditions, such as:
  - Numeric comparisons
  - Alphanumeric comparisons
  - DBCS comparisons
  - National comparisons
- Class conditions; for example, to test whether a data item:
  - `IS NUMERIC`
  - `IS ALPHABETIC`
  - `IS DBCS`
  - `IS KANJI`
  - `IS NOT KANJI`
- Condition-name conditions, to test the value of a conditional variable that you define
- Sign conditions, to test whether a numeric operand `IS POSITIVE`, `NEGATIVE`, or `ZERO`
- Switch-status conditions, to test the status of UPSI switches that you name in the `SPECIAL-NAMES` paragraph
- Complex conditions, such as:
  - Negated conditions; for example, `NOT (A IS EQUAL TO B)`
  - Combined conditions (conditions combined with logical operators `AND` or `OR`)

RELATED CONCEPTS
"Switches and flags" on page 87

RELATED TASKS
"Defining switches and flags" on page 87
"Resetting switches and flags" on page 88
"Checking for incompatible data (numeric class test)" on page 51
"Comparing national (UTF-16) data" on page 186
"Testing for valid DBCS characters" on page 191

RELATED REFERENCES
"UPSI" on page 305
General relation conditions (*COBOL for AIX Language Reference*)

Class condition (*COBOL for AIX Language Reference*)
Rules for condition-name entries (*COBOL for AIX Language Reference*)
Sign condition (*COBOL for AIX Language Reference*)
Combined conditions (*COBOL for AIX Language Reference*)

Switches and flags

Some program decisions are based on whether the value of a data item is true or false, on or off, yes or no. Control these two-way decisions by using level-88 items with meaningful names (*condition-names*) to act as switches.

Other program decisions depend on the particular value or range of values of a data item. When you use condition-names to give more than just on or off values to a field, the field is generally referred to as a *flag*.

Flags and switches make your code easier to change. If you need to change the values for a condition, you have to change only the value of that level-88 condition-name.

For example, suppose a program uses a condition-name to test a field for a given salary range. If the program must be changed to check for a different salary range, you need to change only the value of the condition-name in the `DATA DIVISION`. You do not need to make changes in the `PROCEDURE DIVISION`.

RELATED TASKS
"Defining switches and flags"
"Resetting switches and flags" on page 88

Defining switches and flags

In the `DATA DIVISION`, define level-88 items that will act as switches or flags, and give them meaningful names.

To test for more than two values with flags, assign more than one condition-name to a field by using multiple level-88 items.

The reader can easily follow your code if you choose meaningful condition-names and if the values assigned to them have some association with logical values.

"Example: switches"
"Example: flags" on page 88

Example: switches

The following examples show how you can use level-88 items to test for various binary-valued (on-off) conditions in your program.

For example, to test for the end-of-file condition for an input file named Transaction-File, you can use the following data definitions:

```
Working-Storage Section.
01  Switches.
    05  Transaction-EOF-Switch  Pic X  value space.
        88  Transaction-EOF     value "y".
```

The level-88 description says that a condition named `Transaction-EOF` is turned on when `Transaction-EOF-Switch` has value 'y'. Referencing `Transaction-EOF` in the `PROCEDURE DIVISION` expresses the same condition as testing `Transaction-EOF-`

Switch = "y". For example, the following statement causes a report to be printed only if Transaction-EOF-Switch has been set to 'y':

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

Example: flags

The following examples show how you can use several level-88 items together with an EVALUATE statement to determine which of several conditions in a program is true.

Consider for example a program that updates a master file. The updates are read from a transaction file. The records in the file contain a field that indicates which of the three functions is to be performed: add, change, or delete. In the record description of the input file, code a field for the function code using level-88 items:

```
01  Transaction-Input Record
    05  Transaction-Type        Pic X.
        88  Add-Transaction      Value "A".
        88  Change-Transaction   Value "C".
        88  Delete-Transaction   Value "D".
```

The code in the PROCEDURE DIVISION for testing these condition-names to determine which function is to be performed might look like this:

```
Evaluate True
  When Add-Transaction
    Perform Add-Master-Record-Paragraph
  When Change-Transaction
    Perform Update-Existing-Record-Paragraph
  When Delete-Transaction
    Perform Delete-Master-Record-Paragraph
End-Evaluate
```

Resetting switches and flags

Throughout your program, you might need to reset switches or flags to the original values they had in their data descriptions. To do so, either use a SET statement or define a data item to move to the switch or flag.

When you use the SET *condition-name* TO TRUE statement, the switch or flag is set to the original value that it was assigned in its data description. For a level-88 item that has multiple values, SET *condition-name* TO TRUE assigns the first value (A in the example below):

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

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

"Example: set switch on"
"Example: set switch off" on page 89

Example: set switch on

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

For example, the SET statement in the following example has the same effect as coding the statement Move "y" to Transaction-EOF-Switch:

```
01  Switches
    05  Transaction-EOF-Switch   Pic X  Value space.
        88  Transaction-EOF             Value "y".
. . .
Procedure Division.
000-Do-Main-Logic.
    Perform 100-Initialize-Paragraph
    Read Update-Transaction-File
      At End Set Transaction-EOF to True
    End-Read
```

The following example shows how to assign a value to a field in an output record based on the transaction code of an input record:

```
01  Input-Record.
    05  Transaction-Type        Pic X(9).
01  Data-Record-Out.
    05  Data-Record-Type        Pic X.
        88 Record-Is-Active         Value "A".
        88 Record-Is-Suspended      Value "S".
        88 Record-Is-Deleted        Value "D".
    05  Key-Field               Pic X(5).
. . .
Procedure Division.
    Evaluate Transaction-Type of Input-Record
      When "ACTIVE"
        Set Record-Is-Active to TRUE
      When "SUSPENDED"
        Set Record-Is-Suspended to TRUE
      When "DELETED"
        Set Record-Is-Deleted to TRUE
    End-Evaluate
```

Example: set switch off

The following example shows how you can set a switch off by coding a MOVE statement that moves a value to a level-88 item.

For example, you can use a data item called SWITCH-OFF to set an on-off switch to off, as in the following code, which resets a switch to indicate that end-of-file has not been reached:

```
01  Switches
    05  Transaction-EOF-Switch       Pic X  Value space.
        88  Transaction-EOF                 Value "y".
01  SWITCH-OFF                       Pic X  Value "n".
. . .
Procedure Division.
    . . .
    Move SWITCH-OFF to Transaction-EOF-Switch
```

Repeating program actions

Use a PERFORM statement to repeat the same code (that is, loop) either a specified number of times or based on the outcome of a decision.

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

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

RELATED TASKS
"Choosing inline or out-of-line PERFORM"
"Coding a loop" on page 91
"Looping through a table" on page 92
"Executing multiple paragraphs or sections" on page 92

RELATED REFERENCES
PERFORM statement (*COBOL for AIX Language Reference*)

Choosing inline or out-of-line PERFORM

An inline PERFORM is an imperative statement that is executed in the normal flow of a program; an out-of-line PERFORM entails a branch to a named paragraph and an implicit return from that paragraph.

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

- Is the PERFORM statement used in several places?

  Use an out-of-line PERFORM when you want to use the same portion of code in several places in your program.
- Which placement of the statement will be easier to read?

  If the code to be performed is short, an inline PERFORM can be easier to read. But if the code extends over several screens, the logical flow of the program might be clearer if you use an out-of-line PERFORM. (Each paragraph in structured programming should perform one logical function, however.)
- What are the efficiency tradeoffs?

  An inline PERFORM avoids the overhead of branching that occurs with an out-of-line PERFORM. But even out-of-line PERFORM coding can improve code optimization, so efficiency gains should not be overemphasized.

In the 1974 COBOL standard, the PERFORM statement is out-of-line and thus requires a branch to a separate paragraph and an implicit return. If the performed paragraph is in the subsequent sequential flow of your program, it is also executed in that logic flow. To avoid this additional execution, place the paragraph outside the normal sequential flow (for example, after the GOBACK) or code a branch around it.

The subject of an inline PERFORM is an imperative statement. Therefore, you must code statements (other than imperative statements) within an inline PERFORM with explicit scope terminators.

"Example: inline PERFORM statement"

Example: inline PERFORM statement

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

```
   Perform 100-Initialize-Paragraph
* The following statement is an inline PERFORM:
   Perform Until Transaction-EOF
      Read Update-Transaction-File Into WS-Transaction-Record
         At End
            Set Transaction-EOF To True
         Not At End
            Perform 200-Edit-Update-Transaction
            If No-Errors
               Perform 300-Update-Commuter-Record
            Else
               Perform 400-Print-Transaction-Errors
* End-If is a required scope terminator
            End-If
            Perform 410-Re-Initialize-Fields
* End-Read is a required scope terminator
         End-Read
   End-Perform
```

Coding a loop

Use the PERFORM . . . TIMES statement to execute a paragraph a specified number of times.

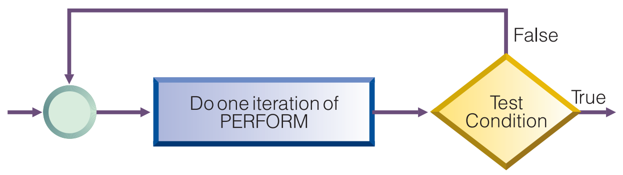
```
PERFORM 010-PROCESS-ONE-MONTH 12 TIMES
INSPECT . . .
```

In the example above, when control reaches the PERFORM statement, the code for the paragraph 010-PROCESS-ONE-MONTH is executed 12 times before control is transferred to the INSPECT statement.

Use the PERFORM . . . UNTIL statement to execute a paragraph until a condition you choose is satisfied. You can use either of the following forms:

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

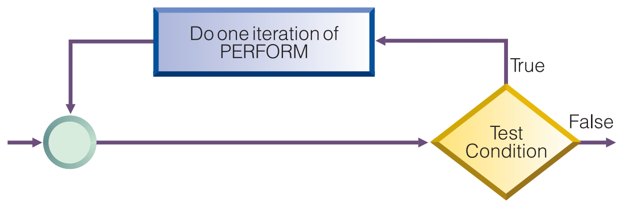
Use the PERFORM . . . WITH TEST AFTER . . . UNTIL statement if you want to execute the paragraph at least once, and test before any subsequent execution. This statement is equivalent to a do-until structure:



In the following example, the implicit WITH TEST BEFORE phrase provides a do-while structure:

```
PERFORM  010-PROCESS-ONE-MONTH
  UNTIL MONTH GREATER THAN 12
INSPECT . . .
```

| When control reaches the PERFORM statement, the condition MONTH GREATER THAN 12 is tested. If the condition is satisfied, control is transferred to the INSPECT statement. If the condition is not satisfied, 010-PROCESS-ONE-MONTH is executed, and the condition is tested again. This cycle continues until the condition tests as true. (To make your program easier to read, you might want to code the WITH TEST BEFORE clause.)



Looping through a table

You can use the PERFORM . . . VARYING statement to initialize a table. In this form of the PERFORM statement, a variable is increased or decreased and tested until a condition is satisfied.

Thus you use the PERFORM statement to control looping through a table. You can use either of these forms:

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

The following section of code shows an example of looping through a table to check for invalid data:

```
PERFORM TEST AFTER VARYING WS-DATA-IX
      FROM 1 BY 1 UNTIL WS-DATA-IX = 12
   IF WS-DATA (WS-DATA-IX) EQUALS SPACES
      SET SERIOUS-ERROR TO TRUE
      DISPLAY ELEMENT-NUM-MSG5
   END-IF
END-PERFORM
INSPECT . . .
```

When control reaches the PERFORM statement above, WS-DATA-IX is set equal to 1 and the PERFORM statement is executed. Then the condition WS-DATA-IX = 12 is tested. If the condition is true, control drops through to the INSPECT statement. If the condition is false, WS-DATA-IX is increased by 1, the PERFORM statement is executed, and the condition is tested again. This cycle of execution and testing continues until WS-DATA-IX is equal to 12.

The loop above controls input-checking for the 12 fields of item WS-DATA. Empty fields are not allowed in the application, so the section of code loops and issues error messages as appropriate.

Executing multiple paragraphs or sections

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

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

RELATED TASKS
"Processing table items using intrinsic functions" on page 78

Chapter 6. Handling strings

COBOL provides language constructs for performing many different operations on string data items.

For example, you can:
- Join or split data items.
- Manipulate null-terminated strings, such as count or move characters.
- Refer to substrings by their ordinal position and, if needed, length.
- Tally and replace data items, such as count the number of times a specific character occurs in a data item.
- Convert data items, such as change to uppercase or lowercase.
- Evaluate data items, such as determine the length of a data item.

RELATED TASKS
"Joining data items (STRING)"
"Splitting data items (UNSTRING)" on page 95
"Manipulating null-terminated strings" on page 98
"Referring to substrings of data items" on page 99
"Tallying and replacing data items (INSPECT)" on page 103
"Converting data items (intrinsic functions)" on page 104
"Evaluating data items (intrinsic functions)" on page 107
Chapter 10, "Processing data in an international environment," on page 169

Joining data items (STRING)

Use the `STRING` statement to join all or parts of several data items or literals into one data item. One `STRING` statement can take the place of several `MOVE` statements.

The `STRING` statement transfers data into a receiving data item in the order that you indicate. In the `STRING` statement you also specify:
- A delimiter for each set of sending fields that, if encountered, causes those sending fields to stop being transferred (`DELIMITED BY` phrase)
- (Optional) Action to be taken if the receiving field is filled before all of the sending data has been processed (`ON OVERFLOW` phrase)
- (Optional) An integer data item that indicates the leftmost character position within the receiving field into which data should be transferred (`WITH POINTER` phrase)

The receiving data item must not be an edited item, or a display or national floating-point item. If the receiving data item has:
- `USAGE DISPLAY`, each identifier in the statement except the `POINTER` identifier must have `USAGE DISPLAY`, and each literal in the statement must be alphanumeric
- `USAGE NATIONAL`, each identifier in the statement except the `POINTER` identifier must have `USAGE NATIONAL`, and each literal in the statement must be national
- `USAGE DISPLAY-1`, each identifier in the statement except the `POINTER` identifier must have `USAGE DISPLAY-1`, and each literal in the statement must be DBCS

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

"Example: STRING statement"

RELATED TASKS
"Handling errors in joining and splitting strings" on page 157

RELATED REFERENCES
STRING statement (*COBOL for AIX Language Reference*)

Example: STRING statement

The following example shows the STRING statement selecting and formatting information from a record into an output line.

The FILE SECTION defines the following record:

```
01  RCD-01.
    05  CUST-INFO.
        10  CUST-NAME    PIC X(15).
        10  CUST-ADDR    PIC X(35).
    05  BILL-INFO.
        10  INV-NO       PIC X(6).
        10  INV-AMT      PIC $$,$$$.99.
        10  AMT-PAID     PIC $$,$$$.99.
        10  DATE-PAID    PIC X(8).
        10  BAL-DUE      PIC $$,$$$.99.
        10  DATE-DUE     PIC X(8).
```

The WORKING-STORAGE SECTION defines the following fields:

```
77  RPT-LINE             PIC X(120).
77  LINE-POS             PIC S9(3).
77  LINE-NO              PIC 9(5) VALUE 1.
77  DEC-POINT            PIC X VALUE ".".
```

The record RCD-01 contains the following information (the symbol *b* indicates a blank space):

```
J.B.bSMITHbbbbb
444bSPRINGbST.,bCHICAGO,bILL.bbbbbb
A14275
$4,736.85
$2,400.00
09/22/76
$2,336.85
10/22/76
```

In the PROCEDURE DIVISION, these settings occur before the STRING statement:

- RPT-LINE is set to SPACES.
- LINE-POS, the data item to be used as the POINTER field, is set to 4.

Here is the STRING statement:

```
STRING
   LINE-NO SPACE CUST-INFO INV-NO SPACE DATE-DUE SPACE
      DELIMITED BY SIZE
   BAL-DUE
      DELIMITED BY DEC-POINT
   INTO RPT-LINE
   WITH POINTER LINE-POS.
```

Because the POINTER field LINE-POS has value 4 before the STRING statement is performed, data is moved into the receiving field RPT-LINE beginning at character position 4. Characters in positions 1 through 3 are unchanged.

The sending items that specify DELIMITED BY SIZE are moved in their entirety to the receiving field. Because BAL-DUE is delimited by DEC-POINT, the moving of BAL-DUE to the receiving field stops when a decimal point (the value of DEC-POINT) is encountered.

STRING results

When the STRING statement is performed, items are moved into RPT-LINE as shown in the table below.

Item	Positions
LINE-NO	4 - 8
Space	9
CUST-INFO	10 - 59
INV-NO	60 - 65
Space	66
DATE-DUE	67 - 74
Space	75
Portion of BAL-DUE that precedes the decimal point	76 - 81

After the STRING statement is performed, the value of LINE-POS is 82, and RPT-LINE has the values shown below.

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

Splitting data items (UNSTRING)

Use the UNSTRING statement to split a sending field into several receiving fields. One UNSTRING statement can take the place of several MOVE statements.

In the UNSTRING statement you can specify:

- Delimiters that, when one of them is encountered in the sending field, cause the current receiving field to stop receiving and the next, if any, to begin receiving (DELIMITED BY phrase)
- A field for the delimiter that, when encountered in the sending field, causes the current receiving field to stop receiving (DELIMITER IN phrase)
- An integer data item that stores the number of characters placed in the current receiving field (COUNT IN phrase)
- An integer data item that indicates the leftmost character position within the sending field at which UNSTRING processing should begin (WITH POINTER phrase)
- An integer data item that stores a tally of the number of receiving fields that are acted on (TALLYING IN phrase)

- Action to be taken if all of the receiving fields are filled before the end of the sending data item is reached (ON OVERFLOW phrase)

The sending data item and the delimiters in the DELIMITED BY phrase must be of category alphabetic, alphanumeric, alphanumeric-edited, DBCS, national, or national-edited.

Receiving data items can be of category alphabetic, alphanumeric, numeric, DBCS, or national. If numeric, a receiving data item must be zoned decimal or national decimal. If a receiving data item has:

- USAGE DISPLAY, the sending item and each delimiter item in the statement must have USAGE DISPLAY, and each literal in the statement must be alphanumeric
- USAGE NATIONAL, the sending item and each delimiter item in the statement must have USAGE NATIONAL, and each literal in the statement must be national
- USAGE DISPLAY-1, the sending item and each delimiter item in the statement must have USAGE DISPLAY-1, and each literal in the statement must be DBCS

"Example: UNSTRING statement"

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 173

RELATED TASKS
"Handling errors in joining and splitting strings" on page 157

RELATED REFERENCES
UNSTRING statement (*COBOL for AIX Language Reference*)
Classes and categories of data (*COBOL for AIX Language Reference*)

Example: UNSTRING statement

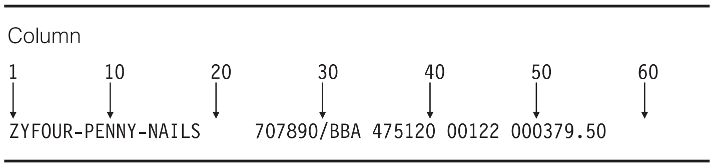
The following example shows the UNSTRING statement transferring selected information from an input record. Some information is organized for printing and some for further processing.

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.



```
ZYFOUR-PENNY-NAILS     707890/BBA 475120 00122 000379.50
```

Here is the UNSTRING statement:

```
* Move subfields of INV-RCD to the subfields of DISPLAY-REC
* and WORK-REC:
     UNSTRING INV-RCD
       DELIMITED BY ALL SPACES  OR "/"  OR DBY-1
       INTO ITEM-NAME    COUNT IN CTR-1
            INV-NO       DELIMITER IN DLTR-1  COUNT IN CTR-2
            INV-CLASS
            M-UNITS      COUNT IN CTR-3
            FIELD-A
            DISPLAY-DOLS DELIMITER IN DLTR-2  COUNT IN CTR-4
       WITH POINTER CHAR-CT
       TALLYING IN  FLDS-FILLED
       ON OVERFLOW  GO TO UNSTRING-COMPLETE.
```

Because the POINTER field CHAR-CT has value 3 before the UNSTRING statement is performed, the two character positions of the CONTROL-CHARS field in INV-RCD are ignored.

UNSTRING results

When the UNSTRING statement is performed, the following steps take place:

1. Positions 3 through 18 (FOUR-PENNY-NAILS) of INV-RCD are placed in ITEM-NAME, left justified in the area, and the four unused character positions are padded with spaces. The value 16 is placed in CTR-1.
2. Because ALL SPACES is coded as a delimiter, the five contiguous space characters in positions 19 through 23 are considered to be one occurrence of the delimiter.
3. Positions 24 through 29 (707890) are placed in INV-NO. The delimiter character slash (/) is placed in DLTR-1, and the value 6 is placed in CTR-2.

4. Positions 31 through 33 (BBA) are placed in INV-CLASS. The delimiter is SPACE, but because no field has been defined as a receiving area for delimiters, the space in position 34 is bypassed.
5. Positions 35 through 40 (475120) are placed in M-UNITS. The value 6 is placed in CTR-3. The delimiter is SPACE, but because no field has been defined as a receiving area for delimiters, the space in position 41 is bypassed.
6. Positions 42 through 46 (00122) are placed in FIELD-A and right justified in the area. The high-order digit position is filled with a zero (0). The delimiter is SPACE, but because no field was defined as a receiving area for delimiters, the space in position 47 is bypassed.
7. Positions 48 through 53 (000379) are placed in DISPLAY-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"

RELATED TASKS
"Handling null-terminated strings" on page 486

RELATED REFERENCES
Alphanumeric literals (*COBOL for AIX Language Reference*)

Example: null-terminated strings

The following example shows several ways in which you can process null-terminated strings.

```
 01  L pic X(20) value z'ab'.
 01  M pic X(20) value z'cd'.
 01  N pic X(20).
 01  N-Length pic 99 value zero.
 01  Y pic X(13) value 'Hello, World!'.
 . . .
* Display null-terminated string:
     Inspect N tallying N-length
       for characters before initial x'00'
     Display 'N: ' N(1:N-Length) ' Length: ' N-Length
     . . .
* Move null-terminated string to alphanumeric, strip null:
     Unstring N delimited by X'00' into X
     . . .
* Create null-terminated string:
     String Y     delimited by size
            X'00' delimited by size
            into N.
     . . .
* Concatenate two null-terminated strings to produce another:
     String L     delimited by x'00'
            M     delimited by x'00'
            X'00' delimited by size
            into N.
```

Referring to substrings of data items

| Refer to a substring of a data item that has USAGE DISPLAY, DISPLAY-1, or NATIONAL
| by using a reference modifier. You can also refer to a substring of an alphanumeric
| or national character string that is returned by an intrinsic function by using a
| reference modifier.

The following example shows how to use a reference modifier to refer to a twenty-character substring of a data item called Customer-Record:

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

You code a reference modifier in parentheses immediately after the data item. As the example shows, a reference modifier can contain two values that are separated by a colon, in this order:

1. Ordinal position (from the left) of the character that you want the substring to start with

2. (Optional) Length of the desired substring in *character positions*

The reference-modifier position and length for an item that has USAGE DISPLAY are expressed in terms of single-byte characters. The reference-modifier position and length for items that have USAGE DISPLAY-1 or NATIONAL are expressed in terms of DBCS character positions and national character positions, respectively.

If you omit the length in a reference modifier (coding only the ordinal position of the first character, followed by a colon), the substring extends to the end of the item. Omit the length where possible as a simpler and less error-prone coding technique.

You can refer to substrings of USAGE DISPLAY data items, including alphanumeric groups, alphanumeric-edited data items, numeric-edited data items, display floating-point data items, and zoned decimal data items, by using reference modifiers. When you reference-modify any of these data items, the result is of category alphanumeric. When you reference-modify an alphabetic data item, the result is of category alphabetic.

You can refer to substrings of USAGE NATIONAL data items, including national groups, national-edited data items, numeric-edited data items, national floating-point data items, and national decimal data items, by using reference modifiers. When you reference-modify any of these data items, the result is of category national. For example, suppose that you define a national decimal data item as follows:

```
01  NATL-DEC-ITEM  Usage National  Pic 999  Value 123.
```

You can use NATL-DEC-ITEM in an arithmetic expression because NATL-DEC-ITEM is of category numeric. But you cannot use NATL-DEC-ITEM(2:1) (the national character 2, which in hexadecimal notation is NX"0032") in an arithmetic expression, because it is of category national.

You can refer to substrings of table entries, including variable-length entries, by using reference modifiers. To refer to a substring of a table entry, code the subscript expression before the reference modifier. For example, assume that PRODUCT-TABLE is a properly coded table of character strings. To move D to the fourth character in the second string in the table, you can code this statement:

```
MOVE 'D' to PRODUCT-TABLE (2), (4:1)
```

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

“Example: arithmetic expressions as reference modifiers” on page 102

Because numeric function identifiers can be used anywhere that arithmetic expressions can be used, you can code a numeric function identifier in a reference modifier as the leftmost character position or as the length, or both.

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

Each number in the reference modifier must have a value of at least 1. The sum of the two numbers must not exceed the total length of the data item by more than 1 character position so that you do not reference beyond the end of the substring.

If the leftmost character position or the length value is a fixed-point noninteger, truncation occurs to create an integer. If either is a floating-point noninteger, rounding occurs to create an integer.

The following options detect out-of-range reference modifiers, and flag violations with a runtime message:

- SSRANGE compiler option
- CHECK runtime option

RELATED CONCEPTS
"Reference modifiers"
"Unicode and the encoding of language characters" on page 173

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

RELATED REFERENCES
"SSRANGE" on page 285
Reference modification (*COBOL for AIX Language Reference*)
Function definitions (*COBOL for AIX 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"
"Example: intrinsic functions as reference modifiers"

RELATED TASKS
"Assigning input from a screen or file (ACCEPT)" on page 34
"Referring to substrings of data items" on page 99
"Using national data (Unicode) in COBOL" on page 174

RELATED REFERENCES
Reference modification (*COBOL for AIX Language Reference*)

Example: arithmetic expressions as reference modifiers

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

Suppose a program has the following data:

```
01  LEFTY     PIC X(30).
01  RIGHTY    PIC X(30)  JUSTIFIED RIGHT.
01  I         PIC 9(9)   USAGE BINARY.
```

The program counts the number of leading spaces and, using arithmetic expressions in a reference modifier, moves the right-justified characters into another field, justified to the left:

```
MOVE SPACES TO LEFTY
MOVE ZERO TO I
INSPECT RIGHTY
   TALLYING I FOR LEADING SPACE.
IF I IS LESS THAN LENGTH OF RIGHTY THEN
   MOVE RIGHTY ( I + 1 : LENGTH OF RIGHTY - I ) TO LEFTY
END-IF
```

The MOVE statement transfers characters from RIGHTY, beginning at the position computed as I + 1 for a length that is computed as LENGTH OF RIGHTY - I, into the field LEFTY.

Example: intrinsic functions as reference modifiers

You can use intrinsic functions in reference modifiers if you do not know the leftmost position or length of a substring at compile time.

For example, the following code fragment causes a substring of Customer-Record to be moved into the data item WS-name. The substring is determined at run time.

```
05  WS-name        Pic x(20).
05  Left-posn      Pic 99.
05  I              Pic 99.
. . .
Move Customer-Record(Function Min(Left-posn I):Function Length(WS-name)) to WS-name
```

If you want to use a noninteger function in a position that requires an integer function, you can use the INTEGER or INTEGER-PART function to convert the result to an integer. For example:

```
Move Customer-Record(Function Integer(Function Sqrt(I)): ) to WS-name
```

RELATED REFERENCES
INTEGER (*COBOL for AIX Language Reference*)
INTEGER-PART (*COBOL for AIX Language Reference*)

Tallying and replacing data items (INSPECT)

Use the INSPECT statement to inspect characters or groups of characters in a data item and to optionally replace them.

Use the INSPECT statement to do the following tasks:

- Count the number of times a specific character occurs in a data item (TALLYING phrase).
- Fill a data item or selected portions of a data item with specified characters such as spaces, asterisks, or zeros (REPLACING phrase).
- Convert all occurrences of a specific character or string of characters in a data item to replacement characters that you specify (CONVERTING phrase).

You can specify one of the following data items as the item to be inspected:

- An elementary item described explicitly or implicitly as USAGE DISPLAY, USAGE DISPLAY-1, or USAGE NATIONAL
- An alphanumeric group item or national group item

If the inspected item has:

- USAGE DISPLAY, each identifier in the statement (except the TALLYING count field) must have USAGE DISPLAY, and each literal in the statement must be alphanumeric
- USAGE NATIONAL, each identifier in the statement (except the TALLYING count field) must have USAGE NATIONAL, and each literal in the statement must be national
- USAGE DISPLAY-1, each identifier in the statement (except the TALLYING count field) must have USAGE DISPLAY-1, and each literal in the statement must be a DBCS literal

"Examples: INSPECT statement"

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 173

RELATED REFERENCES
INSPECT statement (*COBOL for AIX Language Reference*)

Examples: INSPECT statement

The following examples show some uses of the INSPECT statement to examine and replace characters.

In the following example, the INSPECT statement examines and replaces characters in data item DATA-2. The number of times a leading zero (0) occurs in the data item is accumulated in COUNTR. The first instance of the character A that follows the first instance of the character C is replaced by the character 2.

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

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

In the following example, the INSPECT statement examines and replaces characters in data item DATA-3. Each character that precedes the first instance of a quotation mark (") is replaced by the character 0.

```
77  COUNTR           PIC 9   VALUE ZERO.
01  DATA-3           PIC X(8).
. . .
    INSPECT DATA-3
      REPLACING CHARACTERS BY ZEROS BEFORE INITIAL QUOTE
```

DATA-3 before	COUNTR after	DATA-3 after
456"ABEL	0	000"ABEL
ANDES"12	0	00000"12
"TWAS BR	0	"TWAS BR

The following example shows the use of INSPECT CONVERTING with AFTER and BEFORE phrases to examine and replace characters in data item DATA-4. All characters that follow the first instance of the character / but that precede the first instance of the character ? (if any) are translated from lowercase to uppercase.

```
01  DATA-4           PIC X(11).
. . .
    INSPECT DATA-4
      CONVERTING
         "abcdefghijklmnopqrstuvwxyz" TO
         "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
      AFTER INITIAL "/"
      BEFORE INITIAL"?"
```

DATA-4 before	DATA-4 after
a/five/?six	a/FIVE/?six
r/Rexx/RRRr	r/REXX/RRRR
zfour?inspe	zfour?inspe

Converting data items (intrinsic functions)

You can use intrinsic functions to convert character-string data items to several other formats, for example, to uppercase or lowercase, to reverse order, to numbers, or to one code page from another.

You can use the NATIONAL-OF and DISPLAY-OF intrinsic functions to convert to and from national (Unicode) strings.

You can also use the INSPECT statement to convert characters.

"Examples: INSPECT statement" on page 103

RELATED TASKS
"Converting to uppercase or lowercase (UPPER-CASE, LOWER-CASE)"
"Transforming to reverse order (REVERSE)"
"Converting to numbers (NUMVAL, NUMVAL-C)"
"Converting from one code page to another" on page 107

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

| You can use the UPPER-CASE and LOWER-CASE intrinsic functions to easily change the
| case of alphanumeric, alphabetic, or national strings.

```
01  Item-1   Pic x(30)  Value "Hello World!".
01  Item-2   Pic x(30).
. . .
    Display Item-1
    Display Function Upper-case(Item-1)
    Display Function Lower-case(Item-1)
    Move Function Upper-case(Item-1) to Item-2
    Display Item-2
```

The code above displays the following messages on the system logical output device:

```
Hello World!
HELLO WORLD!
hello world!
HELLO WORLD!
```

The DISPLAY statements do not change the actual contents of Item-1, but affect only how the letters are displayed. However, the MOVE statement causes uppercase letters to replace the contents of Item-2.

The conversion uses the case mapping that is defined in the current locale. The length of the function result might differ from the length of the argument.

RELATED TASKS
"Assigning input from a screen or file (ACCEPT)" on page 34
"Displaying values on a screen or in a file (DISPLAY)" on page 35

Transforming to reverse order (REVERSE)

You can reverse the order of the characters in a string by using the REVERSE intrinsic function.

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

For example, the statement above reverses the order of the characters in Orig-cust-name. If the starting value is JOHNSON*bbb*, the value after the statement is performed is *bbb*NOSNHOJ, where *b* represents a blank space.

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 173

Converting to numbers (NUMVAL, NUMVAL-C)

| The NUMVAL and NUMVAL-C functions convert character strings (alphanumeric or
| national literals, or class alphanumeric or class national data items) to numbers.

Use these functions to convert free-format character-representation numbers to numeric form so that you can process them numerically.

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

Use NUMVAL-C when the argument includes a currency symbol or comma or both, as shown in the example above. You can also place an algebraic sign before or after the character string, and the sign will be processed. The arguments must not exceed 18 digits when you compile with the default option ARITH(COMPAT) (*compatibility mode*) nor 31 digits when you compile with ARITH(EXTEND) (*extended mode*), not including the editing symbols.

NUMVAL and NUMVAL-C return long (64-bit) floating-point values in compatibility mode, and return extended-precision (128-bit) floating-point values in extended mode. A reference to either of these functions represents a reference to a numeric data item.

| At most 15 decimal digits can be converted accurately to long-precision floating
| point (as described in the related reference below about conversions and precision).
| If the argument to NUMVAL or NUMVAL-C has more than 15 digits, it is recommended
| that you specify the ARITH(EXTEND) compiler option so that an extended-precision
| function result that can accurately represent the value of the argument is returned.

When you use NUMVAL or NUMVAL-C, you do not need to statically declare numeric data in a fixed format nor input data in a precise manner. For example, suppose you define numbers to be entered as follows:

```
01  X   Pic S999V99  leading sign is separate.
. . .
    Accept X from Console
```

The user of the application must enter the numbers exactly as defined by the PICTURE clause. For example:

```
+001.23
-300.00
```

However, using the NUMVAL function, you could code:

```
01  A   Pic x(10).
01  B   Pic S999V99.
. . .
    Accept A from Console
    Compute B = Function Numval(A)
```

The input could then be:

```
1.23
-300
```

RELATED CONCEPTS
"Formats for numeric data" on page 43
"Data format conversions" on page 48
"Unicode and the encoding of language characters" on page 173

RELATED TASKS
"Converting to or from national (Unicode) representation" on page 182

RELATED REFERENCES
"Conversions and precision" on page 49
"ARITH" on page 248

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 173

RELATED TASKS
"Converting to or from national (Unicode) representation" on page 182

Evaluating data items (intrinsic functions)

You can use intrinsic functions to determine the ordinal position of a character in the collating sequence, to find the largest or smallest item in a series, to find the length of data item, or to determine when a program was compiled.

Use these intrinsic functions:

- CHAR and ORD to evaluate integers and single alphabetic or alphanumeric characters with respect to the collating sequence used in a program
- MAX, MIN, ORD-MAX, and ORD-MIN to find the largest and smallest items in a series of data items, including USAGE NATIONAL data items
- LENGTH to find the length of data items, including USAGE NATIONAL data items
- WHEN-COMPILED to find the date and time when a program was compiled

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 173

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

Evaluating single characters for collating sequence

To find out the ordinal position of a given alphabetic or alphanumeric character in the collating sequence, use the ORD function with the character as the argument. ORD returns an integer that represents that ordinal position.

You can use a one-character substring of a data item as the argument to ORD:

```
IF Function Ord(Customer-record(1:1)) IS > 194 THEN . . .
```

If you know the ordinal position in the collating sequence of a character, and want to find the character that it corresponds to, use the CHAR function with the integer ordinal position as the argument. CHAR returns the desired character. For example:

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

RELATED REFERENCES
CHAR (*COBOL for AIX Language Reference*)
ORD (*COBOL for AIX Language Reference*)

Finding the largest or smallest data item

To determine which of two or more alphanumeric, alphabetic, or national data items has the largest value, use the MAX or ORD-MAX intrinsic function. To determine which item has the smallest value, use MIN or ORD-MIN. These functions evaluate according to the collating sequence.

| To compare numeric items, including those that have USAGE NATIONAL, you can use
| MAX, ORD-MAX, MIN, or ORD-MIN. With these intrinsic functions, the algebraic values of the arguments are compared.

The MAX and MIN functions return the content of one of the arguments that you supply. For example, suppose that your program has the following data definitions:

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

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

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

If you used MIN instead, then THOMAS*bbbb* would be assigned.

The functions ORD-MAX and ORD-MIN return an integer that represents the ordinal position (counting from the left) of the argument that has the largest or smallest value in the list of arguments that you supply. If you used the ORD-MAX function in the example above, the compiler would issue an error message because the reference to a numeric function is not in a valid place. The following statement is a valid use of ORD-MAX:

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

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

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

RELATED TASKS
"Performing arithmetic" on page 51

"Processing table items using intrinsic functions" on page 78
"Returning variable-length results with alphanumeric or national functions"

RELATED REFERENCES
MAX (*COBOL for AIX Language Reference*)
MIN (*COBOL for AIX Language Reference*)
ORD-MAX (*COBOL for AIX Language Reference*)
ORD-MIN (*COBOL for AIX Language Reference*)

Returning variable-length results with alphanumeric or national functions

The results of alphanumeric or national functions could be of varying lengths and values depending on the function arguments.

In the following example, the amount of data moved to R3 and the results of the COMPUTE statement depend on the values and sizes of R1 and R2:

```
01  R1    Pic x(10) value "e".
01  R2    Pic x(05) value "f".
01  R3    Pic x(20) value spaces.
01  L     Pic 99.
. . .
    Move Function Max(R1 R2) to R3
    Compute L = Function Length(Function Max(R1 R2))
```

This code has the following results:

- R2 is evaluated to be larger than R1.
- The string '*fbbbb*' is moved to R3, where *b* represents a blank space. (The unfilled character positions in R3 are padded with spaces.)
- L evaluates to the value 5.

If R1 contained 'g' instead of 'e', the code would have the following results:

- R1 would evaluate as larger than R2.
- The string '*gbbbbbbbbb*' would be moved to R3. (The unfilled character positions in R3 would be padded with spaces.)
- The value 10 would be assigned to L.

If a program uses national data for function arguments, the lengths and values of the function results could likewise vary. For example, the following code is identical to the fragment above, but uses national data instead of alphanumeric data.

```
01  R1    Pic n(10) national value "e".
01  R2    Pic n(05) national value "f".
01  R3    Pic n(20) national value spaces.
01  L     Pic 99    national.
. . .
    Move Function Max(R1 R2) to R3
    Compute L = Function Length(Function Max(R1 R2))
```

This code has the following results, which are similar to the first set of results except that these are for national characters:

- R2 is evaluated to be larger than R1.
- The string NX"0066 0020 0020 0020 0020" (the equivalent in national characters of '*fbbbb*', where *b* represents a blank space), shown here in hexadecimal notation with added spaces for readability, is moved to R3. The unfilled character positions in R3 are padded with national spaces.

- L evaluates to the value 5, the length in national character positions of R2.

You might be dealing with variable-length output from alphanumeric or national functions. Plan your program accordingly. For example, you might need to think about using variable-length files when the records that you are writing could be of different lengths:

```
File Section.
FD  Output-File Recording Mode V.
01  Short-Customer-Record  Pic X(50).
01  Long-Customer-Record   Pic X(70).
Working-Storage Section.
01  R1    Pic x(50).
01  R2    Pic x(70).
. . .
    If R1 > R2
      Write Short-Customer-Record from R1
    Else
      Write Long-Customer-Record from R2
    End-if
```

RELATED TASKS
"Finding the largest or smallest data item" on page 108
"Performing arithmetic" on page 51

RELATED REFERENCES
MAX (*COBOL for AIX Language Reference*)

Finding the length of data items

You can use the LENGTH function in many contexts (including tables and numeric data) to determine the length of an item. For example, you can use the LENGTH function to determine the length of an alphanumeric or national literal, or a data item of any type except DBCS.

The LENGTH function returns the length of a national item (a literal, or any item that has USAGE NATIONAL, including national group items) as an integer equal to the length of the argument in national character positions. It returns the length of any other data item as an integer equal to the length of the argument in alphanumeric character positions.

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

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

You can also use the LENGTH OF special register, which returns the length in bytes even for national data. Coding either Function Length(Customer-name) or LENGTH OF Customer-name returns the same result for alphanumeric items: the length of Customer-name in bytes.

You can use the LENGTH function only where arithmetic expressions are allowed. However, you can use the LENGTH OF special register in a greater variety of contexts. For example, you can use the LENGTH OF special register as an argument to an intrinsic function that 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 51
"Creating variable-length tables (DEPENDING ON)" on page 73
"Processing table items using intrinsic functions" on page 78

RELATED REFERENCES
"ADDR" on page 246
LENGTH (*COBOL for AIX Language Reference*)
LENGTH OF (*COBOL for AIX Language Reference*)

Finding the date of compilation

You can use the `WHEN-COMPILED` intrinsic function to determine when a program was compiled. The 21-character result indicates the four-digit year, month, day, and time (in hours, minutes, seconds, and hundredths of seconds) of compilation, and the difference in hours and minutes from Greenwich mean time.

The first 16 positions are in the following format:

```
YYYYMMDDhhmmsshh
```

You can instead use the `WHEN-COMPILED` special register to determine the date and time of compilation in the following format:

```
MM/DD/YYhh.mm.ss
```

The `WHEN-COMPILED` special register supports only a two-digit year, and carries the time out only to seconds. You can use this special register only as the sending field in a `MOVE` statement.

RELATED REFERENCES
WHEN-COMPILED (*COBOL for AIX Language Reference*)

Chapter 7. Processing files

Reading data from files and writing data to files is an essential part of most COBOL programs. Your program can retrieve information, process it as you request, and then write the results.

Before the processing, however, you need to identify the files and describe their physical structure, and indicate whether they are organized as sequential, relative, indexed, or line sequential. Identifying files entails naming them and their file system. You might also want to set up a file status field that you can check later to make sure that the processing worked properly.

The major tasks you can perform in processing a file are first opening the file and then reading it, and (depending on the type of file organization and access) adding, replacing, or deleting records.

RELATED CONCEPTS
"File systems" on page 115

RELATED TASKS
"Identifying files"
"Specifying a file organization and access mode" on page 122
"Protecting against errors when opening files" on page 125
"Setting up a field for file status" on page 126
"Describing the structure of a file in detail" on page 126
"Coding input and output statements for files" on page 127
"Enabling COBOL applications to access SFS files" on page 136

Identifying files

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

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

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

FileSystemID identifies one of the following file systems:

VSAM
Virtual storage access method system. You can abbreviate VSAM as `VSA`.

Specify the VSAM file system when you want to access VSAM files or Encina® SFS files.

STL Standard language file system

RSD Record sequential delimited file system

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

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

RELATED CONCEPTS
"File systems" on page 115

RELATED TASKS
"Identifying VSAM files"
"Identifying Encina SFS files"
"Identifying STL files" on page 115
"Identifying RSD files" on page 115

RELATED REFERENCES
"VSAM file system" on page 116
"SFS file system" on page 117
"STL file system" on page 118
"RSD file system" on page 120
"Runtime environment variables" on page 211
"FILESYS" on page 303

Identifying VSAM files

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

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

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

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

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

```
SELECT file1 ASSIGN TO MYFILE
```

Identifying Encina SFS files

To use the VSAM file system for an SFS file, identify the file with the `SELECT` clause as shown below.

```
SELECT file1 ASSIGN TO Myfile
```

Myfile, the COBOL *assignment-name*, is a runtime environment variable. Use an `export` command of the following form to assign a value to the environment variable at run time:

```
export Myfile="/.:/myEnc/mySFS/mySFSfil(/.:/myEnc/mySFS/  \
  mySFSfil;myaltfil1,/.:/myEnc/mySFS/mySFSfil;myaltfil2)"
```

This command provides the following information:

- `/.:/myEnc/mySFS/` is the fully qualified name for the Encina server.
- `mySFSfil` is the base SFS file.
- `/.:/myEnc/mySFS/mySFSfil` is the fully qualified base file name.
- `myaltfil1` and `myaltfil2` are the alternate index files.

For an alternate index file, the file name must be in the format of its fully qualified base file name followed by a semicolon (`;`) and the alternate index file name: `/.:/myEnc/mySFS/mySFSfil;myaltfil1`.

A comma is required between specifications of alternate index files in the `export` command.

RELATED REFERENCES
"SFS file system" on page 117

Identifying STL files

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

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

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

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

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

```
SELECT file1 ASSIGN TO MYFILE
```

Identifying RSD files

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

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

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

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

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

```
SELECT file1 ASSIGN TO MYFILE
```

File systems

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

COBOL for AIX supports the following file systems:
- The VSAM file system, which lets you access VSAM files or Encina SFS files. (Encina SFS is the file system used by CICS.) The VSAM file system is provided with COBOL for AIX, whereas Encina SFS is supplied as part of CICS or can be ordered independently of CICS.
- The STL (standard language) file system, which provides the basic facilities for local files. It is provided with COBOL for AIX, and supports sequential, relative, and indexed files.
- The RSD (record sequential delimited) file system, which lets you share data with programs written in other languages. RSD files are sequential only and support all COBOL data types in records. Text data in records can be edited by most file editors.

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

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

RELATED CONCEPTS
"STL and VSAM files that are larger than 2 GB" on page 121

RELATED TASKS
"Identifying files" on page 113

RELATED REFERENCES
"VSAM file system"
"SFS file system" on page 117
"STL file system" on page 118
"RSD file system" on page 120
"File-system summary" on page 121
"FILESYS" on page 303
Appendix C, "zSeries host data format considerations," on page 581

VSAM file system

The VSAM file system supports VSAM files and SFS files. It supports sequential, indexed, and relative files. LINE SEQUENTIAL is the only file-organization type that is not supported.

When you create a VSAM file using OPEN OUTPUT and WRITE statements, multiple files are created in the native byte-stream file system:
- The file attributes are stored in files that have suffix .DDMEA.
- The primary index is stored in a file whose name begins with the character @, followed by the file name.
- The alternate indexes are stored in files that end with suffix .@01, .@02, and so forth.

These local VSAM-managed files must not be accessed using other non-VSAM software.

Restriction: The VSAM file system is not safe for use with multiple threads.

The VSAM file system conforms to Standard COBOL 85.

RELATED CONCEPTS
"STL and VSAM files that are larger than 2 GB" on page 121
"File organization and access mode" on page 122

RELATED TASKS
"Identifying VSAM files" on page 114

RELATED REFERENCES
"File-system summary" on page 121
"SFS file system" on page 117
VSAM File System Reply Messages

SFS file system

The Encina SFS (Structured File Server) file system is a record-oriented file system that supports sequential, relative, and indexed COBOL files. It provides the basic facilities that you need for accessing files sequentially, randomly, or dynamically.

You can process SFS files with the standard file operations such as read, insert, update, or delete.

| **Restriction:** You cannot process SFS files with 64-bit COBOL for AIX programs.

To access SFS files, you specify the VSAM file system. See the related task about identifying files for details.

Each SFS file has one primary index, which defines the physical ordering of the records in the file, and can have any number of secondary indices, which provide alternate sequences in which the records can be accessed.

You specify the full name of an SFS file by the name of the SFS server on which it resides, followed by the name of the file itself. For example, if the server name is /.:/encina/sfs/*my_server* and the file name is *myfile*, then the full file name is /.:/encina/sfs/*my_server*/*myfile*. (The name in this form is referred to as a *fully qualified* file name.)

The SFS file system is nonhierarchical. That is, you can specify only individual file names, not directory names, after the server name.

The SFS file system supports three types of file organization, known as entry-sequenced (that is, sequential), relative, and clustered (that is, indexed).

All data in SFS files is managed by an SFS server. SFS provides a system administration tool, `sfsadmin`, for performing administrative functions such as creating files and indices, determining which volumes are available on the SFS server, and so on, through a command-line interface. For details, see the Encina publication in the related reference below.

COBOL access to SFS files is nontransactional; that is, each operation against an SFS file is performed either in its entirety or not at all (sometimes referred to as *atomic*). In the event of an SFS system failure, the results of a file operation completed by a COBOL application might not be reflected in the SFS file.

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

RELATED CONCEPTS
"File organization and access mode" on page 122

RELATED TASKS
"Identifying files" on page 113
"Identifying Encina SFS files" on page 114
"Enabling COBOL applications to access SFS files" on page 136
"Improving SFS performance" on page 139
"Compiling for SFS files" on page 220
Encina SFS Programming Guide

RELATED REFERENCES
"File-system summary" on page 121
Encina Administration Guide Volume 2: Server Administration

STL file system

The STL file system (standard language file system) supports sequential, indexed, and relative files. It provides the basic file facilities for accessing files.

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

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

RELATED CONCEPTS
"STL and VSAM files that are larger than 2 GB" on page 121
"File organization and access mode" on page 122

RELATED TASKS
"Identifying STL files" on page 115

RELATED REFERENCES
"STL file system return codes"
"File-system summary" on page 121

STL file system return codes

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

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

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

Table 9. **STL file system return codes**

Code	Meaning	Notes
0	Successful completion	The input-output operation completed successfully.
1	Invalid operation	This return code should not occur; it indicates an error in the file system.
2	I/O error	A call to an operating system I/O routine returned an error code.
3	File not open	Attempt to perform an operation (other than `OPEN`) on a file that is not open
4	Key value not found	Attempt to read a record using a key that is not in the file

Table 9. **STL file system return codes** *(continued)*

Code	Meaning	Notes
5	Duplicate key value	Attempt to use a key a second time for a key that does not allow duplicates
6	Invalid key number	This return code should not occur; it indicates an error in the file system.
7	Different key number	This return code should not occur; it indicates an error in the file system.
8	Invalid flag for the operation	This return code should not occur; it indicates an error in the file system.
9	End-of-file	End-of-file was detected. This is not an error.
10	I/O operation must be preceded by an I/O GET operation	The operation is looking for the current record, and the current record has not been defined.
11	Error return from get space routine	The operating system indicates that not enough memory is available.
12	Duplicate key accepted	The operation specified a key, and the key is a duplicate.
13	Sequential access and key sequence bad	Sequential access was specified, but the records are not in sequential order.
14	Record length < max key	The record length does not allow enough space for all of the keys.
15	Access to file denied	The operating system reported that it cannot access the file. Either the file does not exist or the user does not have the proper permission of the operating system to access the file.
16	File already exists	You attempted to open a new file, but the operating system reports that the file already exists.
17	(Reserved)	
18	File locked	Attempt to open a file that is already open in exclusive mode
19	File table full	The operating system reports that its file table is full.
20	Handle table full	The operating system reports that it cannot allocate any more file handles.
21	Title does not say STL	Files opened for reading by the STL file system must contain a header record that contains ″STL″ at a certain offset in the file.
22	Bad indexcount argument for create	This return code should not occur; it indicates an error in the file system.
23	Index or relative record > 64 KB	Index and relative records are limited to a length of 64 KB.
24	Error found in file header or data in open of existing file	STL files begin with a header. The header or its associated data has inconsistent values.
25	Indexed open on sequential file	Attempt to open a sequential file as an indexed or relative file

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

Table 10. **STL file system adapter open routine return codes**

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

RELATED TASKS
"Using file status keys" on page 161

RELATED REFERENCES
"STL file system" on page 118
FILE STATUS clause (*COBOL for AIX Language Reference*)

RSD file system

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

RSD files provide good performance. They give you the ability to port files easily between AIX and Windows-based systems and to share files between programs and applications written in different languages.

RSD files support all COBOL data types in records of fixed length. Each record written is followed by a newline control character. For a read operation, the file record area is filled for the record length. If an end-of-file is encountered before the record area is filled, the record area is padded with space characters.

If you edit an RSD file with a text editor, you must ensure that the length of each record is maintained.For example, if you edit file *infile*, and want to ensure that each record has 80 characters, you can pad each record to column 80 with spaces and write the result to file *outfile* by issuing this command:

```
awk '{printf("\%-80s\n",$0)}' infile > outfile
```

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

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

RELATED CONCEPTS
"File organization and access mode" on page 122

RELATED TASKS
"Identifying RSD files" on page 115

STL and VSAM files that are larger than 2 GB

Beginning with Version 3 of IBM COBOL for AIX, STL and VSAM files are not limited to a maximum size of 2 GB as they were in earlier versions of COBOL for AIX.

By default, STL and VSAM files created using `OPEN OUTPUT` statements have metadata (internal control information), referred to as *large-format*, that supports the increased file size. This feature is in general transparent to COBOL application programs. Note that only the maximum file size has increased; the maximum sizes of individual records have not increased.

You can process large-format files with COBOL for AIX Version 2 programs if you run those programs using the current runtime library. Large-format files cannot be processed by COBOL for AIX Version 2 programs that run with the Version 2 runtime library.

STL and VSAM files created by COBOL for AIX programs before Version 3 have *small-format* metadata. You can open small-format files using `OPEN INPUT`, `OPEN I-O`, or `OPEN EXTEND` statements, but the files continue to be limited to the 2 GB maximum size.

For compatibility with COBOL for AIX Version 2, you can specify runtime option `FILEMODE(SMALL)` when you run a COBOL for AIX program to indicate that the program should create small-format STL and VSAM files. You can process those files with Version 2 programs that run with the Version 2 (or later) runtime library.

You can share small-format STL and VSAM files with PL/I programs.

RELATED TASKS
"Identifying files" on page 113

RELATED REFERENCES
"VSAM file system" on page 116
"STL file system" on page 118
"FILEMODE" on page 303
Compiler limits (*COBOL for AIX Language Reference*)

File-system summary

The following table shows which types of files can be processed by which types of COBOL programs. An X indicates that a file can be processed by the type of program shown at the top of the column.

Table 11. **File-system summary**

File system and format	**Can file be larger than 2 GB?**	**32-bit COBOL for AIX program**	**64-bit COBOL for AIX program**	**Version 2 COBOL for AIX program with current runtime library**	**Version 2 COBOL for AIX program with Version 2 runtime library**
STL small-format	No	X	X	X	X
STL large-format[1]	Yes	X	X	X[2]	
VSAM small-format	No	X	X	X	X
VSAM large-format[1]	Yes	X	X	X[2]	
Line-sequential	Yes	X	X	X	X
RSD	Yes	X	X	X	X
Encina SFS	No COBOL limit[3]	X		X	X

1. A file created with runtime option `FILEMODE(LARGE)` in effect. File size is limited to the maximum record size times the maximum number of records. (For details, see the related reference below about compiler limits.)
2. These programs can open existing large-format files using `OPEN INPUT`, `OPEN I-O`, or `OPEN EXTEND`, and can process those files. However, if these programs create any files using `OPEN OUTPUT`, those files will have small format.
3. See Encina publications for detailed limits and requirements.

RELATED CONCEPTS
"STL and VSAM files that are larger than 2 GB" on page 121

RELATED REFERENCES
"ADDR" on page 246
"FILEMODE" on page 303
Encina Administration Guide Volume 2: Server Administration
Compiler limits (*COBOL for AIX Language Reference*)

Specifying a file organization and access mode

In the `FILE-CONTROL` paragraph, you need to define the physical structure of a file and its access mode, as shown below.

```
FILE-CONTROL.
    SELECT file ASSIGN TO FileSystemID-Filename
    ORGANIZATION IS org ACCESS MODE IS access.
```

For *org*, you can choose `SEQUENTIAL` (the default), `LINE SEQUENTIAL`, `INDEXED`, or `RELATIVE`.

For *access*, you can choose `SEQUENTIAL` (the default), `RANDOM`, or `DYNAMIC`.

Sequential and line-sequential files must be accessed sequentially. For indexed or relative files, all three access modes are possible.

File organization and access mode

You can organize your files as sequential, line-sequential, indexed, or relative. The access mode defines how COBOL reads and writes files, but not how files are organized.

You should decide on the file organization and access modes when you design your program.

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

Table 12. **File organization and access mode**

File organization	Order of records	Records can be deleted or replaced?	Access mode
Sequential	Order in which they were written	A record cannot be deleted, but its space can be reused for a same-length record.	Sequential only
Line-sequential	Order in which they were written	No	Sequential only
Indexed	Collating sequence by key field	Yes	Sequential, random, or dynamic
Relative	Order of relative record numbers	Yes	Sequential, random, or dynamic

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

RELATED CONCEPTS
"Sequential file organization"
"Line-sequential file organization" on page 124
"Indexed file organization" on page 124
"Relative file organization" on page 124
"Sequential access" on page 125
"Random access" on page 125
"Dynamic access" on page 125

RELATED TASKS
"Specifying a file organization and access mode" on page 122

Sequential file organization

A sequential file contains records organized by the order in which they were entered. The order of the records is fixed.

Records in sequential files can be read or written only sequentially.

After you place a record into a sequential file, you cannot shorten, lengthen, or delete the record. However, you can update (REWRITE) a record if the length does not change. New records are added at the end of the file.

If the order in which you keep records in a file is not important, sequential organization is a good choice whether there are many records or only a few. Sequential output is also useful for printing reports.

RELATED CONCEPTS
"Sequential access" on page 125

RELATED REFERENCES
"Valid COBOL statements for sequential files" on page 130

Line-sequential file organization

Line-sequential files are like sequential files, except that the records can contain only characters as data. Line-sequential files are supported by the native byte stream files of the operating system.

Line-sequential files that are created with `WRITE` statements with the `ADVANCING` phrase can be directed to a printer or to a disk.

RELATED CONCEPTS
"Sequential file organization" on page 123

RELATED REFERENCES
"Valid COBOL statements for line-sequential files" on page 130

Indexed file organization

An indexed file contains records ordered by a *record key*. A record key uniquely identifies a record and determines the sequence in which it is accessed with respect to other records.

Each record contains a field that contains the record key. A record key for a record might be, for example, an employee number or an invoice number.

An indexed file can also use *alternate indexes*, that is, record keys that let you access the file using a different logical arrangement of the records. For example, you could access a file through employee department rather than through employee number.

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

EBCDIC consideration: As with any change in the collating sequence, if your indexed file is a local EBCDIC file, the EBCDIC keys will not be recognized as such outside of your COBOL program. For example, an external sort program, unless it also has support for EBCDIC, will not sort records in the order that you might expect.

RELATED REFERENCES
"Valid COBOL statements for indexed and relative files" on page 131

Relative file organization

A relative record file contains records ordered by their *relative key*, that is, a record number that represents the location of the record relative to where the file begins.

For example, the first record in a file has a relative record number of 1, the tenth record has a relative record number of 10, and so forth. The records can have fixed length or variable length.

The record transmission modes allowed for relative files are sequential, random, or dynamic. When relative files are read or written sequentially, the sequence is that of the relative record number.

RELATED REFERENCES
"Valid COBOL statements for indexed and relative files" on page 131

Sequential access

For sequential access, code `ACCESS IS SEQUENTIAL` in the `FILE-CONTROL` paragraph.

For indexed files, records are accessed in the order of the key field selected (either primary or alternate), beginning at the current position of the file position indicator.

For relative files, records are accessed in the order of the relative record numbers.

RELATED CONCEPTS
"Random access"
"Dynamic access"

RELATED REFERENCES
"File position indicator" on page 129

Random access

For random access, code `ACCESS IS RANDOM` in the `FILE-CONTROL` paragraph.

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

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

RELATED CONCEPTS
"Sequential access"
"Dynamic access"

Dynamic access

For dynamic access, code `ACCESS IS DYNAMIC` in the `FILE-CONTROL` paragraph.

Dynamic access supports a mixture of sequential and random access in the same program. With dynamic access, you can use one COBOL file definition to perform both sequential and random processing, accessing some records in sequential order and others by their keys.

For example, suppose you have an indexed file of employee records, and the employee's hourly wage forms the record key. Also, suppose your program is interested in those employees who earn between $12.00 and $18.00 per hour and those who earn $25.00 per hour and above. To access this information, retrieve the first record randomly (with a random-retrieval `READ`) based on the key of 1200. Next, begin reading sequentially (using `READ NEXT`) until the salary field exceeds 1800. Then switch back to a random read, this time based on a key of 2500. After this random read, switch back to reading sequentially until you reach the end of the file.

RELATED CONCEPTS
"Sequential access"
"Random access"

Protecting against errors when opening files

If a program tries to open and read a file that does not exist, normally an error occurs.

However, there might be times when opening a nonexistent file makes sense. For such cases, use the optional keyword OPTIONAL with SELECT:

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

Setting up a field for file status

Establish a file status key by using the FILE STATUS clause in the FILE-CONTROL paragraph and data definitions in the DATA DIVISION.

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    . . .
    FILE STATUS IS file-status
WORKING-STORAGE SECTION.
01  file-status  PIC 99.
```

| Specify the file status key *file-status* as a two-character category alphanumeric or
| category national item, or as a two-digit zoned decimal (USAGE DISPLAY) (as shown
| above) or national decimal (USAGE NATIONAL) item.

Restriction: The data item referenced in the FILE STATUS clause cannot be variably located; for example, it cannot follow a variable-length table.

RELATED TASKS
"Using file status keys" on page 161

RELATED REFERENCES
FILE STATUS clause (*COBOL for AIX Language Reference*)

Describing the structure of a file in detail

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

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

In the example above, *filename* is also the file-name you use in the OPEN, READ, and CLOSE statements.

recordname is the name of the record used in WRITE and REWRITE statements. You can specify more than one record for a file.

fieldlength is the logical length of a field, and *type* specifies the format of a field. If you break the record description entry beyond level 01 in this manner, map each element accurately to the corresponding field in the record.

RELATED REFERENCES
Data relationships (*COBOL for AIX Language Reference*)
Level-numbers (*COBOL for AIX Language Reference*)

PICTURE clause (*COBOL for AIX Language Reference*)
USAGE clause (*COBOL for AIX Language Reference*)

Coding input and output statements for files

After you identify and describe the files in the ENVIRONMENT DIVISION and the DATA DIVISION, you can process the file records in the PROCEDURE DIVISION of your program.

Code your COBOL program according to the types of files that you use, whether sequential, line sequential, indexed, or relative. The general format for coding input and output (as shown in the example referenced below) involves opening the file, reading it, writing information into it, and then closing it.

"Example: COBOL coding for files"

RELATED TASKS
"Identifying files" on page 113
"Specifying a file organization and access mode" on page 122
"Protecting against errors when opening files" on page 125
"Setting up a field for file status" on page 126
"Describing the structure of a file in detail" on page 126
"Opening a file" on page 129
"Reading records from a file" on page 132
"Adding records to a file" on page 133
"Replacing records in a file" on page 134
"Deleting records from a file" on page 135

RELATED REFERENCES
"File position indicator" on page 129
"Statements used when writing records to a file" on page 133
"PROCEDURE DIVISION statements used to update files" on page 135

Example: COBOL coding for files

The following example shows the general format of input-output coding. Explanations of user-supplied information (lowercase text in the example) follow the code.

```
IDENTIFICATION DIVISION.
. . .
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT filename ASSIGN TO assignment-name  (1) (2)
    ORGANIZATION IS org ACCESS MODE IS access  (3) (4)
    FILE STATUS IS file-status  (5)
    . . .
DATA DIVISION.
FILE SECTION.
FD  filename
01  recordname  (6)
    nn . . . fieldlength & type  (7) (8)
    nn . . . fieldlength & type
    . . .
WORKING-STORAGE SECTION.
01  file-status    PIC 99.
    . . .
PROCEDURE DIVISION.
```

```
    OPEN iomode filename    (9)
    . . .
    READ filename
    . . .
    WRITE recordname
    . . .
    CLOSE filename
  STOP RUN.
```

(1) ***filename***
Any valid COBOL name. You must use the same file-name in the SELECT clause and FD entry, and in the OPEN, READ, START, DELETE, and CLOSE statements. This name is not necessarily the actual name of the file as known to the system. Each file requires its own SELECT clause, FD entry, and input-output statements. For WRITE and REWRITE, you specify a record defined for the file.

(2) ***assignment-name***
You can code ASSIGN TO *assignment-name* to specify the target file-system ID and the file-name as it is known to the system directly, or you can set a value indirectly by using an environment variable.

If you want to have the system file-name identified at OPEN time, specify ASSIGN USING *data-name*. The value of *data-name* at the time of the execution of the OPEN statement for that file is used. You can optionally precede the system file identification by the file-system type identification.

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

```
SELECT inventory-file ASSIGN USING a-file . . .
. . .
FD inventory-file . . .
. . .
77 a-file PIC X(25) VALUE SPACES.
. . .
    MOVE "/user/inventory/parts" TO a-file
    OPEN INPUT inventory-file
```

The following example illustrates how inventory-file is dynamically (by way of a MOVE statement) associated with the index file parts, and the alternate index files altpart1 and altpart2 are associated with the fully qualified name (/.:/Encina/sfs in this example) for the Encina server.

```
SELECT inventory-file ASSIGN USING a-file . . .
    ORGANIZATION IS INDEXED
    ACCESS MODE IS DYNAMIC
    RECORD KEY IS FILESYSFILE-KEY
    ALTERNATE RECORD KEY IS ALTKEY1
    ALTERNATE RECORD KEY IS ALTKEY2. . . .
. . .
FILE SECTION.
FD inventory-file . . .
. . .
WORKING-STORAGE SECTION.
01 a-file  PIC X(80) VALUE ' '. . . .
. . .
    MOVE "/.:/Encina/sfs/parts(/.:/Encina/sfs/parts;altpart1,/.:/
       Encina/sfs/parts;altpart2)" TO a-file
    OPEN INPUT inventory-file
```

(3) ***org*** Indicates the organization: LINE SEQUENTIAL, SEQUENTIAL, INDEXED, or RELATIVE. If you omit this clause, the default is ORGANIZATION SEQUENTIAL.

(4) ***access***
Indicates the access mode, `SEQUENTIAL`, `RANDOM`, or `DYNAMIC`. If you omit this clause, the default is `ACCESS SEQUENTIAL`.

(5) ***file-status***
The COBOL file status key. You can specify the file status key as a two-character alphanumeric or national data item, or as a two-digit zoned decimal or national decimal item.

(6) ***recordname***
The name of the record used in the `WRITE` and `REWRITE` statements. You can specify more than one record for a file.

(7) ***fieldlength***
The logical length of the field.

(8) ***type***
Must match the record format of the file. If you break the record description entry beyond the level-01 description, map each element accurately against the record's fields.

(9) ***iomode***
Specifies the open mode. If you are only reading from a file, code `INPUT`. If you are only writing to a file, code `OUTPUT` (to open a new file or write over an existing one) or `EXTEND` (to add records to the end of the file). If you are doing both, code `I-O`.

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

File position indicator

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

You do not set the file position indicator anywhere in your program. It is set by successful `OPEN`, `START`, `READ`, `READ NEXT`, and `READ PREVIOUS` statements. Subsequent `READ`, `READ NEXT`, or `READ PREVIOUS` requests use the established file position indicator location and update it.

The file position indicator is not used or affected by the output statements `WRITE`, `REWRITE`, or `DELETE`. The file position indicator has no meaning for random processing.

Opening a file

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

```
PROCEDURE DIVISION.
    . . .
    OPEN iomode filename
```

In the example above, *iomode* specifies the open mode. If you are only reading from the file, code `INPUT` for the open mode. If you are only writing to the file, code either `OUTPUT` (to open a new file or write over an existing one) or `EXTEND` (to add records to the end of the file) for the open mode.

To open a file that already contains records, use `OPEN INPUT`, `OPEN I-O` (not valid for line-sequential files), or `OPEN EXTEND`.

If you use `OPEN OUTPUT` for an SFS file that contains records, the COBOL run time deletes the file and then creates the file with attributes provided by COBOL. If you do not want an SFS file to be deleted, open the file by coding `OPEN I-O`.

If you open a sequential, line-sequential, or relative file as `EXTEND`, the added records are placed after the last existing record in the file. If you open an indexed file as `EXTEND`, each record that you add must have a record key that is higher than the highest record in the file.

RELATED CONCEPTS
"STL and VSAM files that are larger than 2 GB" on page 121
"File organization and access mode" on page 122

RELATED TASKS
"Protecting against errors when opening files" on page 125

RELATED REFERENCES
"Valid COBOL statements for sequential files"
"Valid COBOL statements for line-sequential files"
"Valid COBOL statements for indexed and relative files" on page 131
OPEN statement (*COBOL for AIX Language Reference*)

Valid COBOL statements for sequential files

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

Table 13. **Valid COBOL statements for sequential files**

Access mode	COBOL statement	OPEN INPUT	OPEN OUTPUT	OPEN I-O	OPEN EXTEND
Sequential	`OPEN`	X	X	X	X
	`WRITE`		X		X
	`START`				
	`READ`	X		X	
	`REWRITE`			X	
	`DELETE`				
	`CLOSE`	X	X	X	X

RELATED CONCEPTS
"Sequential file organization" on page 123
"Sequential access" on page 125

Valid COBOL statements for line-sequential files

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

Table 14. **Valid COBOL statements for line-sequential files**

Access mode	COBOL statement	OPEN INPUT	OPEN OUTPUT	OPEN I-O	OPEN EXTEND
Sequential	OPEN	X	X		X
	WRITE		X		X
	START				
	READ	X			
	REWRITE				
	DELETE				
	CLOSE	X	X		X

RELATED CONCEPTS
"Line-sequential file organization" on page 124
"Sequential access" on page 125

Valid COBOL statements for indexed and relative files

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

Table 15. **Valid COBOL statements for indexed and relative files**

Access mode	COBOL statement	OPEN INPUT	OPEN OUTPUT	OPEN I-O	OPEN EXTEND
Sequential	OPEN	X	X	X	X
	WRITE		X		X
	START	X		X	
	READ	X		X	
	REWRITE			X	
	DELETE			X	
	CLOSE	X	X	X	X
Random	OPEN	X	X	X	
	WRITE		X	X	
	START				
	READ	X		X	
	REWRITE			X	
	DELETE			X	
	CLOSE	X	X	X	
Dynamic	OPEN	X	X	X	
	WRITE		X	X	
	START	X		X	
	READ	X		X	
	REWRITE			X	
	DELETE			X	
	CLOSE	X	X	X	

RELATED CONCEPTS
"Indexed file organization" on page 124
"Relative file organization" on page 124
"Sequential access" on page 125
"Random access" on page 125
"Dynamic access" on page 125

Reading records from a file

Use the READ statement to retrieve records from a file. To read a record, you must have opened the file with OPEN INPUT or OPEN I-O (OPEN I-O is not valid for line-sequential files). Check the file status key after each READ.

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

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

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

When you want to read sequentially beginning at a specific record, use START before the READ NEXT or the READ PREVIOUS statement to set the file position indicator to point to a particular record. When you code START followed by READ NEXT, the next record is read and the file position indicator is reset to the next record. When you code START followed by READ PREVIOUS, the previous record is read and the file position indicator is reset to the previous record. You can move the file position indicator randomly by using START, but all reading is done sequentially from that point.

You can continue to read records sequentially, or you can use the START statement to move the file position indicator. For example:

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

When a direct READ is performed for an indexed file based on an alternate index for which duplicates exist, only the first record in the file (base cluster) with that alternate key value is retrieved. You need a series of READ NEXT statements to retrieve each of the data set records with the same alternate key. A file status value of 02 is returned if there are more records with the same alternate key value to be read. A value of 00 is returned when the last record with that key value has been read.

RELATED CONCEPTS
"Sequential access" on page 125
"Random access" on page 125
"Dynamic access" on page 125
"File organization and access mode" on page 122

RELATED TASKS
"Opening a file" on page 129
"Using file status keys" on page 161

RELATED REFERENCES
"File position indicator" on page 129
FILE STATUS clause (*COBOL for AIX Language Reference*)

Statements used when writing records to a file

The following table shows the COBOL statements that you can use when creating or extending a file.

Table 16. **Statements used when writing records to a file**

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

RELATED CONCEPTS
"File organization and access mode" on page 122

RELATED TASKS
"Specifying a file organization and access mode" on page 122
"Opening a file" on page 129
"Setting up a field for file status" on page 126
"Adding records to a file"

RELATED REFERENCES
"PROCEDURE DIVISION statements used to update files" on page 135

Adding records to a file

The COBOL WRITE statement adds a record to a file without replacing any existing records. The record to be added must not be larger than the maximum record size set when the file was defined. Check the file status key after each WRITE statement.

Adding records sequentially: To add records sequentially to the end of a file that has been opened with either OUTPUT or EXTEND, use ACCESS IS SEQUENTIAL and code the WRITE statement.

Sequential and line-sequential files are always written sequentially.

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

For relative files, the records must be in sequence. If you include a RELATIVE KEY data item in the SELECT clause, the relative record number of the record to be written is placed in that data item.

Adding records randomly or dynamically: When you write records to an indexed data set for which you have coded ACCESS IS RANDOM or ACCESS IS DYNAMIC, you can write the records in any order.

RELATED CONCEPTS
"STL and VSAM files that are larger than 2 GB" on page 121
"File organization and access mode" on page 122

RELATED TASKS
"Specifying a file organization and access mode" on page 122
"Using file status keys" on page 161

RELATED REFERENCES
"Statements used when writing records to a file" on page 133
"PROCEDURE DIVISION statements used to update files" on page 135
FILE STATUS clause (*COBOL for AIX Language Reference*)

Replacing records in a file

To replace a record in a file, use REWRITE on a file that you opened as I-O. If the file was not opened as I-O, the record is not replaced and the status key is set to 49. Check the file status key after each REWRITE statement.

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

To replace a record randomly or dynamically, you do not have to first READ the record. Instead, locate the record you want to replace as follows:
- For indexed files, move the record key to the RECORD KEY data item, and then issue the REWRITE.
- For relative files, move the relative record number to the RELATIVE KEY data item, and then issue the REWRITE.

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

RELATED CONCEPTS
"File organization and access mode" on page 122

RELATED TASKS
"Opening a file" on page 129
"Using file status keys" on page 161

RELATED REFERENCES
FILE STATUS clause (*COBOL for AIX Language Reference*)

Deleting records from a file

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

When ACCESS IS SEQUENTIAL, the record to be deleted must first be read by the COBOL program. The DELETE statement removes the record that was just read. If the DELETE statement is not preceded by a successful READ, the record is not deleted, and the file status key value is set to 92.

When ACCESS IS RANDOM or ACCESS IS DYNAMIC, the record to be deleted need not be read by the COBOL program. To delete a record, move the key of the record to the RECORD KEY data item, and then issue the DELETE.

Check the file status key after each DELETE statement.

RELATED CONCEPTS
"File organization and access mode" on page 122

RELATED TASKS
"Opening a file" on page 129
"Reading records from a file" on page 132
"Using file status keys" on page 161

RELATED REFERENCES
FILE STATUS clause (*COBOL for AIX Language Reference*)

PROCEDURE DIVISION statements used to update files

The table below shows the statements that you can use in the PROCEDURE DIVISION for sequential, line-sequential, indexed, and relative files.

Table 17. **PROCEDURE DIVISION statements used to update files**

Access method	Sequential	Line sequential	Indexed	Relative
ACCESS IS SEQUENTIAL	OPEN EXTEND WRITE CLOSE or OPEN I-O READ REWRITE CLOSE	OPEN EXTEND WRITE CLOSE	OPEN EXTEND WRITE CLOSE or OPEN I-O READ REWRITE DELETE CLOSE	OPEN EXTEND WRITE CLOSE or OPEN I-O READ REWRITE DELETE CLOSE
ACCESS IS RANDOM	Not applicable	Not applicable	OPEN I-O READ WRITE REWRITE DELETE CLOSE	OPEN I-O READ WRITE REWRITE DELETE CLOSE

Table 17. **PROCEDURE DIVISION statements used to update files** *(continued)*

Access method	Sequential	Line sequential	Indexed	Relative
ACCESS IS DYNAMIC (sequential processing)	Not applicable	Not applicable	OPEN I-O READ NEXT READ PREVIOUS START CLOSE	OPEN I-O READ NEXT READ PREVIOUS START CLOSE
ACCESS IS DYNAMIC (random processing)	Not applicable	Not applicable	OPEN I-O READ WRITE REWRITE DELETE CLOSE	OPEN I-O READ WRITE REWRITE DELETE CLOSE

RELATED CONCEPTS
"File organization and access mode" on page 122

RELATED TASKS
"Opening a file" on page 129
"Reading records from a file" on page 132
"Adding records to a file" on page 133
"Replacing records in a file" on page 134
"Deleting records from a file" on page 135

RELATED REFERENCES
"Statements used when writing records to a file" on page 133

Enabling COBOL applications to access SFS files

To access Encina SFS files from a COBOL application that runs under AIX, you must follow guidelines for compiling and linking, and for identifying the file system, Encina SFS server, and SFS files.

| 1. Compile and link the COBOL programs in the application by using the `cob2_r` command.
2. Use the VSAM file system. (`FILESYS(VSA)` is the default for the `FILESYS` runtime option.)
3. Ensure that the Encina SFS server that your application will access is running.
4. Set the ENCINA_CDS_ROOT environment variable to the fully qualified name of the Encina SFS server.

   For example:

   ```
   export ENCINA_CDS_ROOT=/.:/cics/sfs/test1
   ```
5. (Optional) If your application creates one or more SFS files, and you want to allocate the files on an SFS data volume that has a name other than encinaSfsVol, you can specify either or both of the following names:
   - The name of the SFS data volume on which the SFS files are to be created. To do so, assign a value to the runtime environment variable ENCINA_SFS_DATA_VOLUME. The data volume must have been defined to the SFS server. If you do not know which data volumes are available to the SFS server, issue the command `sfsadmin list lvols`.

     By default, SFS files are created on the data volume that is named encinaSfsVol.

- The name of the SFS data volume on which alternate index files, if any, are to be created. To do so, assign a value to the runtime environment variable ENCINA_SFS_INDEX_VOLUME. The data volume must have been defined to the SFS server.

  By default, alternate index files are created on the same volume as the corresponding base files.

6. Identify each SFS file by using its fully qualified name:
   - The Encina SFS server name must precede the file name.
   - The alternate index file names must start with the base file name, followed by a semicolon (`;`) and the alternate index name.

   For example, if /.:/cics/sfs/test1 is the Encina SFS server, and VSAM04A is an SFS file that has alternate index VSAM04A1, you could identify VSAM04A by issuing the following `export` command:

   ```
   export VSAM04A="/.:/cics/sfs/test1/VSAM04A(/.:/cics/sfs/test1/VSAM04A;VSAM04A1)"
   ```

For more information about fully qualified names for SFS servers and files, see the related task about identifying Encina SFS files.

"Example: accessing SFS files"

RELATED TASKS
"Identifying Encina SFS files" on page 114
"Compiling for SFS files" on page 220
"Improving SFS performance" on page 139
Encina SFS Programming Guide

RELATED REFERENCES
"SFS file system" on page 117
"Runtime environment variables" on page 211
"FILESYS" on page 303
Encina Administration Guide Volume 2: Server Administration

Example: accessing SFS files

The following example shows COBOL file descriptions that you could code and `sfsadmin` and `export` commands that you could issue to create and access two SFS files.

VSAM04 is an indexed file that has no alternate index. VSAM04A is an indexed file that has one alternate index, VSAM04A1.

COBOL file descriptions

```
. . .
input-output section.
file-control.
    select VSAM04-file
        assign to VSAM04
        access dynamic
        organization is indexed
        record key is VSAM04-rec-num
        file status is VSAM04-status.

    select VSAM04A-file
        assign to VSAM04A
        access dynamic
        organization is indexed
        record key is VSAM04A-rec-num
```

```
        alternate record key is VSAM04A-date-time
        file status is VSAM04A-status.

data division.
file section.
FD  VSAM04-file
    data record is VSAM04-record.
  01 VSAM04-record.
     05 VSAM04-rec-num              pic x(10).
     05 VSAM04-rec-data             pic x(70).
FD  VSAM04A-file
    data record is VSAM04A-record.
  01 VSAM04A-record.
     05 VSAM04A-rec-num             pic x(10).
     05 VSAM04A-date-time.
        07 VSAM04A-date-yyyymmdd     pic 9(8).
        07 VSAM04A-time-hhmmsshh     pic 9(8).
        07 VSAM04A-date-time-counter pic 9(8).
     05 VSAM04A-rec-data            pic x(1000).
```

sfsadmin commands

Create each indexed file by issuing the `sfsadmin create clusteredfile` command, and add an alternate index by issuing the `sfsadmin add index` command:

```
sfsadmin create clusteredfile VSAM04 2 \
PrimaryKey byteArray 10 \
DATA byteArray 70 \
primaryIndex -unique 1 PrimaryKey sfs_SFS_SERV
#
sfsadmin create clusteredfile VSAM04A 3 \
PrimaryKey byteArray 10 \
AltKey1 byteArray 24 \
DATA byteArray 1000 \
primaryIndex -unique 1 PrimaryKey sfs_SFS_SERV
#
sfsadmin add index VSAM04A VSAM04A1 1 AltKey1
```

As shown in the first `sfsadmin create clusteredfile` command above, you must specify the following items:

- The name of the new indexed file (`VSAM04` in this example)
- The number of fields per record (2)
- The description of each field (`PrimaryKey` and `DATA`, each a byte array)
- The name of the primary index (`primaryIndex`)
- | The `-unique` option
- The number of fields in the primary index (1)
- The names of the fields in the primary index (`PrimaryKey`)
- The name of the data volume on which the file is to be stored (`sfs_SFS_SERV`)

| By default, Encina SFS allows duplicate keys in the primary index of a clustered
| file. However, you must specify the `-unique` option as shown above because
| COBOL requires that key values in the primary index be unique within the file.

As shown in the `sfsadmin add index` command above, you must specify the following items:

- The name of the file to which an alternate index is to be added (`VSAM04A` in this example)
- The name of the new index (`VSAM04A1`)
- The number of fields to be used as keys in the new index (1)

- The names of the fields in the new index (`AltKey1`)

For details about the syntax of the commands `sfsadmin create clusteredfile` and `sfsadmin add index`, see the related references below.

export commands

Before you run the program that processes the SFS files, issue these `export` commands to specify the Encina SFS server (`/.:/cics/sfs/test1`) that will access the files, and the data volume (`sfs_SFS_SERV`) that will store the files:

```
# Set environment variables required by the VSAM file system
# for SFS files:
export ENCINA_CDS_ROOT=/.:/cics/sfs/test1
export ENCINA_SFS_DATA_VOLUME=sfs_SFS_SERV
export ENCINA_SFS_INDEX_VOLUME=sfs_SFS_SERV
# Set environment variable to access SFS file VSAM04:
export VSAM04=/.:/cics/sfs/test1/VSAM04
# Set environment variable to access SFS file VSAM04A and the
# alternate index VSAM04A1:
export VSAM04A="/.:/cics/sfs/test1/VSAM04A(/.:/cics/sfs/test1/VSAM04A;VSAM04A1)"
```

RELATED REFERENCES
Encina Administration Guide Volume 2: Server Administration

Improving SFS performance

You can improve the performance of applications that access SFS files in two ways: by using client-side caching on the client machine, and by reducing the frequency of saving changes to SFS files.

RELATED TASKS
"Identifying Encina SFS files" on page 114
"Enabling client-side caching"
"Reducing the frequency of saving changes" on page 140
Encina SFS Programming Guide

RELATED REFERENCES
"SFS file system" on page 117

Enabling client-side caching

By default, records in SFS files are written to and read from the SFS server, and a remote procedure call (RPC) is needed whenever a record is accessed. If you enable *client-side caching*, however, you can improve performance because less time is needed to access records.

With client-side caching, records are stored in local memory (a *cache*) on the client machine and sent (*flushed*) to the server in a single RPC. You can specify either or both of two types of caching: *read caching* and *insert caching*:

- If you enable read caching, the first sequential read of a file causes the current record and a number of neighboring records to be read from the server and placed in the read cache. Subsequent sequential reads, updates, and deletes are made in the read cache rather than on the server.
- If you enable insert caching, insert operations (but not reads, updates, or deletes) are done in the insert cache rather than on the server.

To enable client-side caching for all the SFS files in your application, set the ENCINA_VSAM_CACHE environment variable before you run the application. See the related reference below about runtime environment variables for a syntax diagram for setting the ENCINA_VSAM_CACHE environment variable.

You can code a single value for the cache size to indicate that the same number of pages is to be used for the read cache and for the insert cache, or you can code distinct values for the read and insert cache by separating the values by a colon (`:`). Express size units as numbers of pages. If you code zero as the size of the read cache, insert cache, or both, that type of caching is disabled. For example, the following command sets the size of the read cache to 16 pages and the size of the insert cache to 64 pages for each SFS file in the application:

```
export ENCINA_VSAM_CACHE=16:64
```

You can also specify one or both of the following flags to make client-side caching more flexible:

- To allow uncommitted data (records that are new or modified but have not yet been sent to the server, known as *dirty* records) to be read, specify ALLOW_DIRTY_READS.

  This flag removes the restriction for read caching that the files being accessed must be locked.
- To allow any inserts to be cached, specify INSERTS_DESPITE_UNIQUE_INDICES.

  This flag removes the restriction for insert caching that all active indices for clustered files and active alternate indices for entry-sequenced and relative files must allow duplicates.

For example, the following command allows maximum flexibility:

```
export ENCINA_VSAM_CACHE=16:64:ALLOW_DIRTY_READS,INSERTS_DESPITE_UNIQUE_INDICES
```

To set client-side caching differently for certain files, code a putenv() call that sets ENCINA_VSAM_CACHE before the `OPEN` statement for each file for which you want to change the caching policy. During a program, the environment-variable settings that you make in a putenv() call take precedence over the environment-variable settings that you make in an `export` command.

"Example: setting and accessing environment variables" on page 217

RELATED TASKS
"Setting environment variables" on page 209

RELATED REFERENCES
"SFS file system" on page 117
"Runtime environment variables" on page 211

Reducing the frequency of saving changes

An RPC normally occurs for each write or update operation on an SFS file that does not use client-side caching (that is, the *operational force* feature of SFS is enabled). All file changes that result from input-output operations are committed to disk before control returns to the application.

You can change this behavior so that the result of input-output operations on SFS files might not be committed to disk until the files are closed (that is, a *lazy-write* strategy is used). If each change to an SFS file is not immediately saved, the application can run faster.

To change the default commit behavior for all the SFS files in your application, set the ENCINA_VSAM_AUTO_FLUSH environment variable to `OFF` before you run the application:

```
export ENCINA_VSAM_AUTO_FLUSH=OFF
```

To set the flush value differently for certain files, code a putenv() call that sets ENCINA_VSAM_AUTO_FLUSH before the `OPEN` statement for each file for which you want to change the flush value. During a program, the environment-variable settings that you make in a putenv() call take precedence over the environment-variable settings that you make in an `export` command.

"Example: setting and accessing environment variables" on page 217

If client-side caching is in effect for an SFS file (that is, environment variable ENCINA_VSAM_CACHE is set to a valid nonzero value), the setting of ENCINA_VSAM_AUTO_FLUSH for the file is ignored. Operational force is disabled for that file.

RELATED TASKS
"Setting environment variables" on page 209

RELATED REFERENCES
"SFS file system" on page 117
"Runtime environment variables" on page 211

Chapter 8. Sorting and merging files

You can arrange records in a particular sequence by using a SORT or MERGE statement. You can mix SORT and MERGE statements in the same COBOL program.

SORT statement
: Accepts input (from a file or an internal procedure) that is not in sequence, and produces output (to a file or an internal procedure) in a requested sequence. You can add, delete, or change records before or after they are sorted.

MERGE statement
: Compares records from two or more sequenced files and combines them in order. You can add, delete, or change records after they are merged.

A program can contain any number of sort and merge operations. They can be the same operation performed many times or different operations. However, one operation must finish before another begins.

The steps you take to sort or merge are generally as follows:
1. Describe the sort or merge file to be used for sorting or merging.
2. Describe the input to be sorted or merged. If you want to process the records before you sort them, code an input procedure.
3. Describe the output from sorting or merging. If you want to process the records after you sort or merge them, code an output procedure.
4. Request the sort or merge.
5. Determine whether the sort or merge operation was successful.

RELATED CONCEPTS
"Sort and merge process"

RELATED TASKS
"Describing the sort or merge file" on page 144
"Describing the input to sorting or merging" on page 145
"Describing the output from sorting or merging" on page 147
"Requesting the sort or merge" on page 148
"Determining whether the sort or merge was successful" on page 151
"Stopping a sort or merge operation prematurely" on page 154

RELATED REFERENCES
SORT statement (*COBOL for AIX Language Reference*)
MERGE statement (*COBOL for AIX Language Reference*)

Sort and merge process

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

If there are multiple keys, the records are first sorted according to the content of the first (or primary) key, then according to the content of the second key, and so on.

To sort a file, use the COBOL SORT statement.

During the merging of two or more files (which must already be sorted), the records are combined and ordered according to the contents of one or more keys in each record. You can order the records in either ascending or descending order of each key. As with sorting, the records are first ordered according to the content of the primary key, then according to the content of the second key, and so on.

Use MERGE . . . USING to name the files that you want to combine into one sequenced file. The merge operation compares keys in the records of the input files, and passes the sequenced records one by one to the RETURN statement of an output procedure or to the file that you name in the GIVING phrase.

RELATED TASKS
"Setting sort or merge criteria" on page 149

RELATED REFERENCES
SORT statement (*COBOL for AIX Language Reference*)
MERGE statement (*COBOL for AIX Language Reference*)

Describing the sort or merge file

Describe the sort file to be used for sorting or merging. You need SELECT clauses and SD entries even if you are sorting or merging data items only from WORKING-STORAGE or LOCAL-STORAGE.

Code as follows:

1. Write one or more SELECT clauses in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION to name a sort file. For example:

   ```
   ENVIRONMENT DIVISION.
   INPUT-OUTPUT SECTION.
   FILE-CONTROL.
       SELECT Sort-Work-1 ASSIGN TO SortFile.
   ```

   Sort-Work-1 is the name of the file in your program. Use this name to refer to the file.
2. Describe the sort file in an SD entry in the FILE SECTION of the DATA DIVISION. Every SD entry must contain a record description. For example:

   ```
   DATA DIVISION.
   FILE SECTION.
   SD  Sort-Work-1
       RECORD CONTAINS 100 CHARACTERS.
   01  SORT-WORK-1-AREA.
       05  SORT-KEY-1   PIC  X(10).
       05  SORT-KEY-2   PIC  X(10).
       05  FILLER       PIC  X(80).
   ```

The file described in an SD entry is the working file used for a sort or merge operation. You cannot perform any input or output operations on this file.

RELATED REFERENCES
"FILE SECTION entries" on page 13

Describing the input to sorting or merging

Describe the input file or files for sorting or merging by following the procedure below.

1. Write one or more SELECT clauses in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION to name the input files. For example:

   ```
   ENVIRONMENT DIVISION.
   INPUT-OUTPUT SECTION.
   FILE-CONTROL.
       SELECT Input-File ASSIGN TO InFile.
   ```

   Input-File is the name of the file in your program. Use this name to refer to the file.
2. Describe the input file (or files when merging) in an FD entry in the FILE SECTION of the DATA DIVISION. For example:

   ```
   DATA DIVISION.
   FILE SECTION.
   FD  Input-File
       RECORD CONTAINS 100 CHARACTERS.
   01  Input-Record   PIC X(100).
   ```

RELATED TASKS
"Coding the input procedure" on page 146
"Requesting the sort or merge" on page 148

RELATED REFERENCES
"FILE SECTION entries" on page 13

Example: describing sort and input files for SORT

The following example shows the ENVIRONMENT DIVISION and DATA DIVISION entries needed to describe sort work files and an input file.

```
 ID Division.
 Program-ID. SmplSort.
 Environment Division.
 Input-Output Section.
 File-Control.
*
* Assign name for a working file is treated as documentation.
*
     Select Sort-Work-1 Assign To SortFile.
     Select Sort-Work-2 Assign To SortFile.
     Select Input-File  Assign To InFile.
 . . .
 Data Division.
 File Section.
 SD  Sort-Work-1
     Record Contains 100 Characters.
 01  Sort-Work-1-Area.
     05  Sort-Key-1    Pic  X(10).
     05  Sort-Key-2    Pic  X(10).
     05  Filler        Pic  X(80).
 SD  Sort-Work-2
     Record Contains 30 Characters.
 01  Sort-Work-2-Area.
     05  Sort-Key      Pic  X(5).
     05  Filler        Pic  X(25).
 FD  Input-File
     Record Contains 100 Characters.
 01  Input-Record      Pic  X(100).
 . . .
```

```
 Working-Storage Section.
 01  EOS-Sw           Pic  X.
 01  Filler.
     05  Table-Entry Occurs 100 Times
             Indexed By X1    Pic X(30).
     . . .
```

RELATED TASKS
"Requesting the sort or merge" on page 148

Coding the input procedure

To process the records in an input file before they are released to the sort program, use the INPUT PROCEDURE phrase of the SORT statement.

You can use an input procedure to:
- Release data items to the sort file from WORKING-STORAGE or LOCAL-STORAGE.
- Release records that have already been read elsewhere in the program.
- Read records from an input file, select or process them, and release them to the sort file.

Each input procedure must be contained in either paragraphs or sections. For example, to release records from a table in WORKING-STORAGE or LOCAL-STORAGE to the sort file SORT-WORK-2, you could code as follows:

```
    SORT SORT-WORK-2
      ON ASCENDING KEY SORT-KEY
      INPUT PROCEDURE 600-SORT3-INPUT-PROC
    . . .
600-SORT3-INPUT-PROC SECTION.
    PERFORM WITH TEST AFTER
      VARYING X1 FROM 1 BY 1 UNTIL X1 = 100
      RELEASE SORT-WORK-2-AREA FROM TABLE-ENTRY (X1)
    END-PERFORM.
```

To transfer records to the sort program, all input procedures must contain at least one RELEASE or RELEASE FROM statement. To release A from X, for example, you can code:

```
MOVE X TO A.
RELEASE A.
```

Alternatively, you can code:

```
RELEASE A FROM X.
```

The following table compares the RELEASE and RELEASE FROM statements.

RELEASE	**RELEASE FROM**
MOVE EXT-RECORD TO SORT-EXT-RECORD PERFORM RELEASE-SORT-RECORD . . . RELEASE-SORT-RECORD. RELEASE SORT-RECORD	PERFORM RELEASE-SORT-RECORD . . . RELEASE-SORT-RECORD. RELEASE SORT-RECORD FROM SORT-EXT-RECORD

RELATED REFERENCES
"Restrictions on input and output procedures" on page 148
RELEASE statement (*COBOL for AIX Language Reference*)

Describing the output from sorting or merging

If the output from sorting or merging is a file, describe the file by following the procedure below.

1. Write a SELECT clause in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION to name the output file. For example:

   ```
   ENVIRONMENT DIVISION.
   INPUT-OUTPUT SECTION.
   FILE-CONTROL.
       SELECT Output-File ASSIGN TO OutFile.
   ```

   Output-File is the name of the file in your program. Use this name to refer to the file.
2. Describe the output file (or files when merging) in an FD entry in the FILE SECTION of the DATA DIVISION. For example:

   ```
   DATA DIVISION.
   FILE SECTION.
   FD  Output-File
       RECORD CONTAINS 100 CHARACTERS.
   01  Output-Record   PIC X(100).
   ```

RELATED TASKS
"Coding the output procedure"
"Requesting the sort or merge" on page 148

RELATED REFERENCES
"FILE SECTION entries" on page 13

Coding the output procedure

To select, edit, or otherwise change sorted records before writing them from the sort work file into another file, use the OUTPUT PROCEDURE phrase of the SORT statement.

Each output procedure must be contained in either a section or a paragraph. An output procedure must include both of the following elements:

- At least one RETURN statement or one RETURN statement with the INTO phrase
- Any statements necessary to process the records that are made available, one at a time, by the RETURN statement

The RETURN statement makes each sorted record available to the output procedure. (The RETURN statement for a sort file is similar to a READ statement for an input file.)

You can use the AT END and END-RETURN phrases with the RETURN statement. The imperative statements in the AT END phrase are performed after all the records have been returned from the sort file. The END-RETURN explicit scope terminator delimits the scope of the RETURN statement.

If you use RETURN INTO instead of RETURN, the records will be returned to WORKING-STORAGE, LOCAL-STORAGE, or to an output area.

RELATED REFERENCES
"Restrictions on input and output procedures" on page 148
RETURN statement (*COBOL for AIX Language Reference*)

Restrictions on input and output procedures

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

- The procedure must not contain any SORT or MERGE statements.
- You can use ALTER, GO TO, and PERFORM statements in the procedure to refer to procedure-names outside the input or output procedure. However, control must return to the input or output procedure after a GO TO or PERFORM statement.
- The remainder of the PROCEDURE DIVISION must not contain any transfers of control to points inside the input or output procedure (with the exception of the return of control from a declarative section).
- In an input or output procedure, you can call a program. However, the called program cannot issue a SORT or MERGE statement, and the called program must return to the caller.
- During a SORT or MERGE operation, the SD data item is used. You must not use it in the output procedure before the first RETURN executes. If you move data into this record area before the first RETURN statement, the first record to be returned will be overwritten.

RELATED TASKS
"Coding the input procedure" on page 146
"Coding the output procedure" on page 147

Requesting the sort or merge

To read records from an input file (files for MERGE) without preliminary processing, use SORT . . . USING or MERGE . . . USING and the name of the input file (files) that you declared in a SELECT clause.

To transfer sorted or merged records from the sort or merge program to another file without any further processing, use SORT . . . GIVING or MERGE . . . GIVING and the name of the output file that you declared in a SELECT clause. For example:

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

For SORT . . . USING or MERGE . . . USING, the compiler generates an input procedure to open the file (files), read the records, release the records to the sort or merge program, and close the file (files). The file (files) must not be open when the SORT or MERGE statement begins execution. For SORT . . . GIVING or MERGE . . . GIVING, the compiler generates an output procedure to open the file, return the records, write the records, and close the file. The file must not be open when the SORT or MERGE statement begins execution.

"Example: describing sort and input files for SORT" on page 145

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 150

Restriction: You cannot use an input procedure with the MERGE statement. The source of input to the merge operation must be a collection of already sorted files. However, if you want an output procedure to be performed on the merged records, use MERGE . . . OUTPUT PROCEDURE. For example:

```
MERGE Merge-Work
    ON ASCENDING KEY Merge-Key
    USING Input-File-1 Input-File-2 Input-File-3
    OUTPUT PROCEDURE ProcessOutput.
```

In the FILE SECTION, you must define *Merge-Work* in an SD entry, and the input files in FD entries.

RELATED REFERENCES
SORT statement (*COBOL for AIX Language Reference*)
MERGE statement (*COBOL for AIX Language Reference*)

Setting sort or merge criteria

To set sort or merge criteria, define the keys on which the operation is to be performed.

Do these steps:

1. In the record description of the files to be sorted or merged, define the key or keys.

   Restriction: A key cannot be variably located.
2. In the SORT or MERGE statement, specify the key fields to be used for sequencing by coding the ASCENDING or DESCENDING KEY phrase, or both. When you code more than one key, some can be ascending, and some descending.

   Specify the names of the keys in decreasing order of significance. The leftmost key is the primary key. The next key is the secondary key, and so on.

SORT and MERGE keys can be of class alphabetic, alphanumeric, national (if the compiler option NCOLLSEQ(BIN) is in effect), or numeric (but not numeric of USAGE NATIONAL). If it has USAGE NATIONAL, a key can be of category national or can be a national-edited or numeric-edited data item. A key cannot be a national decimal data item or a national floating-point data item.

The collation order for national keys is determined by the binary order of the keys. If you specify a national data item as a key, any COLLATING SEQUENCE phrase in the SORT or MERGE statement does not apply to that key.

You can mix SORT and MERGE statements in the same COBOL program. A program can perform any number of sort or merge operations. However, one operation must end before another can begin.

RELATED TASKS
"Controlling the collating sequence with a locale" on page 199

RELATED REFERENCES
"NCOLLSEQ" on page 273
SORT statement (*COBOL for AIX Language Reference*)
MERGE statement (*COBOL for AIX Language Reference*)

Choosing alternate collating sequences

You can sort or merge records on a collating sequence that you specify for single-byte character keys. The default collating sequence is the collating sequence specified by the locale setting in effect at compile time unless you code the PROGRAM COLLATING SEQUENCE clause in the OBJECT-COMPUTER paragraph.

To override the default sequence, use the COLLATING SEQUENCE phrase of the SORT or MERGE statement. You can use different collating sequences for each SORT or MERGE statement in your program.

The PROGRAM COLLATING SEQUENCE clause and the COLLATING SEQUENCE phrase apply only to keys of class alphabetic or alphanumeric. The COLLATING SEQUENCE phrase is valid only when a single-byte ASCII code page is in effect.

RELATED TASKS
"Specifying the collating sequence" on page 8
"Controlling the collating sequence with a locale" on page 199
"Setting sort or merge criteria" on page 149

RELATED REFERENCES
OBJECT-COMPUTER paragraph (*COBOL for AIX Language Reference*)
SORT statement (*COBOL for AIX Language Reference*)
Classes and categories of data (*COBOL for AIX Language Reference*)

Example: sorting with input and output procedures

The following example shows the use of an input and an output procedure in a SORT statement. The example also shows how you can define a primary key (SORT-GRID-LOCATION) and a secondary key (SORT-SHIFT) before using them in the SORT statement.

```
DATA DIVISION.
. . .
SD  SORT-FILE
    RECORD CONTAINS 115 CHARACTERS
    DATA RECORD SORT-RECORD.
01  SORT-RECORD.
    05  SORT-KEY.
        10  SORT-SHIFT            PIC X(1).
        10  SORT-GRID-LOCATION    PIC X(2).
        10  SORT-REPORT           PIC X(3).
    05  SORT-EXT-RECORD.
        10  SORT-EXT-EMPLOYEE-NUM   PIC X(6).
        10  SORT-EXT-NAME           PIC X(30).
        10  FILLER                  PIC X(73).
. . .
WORKING-STORAGE SECTION.
01  TAB1.
    05 TAB-ENTRY OCCURS 10 TIMES
           INDEXED BY TAB-INDX.
        10  WS-SHIFT              PIC X(1).
        10  WS-GRID-LOCATION      PIC X(2).
        10  WS-REPORT             PIC X(3).
        10  WS-EXT-EMPLOYEE-NUM   PIC X(6).
        10  WS-EXT-NAME           PIC X(30).
        10  FILLER                PIC X(73).
. . .
PROCEDURE DIVISION.
    . . .
    SORT SORT-FILE
```

```
        ON ASCENDING KEY SORT-GRID-LOCATION SORT-SHIFT
        INPUT PROCEDURE 600-SORT3-INPUT
        OUTPUT PROCEDURE 700-SORT3-OUTPUT.
    . . .
600-SORT3-INPUT.
    PERFORM VARYING TAB-INDX FROM 1 BY 1 UNTIL TAB-INDX > 10
        RELEASE SORT-RECORD FROM TAB-ENTRY(TAB-INDX)
    END-PERFORM.
. . .
700-SORT3-OUTPUT.
    PERFORM VARYING TAB-INDX FROM 1 BY 1 UNTIL TAB-INDX > 10
        RETURN SORT-FILE INTO TAB-ENTRY(TAB-INDX)
            AT END DISPLAY 'Out Of Records In SORT File'
        END-RETURN
    END-PERFORM.
```

RELATED TASKS
"Requesting the sort or merge" on page 148

Determining whether the sort or merge was successful

The SORT or MERGE statement returns a completion code of either 0 (successful completion) or 16 (unsuccessful completion) after each sort or merge has finished. The completion code is stored in the SORT-RETURN special register.

You should test for successful completion after each SORT or MERGE statement. For example:

```
    SORT SORT-WORK-2
        ON ASCENDING KEY SORT-KEY
        INPUT PROCEDURE IS 600-SORT3-INPUT-PROC
        OUTPUT PROCEDURE IS 700-SORT3-OUTPUT-PROC.
    IF SORT-RETURN NOT=0
        DISPLAY "SORT ENDED ABNORMALLY. SORT-RETURN = " SORT-RETURN.
    . . .
600-SORT3-INPUT-PROC SECTION.
    . . .
700-SORT3-OUTPUT-PROC SECTION.
    . . .
```

If you do not reference SORT-RETURN anywhere in your program, the COBOL run time tests the completion code. If it is 16, COBOL issues a runtime diagnostic message and terminates the run unit (or the thread, in a multithreaded environment). The diagnostic message contains a sort or merge error number that can help you determine the cause of the problem.

If you test SORT-RETURN for one or more (but not necessarily all) SORT or MERGE statements, the COBOL run time does not check the completion code. However, you can obtain the sort or merge error number after any SORT or MERGE statement by calling the iwzGetSortErrno service; for example:

```
77  sortErrno    PIC 9(9)   COMP-5.
. . .
    CALL 'iwzGetSortErrno'  USING  sortErrno
. . .
```

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

RELATED REFERENCES
"Sort and merge error numbers" on page 152

Sort and merge error numbers

If you do not reference `SORT-RETURN` in your program, and the completion code from a sort or merge operation is 16, COBOL for AIX issues a runtime diagnostic message that contains one of the nonzero error numbers shown in the table below.

Table 18. **Sort and merge error numbers**

Error number	Description
0	No error
1	Record is out of order.
2	Equal-keyed records were detected.
3	Multiple main functions were specified (internal error).
4	Error in the parameter file
5	Parameter file could not be opened.
6	Operand missing from option
7	Operand missing from extended option
8	Invalid operand in option
9	Invalid operand in extended option
10	An invalid option was specified.
11	An invalid extended option was specified.
12	An invalid temporary directory was specified.
13	An invalid file-name was specified.
14	An invalid field was specified.
15	A field was missing in the record.
16	A field was too short in the record.
17	Syntax error in `SELECT` specification
18	An invalid constant was specified in `SELECT`.
19	Invalid comparison between constant and data type in `SELECT`
20	Invalid comparison between two data types in `SELECT`
21	Syntax error in format specification
22	Syntax error in reformat specification
23	An invalid constant was specified in the reformat specification.
24	Syntax error in sum specification
25	A flag was specified multiple times.
26	Too many outputs were specified.
27	No input source was specified.
28	No output destination was specified.
29	An invalid modifier was specified.
30	Sum is not allowed.
31	Record is too short.
32	Record is too long.
33	An invalid packed or zoned field was detected.
34	Read error on file

Table 18. **Sort and merge error numbers** *(continued)*

Error number	Description
35	Write error on file
36	Cannot open input file.
37	Cannot open message file.
38	VSAM file error
39	Insuficient space in target buffers
40	Not enough temporary disk space
41	Not enough space for output file
42	An unexpected signal was trapped.
43	Error was returned from the input exit.
44	Error was returned from the output exit.
45	Unexpected data was returned from the output user exit.
46	Invalid bytes used value was returned from input exit.
47	Invalid bytes used value was returned from output exit.
48	SMARTsort is not active.
49	Insufficient storage to continue execution
50	Parameter file was too large.
51	Nonmatching single quotation mark
52	Nonmatching quotation mark
53	Conflicting options were specified.
54	Length field in record is invalid.
55	Last field in record is invalid.
56	Required record format was not specified.
57	Cannot open output file.
58	Cannot open temporary file.
59	Invalid file organization
60	User exit is not supported with the specified file organization.
61	Locale is not known to the system.
62	Record contains an invalid multibyte character.
63	A VSAM organization was specified for the file, but the file is not VSAM.
64	No key specified to `SORT` is usable for definition of indexed output file.
65	VSAM fixed record length for the file does not agree with the specified record format.
66	The SMARTsort options file creation failed.
67	A fully qualified, nonrelative path name must be specified as a work directory.
68	A required option must be specified.
69	Path name is not valid.
79	Maximum number of temporary files has been reached.
501	Invalid function
502	Invalid record type

Table 18. **Sort and merge error numbers** *(continued)*

Error number	Description
503	Invalid record length
504	Type length error
505	Invalid type
506	Mismatched number of keys
507	Type is too long.
508	Invalid key offset
509	Invalid ascending or descending key
510	Invalid overlapping keys
511	No key was defined.
512	No input file was specified.
513	No output file was specified.
514	Mixed-type input files
515	Mixed-type output files
516	Invalid input work buffer
517	Invalid output work buffer
518	COBOL input I/O error
519	COBOL output I/O error
520	Unsupported function
521	Invalid key
522	Invalid `USING` file
523	Invalid `GIVING` file
524	No work directory was supplied.
525	Work directory does not exist.
526	Sort common was not allocated.
527	No storage for sort common
528	Binary buffer was not allocated.
529	Line-sequential file buffer was not allocated.
530	Work space allocation failed.
531	FCB allocation failed.

Stopping a sort or merge operation prematurely

To stop a sort or merge operation, move the integer 16 into the `SORT-RETURN` special register.

Move 16 into the register in either of the following ways:

- Use `MOVE` in an input or output procedure.

  Sort or merge processing will be stopped immediately after the next `RELEASE` or `RETURN` statement is performed.
- Reset the register in a declarative section entered during processing of a `USING` or `GIVING` file.

Sort or merge processing will be stopped on exit from the declarative section.

Control then returns to the statement following the SORT or MERGE statement.

Chapter 9. Handling errors

Put code in your programs that anticipates possible system or runtime problems. If you do not include such code, output data or files could be corrupted, and the user might not even be aware that there is a problem.

The error-handling code can take actions such as handling the situation, issuing a message, or halting the program. You might for example create error-detection routines for data-entry errors or for errors as your installation defines them. In any event, coding a warning message is a good idea.

COBOL for AIX contains special elements to help you anticipate and correct error conditions:
- `ON OVERFLOW` in `STRING` and `UNSTRING` operations
- `ON SIZE ERROR` in arithmetic operations
- Elements for handling input or output errors
- `ON EXCEPTION` or `ON OVERFLOW` in `CALL` statements

RELATED TASKS
"Handling errors in joining and splitting strings"
"Handling errors in arithmetic operations" on page 158
"Handling errors in input and output operations" on page 158
"Handling errors when calling programs" on page 165

Handling errors in joining and splitting strings

During the joining or splitting of strings, the pointer used by `STRING` or `UNSTRING` might fall outside the range of the receiving field. A potential overflow condition exists, but COBOL does not let the overflow happen.

Instead, the `STRING` or `UNSTRING` operation is not completed, the receiving field remains unchanged, and control passes to the next sequential statement. If you do not code the `ON OVERFLOW` phrase of the `STRING` or `UNSTRING` statement, you are not notified of the incomplete operation.

Consider the following statement:

```
String Item-1 space Item-2 delimited by Item-3
    into Item-4
    with pointer String-ptr
    on overflow
        Display "A string overflow occurred"
End-String
```

These are the data values before and after the statement is performed:

Data item	`PICTURE`	Value before	Value after
`Item-1`	`X(5)`	AAAAA	AAAAA
`Item-2`	`X(5)`	EEEAA	EEEAA
`Item-3`	`X(2)`	EA	EA
`Item-4`	`X(8)`	*bbbbbbbb*[1]	*bbbbbbbb*[1]

Data item	PICTURE	Value before	Value after
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

Example: checking for division by zero

The following example shows how you can code an ON SIZE ERROR imperative statement so that the program issues an informative message if division by zero occurs.

```
DIVIDE-TOTAL-COST.
    DIVIDE TOTAL-COST BY NUMBER-PURCHASED
        GIVING ANSWER
        ON SIZE ERROR
          DISPLAY "ERROR IN DIVIDE-TOTAL-COST PARAGRAPH"
          DISPLAY "SPENT " TOTAL-COST, " FOR " NUMBER-PURCHASED
          PERFORM FINISH
    END-DIVIDE
    . . .
    FINISH.
    STOP RUN.
```

If division by zero occurs, the program writes a message and halts program execution.

Handling errors in input and output operations

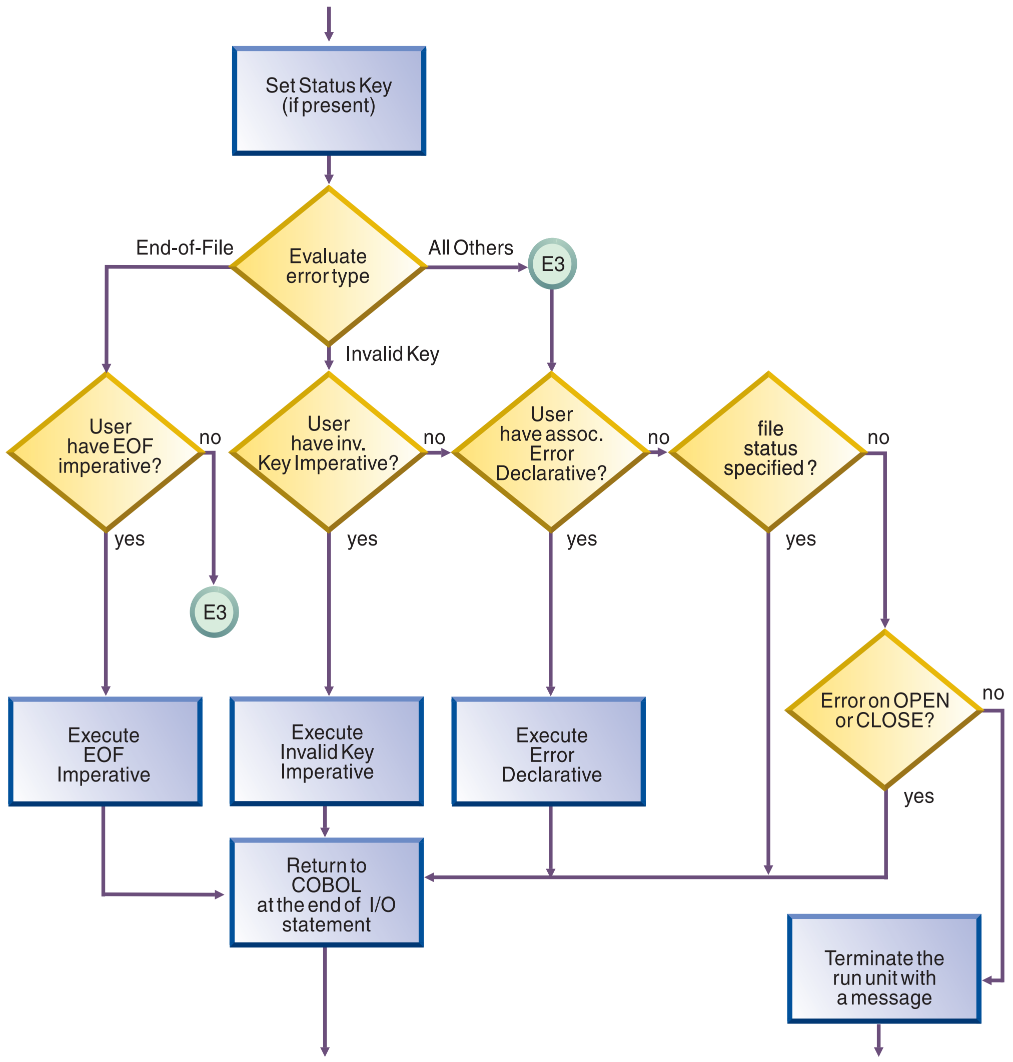
When an input or output operation fails, COBOL does not automatically take corrective action. You choose whether your program will continue running after a less-than-severe input or output error.

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

- End-of-file condition (AT END)
- ERROR declaratives
- FILE STATUS clause and file status key
- File system status code
- Imperative-statement phrases on READ or WRITE statements
- INVALID KEY phrase

To have your program continue, you must code the appropriate error-recovery procedure. You might code, for example, a procedure to check the value of the file status key. If you do not handle an input or output error in any of these ways, a COBOL runtime message is written and the run unit ends.

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



Set Status Key (if present)
Evaluate error type
End-of-File
All Others
E3
Invalid Key
User have EOF imperative?
no
yes
E3
User have inv. Key Imperative?
no
yes
User have assoc. Error Declarative?
no
yes
file status specified ?
no
yes
Error on OPEN or CLOSE?
no
yes
Execute EOF Imperative
Execute Invalid Key Imperative
Execute Error Declarative
Return to COBOL at the end of I/O statement
Terminate the run unit with a message

RELATED TASKS
"Using the end-of-file condition (AT END)" on page 160

"Coding ERROR declaratives"
"Using file status keys" on page 161
"Using file system status codes" on page 162
"Coding INVALID KEY phrases" on page 164

RELATED REFERENCES
File status key (*COBOL for AIX Language Reference*)

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

You code the AT END phrase of the READ statement to handle errors or normal conditions, according to your program design. At end-of-file, the AT END phrase is performed. If you do not code an AT END phrase, the associated ERROR declarative is performed.

In many designs, reading sequentially to the end of a file is done intentionally, and the AT END condition is expected. For example, suppose you are processing a file that contains transactions in order to update a master file:

```
PERFORM UNTIL TRANSACTION-EOF = "TRUE"
  READ UPDATE-TRANSACTION-FILE INTO WS-TRANSACTION-RECORD
    AT END
      DISPLAY "END OF TRANSACTION UPDATE FILE REACHED"
      MOVE "TRUE" TO TRANSACTION-EOF
  END READ
  . . .
END-PERFORM
```

Any NOT AT END phrase is performed only if the READ statement completes successfully. If the READ operation fails because of a condition other than end-of-file, neither the AT END nor the NOT AT END phrase is performed. Instead, control passes to the end of the READ statement after any associated declarative procedure is performed.

You might choose not to code either an AT END phrase or an EXCEPTION declarative procedure, but to code a status key clause for the file. In that case, control passes to the next sequential instruction after the input or output statement that detected the end-of-file condition. At that place, you should have some code that takes appropriate action.

RELATED REFERENCES
AT END phrases (*COBOL for AIX Language Reference*)

Coding ERROR declaratives

You can code one or more ERROR declarative procedures that will be given control if an input or output error occurs during the execution of your program. If you do not code such procedures, your job could be canceled or abnormally terminated after an input or output error occurs.

Place each such procedure in the declaratives section of the PROCEDURE DIVISION. You can code:

- A single, common procedure for the entire program
- Procedures for each file open mode (whether INPUT, OUTPUT, I-O, or EXTEND)
- Individual procedures for each file

In an ERROR declarative procedure, you can code corrective action, retry the operation, continue, or end execution. (If you continue processing a blocked file, though, you might lose the remaining records in a block after the record that caused the error.) You can use the ERROR declaratives procedure in combination with the file status key if you want a further analysis of the error.

RELATED REFERENCES
EXCEPTION/ERROR declarative (*COBOL for AIX Language Reference*)

Using file status keys

After each input or output statement is performed on a file, the system updates values in the two digit positions of the file status key. In general, a zero in the first position indicates a successful operation, and a zero in both positions means that nothing abnormal occurred.

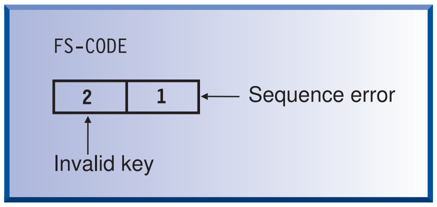
Establish a file status key by coding:
- The FILE STATUS clause in the FILE-CONTROL paragraph:
  ```
  FILE STATUS IS data-name-1
  ```
- Data definitions in the DATA DIVISION (WORKING-STORAGE, LOCAL-STORAGE, or LINKAGE SECTION), for example:
  ```
  WORKING-STORAGE SECTION.
  01  data-name-1  PIC 9(2)  USAGE NATIONAL.
  ```

Specify the file status key *data-name-1* as a two-character category alphanumeric or category national item, or as a two-digit zoned decimal or national decimal item. This *data-name-1* cannot be variably located.

Your program can check the file status key to discover whether an error occurred, and, if so, what type of error occurred. For example, suppose that a FILE STATUS clause is coded like this:

```
FILE STATUS IS FS-CODE
```

FS-CODE is used by COBOL to hold status information like this:



Follow these rules for each file:
- Define a different file status key for each file.

  Doing so means that you can determine the cause of a file input or output exception, such as an application logic error or a disk error.
- Check the file status key after each input or output request.

  If the file status key contains a value other than 0, your program can issue an error message or can take action based on that value.

  You do not have to reset the file status key code, because it is set after each input or output attempt.

In addition to the file status key, you can code a second identifier in the `FILE STATUS` clause to get more detailed information about file-system input or output requests. See the related reference below about file system status codes.

You can use the file status key alone or in conjunction with the `INVALID KEY` option, or to supplement the `EXCEPTION` or `ERROR` declarative. Using the file status key in this way gives you precise information about the results of each input or output operation.

"Example: file status key"
"Example: checking file system status codes" on page 163

RELATED TASKS
"Setting up a field for file status" on page 126
"Using file system status codes"

RELATED REFERENCES
FILE STATUS clause (*COBOL for AIX Language Reference*)
File status key (*COBOL for AIX Language Reference*)

Example: file status key

The following example shows how you can perform a simple check of the file status key after opening a file.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  SIMCHK.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT MASTERFILE ASSIGN TO AS-MASTERA
    FILE STATUS IS MASTER-CHECK-KEY
    . . .
DATA DIVISION.
. . .
WORKING-STORAGE SECTION.
01  MASTER-CHECK-KEY       PIC X(2).
. . .
PROCEDURE DIVISION.
    OPEN INPUT MASTERFILE
    IF MASTER-CHECK-KEY NOT = "00"
        DISPLAY "Nonzero file status returned from OPEN " MASTER-CHECK-KEY
    . . .
```

Using file system status codes

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

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

The data item *data-name-1* specifies the two-digit COBOL file status key, which must be a two-character category alphanumeric or category national item, or a two-digit zoned decimal or national decimal item. The data item *data-name-8* specifies a data item that contains the file-system status code when the COBOL file status key is not zero. *data-name-8* is at least 6 bytes long, and must be an alphanumeric item.

For STL file system input and output requests, if *data-name-8* is 6 bytes long, it contains the status code; if *data-name-8* is longer than 6 bytes, it also contains a message with further information:

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

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

For VSAM file-system input and output requests, define *data-name-8* with `PICTURE X(`*n*`)` and `USAGE DISPLAY` attributes, where *n* is 6 or greater. The `PICTURE` string value represents the first *n* bytes of the VSAM reply message structure. If the size *m* of the reply message structure is less than *n*, only the first *m* bytes contain useful information.

"Example: checking file system status codes"

RELATED REFERENCES
"VSAM file system" on page 116
"STL file system" on page 118
VSAM File System Reply Messages

Example: checking file system status codes

The following example reads an indexed file starting at the fifth record and checks the file status key after each input or output request. It displays the VSAM codes when the file status key is not zero.

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

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FILESYSFILE ASSIGN TO FILESYSFILE
    ORGANIZATION IS INDEXED
    ACCESS DYNAMIC
    RECORD KEY IS FILESYSFILE-KEY
    FILE STATUS IS FS-CODE, FILESYS-CODE.
DATA DIVISION.
FILE SECTION.
FD  FILESYSFILE
    RECORD  30.
01  FILESYSFILE-REC.
    10 FILESYSFILE-KEY          PIC X(6).
    10 FILLER                   PIC X(24).
WORKING-STORAGE SECTION.
01  RETURN-STATUS.
    05 FS-CODE                  PIC XX.
    05 FILESYS-CODE             PIC X(6).
PROCEDURE DIVISION.
    OPEN  INPUT FILESYSFILE.
    DISPLAY "OPEN INPUT FILESYSFILE FS-CODE: " FS-CODE.

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

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

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

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

    CLOSE FILESYSFILE.
    STOP RUN.

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

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

Coding INVALID KEY phrases

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

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

INVALID KEY phrases differ from ERROR declaratives in several ways. INVALID KEY phrases:

- Operate for only limited types of errors. ERROR declaratives encompass all forms.
- Are coded directly in the input or output verb. 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 165

Example: FILE STATUS and INVALID KEY

The following example shows how you can use the file status code and the INVALID KEY phrase to determine more specifically why an input or output statement failed.

Assume that you have a file that contains master customer records and you need to update some of these records with information from a transaction update file. The program reads each transaction record, finds the corresponding record in the master file, and makes the necessary updates. The records in both files contain a field for a customer number, and each record in the master file has a unique customer number.

The FILE-CONTROL entry for the master file of customer records includes statements that define indexed organization, random access, MASTER-CUSTOMER-NUMBER as the prime record key, and CUSTOMER-FILE-STATUS as the file status key.

```
.
.  (read the update transaction record)
.
MOVE "TRUE" TO TRANSACTION-MATCH
MOVE UPDATE-CUSTOMER-NUMBER TO MASTER-CUSTOMER-NUMBER
READ MASTER-CUSTOMER-FILE INTO WS-CUSTOMER-RECORD
  INVALID KEY
    DISPLAY "MASTER CUSTOMER RECORD NOT FOUND"
    DISPLAY "FILE STATUS CODE IS: " CUSTOMER-FILE-STATUS
    MOVE "FALSE" TO TRANSACTION-MATCH
END-READ
```

Handling errors when calling programs

When a program dynamically calls a separately compiled program, the called program might be unavailable. For example, the system might be out of storage or unable to locate the load module. If the CALL statement does not have an ON EXCEPTION or ON OVERFLOW phrase, your application might abend.

Use the ON EXCEPTION phrase to perform a series of statements and to perform your own error handling. For example, in the code fragment below, if program REPORTA is unavailable, control passes to the ON EXCEPTION phrase.

```
MOVE "REPORTA" TO REPORT-PROG
CALL REPORT-PROG
  ON EXCEPTION
    DISPLAY "Program REPORTA not available, using REPORTB."
    MOVE "REPORTB" TO REPORT-PROG
    CALL REPORT-PROG
    END-CALL
END-CALL
```

| The ON EXCEPTION phrase applies only to the availability of the called program on
| its initial load. If the called program is loaded but fails for any other reason (such
| as initialization), the ON EXCEPTION phrase is not performed.

Part 2. Enabling programs for international environments

Chapter 10. Processing data in an international environment . . . 169
COBOL statements and national data . . . 170
Intrinsic functions and national data. . . . 172
Unicode and the encoding of language characters 173
Using national data (Unicode) in COBOL . . . 174
  Defining national data items . . . 174
  Using national literals . . . 175
  Using national-character figurative constants 176
  Defining national numeric data items . . . 177
  National groups . . . 177
  Using national groups . . . 179
    Using national groups as elementary items 180
    Using national groups as group items . . . 180
  Storage of character data . . . 181
Converting to or from national (Unicode) representation . . . 182
  Converting alphanumeric, DBCS, and integer to national (MOVE) . . . 182
  Converting alphanumeric or DBCS to national (NATIONAL-OF) . . . 183
  Converting national to alphanumeric (DISPLAY-OF) . . . 184
  Overriding the default code page. . . . 184
  Example: converting to and from national data 184
Processing UTF-8 data . . . 185
Processing Chinese GB 18030 data . . . 186
Comparing national (UTF-16) data . . . 186
  Comparing two class national operands . . . 187
  Comparing class national and class numeric operands . . . 188
  Comparing national numeric and other numeric operands . . . 188
  Comparing national and other character-string operands . . . 189
  Comparing national data and alphanumeric-group operands. . . . 189
Coding for use of DBCS support . . . 189
  Declaring DBCS data . . . 190
  Using DBCS literals . . . 190
    Comparing DBCS literals . . . 191
  Testing for valid DBCS characters . . . 191
  Processing alphanumeric data items that contain DBCS data . . . 192

Chapter 11. Setting the locale . . . 193
The active locale . . . 193
Specifying the code page for character data . . . 194
Using environment variables to specify a locale 195
  Determination of the locale from system settings 196
  Types of messages for which translations are available . . . 197
Locales and code pages that are supported . . . 197
Controlling the collating sequence with a locale 199
  Controlling the alphanumeric collating sequence with a locale. . . . 200
  Controlling the DBCS collating sequence with a locale . . . 201
  Controlling the national collating sequence with a locale . . . 202
  Intrinsic functions that depend on collating sequence . . . 202
Accessing the active locale and code-page values 203
  Example: get and convert a code-page ID . . . 204

Chapter 10. Processing data in an international environment

COBOL for AIX supports Unicode UTF-16 as national character data at run time. UTF-16 provides a consistent and efficient way to encode plain text. Using UTF-16, you can develop software that will work with various national languages.

Use these COBOL facilities to code and compile programs that process national data and culturally sensitive collation orders for such data:

- Data types and literals:
  - Character data types, defined with the `USAGE NATIONAL` clause and a `PICTURE` clause that defines data of category national, national-edited, or numeric-edited
  - Numeric data types, defined with the `USAGE NATIONAL` clause and a `PICTURE` clause that defines a numeric data item (a *national decimal item*) or an external floating-point data item (a *national floating-point item*)
  - National literals, specified with literal prefix `N` or `NX`
  - Figurative constant `ALL` *national-literal*
  - Figurative constants `QUOTE`, `SPACE`, `HIGH-VALUE`, `LOW-VALUE`, or `ZERO`, which have national character (UTF-16) values when used in national-character contexts
- The COBOL statements shown in the related reference below about COBOL statements and national data
- Intrinsic functions:
  - `NATIONAL-OF` to convert an alphanumeric or double-byte character set (DBCS) character string to `USAGE NATIONAL` (UTF-16)
  - `DISPLAY-OF` to convert a national character string to `USAGE DISPLAY` in a selected code page (EBCDIC, ASCII, EUC, or UTF-8)
  - The other intrinsic functions shown in the related reference below about intrinsic functions and national data
- The `GROUP-USAGE NATIONAL` clause to define groups that contain only `USAGE NATIONAL` data items and that behave like elementary category national items in most operations
- Compiler options:
  - `NSYMBOL` to control whether national or DBCS processing is used for the `N` symbol in literals and `PICTURE` clauses
  - `NCOLLSEQ` to specify the collating sequence for comparison of national operands

You can also take advantage of implicit conversions of alphanumeric or DBCS data items to national representation. The compiler performs such conversions (in most cases) when you move these items to national data items, or compare these items with national data items.

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 173
"National groups" on page 177

RELATED TASKS
"Using national data (Unicode) in COBOL" on page 174
"Converting to or from national (Unicode) representation" on page 182

"Processing UTF-8 data" on page 185
"Processing Chinese GB 18030 data" on page 186
"Comparing national (UTF-16) data" on page 186
"Coding for use of DBCS support" on page 189
Chapter 11, "Setting the locale," on page 193

RELATED REFERENCES
"COBOL statements and national data"
"Intrinsic functions and national data" on page 172
"NCOLLSEQ" on page 273
"NSYMBOL" on page 273
Classes and categories of data (*COBOL for AIX Language Reference*)
Data categories and PICTURE rules (*COBOL for AIX Language Reference*)
MOVE statement (*COBOL for AIX Language Reference*)
General relation conditions (*COBOL for AIX Language Reference*)

COBOL statements and national data

You can use national data with the PROCEDURE DIVISION and compiler-directing statements shown in the table below.

Table 19. **COBOL statements and national data**

COBOL statement	Can be national	Comment	For more information
ACCEPT	*identifier-1*, *identifier-2*	*identifier-1* is converted from the code page indicated by the runtime locale only if input is from the terminal.	"Assigning input from a screen or file (ACCEPT)" on page 34
ADD	All identifiers can be numeric items that have USAGE NATIONAL. *identifier-3* (GIVING) can be numeric-edited with USAGE NATIONAL.		"Using COMPUTE and other arithmetic statements" on page 51
CALL	*identifier-2*, *identifier-3*, *identifier-4*, *identifier-5*; *literal-2*, *literal-3*		"Passing data" on page 481
COMPUTE	*identifier-1* can be numeric or numeric-edited with USAGE NATIONAL. *arithmetic-expression* can contain numeric items that have USAGE NATIONAL.		"Using COMPUTE and other arithmetic statements" on page 51
COPY . . . REPLACING	*operand-1*, *operand-2* of the REPLACING phrase		Chapter 15, "Compiler-directing statements," on page 295
DISPLAY	*identifier-1*	*identifier-1* is converted to the code page associated with the current locale.	"Displaying values on a screen or in a file (DISPLAY)" on page 35

Table 19. **COBOL statements and national data** *(continued)*

COBOL statement	Can be national	Comment	For more information
`DIVIDE`	All identifiers can be numeric items that have `USAGE NATIONAL`. *identifier-3* (`GIVING`) and *identifier-4* (`REMAINDER`) can be numeric-edited with `USAGE NATIONAL`.		"Using COMPUTE and other arithmetic statements" on page 51
`INITIALIZE`	*identifier-1*; *identifier-2* or *literal-1* of the `REPLACING` phrase	If you specify `REPLACING NATIONAL` or `REPLACING NATIONAL-EDITED`, *identifier-2* or *literal-1* must be valid as a sending operand in a move to *identifier-1*.	"Examples: initializing data items" on page 28
`INSPECT`	All identifiers and literals. (*identifier-2*, the `TALLYING` integer data item, can have `USAGE NATIONAL`.)	If any of these (other than *identifier-2*, the `TALLYING` identifier) have `USAGE NATIONAL`, all must be national.	"Tallying and replacing data items (INSPECT)" on page 103
`INVOKE`	Method-name as *identifier-2* or *literal-1*; *identifier-3* or *literal-2* in the `BY VALUE` phrase		"Invoking methods (INVOKE)" on page 418
`MERGE`	Merge keys, if you specify `NCOLLSEQ(BIN)`	The `COLLATING SEQUENCE` phrase does not apply.	"Setting sort or merge criteria" on page 149
`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
`MULTIPLY`	All identifiers can be numeric items that have `USAGE NATIONAL`. *identifier-3* (`GIVING`) can be numeric-edited with `USAGE NATIONAL`.		"Using COMPUTE and other arithmetic statements" on page 51
`SEARCH ALL` (binary search)	Both the key data item and its object of comparison	The key data item and its object of comparison must be compatible according to the rules of comparison. If the object of comparison is of class national, the key must be also.	"Doing a binary search (SEARCH ALL)" on page 77
`SORT`	Sort keys, if you specify `NCOLLSEQ(BIN)`	The `COLLATING SEQUENCE` phrase does not apply.	"Setting sort or merge criteria" on page 149
`STRING`	All identifiers and literals. (*identifier-4*, the `POINTER` integer data item, can have `USAGE NATIONAL`.)	If *identifier-3*, the receiving data item, is national, all identifiers and literals (other than *identifier-4*, the `POINTER` identifier) must be national.	"Joining data items (STRING)" on page 93

Table 19. **COBOL statements and national data** *(continued)*

COBOL statement	Can be national	Comment	For more information
`SUBTRACT`	All identifiers can be numeric items that have `USAGE NATIONAL`. *identifier-3* (`GIVING`) can be numeric-edited with `USAGE NATIONAL`.		"Using COMPUTE and other arithmetic statements" on page 51
`UNSTRING`	All identifiers and literals. (*identifier-6* and *identifier-7*, the `COUNT` and `TALLYING` integer data items, respectively, can have `USAGE NATIONAL`.)	If *identifier-4*, a receiving data item, has `USAGE NATIONAL`, the sending data item and each delimiter must have `USAGE NATIONAL`, and each literal must be national.	"Splitting data items (UNSTRING)" on page 95
`XML GENERATE`	*identifier-1* (the generated XML document); *identifier-2* (the source field or fields); *identifier-4* or *literal-4* (the namespace identifier); *identifier-5* or *literal-5* (the namespace prefix)		Chapter 22, "Producing XML output," on page 379
`XML PARSE`	*identifier-1* (the XML document)	The `XML-NTEXT` special register contains national character document fragments during parsing.	Chapter 21, "Processing XML input," on page 361

RELATED TASKS
"Defining numeric data" on page 39
"Displaying numeric data" on page 41
"Using national data (Unicode) in COBOL" on page 174
"Comparing national (UTF-16) data" on page 186

RELATED REFERENCES
"NCOLLSEQ" on page 273
Classes and categories of data (*COBOL for AIX Language Reference*)

Intrinsic functions and national data

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

Table 20. **Intrinsic functions and national character data**

Intrinsic function	Function type	For more information
`DISPLAY-OF`	Alphanumeric	"Converting national to alphanumeric (DISPLAY-OF)" on page 184
`LENGTH`	Integer	"Finding the length of data items" on page 110
`LOWER-CASE`, `UPPER-CASE`	National	"Converting to uppercase or lowercase (UPPER-CASE, LOWER-CASE)" on page 105
`NUMVAL`, `NUMVAL-C`	Numeric	"Converting to numbers (NUMVAL, NUMVAL-C)" on page 105
`MAX`, `MIN`	National	"Finding the largest or smallest data item" on page 108

Table 20. **Intrinsic functions and national character data** *(continued)*

Intrinsic function	Function type	For more information
`ORD-MAX`, `ORD-MIN`	Integer	"Finding the largest or smallest data item" on page 108
`REVERSE`	National	"Transforming to reverse order (REVERSE)" on page 105

You can use national decimal arguments wherever zoned decimal arguments are allowed. You can use national floating-point arguments wherever display floating-point arguments are allowed. (See the related reference below about arguments for a complete list of intrinsic functions that can take integer or numeric arguments.)

RELATED TASKS
"Defining numeric data" on page 39
"Using national data (Unicode) in COBOL" on page 174

RELATED REFERENCES
Arguments (*COBOL for AIX Language Reference*)
Classes and categories of data (*COBOL for AIX Language Reference*)

Unicode and the encoding of language characters

COBOL for AIX provides basic runtime support for Unicode, which can handle tens of thousands of characters that cover all commonly used characters and symbols in the world.

A *character set* is a defined set of characters, but is not associated with a coded representation. A *coded character set* (also referred to in this documentation as a *code page*) is a set of unambiguous rules that relate the characters of the set to their coded representation. Each code page has a name and is like a table that sets up the symbols for representing a character set; each symbol is associated with a unique bit pattern, or *code point*. Each code page also has a *coded character set identifier* (*CCSID*), which is a value from 1 to 65,536.

Unicode has several encoding schemes, called *Unicode Transformation Format (UTF)*, such as UTF-8, UTF-16, and UTF-32. COBOL for AIX uses UTF-16 (CCSID 1200) in big-endian format as the representation for national literals and data items that have `USAGE NATIONAL`.

UTF-8 represents ASCII invariant characters a-z, A-Z, 0-9, and certain special characters such as ' @ , . + - = / * ( ) the same way that they are represented in ASCII. UTF-16 represents these characters as `NX'00`*`nn`*`'`, where `X'`*`nn`*`'` is the representation of the character in ASCII.

For example, the string `'ABC'` is represented in UTF-16 as `NX'004100420043'`. In UTF-8, `'ABC'` is represented as `X'414243'`.

One or more *encoding units* are used to represent a character from a coded character set. For UTF-16, an encoding unit takes 2 bytes of storage. Any character defined in any EBCDIC, ASCII, or EUC code page is represented in one UTF-16 encoding unit when the character is converted to the national data representation.

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

RELATED TASKS
"Converting to or from national (Unicode) representation" on page 182

RELATED REFERENCES
"Storage of character data" on page 181
"Locales and code pages that are supported" on page 197
Character sets and code pages (*COBOL for AIX Language Reference*)

Using national data (Unicode) in COBOL

In COBOL for AIX, 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 173
"National groups" on page 177

RELATED TASKS
"Defining national data items"
"Using national literals" on page 175
"Using national-character figurative constants" on page 176
"Defining national numeric data items" on page 177
"Using national groups" on page 179
"Converting to or from national (Unicode) representation" on page 182
"Comparing national (UTF-16) data" on page 186

RELATED REFERENCES
"Storage of character data" on page 181
Classes and categories of data (*COBOL for AIX Language Reference*)

Defining national data items

Define national data items with the `USAGE NATIONAL` clause to hold national (UTF-16) character strings.

You can define national data items of the following categories:
- National

- National-edited
- Numeric-edited

To define a category national data item, code a PICTURE clause that contains only one or more PICTURE symbols N.

To define a national-edited data item, code a PICTURE clause that contains at least one of each of the following symbols:
- Symbol N
- Simple insertion editing symbol B, 0, or /

To define a numeric-edited data item of class national, code a PICTURE clause that defines a numeric-edited item (for example, -$999.99) and code a USAGE NATIONAL clause. You can use a numeric-edited data item that has USAGE NATIONAL in the same way that you use a numeric-edited item that has USAGE DISPLAY.

You can also define a data item as numeric-edited by coding the BLANK WHEN ZERO clause for an elementary item that is defined as numeric by its PICTURE clause.

If you code a PICTURE clause but do not code a USAGE clause for data items that contain only one or more PICTURE symbols N, you can use the compiler option NSYMBOL(NATIONAL) to ensure that such items are treated as national data items instead of as DBCS items.

RELATED TASKS
"Displaying numeric data" on page 41

RELATED REFERENCES
"NSYMBOL" on page 273
BLANK WHEN ZERO clause (*COBOL for AIX Language Reference*)

Using national literals

To specify national literals, use the prefix character N and compile with the option NSYMBOL(NATIONAL).

You can use either of these notations:
- N"*character-data*"
- N'*character-data*'

If you compile with the option NSYMBOL(DBCS), the literal prefix character N specifies a DBCS literal, not a national literal.

To specify a national literal as a hexadecimal value, use the prefix NX. You can use either of these notations:
- NX"*hexadecimal-digits*"
- NX'*hexadecimal-digits*'

Each of the following MOVE statements sets the national data item Y to the UTF-16 value of the characters 'AB':

```
01 Y pic NN usage national.
. . .
    Move NX"00410042" to Y
    Move N"AB"        to Y
    Move "AB"         to Y
```

Do not use alphanumeric hexadecimal literals in contexts that call for national literals, because such usage is easily misunderstood. For example, the following statement also results in moving the UTF-16 characters 'AB' (not the hexadecimal bit pattern 4142) to Y, where Y is defined as USAGE NATIONAL:

```
Move X"4142" to Y
```

| You cannot use national literals in the SPECIAL-NAMES paragraph or as
| program-names. You can use a national literal to name an object-oriented method
| in the METHOD-ID paragraph or to specify a method-name in an INVOKE statement.

Use the SOSI compiler option to control how shift-out and shift-in characters within a national literal are handled.

RELATED TASKS
"Using literals" on page 25

RELATED REFERENCES
"NSYMBOL" on page 273
"SOSI" on page 281
National literals (*COBOL for AIX Language Reference*)

Using national-character figurative constants

| You can use the figurative constant ALL *national-literal* in a context that requires
| national characters. ALL *national-literal* represents all or part of the string that is generated by successive concatenations of the encoding units that make up the national literal.

| You can use the figurative constants QUOTE, SPACE, HIGH-VALUE, LOW-VALUE, or ZERO in a context that requires national characters, such as a MOVE statement, an implicit move, or a relation condition that has national operands. In these contexts, the figurative constant represents a national-character (UTF-16) value.

| When you use the figurative constant HIGH-VALUE in a context that requires
| national characters, its value is NX'FFFF'. When you use LOW-VALUE in a context
| that requires national characters, its value is NX'0000'. You can use HIGH-VALUE or
| LOW-VALUE in a context that requires national characters only if the NCOLLSEQ(BIN)
| compiler option is in effect.

| **Restrictions:** You must not use HIGH-VALUE or the value assigned from HIGH-VALUE
| in a way that results in conversion of the value from one data representation to
| another (for example, between USAGE DISPLAY and USAGE NATIONAL, or between
| ASCII and EBCDIC when the CHAR(EBCDIC) compiler option is in effect). X'FF' (the
| value of HIGH-VALUE in an alphanumeric context when the EBCDIC collating
| sequence is being used) does not represent a valid EBCDIC or ASCII character, and
| NX'FFFF' does not represent a valid national character. Conversion of such a value
| to another representation results in a *substitution character* being used (not X'FF' or
| NX'FFFF'). Consider the following example:

```
01 natl-data  PIC NN  Usage National.
01 alph-data  PIC XX.
. . .
    MOVE HIGH-VALUE TO natl-data, alph-data
    IF natl-data = alph-data. . .
```

The `IF` statement above evaluates as false even though each of its operands was set to `HIGH-VALUE`. Before an elementary alphanumeric operand is compared to a national operand, the alphanumeric operand is treated as though it were moved to a temporary national data item, and the alphanumeric characters are converted to the corresponding national characters. When `X'FF'` is converted to UTF-16, however, the UTF-16 item gets a substitution character value and so does not compare equally to `NX'FFFF'`.

RELATED TASKS
"Converting to or from national (Unicode) representation" on page 182
"Comparing national (UTF-16) data" on page 186

RELATED REFERENCES
"CHAR" on page 250
"NCOLLSEQ" on page 273
Figurative constants (*COBOL for AIX Language Reference*)
DISPLAY-OF (*COBOL for AIX Language Reference*)

Defining national numeric data items

Define data items with the `USAGE NATIONAL` clause to hold numeric data that is represented in national characters (UTF-16). You can define national decimal items and national floating-point items.

To define a national decimal item, code a `PICTURE` clause that contains only the symbols `9`, `P`, `S`, and `V`. If the `PICTURE` clause contains `S`, the `SIGN IS SEPARATE` clause must be in effect for that item.

To define a national floating-point item, code a `PICTURE` clause that defines a floating-point item (for example, `+99999.9E-99`).

You can use national decimal items in the same way that you use zoned decimal items. You can use national floating-point items in the same way that you use display floating-point items.

RELATED TASKS
"Defining numeric data" on page 39
"Displaying numeric data" on page 41

RELATED REFERENCES
SIGN clause (*COBOL for AIX 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 93
"Splitting data items (UNSTRING)" on page 95
"Tallying and replacing data items (INSPECT)" on page 103
"Using national groups" on page 179

RELATED REFERENCES
GROUP-USAGE clause (*COBOL for AIX Language Reference*)

Using national groups

To define a group data item as a national group, code a GROUP-USAGE NATIONAL clause at the group level for the item. The group can contain only data items that explicitly or implicitly have USAGE NATIONAL.

The following data description entry specifies that a level-01 group and its subordinate groups are national group items:

```
01  Nat-Group-1    GROUP-USAGE NATIONAL.
    02  Group-1.
        04  Month    PIC 99.
        04  DayOf    PIC 99.
        04  Year     PIC 9999.
    02  Group-2    GROUP-USAGE NATIONAL.
        04  Amount   PIC 9(4).99  USAGE NATIONAL.
```

In the example above, Nat-Group-1 is a national group, and its subordinate groups Group-1 and Group-2 are also national groups. A GROUP-USAGE NATIONAL clause is implied for Group-1, and USAGE NATIONAL is implied for the subordinate items in Group-1. Month, DayOf, and Year are national decimal items, and Amount is a numeric-edited item that has USAGE NATIONAL.

You can subordinate national groups within alphanumeric groups as in the following example:

```
01  Alpha-Group-1.
    02  Group-1.
        04  Month   PIC 99.
        04  DayOf   PIC 99.
        04  Year    PIC 9999.
    02  Group-2   GROUP-USAGE NATIONAL.
        04  Amount  PIC 9(4).99.
```

In the example above, Alpha-Group-1 and Group-1 are alphanumeric groups; USAGE DISPLAY is implied for the subordinate items in Group-1. (If Alpha-Group-1 specified USAGE NATIONAL at the group level, USAGE NATIONAL would be implied for each of the subordinate items in Group-1. However, Alpha-Group-1 and Group-1 would be alphanumeric groups, not national groups, and would behave like alphanumeric groups during operations such as moves and compares.) Group-2 is a national group, and USAGE NATIONAL is implied for the numeric-edited item Amount.

You cannot subordinate alphanumeric groups within national groups. All elementary items within a national group must be explicitly or implicitly described as USAGE NATIONAL, and all group items within a national group must be explicitly or implicitly described as GROUP-USAGE NATIONAL.

RELATED CONCEPTS
"National groups" on page 177

RELATED TASKS
"Using national groups as elementary items" on page 180
"Using national groups as group items" on page 180

RELATED REFERENCES
GROUP-USAGE clause (*COBOL for AIX Language Reference*)

Using national groups as elementary items

In most cases, you can use a national group as though it were an elementary data item.

In the following example, a national group item, `Group-1`, is moved to a national-edited item, `Edited-date`. Because `Group-1` is treated as an elementary data item during the move, editing takes place in the receiving data item. The value in `Edited-date` after the move is `06/23/2008` in national characters.

```
01  Edited-date  PIC NN/NN/NNNN  USAGE NATIONAL.
01  Group-1    GROUP-USAGE NATIONAL.
    02 Month     PIC 99   VALUE 06.
    02 DayOf     PIC 99   VALUE 23.
    02 Year      PIC 9999 VALUE 2008.
    . . .
    MOVE Group-1 to Edited-date.
```

If `Group-1` were instead an alphanumeric group in which each of its subordinate items had `USAGE NATIONAL` (specified either explicitly with a `USAGE NATIONAL` clause on each elementary item, or implicitly with a `USAGE NATIONAL` clause at the group level), a group move, rather than an elementary move, would occur. Neither editing nor conversion would take place during the move. The value in the first eight character positions of `Edited-date` after the move would be `06232008` in national characters, and the value in the remaining two character positions would be 4 bytes of alphanumeric spaces.

RELATED TASKS
"Assigning values to group data items (MOVE)" on page 33
"Comparing national data and alphanumeric-group operands" on page 189
"Using national groups as group items"

RELATED REFERENCES
MOVE statement (*COBOL for AIX Language Reference*)

Using national groups as group items

In some cases when you use a national group, it is handled with group semantics; that is, the elementary items in the group are recognized or processed.

In the following example, an `INITIALIZE` statement that acts upon national group item `Group-OneN` causes the value 15 in national characters to be moved to only the numeric items in the group:

```
01  Group-OneN    Group-Usage National.
    05  Trans-codeN     Pic N   Value "A".
    05  Part-numberN    Pic NN  Value "XX".
    05  Trans-quanN     Pic 99  Value 10.
    . . .
    Initialize Group-OneN Replacing Numeric Data By 15
```

Because only `Trans-quanN` in `Group-OneN` above is numeric, only `Trans-quanN` receives the value 15. The other subordinate items are unchanged.

The table below summarizes the cases where national groups are processed with group semantics.

Table 21. **National group items that are processed with group semantics**

Language feature	Uses of national group items	Comment
`CORRESPONDING` phrase of the `ADD`, `SUBTRACT`, or `MOVE` statement	Specify a national group item for processing as a group in accordance with the rules of the `CORRESPONDING` phrase.	Elementary items within the national group are processed like elementary items that have `USAGE NATIONAL` within an alphanumeric group.
`INITIALIZE` statement	Specify a national group for processing as a group in accordance with the rules of the `INITIALIZE` statement.	Elementary items within the national group are initialized like elementary items that have `USAGE NATIONAL` within an alphanumeric group.
Name qualification	Use the name of a national group item to qualify the names of elementary data items and of subordinate group items in the national group.	Follow the same rules for qualification as for an alphanumeric group.
`THROUGH` phrase of the `RENAMES` clause	To specify a national group item in the `THROUGH` phrase, use the same rules as for an alphanumeric group item.	The result is an alphanumeric group item.
`FROM` phrase of the `XML GENERATE` statement	Specify a national group item in the `FROM` phrase for processing as a group in accordance with the rules of the `XML GENERATE` statement.	Elementary items within the national group are processed like elementary items that have `USAGE NATIONAL` within an alphanumeric group.

RELATED TASKS
"Initializing a structure (INITIALIZE)" on page 30
"Initializing a table (INITIALIZE)" on page 68
"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 110
"Generating XML output" on page 379

RELATED REFERENCES
Qualification (*COBOL for AIX Language Reference*)
RENAMES clause (*COBOL for AIX 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 22. **Encoding and size of alphanumeric, DBCS, and national data**

Characteristic	`DISPLAY`	`DISPLAY-1`	`NATIONAL`
Character encoding unit	1 byte	2 bytes	2 bytes
Code page	ASCII, EUC, UTF-8, or EBCDIC[3]	ASCII DBCS or EBCDIC DBCS[3]	UTF-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 22. **Encoding and size of alphanumeric, DBCS, and national data** *(continued)*

Characteristic	DISPLAY	DISPLAY-1	NATIONAL

1. National literals in your source program are converted to UTF-16 for use at run time.
2. Most characters are represented in UTF-16 using one encoding unit. In particular, the following characters are represented using a single UTF-16 encoding unit per character:
   - COBOL characters A-Z, a-z, 0-9, space, + - * / = $, ; . " ( ) > < :'
   - All characters that are converted from an EBCDIC, ASCII, or EUC code page
3. Depending on the locale, the `CHAR(NATIVE)` or `CHAR(EBCDIC)` option, and the EBCDIC_CODEPAGE environment variable settings

RELATED CONCEPTS
"Unicode and the encoding of language characters" on page 173

RELATED TASKS
"Specifying the code page for character data" on page 194

RELATED REFERENCES
"CHAR" on page 250

Converting to or from national (Unicode) representation

You can implicitly or explicitly convert data items to national (UTF-16) representation.

You can implicitly convert alphabetic, alphanumeric, DBCS, or integer data to national data by using the `MOVE` statement. Implicit conversions also take place in other COBOL statements, such as `IF` statements that compare an alphanumeric data item with a data item that has `USAGE NATIONAL`.

You can explicitly convert to and from national data items by using the intrinsic functions `NATIONAL-OF` and `DISPLAY-OF`, respectively. By using these intrinsic functions, you can specify a code page for the conversion that is different from the code page that is in effect for a data item.

RELATED TASKS
"Converting alphanumeric, DBCS, and integer to national (MOVE)"
"Converting alphanumeric or DBCS to national (NATIONAL-OF)" on page 183
"Converting national to alphanumeric (DISPLAY-OF)" on page 184
"Overriding the default code page" on page 184
"Comparing national (UTF-16) data" on page 186
Chapter 11, "Setting the locale," on page 193

Converting alphanumeric, DBCS, and integer to national (MOVE)

You can use a `MOVE` statement to implicitly convert data to national representation.

You can move the following kinds of data to category national or national-edited data items, and thus convert the data to national representation:

- Alphabetic
- Alphanumeric

- Alphanumeric-edited
- DBCS
- Integer of `USAGE DISPLAY`
- Numeric-edited of `USAGE DISPLAY`

You can likewise move the following kinds of data to numeric-edited data items that have `USAGE NATIONAL`:
- Alphanumeric
- Display floating-point (floating-point of `USAGE DISPLAY`)
- Numeric-edited of `USAGE DISPLAY`
- Integer of `USAGE DISPLAY`

For complete rules about moves to national data, see the related reference about the `MOVE` statement.

For example, the `MOVE` statement below moves the alphanumeric literal `"AB"` to the national data item `UTF16-Data`:

```
01  UTF16-Data  Pic N(2) Usage National.
    . . .
    Move "AB" to UTF16-Data
```

After the `MOVE` statement above, `UTF16-Data` contains `NX'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 41
“Coding for use of DBCS support” on page 189

RELATED REFERENCES
MOVE statement (*COBOL for AIX Language Reference*)

Converting alphanumeric or DBCS to national (NATIONAL-OF)

Use the `NATIONAL-OF` intrinsic function to convert alphabetic, alphanumeric, or DBCS data to a national data item. Specify the source code page as the second argument if the source is encoded in a different code page than is in effect for the data item.

“Example: converting to and from national data” on page 184

RELATED TASKS
“Processing UTF-8 data” on page 185
“Processing Chinese GB 18030 data” on page 186
“Processing alphanumeric data items that contain DBCS data” on page 192

RELATED REFERENCES
NATIONAL-OF (*COBOL for AIX Language Reference*)

Converting national to alphanumeric (DISPLAY-OF)

| Use the DISPLAY-OF intrinsic function to convert national data to an alphanumeric
| (USAGE DISPLAY) character string that is represented in a code page that you specify
| as the second argument.

| If you omit the second argument, the output code page is determined from the
| runtime locale.

If you specify an EBCDIC or ASCII code page that combines single-byte character set (SBCS) and DBCS characters, the returned string might contain a mixture of SBCS and DBCS characters. The DBCS substrings are delimited by shift-in and shift-out characters if the code page in effect for the function is an EBCDIC code page.

"Example: converting to and from national data"

RELATED CONCEPTS
"The active locale" on page 193

RELATED TASKS
"Processing UTF-8 data" on page 185
"Processing Chinese GB 18030 data" on page 186

RELATED REFERENCES
DISPLAY-OF (*COBOL for AIX Language Reference*)

Overriding the default code page

In some cases, you might need to convert data to or from a code page that differs from the code page that is in effect at run time. To do so, convert the item by using a conversion function in which you explicitly specify the code page.

If you specify a code page as an argument to the DISPLAY-OF intrinsic function, and the code page differs from the code page that is in effect at run time, do not use the function result in any operations that involve implicit conversion (such as an assignment to, or comparison with, a national data item). Such operations assume the runtime code page.

Example: converting to and from national data

The following example shows the NATIONAL-OF and DISPLAY-OF intrinsic functions and the MOVE statement for converting to and from national (UTF-16) data items. It also demonstrates the need for explicit conversions when you operate on strings that are encoded in multiple code pages.

```
* . . .
 01  Data-in-Unicode          pic N(100) usage national.
 01  Data-in-Greek            pic X(100).
 01  other-data-in-US-English pic X(12) value "PRICE in $ =".
* . . .
     Read Greek-file into Data-in-Greek
     Move function National-of(Data-in-Greek, "ISO8859-7")
         to Data-in-Unicode
* . . . process Data-in-Unicode here . . .
```

```
     Move function Display-of(Data-in-Unicode, "ISO8859-7")
         to Data-in-Greek
     Write Greek-record from Data-in-Greek
```

The example above works correctly because the input code page is specified. `Data-in-Greek` is converted as data represented in ISO8859-7 (Ascii Greek). However, the following statement results in an incorrect conversion unless all the characters in the item happen to be among those that have a common representation in both the Greek and the English code pages:

```
Move Data-in-Greek to Data-in-Unicode
```

Assuming that the locale in effect is en_US.ISO8859-1, the `MOVE` statement above converts `Data-in-Greek` to Unicode based on the code page ISO8859-1 to UTF-16BE conversion. This conversion does not produce the expected results because `Data-in-Greek` is encoded in ISO8859-7.

If you set the locale to el_GR.ISO8859-7 (that is, your program handles ASCII data in Greek), you can code the same example correctly as follows:

```
* . . .
 01  Data-in-Unicode pic N(100) usage national.
 01  Data-in-Greek   pic X(100).
* . . .
     Read Greek-file into Data-in-Greek
* . . . process Data-in-Greek here ...
* . . . or do the following (if need to process data in Unicode):
     Move Data-in-Greek to Data-in-Unicode
* . . . process Data-in-Unicode
     Move function Display-of(Data-in-Unicode) to Data-in-Greek
     Write Greek-record from Data-in-Greek
```

RELATED TASKS
Chapter 11, "Setting the locale," on page 193

Processing UTF-8 data

When you need to process UTF-8 data, first convert the UTF-8 data to UTF-16 in a national data item. After processing the national data, convert it back to UTF-8 for output. For the conversions, use the intrinsic functions `NATIONAL-OF` and `DISPLAY-OF`, respectively. Use code page 1208 for UTF-8 data.

You need to do two steps to convert ASCII or EBCDIC data to UTF-8 (unless the code page of the locale in effect is UTF-8, in which case the native alphanumeric data is already encoded in UTF-8):

1. Use the function `NATIONAL-OF` to convert the ASCII or EBCDIC string to a national (UTF-16) string.
2. Use the function `DISPLAY-OF` to convert the national string to UTF-8.

The following example converts Greek EBCDIC data to UTF-8:

```
01 Greek-EBCDIC pic X(10) value "αβγδεζηθ".
01 UnicodeString pic N(10).
01 UTF-8-String pic X(20).
    Move function National-of(Greek-EBCDIC, 00875) to UnicodeString
    Move function Display-of(UnicodeString, 01208) to UTF-8-String
```

Usage note: Use care if you use reference modification to refer to data encoded in UTF-8. UTF-8 characters are encoded with a varying number of bytes per character. Avoid operations that might split a multibyte character.

RELATED TASKS
"Referring to substrings of data items" on page 99
"Converting to or from national (Unicode) representation" on page 182
"Parsing XML documents encoded in UTF-8" on page 374

Processing Chinese GB 18030 data

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

COBOL for AIX supports GB 18030. If the code page specified for the locale in effect is GB18030 (a code page that supports GB 18030), `USAGE DISPLAY` data items that contain GB 18030 characters encoded in GB18030 can be processed in a program. GB 18030 characters take 1 to 4 bytes each. Therefore the program logic must be sensitive to the multibyte nature of the data.

You can process GB 18030 characters in these ways:
- Use national data items to define and process GB 18030 characters that are represented in UTF-16, CCSID 01200.
- Process data in any code page (including GB18030, which has CCSID 1392) by converting the data to UTF-16, processing the UTF-16 data, and then converting the data back to the original code-page representation.

When you need to process Chinese GB 18030 data that requires conversion, first convert the input data to UTF-16 in a national data item. After you process the national data item, convert it back to Chinese GB 18030 for output. For the conversions, use the intrinsic functions `NATIONAL-OF` and `DISPLAY-OF`, respectively, and specify GB18030 or 1392 as the second argument of each function.

The following example illustrates these conversions:

```
 01 Chinese-ASCII pic X(16) value "奥林匹克运动会".
 01 Chinese-GB18030-String pic X(16).
 01 UnicodeString pic N(14).
 . . .
     Move function National-of(Chinese-ASCII, 1392) to UnicodeString
*  Process data in Unicode
     Move function Display-of(UnicodeString, 1392) to Chinese-GB18030-String
```

RELATED TASKS
"Converting to or from national (Unicode) representation" on page 182
"Coding for use of DBCS support" on page 189

RELATED REFERENCES
"Storage of character data" on page 181

Comparing national (UTF-16) data

You can compare national (UTF-16) data, that is, national literals and data items that have `USAGE NATIONAL` (whether of class national or class numeric), explicitly or implicitly with other kinds of data in relation conditions.

You can code conditional expressions that use national data in the following statements:

- `EVALUATE`
- `IF`
- `INSPECT`
- `PERFORM`
- `SEARCH`
- `STRING`
- `UNSTRING`

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

RELATED TASKS
"Comparing two class national operands"
"Comparing class national and class numeric operands" on page 188
"Comparing national numeric and other numeric operands" on page 188
"Comparing national and other character-string operands" on page 189
"Comparing national data and alphanumeric-group operands" on page 189

RELATED REFERENCES
Relation conditions (*COBOL for AIX Language Reference*)
General relation conditions (*COBOL for AIX Language Reference*)
National comparisons (*COBOL for AIX Language Reference*)
Group comparisons (*COBOL for AIX Language Reference*)

Comparing two class national operands

| You can compare the character values of two operands of class national.

| Either operand (or both) can be any of the following types of items:
| - A national group
| - An elementary category national or national-edited data item
| - A numeric-edited data item that has `USAGE NATIONAL`

| One of the operands can instead be a national literal or a national intrinsic
| function.

Use the `NCOLLSEQ` compiler option to determine which type of comparison to perform:

`NCOLLSEQ(BINARY)`
| When you compare two class national operands of the same length, they are determined to be equal if all pairs of the corresponding characters are equal. Otherwise, comparison of the binary values of the first pair of unequal characters determines the operand with the larger binary value.

When you compare operands of unequal lengths, the shorter operand is treated as if it were padded on the right with default UTF-16 space characters (`NX'0020'`) to the length of the longer operand.

`NCOLLSEQ(LOCALE)`
When you use a locale-based comparison, the operands are compared by using the algorithm for collation order that is associated with the locale in

effect. Trailing spaces are truncated from the operands, except that an operand that consists of all spaces is truncated to a single space.

When you compare operands of unequal lengths, the shorter operand is not extended with spaces because such an extension could alter the expected results for the locale.

The `PROGRAM COLLATING SEQUENCE` clause does not affect the comparison of two class national operands.

RELATED CONCEPTS
"National groups" on page 177

RELATED TASKS
"Using national groups" on page 179

RELATED REFERENCES
"NCOLLSEQ" on page 273
National comparisons (*COBOL for AIX Language Reference*)

Comparing class national and class numeric operands

You can compare national literals or class national data items to integer literals or numeric data items that are defined as integer (that is, national decimal items or zoned decimal items). At most one of the operands can be a literal.

You can also compare national literals or class national data items to floating-point data items (that is, display floating-point or national floating-point items).

Numeric operands are converted to national (UTF-16) representation if they are not already in national representation. A comparison is made of the national character values of the operands.

RELATED REFERENCES
General relation conditions (*COBOL for AIX 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 177

RELATED REFERENCES
General relation conditions (*COBOL for AIX Language Reference*)

Comparing national and other character-string operands

You can compare the character value of a national literal or class national data item with the character value of any of the following other character-string operands: alphabetic, alphanumeric, alphanumeric-edited, DBCS, or numeric-edited of USAGE DISPLAY.

These operands are treated as if they were moved to an elementary national data item. The characters are converted to national (UTF-16) representation, and the comparison proceeds with two national character operands.

RELATED TASKS
"Using national-character figurative constants" on page 176
"Comparing DBCS literals" on page 191

RELATED REFERENCES
National comparisons (*COBOL for AIX Language Reference*)

Comparing national data and alphanumeric-group operands

You can compare a national literal, a national group item, or any elementary data item that has USAGE NATIONAL to an alphanumeric group.

Neither operand is converted. The national operand is treated as if it were moved to an alphanumeric group item of the same size in bytes as the national operand, and the two groups are compared. An alphanumeric comparison is done regardless of the representation of the subordinate items in the alphanumeric group operand.

For example, Group-XN is an alphanumeric group that consists of two subordinate items that have USAGE NATIONAL:

```
01  Group-XN.
    02 TransCode PIC NN   Value "AB"  Usage National.
    02 Quantity  PIC 999  Value 123   Usage National.
    . . .
    If N"AB123" = Group-XN  Then Display "EQUAL"
    Else Display "NOT EQUAL".
```

When the IF statement above is executed, the 10 bytes of the national literal N"AB123" are compared byte by byte to the content of Group-XN. The items compare equally, and "EQUAL" is displayed.

RELATED REFERENCES
Group comparisons (*COBOL for AIX Language Reference*)

Coding for use of DBCS support

IBM COBOL for AIX supports using applications in any of many national languages, including languages that use double-byte character sets (DBCS).

The following list summarizes the support for DBCS:
- DBCS characters in user-defined words (multibyte names)
- DBCS characters in comments
- DBCS data items (defined with PICTURE N, G, or G and B)
- DBCS literals

- Collating sequence
- SOSI compiler option

RELATED TASKS
"Declaring DBCS data"
"Using DBCS literals"
"Testing for valid DBCS characters" on page 191
"Processing alphanumeric data items that contain DBCS data" on page 192
Chapter 11, "Setting the locale," on page 193
"Controlling the collating sequence with a locale" on page 199

RELATED REFERENCES
"SOSI" on page 281

Declaring DBCS data

Use the PICTURE and USAGE clauses to declare DBCS data items. DBCS data items can use PICTURE symbols G, G and B, or N. Each DBCS character position is 2 bytes in length.

You can specify a DBCS data item by using the USAGE DISPLAY-1 clause. When you use PICTURE symbol G, you must specify USAGE DISPLAY-1. When you use PICTURE symbol N but omit the USAGE clause, USAGE DISPLAY-1 or USAGE NATIONAL is implied depending on the setting of the NSYMBOL compiler option.

If you use a VALUE clause with the USAGE clause in the declaration of a DBCS item, you must specify a DBCS literal or the figurative constant SPACE or SPACES.

If a data item has USAGE DISPLAY-1 (either explicitly or implicitly), the selected locale must indicate a code page that includes DBCS characters. If the code page of the locale does not include DBCS characters, such data items are flagged as errors.

For the purpose of handling reference modifications, each character in a DBCS data item is considered to occupy the number of bytes that corresponds to the code-page width (that is, 2).

RELATED TASKS
Chapter 11, "Setting the locale," on page 193

RELATED REFERENCES
"Locales and code pages that are supported" on page 197
"NSYMBOL" on page 273

Using DBCS literals

You can use the prefix N or G to represent a DBCS literal.

That is, you can specify a DBCS literal in either of these ways:
- N'*dbcs characters*' (provided that the compiler option NSYMBOL(DBCS) is in effect)
- G'*dbcs characters*'

You can use quotation marks (") or single quotation marks (') as the delimiters of a DBCS literal irrespective of the setting of the APOST or QUOTE compiler option. You must code the same opening and closing delimiter for a DBCS literal.

If the `SOSI` compiler option is in effect, the shift-out (SO) control character `X'1E'` must immediately follow the opening delimiter, and the shift-in (SI) control character `X'1F'` must immediately precede the closing delimiter.

In addition to DBCS literals, you can use alphanumeric literals to specify any character in one of the supported code pages. However, if the `SOSI` compiler option is in effect, any string of DBCS characters that is within an alphanumeric literal must be delimited by the SO and SI characters.

You cannot continue an alphanumeric literal that contains multibyte characters. The length of a DBCS literal is likewise limited by the available space in Area B on a single source line. The maximum length of a DBCS literal is thus 28 double-byte characters.

An alphanumeric literal that contains multibyte characters is processed byte by byte, that is, with semantics appropriate for single-byte characters, except when it is converted explicitly or implicitly to national data representation, as for example in an assignment to or comparison with a national data item.

RELATED TASKS
"Comparing DBCS literals"
"Using figurative constants" on page 26

RELATED REFERENCES
"NSYMBOL" on page 273
"QUOTE/APOST" on page 278
"SOSI" on page 281
DBCS literals (*COBOL for AIX Language Reference*)

Comparing DBCS literals

Comparisons of DBCS literals are based on the compile-time locale. Therefore, do not use DBCS literals within a statement that expresses an implied relational condition between two DBCS literals (such as `VALUE G'`*literal-1*`' THRU G'`*literal-2*`'`) unless the intended runtime locale is the same as the compile-time locale.

RELATED TASKS
"Comparing national (UTF-16) data" on page 186
Chapter 11, "Setting the locale," on page 193

RELATED REFERENCES
"COLLSEQ" on page 253
DBCS literals (*COBOL for AIX Language Reference*)
DBCS comparisons (*COBOL for AIX Language Reference*)

Testing for valid DBCS characters

The Kanji class test tests for valid Japanese graphic characters. This testing includes Katakana, Hiragana, Roman, and Kanji character sets.

Kanji and DBCS class tests are defined to be consistent with their zSeries definitions. Both class tests are performed internally by converting the double-byte characters to the double-byte characters defined for z/OS. The converted double-byte characters are tested for DBCS and Japanese graphic characters.

The Kanji class test is done by checking the converted characters for the range `X'41'` through `X'7E'` in the first byte and `X'41'` through `X'FE'` in the second byte, plus the space character `X'4040'`.

The DBCS class test tests for valid graphic characters for the code page.

The DBCS class test is done by checking the converted characters for the range `X'41'` through `X'FE'` in both the first and second byte of each character, plus the space character `X'4040'`.

RELATED TASKS
"Coding conditional expressions" on page 86

RELATED REFERENCES
Class condition (*COBOL for AIX Language Reference*)

Processing alphanumeric data items that contain DBCS data

If you use byte-oriented operations (for example, `STRING`, `UNSTRING`, or reference modification) on an alphanumeric data item that contains DBCS characters, results are unpredictable. You should instead convert the item to a national data item before you process it.

That is, do these steps:

1. Convert the item to UTF-16 in a national data item by using a `MOVE` statement or the `NATIONAL-OF` intrinsic function.
2. Process the national data item as needed.
3. Convert the result back to an alphanumeric data item by using the `DISPLAY-OF` intrinsic function.

RELATED TASKS
"Joining data items (STRING)" on page 93
"Splitting data items (UNSTRING)" on page 95
"Referring to substrings of data items" on page 99
"Converting to or from national (Unicode) representation" on page 182

Chapter 11. Setting the locale

You can write applications to reflect the cultural conventions of the locale in effect when the applications are run. Cultural conventions include sort order, character classification, and national language; and formats of dates and times, numbers, monetary units, postal addresses, and telephone numbers.

With COBOL for AIX, you can select the appropriate code pages and collating sequences, and you can use language elements and compiler options to handle | Unicode, single-byte character sets, and double-byte character sets (DBCS).

RELATED CONCEPTS
"The active locale"

RELATED TASKS
"Specifying the code page for character data" on page 194
"Using environment variables to specify a locale" on page 195
"Controlling the collating sequence with a locale" on page 199
"Accessing the active locale and code-page values" on page 203

The active locale

A *locale* is a collection of data that encodes information about a cultural environment. The active locale is the locale that is in effect when you compile or run your program. You can establish a cultural environment for an application by specifying the active locale.

Only one locale can be active at a time.

The active locale affects the behavior of these culturally sensitive interfaces for the entire program:
- Code pages used for character data
- Messages
- Collating sequence
- Date and time formats
- Character classification and case conversion

The active locale does not affect the following items, for which Standard COBOL 85 defines specific language and behavior:
- Decimal point and grouping separators
- Currency sign

The active locale determines the code page for compiling and running programs:
- The code page that is used for compilation is based on the locale setting at compile time.
- The code page that is used for running an application is based on the locale setting at run time.

The evaluation of literal values in the source program is handled with the locale that is active at compile time. For example, the conversion of national literals from the source representation to UTF-16 for running the program uses the compile-time locale.

COBOL for AIX determines the setting of the active locale from a combination of the applicable environment variables and system settings. Environment variables are used first. If an applicable locale category is not defined by environment variables, COBOL uses defaults and system settings.

RELATED CONCEPTS
"Determination of the locale from system settings" on page 196

RELATED TASKS
"Specifying the code page for character data"
"Using environment variables to specify a locale" on page 195
"Controlling the collating sequence with a locale" on page 199

RELATED REFERENCES
"Types of messages for which translations are available" on page 197

Specifying the code page for character data

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

At run time, you can use the characters that are represented in a supported code page in data items described with `USAGE DISPLAY`, `USAGE DISPLAY-1`, or `USAGE NATIONAL`.

The code page that is in effect for a particular data item depends on the following aspects:
- Which `USAGE` clause you used
- Whether you used the `NATIVE` phrase with the `USAGE` clause
- Whether you used the `CHAR(NATIVE)` or `CHAR(EBCDIC)` compiler option
- The value of the EBCDIC_CODEPAGE environment variable
- Which locale is active

For `USAGE NATIONAL` data items, the code page is always UTF-16 in big-endian format.

For `USAGE DISPLAY` data items, COBOL for AIX chooses between ASCII, UTF-8, EUC, and EBCDIC code pages as follows:
- Data items that are described with the `NATIVE` phrase in the `USAGE` clause or that are compiled with the `CHAR(NATIVE)` option in effect are encoded in an ASCII, EUC, or UTF-8 code page.
- Data items that are described without the `NATIVE` phrase in the `USAGE` clause and that are compiled with the `CHAR(EBCDIC)` option in effect are encoded in an EBCDIC code page.

For `USAGE DISPLAY-1` data items, COBOL for AIX chooses between ASCII and EBCDIC code pages as follows:

- Data items that are described with the `NATIVE` phrase in the `USAGE` clause or that are compiled with the `CHAR(NATIVE)` option in effect are encoded in an ASCII DBCS code page.
- Data items that are described without the `NATIVE` phrase in the `USAGE` clause and that are compiled with the `CHAR(EBCDIC)` option in effect are encoded in an EBCDIC DBCS code page.

COBOL determines the appropriate code page as follows:

ASCII, UTF-8, EUC
From the active locale at run time

EBCDIC
From the EBCDIC_CODEPAGE environment variable, if set, otherwise the default EBCDIC code page from the current locale setting

RELATED TASKS
"Using environment variables to specify a locale"

RELATED REFERENCES
"Locales and code pages that are supported" on page 197
"Runtime environment variables" on page 211
"CHAR" on page 250
COBOL words with single-byte characters
(*COBOL for AIX Language Reference*)
User-defined words with multibyte characters
(*COBOL for AIX Language Reference*)

Using environment variables to specify a locale

Use any of several environment variables to provide the locale information for a COBOL program.

To specify a code page to use for all of the locale categories (messages, collating sequence, date and time formats, character classification, and case conversion), use LC_ALL.

To set the value for a specific locale category, use the appropriate environment variable:

- Use LC_MESSAGES to specify the format for affirmative and negative responses. You can use it also to affect whether messages (for example, error messages and listing headers) are in US English or Japanese. For any locale other than Japanese, US English is used.
- Use LC_COLLATE to specify the collating sequence in effect for greater-than or less-than comparisons, such as in relation conditions or in the `SORT` and `MERGE` statements.
- Use LC_TIME to specify the format of the date and time shown on the compiler listings. All other date and time values are controlled through COBOL language syntax.
- Use LC_CTYPE to specify character classification, case conversion, and other character attributes.

Any locale category that has not been specified by one of the locale environment variables above is set from the value of the LANG environment variable.

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

```
SET LC_xxxx=ll_CC.codepageID
```

Here `LC_xxxx` is the name of the locale category, *ll* is a lowercase two-letter language code, *CC* is an uppercase two-letter ISO country code, and *codepageID* is the code page to be used for native `DISPLAY` and `DISPLAY-1` data. COBOL for AIX uses the POSIX-defined locale conventions.

For example, to set the locale to Canadian French encoded in ISO8859-1, issue this command in the command window from which you compile and run a COBOL application:

```
export LC_ALL=fr_CA.ISO8859-1
```

You must code a valid value for the locale name (*ll_CC*), and the code page (*codepageID*) that you specify must be valid for the locale name. Valid values are shown in the table of supported locales and code pages referenced below.

RELATED CONCEPTS
"Determination of the locale from system settings"

RELATED TASKS
"Specifying the code page for character data" on page 194

RELATED REFERENCES
"Locales and code pages that are supported" on page 197
"Runtime environment variables" on page 211

Determination of the locale from system settings

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

When the language and country codes are determined from environment variables, but the code page is not, COBOL for AIX uses the default system code page for the language and country-code combination.

Multiple code pages might apply to the language and country-code combination. If you do not want the AIX system to select a default code page, you must specify the code page explicitly.

UTF-8: UTF-8 encoding is supported for any language and country-code combination.

RELATED TASKS
"Specifying the code page for character data" on page 194
"Using environment variables to specify a locale" on page 195
"Accessing the active locale and code-page values" on page 203
"Setting environment variables" on page 209

RELATED REFERENCES
"Locales and code pages that are supported" on page 197
"Runtime environment variables" on page 211

Types of messages for which translations are available

The following messages are enabled for national language support: compiler, runtime, and debugger user interface messages, and listing headers (including locale-based date and time formats).

Appropriate text and formats, as specified in the active locale, are used for these messages and the listing headers.

See the related reference below for information about the LANG and NLSPATH environment variables, which affect the language and locale of messages.

RELATED CONCEPTS
"The active locale" on page 193

RELATED TASKS
"Using environment variables to specify a locale" on page 195

RELATED REFERENCES
"Runtime environment variables" on page 211

Locales and code pages that are supported

The following table shows the locales that could be available on your system, and the code pages that are supported for each locale. COBOL for AIX supports the locales that are available on the system at compile time and at run time.

The locales that are available on the system are normally located at /usr/lib/nls/loc.

Table 23. **Supported locales and code pages**

Locale name[1]	Language[2]	Country or region[3]	ASCII-based code pages[4]	EBCDIC code pages[5]	Language group
any			UTF-8	EBCDIC code pages that are applicable to the locale are based on the *language* and *COUNTRY* portions of the locale name regardless of the code-page value of the locale.	
ar_AA	Arabic	Arabic countries	ISO8859-6	IBM-16804, IBM-420	Arabic
be_BY	Byelorussian	Belarus	ISO8859-5	IBM-1025, IBM-1154	Latin 5
bg_BG	Bulgarian	Bulgaria	ISO8859-5	IBM-1025, IBM-1154	Latin 5
ca_ES	Catalan	Spain	ISO8859-1, IBM-850, 8859-15	IBM-285, IBM-1145	Latin 1
cs_CZ	Czech	Czech Republic	ISO8859-2	IBM-870, IBM-1153	Latin 2
da_DK	Danish	Denmark	ISO8859-1, IBM-850, 8859-15	IBM-277, IBM-1142	Latin 1
de_CH	German	Switzerland	ISO8859-1, IBM-850, 8859-15	IBM-500, IBM-1148	Latin 1
de_DE	German	Germany	ISO8859-1, IBM-850, 8859-15	IBM-273, IBM-1141	Latin 1
el_GR	Greek	Greece	ISO8859-7	IBM-4971, IBM-875	Greek

Table 23. **Supported locales and code pages** *(continued)*

Locale name[1]	Language[2]	Country or region[3]	ASCII-based code pages[4]	EBCDIC code pages[5]	Language group
en_AU	English	Australia	ISO8859-1, IBM-850, 8859-15	IBM-037, IBM-1140	Latin 1
en_BE	English	Belgium	ISO8859-1, IBM-850, 8859-15	IBM-500, IBM-1148	Latin 1
en_GB	English	United Kingdom	ISO8859-1, IBM-850, 8859-15	IBM-037, IBM-1140	Latin 1
en_JP	English	Japan	ISO8859-1, IBM-850, 8859-15	IBM-037, IBM-1140	Latin 1
en_US	English	United States	ISO8859-1, IBM-850, 8859-15	IBM-037, IBM-1140	Latin 1
en_ZA	English	South Africa	ISO8859-1, IBM-850, 8859-15	IBM-037, IBM-1140	Latin 1
es_ES	Spanish	Spain	ISO8859-1, IBM-850, 8859-15	IBM-284, IBM-1145	Latin 1
fi_FI	Finnish	Finland	ISO8859-1, IBM-850, 8859-15	IBM-278, IBM-1143	Latin 1
fr_BE	French	Belgium	ISO8859-1, IBM-850, 8859-15	IBM-297, IBM-1148	Latin 1
fr_CA	French	Canada	ISO8859-1, IBM-850, 8859-15	IBM-037, IBM-1140	Latin 1
fr_CH	French	Switzerland	ISO8859-1, IBM-850, 8859-15	IBM-500, IBM-1148	Latin 1
fr_FR	French	France	ISO8859-1, IBM-850, 8859-15	IBM-297, IBM-1148	Latin 1
hr_HR	Croatian	Croatia	ISO8859-2	IBM-870, IBM-1153	Latin 2
hu_HU	Hungarian	Hungary	ISO8859-2	IBM-870, IBM-1153	Latin 2
is_IS	Icelandic	Iceland	ISO8859-1, IBM-850, 8859-15	IBM-871, IBM-1149	Latin 1
it_CH	Italian	Switzerland	ISO8859-1, IBM-850, 8859-15	IBM-500, IBM-1148	Latin 1
it_IT	Italian	Italy	ISO8859-1, IBM-850, 8859-15	IBM-280, IBM-1144	Latin 1
iw_IL	Hebrew	Israel	ISO8859-8	IBM-12712, IBM-424	Hebrew
Iw_IL	Hebrew	Israel	IBM-856	IBM-12712, IBM-424	Hebrew
ja_JP	Japanese	Japan	IBM-eucJP	IBM-930, IBM-939, IBM-1390, IBM-1399	Ideographic languages
Ja_JP	Japanese	Japan	IBM-943, IBM-932	IBM-930, IBM-939, IBM-1390, IBM-1399	Ideographic languages
ko_KR	Korean	Korea, Republic of	IBM-eucKR	IBM-933, IBM-1364	Ideographic languages
lt_LT	Lithuanian	Lithuania	IBM-1257	n/a	Lithuanian
lv_LV	Latvian	Latvia	IBM-1257	n/a	Latvian
mk_MK	Macedonian	Macedonia	ISO8859-5	IBM-1025, IBM-1154	Latin 5
nl_BE	Dutch	Belgium	ISO8859-1, IBM-850, 8859-15	IBM-500, IBM-1148	Latin 1
nl_NL	Dutch	Netherlands	ISO8859-1, IBM-850, 8859-15	IBM-037, IBM-1140	Latin 1
no_NO	Norwegian	Norway	ISO8859-1, IBM-850, 8859-15	IBM-277, IBM-1142	Latin 1
pl_PL	Polish	Poland	ISO8859-2	IBM-870, IBM-1153	Latin 2
pt_BR	Portuguese	Brazil	ISO8859-1, IBM-850, 8859-15	IBM-037, IBM-1140	Latin 1
pt_PT	Portuguese	Portugal	ISO8859-1, IBM-850, 8859-15	IBM-037, IBM-1140	Latin 1
ro_RO	Romanian	Romania	ISO8859-2	IBM-870, IBM-1153	Latin 2
ru_RU	Russian	Russian federation	ISO8859-5	IBM-1025, IBM-1154	Latin 5
sh_SP	Serbian (Latin)	Serbia	ISO8859-2	IBM-870, IBM-1153	Latin 2
sk_SK	Slovak	Slovakia	ISO8859-2	IBM-870, IBM-1153	Latin 2
sl_SI	Slovenian	Slovenia	ISO8859-2	IBM-870, IBM-1153	Latin 2

Table 23. **Supported locales and code pages** *(continued)*

Locale name[1]	Language[2]	Country or region[3]	ASCII-based code pages[4]	EBCDIC code pages[5]	Language group
sq_AL	Albanian	Albania	ISO8859-1, IBM-850, 8859-15	IBM-500, IBM-1148	Latin 1
sv_SE	Swedish	Sweden	ISO8859-1, IBM-850, 8859-15	IBM-278, IBM-1143	Latin 1
th_TH	Thai	Thailand	TIS-620	IBM-9030	Thai
tr_TR	Turkish	Turkey	ISO8859-9	IBM-1026, IBM-1155	Turkish
uk_UA	Ukranian	Ukraine	ISO8859-5, IBM-1124	IBM-1123, IBM-1154	Latin 5
zh_CN	Chinese (simplified)	China	IBM-eucCN	IBM-1388	Ideographic languages
Zh_CN	Chinese	China	GB18030	IBM-1388	Ideographic languages
zh_TW	Chinese (traditional)	Taiwan	IBM-eucTW	IBM-1371, IBM-937	Ideographic languages

1. Shows the valid combinations of ISO language code and ISO country code (*language_COUNTRY*) that are supported. The case of each character in the locale name shown in the table is significant and might not reflect the casing of a locale name with a specific code page selected (or implied) for the locale. See the locale file name in the locale directory (normally at /usr/lib/nls/loc) for the proper casing of each character for the locale name selected. A locale name with the UTF-8 code page is in general entirely uppercase (for example, ZH_CN.UTF-8).
2. Shows the associated language.
3. Shows the associated country or region.
4. Shows the code pages that are valid as the code-page ID for the locale that has the corresponding *language_COUNTRY* value. These table entries are not definitive. For the current list of valid locales, consult your system documentation for the specific version and configuration of AIX that you are running. The locale that you select must be valid, that is, installed both where you develop and where you run the program.
5. Shows the code pages that are valid as the code-page ID for the locale that has the corresponding *language_COUNTRY* value. These code pages are valid as content for the EBCDIC_CODEPAGE environment variable. If the EBCDIC_CODEPAGE environment variable is not set, the rightmost code-page entry shown in this column is selected as the EBCDIC code page for the corresponding locale.

RELATED TASKS
"Specifying the code page for character data" on page 194
"Using environment variables to specify a locale" on page 195

Controlling the collating sequence with a locale

Various operations, such as comparisons, sorting, and merging, use the collating sequence that is in effect for the program and data items. How you control the collating sequence depends on the code page in effect for the class of the data: alphabetic, alphanumeric, DBCS, or national.

A locale-based collating sequence for items with class alphabetic, alphanumeric, or DBCS applies only when the `COLLSEQ(LOCALE)` compiler option is in effect, not when `COLLSEQ(BIN)` or `COLLSEQ(EBCDIC)` is in effect. Similarly, a locale-based collating sequence for class national items applies only when the `NCOLLSEQ(LOCALE)` compiler option is in effect, not when `NCOLLSEQ(BIN)` is in effect.

When the `COLLSEQ(LOCALE)` or `NCOLLSEQ(LOCALE)` compiler option is in effect, the compile-time locale is used for language elements that have syntax or semantic rules that are affected by locale-based collation order, such as:

- `THRU` phrase in a condition-name `VALUE` clause

- *literal-3* THRU *literal-4* phrase in the EVALUATE statement
- *literal-1* THRU *literal-2* phrase in the ALPHABET clause
- Ordinal positions of characters specified in the SYMBOLIC CHARACTERS clause
- THRU phrase in the CLASS clause

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

RELATED TASKS
"Specifying the collating sequence" on page 8
"Specifying the code page for character data" on page 194
"Using environment variables to specify a locale" on page 195
"Controlling the alphanumeric collating sequence with a locale"
"Controlling the DBCS collating sequence with a locale" on page 201
"Controlling the national collating sequence with a locale" on page 202
"Accessing the active locale and code-page values" on page 203
"Setting sort or merge criteria" on page 149

RELATED REFERENCES
"Locales and code pages that are supported" on page 197
"COLLSEQ" on page 253
"NCOLLSEQ" on page 273

Controlling the alphanumeric collating sequence with a locale

The collating sequence for single-byte alphanumeric characters for the program collating sequence is based on either the locale at compile time or the locale at run time.

If you specify PROGRAM COLLATING SEQUENCE in the source program, the collating sequence is set at compile time and is used regardless of the locale at run time. If instead you set the collating sequence by using the COLLSEQ compiler option, the locale at run time takes precedence.

If the code page in effect is a single-byte ASCII code page, you can specify the following clauses in the SPECIAL-NAMES paragraph:
- ALPHABET clause
- SYMBOLIC CHARACTERS clause
- CLASS clause

If you specify these clauses when the source code page in effect includes DBCS characters, the clauses will be diagnosed and treated as comments. The rules of the COBOL user-defined alphabet-name and symbolic characters assume a character-by-character collating sequence, not a collating sequence that depends on a sequence of multiple characters.

If you specify the PROGRAM COLLATING SEQUENCE clause in the OBJECT-COMPUTER paragraph, the collating sequence that is associated with the *alphabet-name* is used to determine the truth value of alphanumeric comparisons. The PROGRAM COLLATING SEQUENCE clause also applies to sort and merge keys of USAGE DISPLAY unless you specify the COLLATING SEQUENCE phrase in the SORT or MERGE statement.

If you do not specify the COLLATING SEQUENCE phrase or the PROGRAM COLLATING SEQUENCE clause, the collating sequence in effect is NATIVE by default, and it is based on the active locale setting. This setting applies to SORT and MERGE statements and to the program collating sequence.

The collating sequence affects the processing of the following items:

- ALPHABET clause (for example, *literal-1* THRU *literal-2*)
- SYMBOLIC CHARACTERS specifications
- VALUE range specifications for level-88 items, relation conditions, and SORT and MERGE statements

RELATED TASKS
"Specifying the collating sequence" on page 8
"Controlling the collating sequence with a locale" on page 199
"Controlling the DBCS collating sequence with a locale"
"Controlling the national collating sequence with a locale" on page 202
"Setting sort or merge criteria" on page 149

RELATED REFERENCES
"COLLSEQ" on page 253
Classes and categories of data (*COBOL for AIX Language Reference*)
Alphanumeric comparisons (*COBOL for AIX Language Reference*)

Controlling the DBCS collating sequence with a locale

The locale-based collating sequence at run time always applies to DBCS data, except for comparisons of literals.

You can use a data item or literal of class DBCS in a relation condition with any
| relational operator. The other operand must be of class DBCS or class national, or
| be an alphanumeric group. No distinction is made between DBCS items and edited DBCS items.

When you compare two DBCS operands, the collating sequence is determined by the active locale if the COLLSEQ(LOCALE) compiler option is in effect. Otherwise, the collating sequence is determined by the binary values of the DBCS characters. The PROGRAM COLLATING SEQUENCE clause has no effect on comparisons that involve data
| items or literals of class DBCS.

When you compare a DBCS item to a national item, the DBCS operand is treated as if it were moved to an elementary national item of the same length as the DBCS operand. The DBCS characters are converted to national representation, and the comparison proceeds with two national character operands.

| When you compare a DBCS item to an alphanumeric group, no conversion or editing is done. The comparison proceeds as for two alphanumeric character operands. The comparison operates on bytes of data without regard to data representation.

RELATED TASKS
"Specifying the collating sequence" on page 8
"Using DBCS literals" on page 190
"Controlling the collating sequence with a locale" on page 199
"Controlling the alphanumeric collating sequence with a locale" on page 200
"Controlling the national collating sequence with a locale" on page 202

RELATED REFERENCES
"COLLSEQ" on page 253
Classes and categories of data (*COBOL for AIX Language Reference*)
Alphanumeric comparisons (*COBOL for AIX Language Reference*)
DBCS comparisons (*COBOL for AIX Language Reference*)
Group comparisons (*COBOL for AIX Language Reference*)

Controlling the national collating sequence with a locale

| You can use national literals or data items of `USAGE NATIONAL` in a relation condition with any relational operator. The `PROGRAM COLLATING SEQUENCE` clause has no effect on comparisons that involve national operands.

Use the `NCOLLSEQ(LOCALE)` compiler option to effect comparisons based on the algorithm for collation order that is associated with the active locale at run time. If `NCOLLSEQ(BINARY)` is in effect, the collating sequence is determined by the binary values of the national characters.

| Keys used in a `SORT` or `MERGE` statement can be class national only if the `NCOLLSEQ(BIN)` option is in effect.

RELATED TASKS
"Comparing national (UTF-16) data" on page 186
"Controlling the collating sequence with a locale" on page 199
"Controlling the DBCS collating sequence with a locale" on page 201
"Setting sort or merge criteria" on page 149

RELATED REFERENCES
"NCOLLSEQ" on page 273
Classes and categories of data (*COBOL for AIX Language Reference*)
National comparisons (*COBOL for AIX Language Reference*)

Intrinsic functions that depend on collating sequence

The following intrinsic functions depend on the ordinal positions of characters.

For an ASCII code page, these intrinsic functions are supported based on the collating sequence in effect. For an EUC code page or a code page that includes DBCS characters, the ordinal positions of single-byte characters are assumed to correspond to the hexadecimal representations of the single-byte characters. For example, the ordinal position for 'A' is 66 (`X'41'` + 1) and the ordinal position for '*' is 43 (`X'2A'` + 1).

Table 24. **Intrinsic functions that depend on collating sequence**

Intrinsic function	Returns:	Comments
`CHAR`	Character that corresponds to the ordinal-position argument	
`MAX`	Content of the argument that contains the maximum value	The arguments can be alphabetic, alphanumeric, national, or numeric.[1]
`MIN`	Content of the argument that contains the minimum value	The arguments can be alphabetic, alphanumeric, national, or numeric.[1]

Table 24. **Intrinsic functions that depend on collating sequence** *(continued)*

Intrinsic function	Returns:	Comments
ORD	Ordinal position of the character argument	
ORD-MAX	Integer ordinal position in the argument list of the argument that contains the maximum value	The arguments can be alphabetic, alphanumeric, national, or numeric.[1]
ORD-MIN	Integer ordinal position in the argument list of the argument that contains the minimum value	The arguments can be alphabetic, alphanumeric, national, or numeric.[1]
1. Code page and collating sequence are not applicable when the function has numeric arguments.		

These intrinsic functions are not supported for the DBCS data type.

RELATED TASKS
"Specifying the collating sequence" on page 8
"Comparing national (UTF-16) data" on page 186
"Controlling the collating sequence with a locale" on page 199

Accessing the active locale and code-page values

To verify the locale that is in effect at compile time, check the last few lines of the compiler listing.

For some applications, you might want to verify the locale and the EBCDIC code page that are active at run time, and convert a code-page ID to the corresponding CCSID. You can use callable library routines to perform these queries and conversions.

To access the locale and the EBCDIC code page that are active at run time, call the library function _iwzGetLocaleCP as follows:

```
CALL "_iwzGetLocaleCP" USING output1, output2
```

The variable *output1* is an alphanumeric item of 20 characters that represents the null-terminated locale value in the following format:
- Two-character language code
- An underscore (_)
- Two-character country code
- A period (.)
- The code-page value for the locale

For example, `en_US.IBM-1252` is the locale value of language code `en`, country code `US`, and code page `IBM-1252`.

The variable *output2* is an alphanumeric item of 10 characters that represents the null-terminated EBCDIC code-page ID in effect, such as `IBM-1140`.

To convert a code-page ID to the corresponding CCSID, call the library function _iwzGetCCSID as follows:

```
CALL "_iwzGetCCSID" USING input, output RETURNING returncode
```

input is an alphanumeric item that represents a null-terminated code-page ID.

output is a signed 4-byte binary item, such as one defined as `PIC S9(5) COMP-5`. Either the CCSID that corresponds to the input code-page ID string or the error code of -1 is returned.

returncode is a signed 4-byte binary data item, which is set as follows:

0 Successful.

1 The code-page ID is valid but does not have an associated CCSID; *output* is set to -1.

-1 The code-page ID is not a valid code page; *output* is set to -1.

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

"Example: get and convert a code-page ID"

RELATED TASKS
Chapter 11, "Setting the locale," on page 193

RELATED REFERENCES
"DYNAM" on page 258
"PGMNAME" on page 276
Chapter 15, "Compiler-directing statements," on page 295

Example: get and convert a code-page ID

The following example shows how you can use the callable services _iwzGetLocaleCP and _iwzGetCCSID to retrieve the locale and EBCDIC code page that are in effect, respectively, and convert a code-page ID to the corresponding CCSID.

```
cbl pgmname(lm)
       Identification Division.
       Program-ID.  "Samp1".
       Data Division.
       Working-Storage Section.
       01 locale-in-effect.
          05 ll-cc            pic x(5).
          05 filler-period    pic x.
          05 ASCII-CP         Pic x(14).
       01 EBCDIC-CP           pic x(10).
       01 CCSID               pic s9(5) comp-5.
       01 RC                  pic s9(5) comp-5.
       01 n                   pic 99.

       Procedure Division.
       Get-locale-and-codepages section.
       Get-locale.
           Display "Start Samp1."
           Call "_iwzGetLocaleCP"
             using locale-in-effect, EBCDIC-CP
           Move 0 to n
           Inspect locale-in-effect
             tallying n for characters before initial x'00'
           Display "locale in effect: " locale-in-effect (1 : n)
           Move 0 to n
```

```
    Inspect EBCDIC-CP
       tallying n for characters before initial x'00'
    Display "EBCDIC code page in effect: "
      EBCDIC-CP (1 : n).

Get-CCSID-for-EBCDIC-CP.
    Call "_iwzGetCCSID" using EBCDIC-CP, CCSID returning RC
    Evaluate RC
      When 0
        Display "CCSID for " EBCDIC-CP (1 : n) " is " CCSID
      When 1
        Display EBCDIC-CP (1 : n)
                " does not have a CCSID value."
      When other
        Display EBCDIC-CP (1 : n) " is not a valid code page."
    End-Evaluate.

Done.
    Goback.
```

If you set the locale to ja_JP.IBM-943 (`set LC_ALL=ja_JP.IBM-943`), the output from the sample program is:

```
Start Samp1.
locale in effect: ja_JP.IBM-943
EBCDIC code page in effect: IBM-1399
CCSID for IBM-1399 is 0000001399
```

RELATED TASKS

"Using environment variables to specify a locale" on page 195

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

Chapter 12. Compiling, linking, and running programs . . . 209
Setting environment variables . . . 209
  Compiler environment variables . . . 210
  Runtime environment variables . . . 211
    TZ environment parameter variables . . . 217
  Example: setting and accessing environment variables . . . 217
Compiling programs . . . 218
  Compiling from the command line . . . 219
    Examples: using cob2 for compiling . . . 220
  Compiling using shell scripts . . . 220
  Compiling for SFS files . . . 220
  Specifying compiler options with the PROCESS (CBL) statement . . . 221
  Modifying the default compiler configuration 221
    Tailoring your compilation . . . 222
    Stanzas in the configuration file . . . 223
Correcting errors in your source program . . . 224
  Severity codes for compiler error messages . . 225
  Generating a list of compiler error messages . . 225
    Messages and listings for compiler-detected errors . . . 225
    Format of compiler error messages . . . 226
cob2, cob2_j, and cob2_r options . . . 227
  Options that apply to compiling . . . 227
  Options that apply to linking . . . 228
  Options that apply to both compiling and linking . . . 228
Linking programs . . . 229
  Passing options to the linker . . . 230
    Examples: using an invocation command for linking . . . 231
  Linker input and output files . . . 232
    Linker search rules . . . 233
    Linker file-name defaults . . . 233
Correcting errors in linking. . . . 234
Running programs . . . 234
Redistributing COBOL for AIX library components 235

Chapter 13. Compiling, linking, and running OO applications . . . 237
Compiling OO applications. . . . 237
Preparing OO applications . . . 238
Example: compiling and linking a COBOL class definition. . . . 239
Running OO applications . . . 239
  Running OO applications that start with a main method . . . 240
  Running OO applications that start with a COBOL program . . . 240
| Java SDK support . . . 241

Chapter 14. Compiler options . . . 243
| Conflicting compiler options . . . 245
ADATA . . . 246
| ADDR. . . . 246
ARITH . . . 248
BINARY . . . 249
CALLINT . . . 249
CHAR. . . . 250
| CICS . . . 252
COLLSEQ . . . 253
COMPILE . . . 255
CURRENCY. . . . 255
DATEPROC . . . 256
DIAGTRUNC . . . 257
DYNAM . . . 258
ENTRYINT . . . 259
EXIT . . . 259
  Character string formats. . . . 261
  User-exit work area . . . 261
  Parameter list for exit modules . . . 261
  Using INEXIT . . . 262
  Using LIBEXIT . . . 262
  Using PRTEXIT. . . . 263
  Using ADEXIT . . . 264
FLAG . . . 264
FLAGSTD . . . 265
FLOAT . . . 267
LIB . . . 268
LINECOUNT . . . 268
LIST . . . 269
LSTFILE . . . 269
MAP . . . 270
MAXMEM . . . 271
| MDECK . . . 272
NCOLLSEQ . . . 273
NSYMBOL . . . 273
NUMBER . . . 274
OPTIMIZE . . . 275
PGMNAME . . . 276
  PGMNAME(UPPER) . . . 277
  PGMNAME(MIXED) . . . 277
PROFILE . . . 277
QUOTE/APOST . . . 278
SEPOBJ . . . 278
  Batch compilation . . . 279
SEQUENCE . . . 280
SIZE . . . 280
SOSI . . . 281
SOURCE . . . 282
SPACE . . . 283
SPILL . . . 284
SQL . . . 284
SSRANGE . . . 285
TERMINAL . . . 286
TEST . . . 286

THREAD 287
TRUNC 288
TRUNC example 1 289
TRUNC example 2 290
VBREF 290
WSCLEAR 291
XREF 291
YEARWINDOW 292
ZWB 293

Chapter 15. Compiler-directing statements . . 295

Chapter 16. Runtime options 301
CHECK 301
DEBUG 302
ERRCOUNT 302
| FILEMODE 303
FILESYS 303
TRAP 304
UPSI 305

Chapter 17. Debugging 307
Debugging with source language 307
Tracing program logic 307
Finding and handling input-output errors . . . 308
Validating data 309
Finding uninitialized data 309
Generating information about procedures . . . 309
Example: USE FOR DEBUGGING 310
Debugging using compiler options 311
Finding coding errors 312
Finding line sequence problems 312
Checking for valid ranges 312
Selecting the level of error to be diagnosed . . 313
Example: embedded messages. 314
Finding program entity definitions and references 315
Listing data items 316
Using the debugger 316
Getting listings 316
Example: short listing 318
Example: SOURCE and NUMBER output . . . 320
Example: MAP output 320
Example: embedded map summary 321
Terms and symbols used in MAP output . . 322
Example: nested program map 323
Example: XREF output: data-name cross-references. 324
Example: XREF output: program-name cross-references. 325
| Example: XREF output: COPY/BASIS
| cross-references. 325
Example: embedded cross-reference 326
Example: VBREF compiler output 326
Debugging with messages that have offset information 327
Debugging user exits 328
Debugging assembler routines. 328

Chapter 12. Compiling, linking, and running programs

The following sections explain how to set environment variables, compile, link, run, correct errors, and redistribute library components.

RELATED TASKS
"Setting environment variables"
"Compiling programs" on page 218
"Correcting errors in your source program" on page 224
"Linking programs" on page 229
"Correcting errors in linking" on page 234
"Running programs" on page 234
"Redistributing COBOL for AIX library components" on page 235

RELATED REFERENCES
"cob2, cob2_j, and cob2_r options" on page 227

Setting environment variables

You use environment variables to set values that programs need. Specify the value of an environment variable by using the `export` command or the putenv() POSIX function. If you do not set an environment variable, either a default value is applied or the variable is not defined.

An *environment variable* defines some aspect of a user environment or a program environment that can vary. For example, you use the COBPATH environment variable to define the locations where the COBOL run time can find a program when another program dynamically calls it. Environment variables are used by both the compiler and runtime libraries.

When you installed IBM COBOL for AIX, the installation process set environment variables to access the COBOL for AIX compiler and runtime libraries. To compile and run a simple COBOL program, the only environment variables that need to be set are NLSPATH and LANG. These variables are usually set at installation.

You can change the value of an environment variable in either of two places by using the `export` command:

- At the prompt in a command shell (for example, in an XTERM window). This environment variable definition applies to programs (processes or child processes) that you run from that shell or from any of its descendants (that is, any shells called directly or indirectly from that shell).
- In the .profile file in your home directory. If you define environment variables in the .profile file, the values of these variables are defined automatically whenever you begin an AIX session, and the values apply to all shell processes.

You can also set environment variables from within a COBOL program by using the putenv() POSIX function, and access the environment variables by using the getenv() POSIX function.

Some environment variables (such as COBPATH and NLSPATH) define directories in which to search for files. If multiple directory paths are listed, they are delimited by colons. Paths that are defined by environment variables are evaluated in order, from the first path to the last in the `export` statement. If multiple files that have the same name are defined in the paths of an environment variable, the *first* located copy of the file is used.

For example, the following `export` command sets the COBPATH environment variable (which defines the locations where the COBOL run time can find dynamically accessed programs) to include two directories, the first of which is searched first:

```
export COBPATH=/users/me/bin:/mytools/bin
```

"Example: setting and accessing environment variables" on page 217

RELATED TASKS
"Tailoring your compilation" on page 222

RELATED REFERENCES
"Compiler environment variables"
"Runtime environment variables" on page 211

Compiler environment variables

The COBOL compiler uses the following environment variables.

Because COBOL words are defined to be case insensitive, all letters in COBOL words are treated as uppercase, including *library-name* and *text-name*. Thus environment variable names that correspond to such names must be uppercase. For example, the environment variable name that corresponds to `COPY MyCopy` is MYCOPY.

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

If COBCPYEXT is not defined, the following suffixes are searched: CPY, CBL, COB, and the lowercase equivalents cpy, cbl, and cob.

COBLSTDIR
Specifies the directory into which the compiler listing file is written. Specify any valid path. To indicate an absolute path, specify a leading slash. Otherwise, the path is relative to the current directory. A trailing slash is optional.

If COBLSTDIR is not defined, the compiler listing is written into the current directory.

COBOPT
Specifies compiler options. To specify multiple compiler options, separate each option by a space or comma. Surround the list of options with quotation marks if the list contains blanks or characters that are significant to the AIX command shell. For example:

```
export COBOPT="TRUNC(OPT) TERMINAL"
```

Default values apply to individual compiler options.

DB2DBDFT
: Specifies the database for compiling programs that have embedded SQL statements.

library-name
: If you specify *library-name* as a user-defined word, the name is used as an environment variable, and the value of the environment variable is used as the path to locate the copybook.

  If you do not specify a library-name, the compiler searches the library paths in the following order:

  1. Current directory
  2. Paths specified by the `-I`*xxx* option, if set
  3. Paths specified by the SYSLIB environment variable

  The search ends when the file is found.

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

| **LANG**
| : See the related reference about runtime environment variables below for
| details.

| **NLSPATH**
| : See the related reference about runtime environment variables below for
| details.

SYSLIB
: Specifies paths to be used for COBOL `COPY` statements that have text-names that are unqualified by library-names. It also specifies paths to be used for `SQL INCLUDE` statements.

text-name
: If you specify *text-name* as a user-defined word, the name is used as an environment variable, and the value of the environment variable is used as the file-name and possibly the path name of the copybook.

  To specify multiple path names, delimit them with a colon (**:**) or a semicolon (**;**).

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

RELATED CONCEPTS
"DB2 coprocessor" on page 331

RELATED TASKS
"Using SQL INCLUDE with the DB2 coprocessor" on page 332

RELATED REFERENCES
"Runtime environment variables"
"cob2, cob2_j, and cob2_r options" on page 227
Chapter 14, "Compiler options," on page 243
Chapter 15, "Compiler-directing statements," on page 295

Runtime environment variables

The COBOL runtime library uses the following environment variables.

assignment-name

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

```
export OUTPUTFILE=january.car_results
```

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

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

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

```
SELECT CARPOOL ASSIGN TO OUTPUTFILE
```

Because `OUTPUTFILE` is defined in an environment variable, this statement results in data being written to the file january.car_results.

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

CLASSPATH

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

COBJVMINITOPTIONS

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

COBMSGS

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

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

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

COBPATH

Specifies the directory paths to be used by the COBOL run time to locate dynamically accessed programs such as shared libraries.

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

```
export COBPATH=/pgmpath/pgmshlib
```

COBRTOPT

Specifies the COBOL runtime options.

Separate runtime options by a comma or a colon. Use parentheses or equal signs (=) as the delimiter for suboptions. Options are not case sensitive. For example, the following two commands are equivalent:

```
export COBRTOPT="check(ON),UPSI(00000000)"
export COBRTOPT='CHECK=on:UPSI=00000000"
```

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

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

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

```
export EBCDIC_CODEPAGE=codepage
```

If EBCDIC_CODEPAGE is not set, the default EBCDIC code page is selected based on the current locale, as described in the related reference below about the locales and code pages supported. When the CHAR(EBCDIC) compiler option is in effect and multiple EBCDIC code pages are applicable to the locale in effect, you must set the EBCDIC_CODEPAGE environment variable unless the default EBCDIC code page for the locale is acceptable.

ENCINA_CDS_ROOT
Specifies the Encina SFS server name. For example:

```
export ENCINA_CDS_ROOT=/.:/cics/sfs/test1
```

ENCINA_SFS_DATA_VOLUME
Specifies the name of the SFS data volume on which SFS files are to be created. For example:

```
export ENCINA_SFS_DATA_VOLUME=sfs_SFS_SERV
```

This data volume must have been defined to the SFS server that your application accesses.

If this variable is not set, the default name encinaSfsVol is used.

ENCINA_SFS_INDEX_VOLUME
Specifies the name of the SFS data volume on which alternate index files are to be created. For example:

```
export ENCINA_SFS_INDEX_VOLUME=sfs_SFS_SERV
```

This data volume must have been defined to the SFS server that your application accesses.

If this variable is not set, alternate index files are created on the same data volume as the corresponding base index files.

ENCINA_VSAM_AUTO_FLUSH
Specifies whether all changes to Encina SFS files for each input-output operation are committed to disk before control is returned to the application (that is, whether the *operational force* feature of SFS is enabled). You can improve the performance of applications that use SFS files by specifying OFF for this environment variable. For example:

```
export ENCINA_VSAM_AUTO_FLUSH=OFF
```

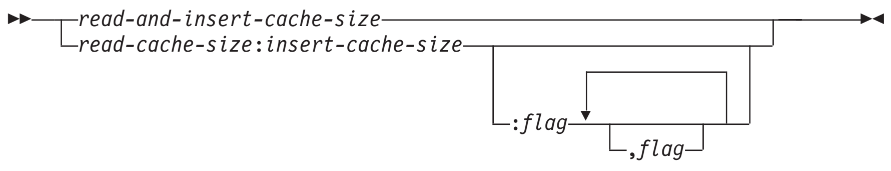
When this environment variable is set to OFF, SFS uses a *lazy-write* strategy; that is, changes to SFS files might not be committed to disk until the files are closed.

If SFS client-side caching is in effect (that is, environment variable ENCINA_VSAM_CACHE is set to a valid nonzero value), the setting of ENCINA_VSAM_AUTO_FLUSH is ignored. Operational force is disabled.

If SFS client-side caching is not in effect, the value of ENCINA_VSAM_AUTO_FLUSH defaults to ON.

ENCINA_VSAM_CACHE
Specifies whether client-side caching is enabled for SFS files. You can improve the performance of applications that use SFS files by enabling client-side caching.

ENCINA_VSAM_CACHE syntax



Size units are in numbers of pages. A size of zero indicates that caching is disabled. The possible flags are:

ALLOW_DIRTY_READS
Removes the restriction for read caching that the files being accessed must be locked.

INSERTS_DESPITE_UNIQUE_INDICES
Removes the restriction for insert caching that inserts are cached only if all active indices for clustered files and all active alternate indices for entry-sequenced and relative files allow duplicates.

For example, the following command sets the read-cache size to 16 pages and the insert-cache size to 64 pages. It also enables dirty reads and enables inserts despite unique indices:

```
export ENCINA_VSAM_CACHE=16:64:ALLOW_DIRTY_READS, \
    INSERTS_DESPITE_UNIQUE_INDICES
```

By default, client-side caching is not enabled.

LANG
Specifies the locale (as described in the related task about using environment variables to specify a locale). LANG also influences the value of the NLSPATH environment variable as described below.

For example, the following command sets the language locale name to U.S. English:

```
export LANG=en_US
```

LC_ALL
Specifies the locale. A locale setting that uses LC_ALL overrides any setting that uses LANG or any other LC_*xx* environment variable (as described in the related task about using environment variables to specify a locale).

LC_COLLATE
Specifies the collation behavior for the locale. This setting is overridden if LC_ALL if specified.

LC_CTYPE
Specifies the code page for the locale. This setting is overridden if LC_ALL if specified.

LC_MESSAGES
Specifies the language for messages for the locale. This setting is overridden if LC_ALL if specified.

LC_TIME
Determines the locale for date and time formatting information. This setting is overridden if LC_ALL if specified.

LIBPATH
Specifies the full path name for the COBOL runtime library. The default is set at installation.

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

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

NLSPATH
Specifies the full path name of message catalogs and help files. NLSPATH must always be set, and is given an initial value of `usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/%L/%N.cat` during installation. `%L` is substituted by the value specified by the LANG environment variable. `%N` is substituted by the message catalog name used by COBOL.

COBOL for AIX installation puts the messages in usr/lib/nls/msg/*xx*, where *xx* is any language that COBOL for AIX supports. The default is en_US.

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

```
DIR=xxxx
nlspath=$DIR/%L/%N:$NLSPATH
export NLSPATH
```

xxxx is the directory where COBOL was installed. The directory *xxxx* must contain a directory *xxxx*/en_US (in the case of a U.S. English language setup) that contains the COBOL message catalog.

Messages in the following languages are included with the product:

en_US English

ja_JP, Ja_JP
Japanese

You can specify the languages for the messages and for the locale setting differently. For example, you can set the environment variable LANG to `en_US` and set the environment variable LC_ALL to `ja_JP.IBM-eucJP`. In this example, any COBOL compiler or runtime messages will be in English, whereas native ASCII (`DISPLAY` or `DISPLAY-1`) data in the program is treated as encoded in code page IBM-eucJP (ASCII Japanese code page).

| The compiler uses the combination of the NLSPATH and the LANG environment variable values to access the message catalog. If NLSPATH is validly set but LANG is not set to one of the locale values shown above, a warning message is generated and the compiler defaults to the en_US message catalog. If the NLSPATH value is invalid, a terminating error message is generated.

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

PATH Specifies the directory paths of executable programs.

SYSIN, SYSIPT, SYSOUT, SYSLIST, SYSLST, CONSOLE, SYSPUNCH, SYSPCH
These COBOL environment names are used as the environment variable names that correspond to the mnemonic names used in `ACCEPT` and `DISPLAY` statements.

If the environment variable is not set, by default SYSIN and SYSIPT are directed to the logical input device (keyboard). SYSOUT, SYSLIST, SYSLST, and CONSOLE are directed to the system logical output device (screen). SYSPUNCH and SYSPCH are not assigned a default value and are not valid unless you explicitly define them.

For example, the following command defines `CONSOLE`:

```
export CONSOLE=/users/mypath/myfile
```

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

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

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

For example:

```
export TMP=/tmp
```

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

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

When TZ is not present, the default is EST5EDT, the default locale value. When only the standard time zone is specified, the default value of *n* (difference in hours from GMT) is 0 instead of 5.

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

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

```
export TZ=CST6CDT
```

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

RELATED TASKS
"Using environment variables to specify a locale" on page 195
"Identifying files" on page 113
"Enabling COBOL applications to access SFS files" on page 136

"Improving SFS performance" on page 139
"Setting environment variables" on page 209

RELATED REFERENCES
"Locales and code pages that are supported" on page 197
"TZ environment parameter variables"
"CHAR" on page 250
Chapter 16, "Runtime options," on page 301

TZ environment parameter variables

The values for the TZ variable are defined below.

Table 25. **TZ environment parameter variables**

Variable	Description	Default value
SSS	Standard time-zone identifier. This must be three characters, must begin with a letter, and can contain spaces.	EST
n	Difference (in hours) between the standard time zone and coordinated universal time (UTC), formerly called Greenwich mean time (GMT). A positive number denotes time zones west of the Greenwich meridian. A negative number denotes time zones east of the Greenwich meridian.	5
DDD	Daylight saving time (DST) zone identifier. This must be three characters, must begin with a letter, and can contain spaces.	EDT
sm	Starting month (1 to 12) of DST	4
sw	Starting week (-4 to 4) of DST	1
sd	Starting day of DST: 0 to 6 if *sw* is not zero; 1 to 31 if *sw* is zero	0
st	Starting time (in seconds) of DST	3600
em	Ending month (1 to 12) of DST	10
ew	Ending week (-4 to 4) of DST	-1
ed	Ending day of DST: 0 to 6 if *ew* is not zero; 1 to 31 if *ew* is zero	0
et	Ending time (in seconds) of DST	7200
shift	Amount of time change (in seconds)	3600

Example: setting and accessing environment variables

The following example shows how you can access and set environment variables from a COBOL program by calling the standard POSIX functions getenv() and putenv().

Because getenv() and putenv() are C functions, you must pass arguments `BY VALUE`. Pass character strings as `BY VALUE` pointers that point to null-terminated strings. Compile programs that call these functions with the `NODYNAM` and `PGMNAME(LONGMIXED)` options.

```
CBL pgmname(longmixed),nodynam
Identification division.
Program-id. "envdemo".
Data division.
Working-storage section.
01 P pointer.
01 PATH pic x(5) value Z"PATH".
01 var-ptr pointer.
```

```
 01 var-len pic 9(4) binary.
 01 putenv-arg pic x(14) value Z"MYVAR=ABCDEFG".
 01 rc pic 9(9) binary.
 Linkage section.
 01 var pic x(5000).
 Procedure division.
* Retrieve and display the PATH environment variable
     Set P to address of PATH
     Call "getenv" using by value P returning var-ptr
     If var-ptr = null then
         Display "PATH not set"
     Else
         Set address of var to var-ptr
         Move 0 to var-len
         Inspect var tallying var-len
           for characters before initial X"00"
         Display "PATH = " var(1:var-len)
     End-if
* Set environment variable MYVAR to ABCDEFG
     Set P to address of putenv-arg
     Call "putenv" using by value P returning rc
     If rc not = 0 then
         Display "putenv failed"
         Stop run
     End-if
     Goback.
```

Compiling programs

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

You can:

- Set the COBOPT environment variable from the command line.
- Specify compiler options as options to the `cob2`, `cob2_j`, or `cob2_r` command. You can use these commands on the command line, in a shell script, or in a makefile.
- Use `PROCESS` (`CBL`) or `*CONTROL` statements. An option that you specify by using `PROCESS` overrides every other option specification.

RELATED TASKS
"Compiling from the command line" on page 219
"Compiling using shell scripts" on page 220
"Compiling for SFS files" on page 220
"Specifying compiler options with the PROCESS (CBL) statement" on page 221
"Modifying the default compiler configuration" on page 221
Chapter 25, "Porting applications between platforms," on page 457

RELATED REFERENCES
"Compiler environment variables" on page 210
"cob2, cob2_j, and cob2_r options" on page 227
Chapter 14, "Compiler options," on page 243
"Conflicting compiler options" on page 245
Appendix A, "Migration and compatibility," on page 569
Appendix B, "Summary of differences with host COBOL," on page 573

Compiling from the command line

To compile a COBOL program from the command line, issue the `cob2`, `cob2_j`, or `cob2_r` command, depending on the type of the program. (Each command also invokes the linker.)

cob2 command syntax

```
►►──┬─cob2───┬──┬─────────┬─filenames─────────────────────────────►◄
    ├─cob2_j─┤  └─options─┘
    └─cob2_r─┘
```

Choose the command as follows:

- `cob2_j` for object-oriented client programs and class definitions
- `cob2_r` for:
  - Applications that use the SFS file system
  - Non-CICS applications that you compile with the `THREAD` compiler option.
  - CICS applications. Compile them with the `NOTHREAD` compiler option (the default).
- `cob2` otherwise

When you compile by using the `cob2_j` command, the `THREAD` compiler option is automatically included as a default invocation option.

To compile multiple files, specify the file-names at any position in the command line, using spaces to separate options and file-names. For example, the following two commands are equivalent:

```
cob2 -g -qlib filea.cbl fileb.cbl -qflag(w)
cob2 filea.cbl -g -qflag(w) fileb.cbl -qlib
```

Each of the four commands accepts compiler and linker options in any order on the command line. Any options that you specify apply to all files on the command line. (For details about which libraries are linked when you use one of the commands, see the related task about linking programs.)

Only files with certain suffixes (such as .cbl) are passed to the compiler. All other files are passed to the linker.

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

"Examples: using cob2 for compiling" on page 220

RELATED TASKS
"Modifying the default compiler configuration" on page 221
"Linking programs" on page 229

RELATED REFERENCES
"cob2, cob2_j, and cob2_r options" on page 227
Chapter 14, "Compiler options," on page 243

Examples: using cob2 for compiling

The following examples show the output produced for various `cob2` invocations.

Table 26. **Output from the cob2 command**

To compile:	Enter:	These files are produced:
alpha.cbl	`cob2 -c alpha.cbl`	alpha.o
alpha.cbl and beta.cbl	`cob2 -c alpha.cbl beta.cbl`	alpha.o, beta.o
alpha.cbl with the `LIST` and `ADATA` options	`cob2 -qlist,adata alpha.cbl`	alpha.wlist, alpha.o[1], alpha.lst, alpha.adt, and a.out
1. If the linking is successful, this file is deleted.		

RELATED TASKS
"Compiling from the command line" on page 219

RELATED REFERENCES
"ADATA" on page 246
"LIST" on page 269

Compiling using shell scripts

You can use a shell script to automate `cob2`, `cob2_j`, or `cob2_r` commands.

To prevent invalid syntax from being passed to the command, however, follow these guidelines:

- Use an equal sign and colon rather than parentheses to delimit compiler suboptions.
- Use underscores rather than single quotation marks to delimit compiler suboptions.
- Do not use any blanks in the option string unless you enclose the string in quotation marks ("").

Table 27. **Examples of compiler-option syntax in a shell script**

Use in shell script	Use on command line
`-qFLOAT=NATIVE:,CHAR=NATIVE:`	`-qFLOAT(NATIVE),CHAR(NATIVE)`
`-qEXIT=INEXIT=_String_,MYMODULE::`	`-qEXIT(INEXIT('String',MYMODULE))`

Compiling for SFS files

| When you use the SFS file system, compile your programs by using the `cob2_r` command.

For example:

```
cob2_r alpha.cbl
```

You can use the `-#` option to see which libraries are linked; for example:

```
cob2_r -# alpha.o
```

RELATED TASKS
"Enabling COBOL applications to access SFS files" on page 136

RELATED REFERENCES
"SFS file system" on page 117
"cob2, cob2_j, and cob2_r options" on page 227

Specifying compiler options with the PROCESS (CBL) statement

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

CBL/PROCESS statement syntax

```
►►──┬─CBL─────┬──┬──────────────┬──────────────────────────►◄
    └─PROCESS─┘  └─options-list─┘
```

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

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

Use one or more blanks to separate `PROCESS` from the first option in *options-list*. Separate options with a comma or a blank. Do not insert spaces between individual options and their suboptions.

You can use more than one `PROCESS` statement. If you do so, the `PROCESS` statements must follow each another with no intervening statements. You cannot continue options across multiple `PROCESS` statements.

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

RELATED REFERENCES
CBL (PROCESS) statement (*COBOL for AIX Language Reference*)

Modifying the default compiler configuration

The default options used by the `cob2` command are obtained from the configuration file, which is by default /etc/cob2.cfg. You can display the options used by `cob2` (or `cob2_j`, or `cob2_r`) by specifying the `-#` option on the command.

If you are using the default configuration file, the command cob2 -# abc.cbl displays this output:

```
exec: /usr/lpp/cobol/bin/IGYCCOB2 abc.cbl
exec: /usr/bin/ld -bpT:0x10000000 -bpD:0x20000000 -bhalt:5 /lib/crt0.o abc.o
  -lcob2s -L/usr/lpp/cobol/lib -lsmrtlite -lC128 -lC -lc
```

If you edit or display /etc/cob2.cfg (using the cat command), you see a line that specifies the COBOL compiler to use:

```
DEFLT: compiler1 = /usr/lpp/cobol/bin/IGYCCOB2
```

Other lines in the file specify the linker and various options.

You can modify the cob2.cfg configuration file, for example to change the default options or the location of the compiler and libraries. For example, the cob2_j stanza of cob2.cfg specifies the Java library-names. You might need to update the Java library-names to indicate the release level of Java that you are using.

Instead of modifying the default configuration file, you can tailor a copy of the file for your purposes.

RELATED CONCEPTS
"Java SDK support" on page 241

RELATED TASKS
"Tailoring your compilation"

RELATED REFERENCES
"cob2, cob2_j, and cob2_r options" on page 227
"Stanzas in the configuration file" on page 223

Tailoring your compilation

To tailor a compilation to your needs, you can change a copy of the default configuration file and use it in various ways.

The configuration file /etc/cob2.cfg has sections, called *stanzas*, that begin cob2:, cob2_j:, and cob2_r:. To list these stanzas, use the command ls /usr/bin/cob2*. The resulting list shows:

```
/usr/bin/cob2
/usr/bin/cob2_j
/usr/bin/cob2_r
```

| The last two files are linked to the first one. The commands cob2, cob2_j, and cob2_r execute the same module; however, cob2 uses the cob2 stanza, cob2_j uses the cob2_j stanza, and so forth.

You can tailor your compilation by doing the following steps:
1. Make a copy of the default configuration file /etc/cob2.cfg.
2. Change your copy to support specific compilation requirements or other COBOL compilation environments.
3. (Optional) Issue the cob2 command with the -# option to display the effect of your changes.
4. Use your copy instead of /etc/cob2.cfg.

To selectively use a modified copy of the configuration file for a specific compilation, issue the cob2 (or cob2_j or cob2_r) command with the -F option. For example, to use /u/myhome/myconfig.cfg instead of /etc/cob2.cfg as the configuration file to compile myfile.cbl, issue the command:

```
cob2 myfile.cbl -F/u/myhome/myconfig.cfg
```

If you add your own stanza, such as mycob2, you can specify it with the -F option:

```
cob2 myfile.cbl -F/u/myhome/myconfig.cfg:mycob2
```

Or you can define a mycob2 command:

```
ln -s /usr/bin/cob2 /u/myhome/mycob2
mycob2 myfile.cbl -F/u/myhome/myconfig
```

Or you can add mycob2 to /etc/cob2.cfg:

```
ln -s /usr/bin/cob2 mycob2
mycob2 myfile.cbl
```

Whichever directory you name in the ln command (such as /u/myhome above) must be in your PATH environment variable.

For a list of the attributes in each stanza, see the related reference below about stanzas. You can change any of the items. For example, to add a library to the list of libraries used by the linker, change the runlib2 attribute.

RELATED TASKS
"Setting environment variables" on page 209

RELATED REFERENCES
"cob2, cob2_j, and cob2_r options" on page 227
"Stanzas in the configuration file"

Stanzas in the configuration file

A stanza in the configuration file can contain any of several attributes, as shown in the following table.

| Attribute names that have suffix _64 are used when you specify the -q64 option of
| the cob2 command. When -q32 is specified or defaulted, the corresponding
| nonsuffixed attribute name is used.

Table 28. **Stanza attributes**

Attribute	Description
compiler1	Path name of the COBOL compiler component. The default for cob2 is /usr/lpp/cobol/bin/IGYCCOB2.
crt crt_64	Path name of the object file passed as the first parameter to the linker. If you do not specify either the -p or -pg option, the value of crt is used. The default is /lib/crt0.o or /lib/crt0_64.o.
gcrt gcrt_64	Path name of the object file passed as the first parameter to the linker. If you specify the -pg option, the value of gcrt is used. The default is /lib/gcrt0.o or /lib/gcrt0_64.o.
ld	Path name to be used to link COBOL programs. The default is /usr/bin/ld.
ldflags ldflags_64	A string of option flags, separated by commas, to be passed to the linker.

Table 28. **Stanza attributes** *(continued)*

Attribute	Description
mcrt mcrt_64	Path name of the object file passed as the first parameter to the linker if you specified the `-p` option. The default is /lib/mcrt0.o or /lib/mcrt0_64.o.
proflibs	Profiling libraries used by the linker, separated by commas. The default is -l/lib/profiled, -l/usr/lib/profiled.
runlib1	A library option, followed by runlib2, that the compiler passes as a parameter to the linker to specify the COBOL main library for linking for both profiling and nonprofiling. The default is -lcob2s.
runlib2 runlib2_64	Libraries used by the linker, separated by commas; used for both profiling and nonprofiling. The default is stanza-dependent.
use	Stanza from which attributes are taken, in addition to the local stanza. For single-valued attributes, values in the `use` stanza apply if no value is provided in the local, or default, stanza. For comma-separated lists, the values from the `use` stanza are added to the values from the local stanza.

RELATED TASKS
"Modifying the default compiler configuration" on page 221
"Tailoring your compilation" on page 222

RELATED REFERENCES
"cob2, cob2_j, and cob2_r options" on page 227

Correcting errors in your source program

Messages about source-code errors indicate where the error occurred (`LINEID`). The text of a message tells you what the problem is. With this information, you can correct the source program.

Although you should try to correct errors, it is not necessary to fix all of them. You can leave a warning-level or informational-level message in a program without much risk, and you might decide that the recoding and compilation that are needed to remove the error are not worth the effort. Severe-level and error-level errors, however, indicate probable program failure and should be corrected.

In contrast with the four lower levels of errors, an unrecoverable (U-level) error might not result from a mistake in your source program. It could come from a flaw in the compiler itself or in the operating system. In any case, the problem must be resolved, because the compiler is forced to end early and does not produce complete object code or listing. If the message occurs for a program that has many S-level syntax errors, correct those errors and compile the program again. You can also resolve job set-up problems (problems such as missing data-set definitions or insufficient storage for compiler processing) by making changes to the compile job. If your compile job setup is correct and you have corrected the S-level syntax errors, you need to contact IBM to investigate other U-level errors.

After correcting the errors in your source program, recompile the program. If this second compilation is successful, proceed to the link-editing step. If the compiler still finds problems, repeat the above procedure until only informational messages are returned.

RELATED TASKS
"Generating a list of compiler error messages"
"Linking programs" on page 229

RELATED REFERENCES
"Messages and listings for compiler-detected errors"

Severity codes for compiler error messages

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

Table 29. **Severity codes for compiler error messages**

Level of message	Return code	Purpose
Informational (I)	0	To inform you. No action is required and the program runs correctly.
Warning (W)	4	To indicate a possible error. The program probably runs correctly as written.
Error (E)	8	To indicate a condition that is definitely an error. The compiler attempted to correct the error, but the results of program execution might not be what you expect. You should correct the error.
Severe (S)	12	To indicate a condition that is a serious error. The compiler was unable to correct the error. The program does not run correctly, and execution should not be attempted. Object code might not be created.
Unrecoverable (U)	16	To indicate an error condition of such magnitude that the compilation was terminated.

Generating a list of compiler error messages

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

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

```
Identification Division.
Program-ID. ErrMsg.
```

RELATED REFERENCES
"Messages and listings for compiler-detected errors"
"Format of compiler error messages" on page 226

Messages and listings for compiler-detected errors

As the compiler processes your source program, it checks for COBOL language errors. For each error found, the compiler issues a message. These messages are collated in the compiler listing (subject to the `FLAG` option).

The compiler listing file has the same name as the compiler source file, but with the suffix .lst. For example, the listing file for myfile.cbl is myfile.lst. The listing file is written to the directory from which the `cob2` (or `cob2_j` or `cob2_r`) command was issued.

Each message in the listing provides the following information:

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

Wherever possible, the message provides specific instructions for correcting the error.

The messages for errors found during processing of compiler options, `CBL` and `PROCESS` statements, and `BASIS`, `COPY`, or `REPLACE` statements are displayed near the top of the listing.

The messages for compilation errors found in your program (ordered by line number) are displayed near the end of the listing for each program.

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

RELATED TASKS
"Correcting errors in your source program" on page 224
"Generating a list of compiler error messages" on page 225

RELATED REFERENCES
"Format of compiler error messages"
"Severity codes for compiler error messages" on page 225
"FLAG" on page 264

Format of compiler error messages

Each message issued by the compiler has a source line number, a message identifier, and message text.

Each message has the following form:

```
nnnnnn 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, these are your original source program numbers. If you specified `NONUMBER`, the numbers are those generated by the compiler.

IGY
: The prefix that identifies this message as coming from the COBOL compiler.

pp
: Two characters that identify which phase or subphase of the compiler discovered the error. As an application programmer, you can ignore this information. If you are diagnosing a suspected compiler error, contact IBM for support.

xxxx
: A four-digit number that identifies the error message.

l
: A character that indicates the severity level of the error: I, W, E, S, or U.

message-text
: The message text, which in the case of an error message is a short explanation of the condition that caused the error.

Tip: If you used the `FLAG` option to suppress messages, there might be additional errors in your program.

RELATED REFERENCES
"Severity codes for compiler error messages" on page 225
"FLAG" on page 264

cob2, cob2_j, and cob2_r options

The options listed below apply to all of the invocation commands: `cob2`, `cob2_j`, and `cob2_r`.

Options that apply to compiling

-c Compiles programs but does not link them.

-comprc_ok=*n*
Controls the behavior upon return from the compiler. If the return code is less than or equal to *n*, the command continues to the link step, or in the compile-only case, exits with a zero return code. If the return code generated by the compiler is greater than *n*, the command exits with the same return code returned by the compiler.

The default is `-comprc_ok=4`.

-host Sets these compiler options for host COBOL data representation and language semantics:

- `CHAR(EBCDIC)`
- `COLLSEQ(EBCDIC)`
- `NCOLLSEQ(BIN)`
- `FLOAT(S390)`

The `-host` option changes the format of COBOL program command-line arguments from an array of pointers to an EBCDIC character string that has a halfword prefix that contains the string length. For additional information, see the related task below about using command-line arguments.

If you use the `-host` option and want a main routine in a C or PL/I object file to be the main entry point of your application, you must use the `-cmain` linker option as described below.

-I*xxx* Adds path *xxx* to the directories to be searched for copybooks if neither a library-name nor SYSLIB is specified. (This option is the uppercase letter *I*, not the lowercase letter *l*.)

Only a single path is allowed for each `-I` option. To add multiple paths, use multiple `-I` options. Do not insert spaces between `-I` and *xxx*.

If you use the `COPY` statement, you must ensure that the `LIB` compiler option is in effect.

-q*xxx* Passes options to the compiler, where *xxx* is any compiler option or set of compiler options. Do not insert spaces between `-q` and *xxx*.

If a parenthesis is part of the compiler option or suboption, or if a series of options is specified, include them in quotation marks.

To specify multiple options, delimit each option by a blank or comma. For example, the following two option strings are equivalent:

```
-qoptiona,optionb
-q"optiona optionb"
```

If you plan to use a shell script to automate your `cob2` tasks, a special syntax is provided for the `-q`*`xxx`* option. For details, see the related task about compiling using shell scripts.

Options that apply to linking

-cmain
(Has an effect only if you also specify `-host`.) Makes a C or PL/I object file that contains a main routine the main entry point in the executable file. In C, a main routine is identified by the function name main(). In PL/I, a main routine is identified by the `PROC OPTIONS(MAIN)` statement.

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

-main:*xxx*
Makes *xxx* the first file in the list of files passed to the linker. The purpose of this option is to make the specified file the main program in the executable file. *xxx* must uniquely identify the object file or the archive library, and the suffix must be either .o or .a, respectively.

If `-main` is not specified, the first object, archive library, or source file specified in the command is the first file in the list of files passed to the linker.

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

-o *xxx* Names the executable module *xxx*, where *xxx* is any name. If the `-o` option is not used, the name of the executable module defaults to a.out.

-p Instructs the linker to set up the object file produced by the compiler for AIX profiling (command used to evaluate program performance) with minimum profiling information.

-pg Instructs the linker to set up the object file produced by the compiler for AIX graph profiling (command used to evaluate program performance) with maximum profiling information.

Options that apply to both compiling and linking

-F*xxx* Uses *xxx* as a configuration file or a stanza rather than the defaults specified in the /etc/cob2.cfg configuration file. *xxx* has one of the following forms:

- *configuration_file*:*stanza*
- *configuration_file*
- :*stanza*

-g Produces symbolic information used by the debugger.

-q32 Specifies that a 32-bit object program is to be generated. Sets the `ADDR(32)` compiler option.

-q64 Specifies that a 64-bit object program is to be generated. Sets the `ADDR(64)` compiler option. Sets the `-b64` linker option, which instructs the linker to create a 64-bit executable module.

-v	Displays compile and link steps, and executes them.
-#	Displays compile and link steps, but does not execute them.
-?, ?	Displays help for the `cob2`, `cob2_j`, or `cob2_r` command.

RELATED TASKS
"Compiling from the command line" on page 219
"Compiling using shell scripts" on page 220
"Passing options to the linker" on page 230
"Using command-line arguments" on page 495

RELATED REFERENCES
"Compiler environment variables" on page 210
"Runtime environment variables" on page 211
"ADDR" on page 246

Linking programs

Use the linker to link specified object files and create an executable file or a shared object.

You have a choice of ways to start the linker:

- With the `cob2`, `cob2_j`, or `cob2_r` command:

  Each of these commands calls the linker unless you specify the `-c` option. For details about which command to use for which type of application, see the related task about compiling from the command line.

  `cob2_j` and `cob2_r` link with the COBOL multithreaded libraries. Additionally, `cob2_j` links with the Java Native Interface (JNI) runtime libraries. `cob2` links with the COBOL static runtime library.

  Linking with C: The linker accepts .o files, but does not accept .c files. If you want to link C and COBOL files together, first produce .o files for the C source files by using the `xlc -c` command.
- With the `ld` command:

  You can invoke the linker explicitly with the `ld` command. However, the compiler invocation commands pass several linker options and library-names to the linker automatically. In most cases, it is better to use one of the compiler invocation commands to link-edit your .o files.
- Through a makefile:

  You can write a makefile to organize the sequence of actions (such as compiling and linking) that are required to build your program. In the makefile you can use linker statements to specify the kind of output that you want.

You can specify linker options using any of the methods above.

"Examples: using an invocation command for linking" on page 231

RELATED TASKS
"Passing options to the linker" on page 230
"Compiling from the command line" on page 219
Chapter 28, "Building shared libraries," on page 497

RELATED REFERENCES
"cob2, cob2_j, and cob2_r options" on page 227

"Linker input and output files" on page 232
"Linker search rules" on page 233
Appendix A, "Migration and compatibility," on page 569
AIX General Programming Concepts: Writing and Debugging Programs (make)

Passing options to the linker

You can specify linker options in any of these ways: an invocation command (`cob2`, `cob2_j`, or `cob2_r`), the `ld` command, or makefile statements.

Any options that you specify in an invocation command that are not recognized by the command are passed to the linker.

Several invocation command options influence the linking of your program. For details about the set of options for a given command, consult documentation for that command. For details about the full set of available linker options, see the related reference about the `ld` command.

The following table shows the options that you are more likely to need for your COBOL programs. Precede each option with a hyphen (-) as shown.

Table 30. **Common linker options**

Option	Purpose
`-bdynamic`, `-bshared`, or `-bstatic`	Controls the processing of `-l` options and the way that shared objects are processed
`-bD:`*`nnnn`*	Instructs the linker to allocate *nnnn* bytes of heap space for static user data. (The default value is 0.) This space might be needed to run programs that have very large data items defined, particularly data items in the `WORKING-STORAGE SECTION`.
`-bexpall`	Exports all global symbols, except imported symbols, unreferenced symbols defined in archive members, and symbols beginning with an underscore (_). When you use this option, you might be able to avoid using an export file.
`-bE:`*`foo`*`.exp`	Exports the symbols that are named in the export file *foo*.exp
`-bhalt:`*`error_level`*	Specifies the maximum error level allowed before the processing of linker commands halts
`-bI:`*`foo`*`.imp` or `-bimport:`*`foo`*`.imp`	Imports the symbols from the import file *foo*.imp
`-bloadmap:`*`name`*	Requests that a log of linker actions and messages is saved in file *name*
`-bmap:`*`name`*	Writes an address map of the output object file to file *name*
`-bmaxdata:`*`bytes`* or `-bmaxstack:`*`bytes`*	Specifies the maximum amount of space to reserve for the program data segment and stack segment for programs for which the size of these regions is a constraint
`-bM:SRE`	Makes the output module a shared reentrant object
`-bnoentry`	Indicates that this object does not have a main entry point
`-bnso`	Statically links a shared object as an ordinary object file
`-brtl` or `-bnortl`	Determines which algorithm is used to find libraries that you specify with the `-l` option
`-L`*`dir`*	Specifies the directory to search for libraries specified by the `-l` option (default: /usr/lib)

Table 30. **Common linker options** *(continued)*

Option	Purpose
-l*name*	Searches the specified library-file, where *name* selects the file lib*name*.a (default: libraries listed in cob2.cfg)
-S*nnnn*	Instructs the linker to allocate *nnnn* bytes of user stack. (The default value is 0.) This space might be needed to run programs that have very large data items defined, particularly data items in the LOCAL-STORAGE SECTION.

"Examples: using an invocation command for linking"

RELATED REFERENCES
"cob2, cob2_j, and cob2_r options" on page 227
"Linker input and output files" on page 232
"Linker search rules" on page 233
AIX Commands Reference, Volume 3 (ld command)

Examples: using an invocation command for linking

You can use any of the COBOL invocation commands to both compile and link your programs. The following examples illustrate the use of cob2.

- To link two files together after they are compiled, do not use the -c option. For example, to compile and link alpha.cbl and beta.cbl and generate a.out, enter:

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

  This command creates alpha.o and beta.o, then links alpha.o, beta.o, and the COBOL libraries. If the link step is successful, it produces an executable program named a.out.
- To link a compiled file with a library, enter:

  ```
  cob2 foo.cbl -brtl -bshared -lmylib
  ```

  This command causes the linker to search for the library libmylib.so first, and then the library libmylib.a in each directory in the search path consecutively until either is encountered.
- To use options to limit the search for a library, enter:

  ```
  cob2 foo.cbl -bdynamic -llib1 -bstatic -llib2 -brtl
  ```

  In this case, to satisfy the first library specification, the linker searches for the library liblib1.so first and then the library liblib1.a in each directory (as described in the previous example). However, at the same time the linker only searches for liblib2.a in those same libraries.
- To use -b linker options on the invocation command line to create statically linked object files, enter:

  ```
  cob2 -bnso -bI:/usr/lib/syscalls.exp file1.cbl file2.cbl
  ```
- To compile and link in separate steps, enter commands such as these:

  ```
  cob2 -c file1.cbl                # Produce one object file
  cob2 -c file2.cbl file3.cbl      # Or multiple object files
  cob2 file1.o file2.o file3.o     # Link with appropriate libraries
  ```
- To combine beta.cbl and gamma.cbl into a shared library object called sh_bg, enter:

  ```
  cob2 -o sh_bg beta.cbl gamma.cbl -bexpall -bM:SRE -bnoentry
  ```

  "Example: creating a sample shared library" on page 498

Linker input and output files

The linker takes object files, links them with each other and with any library files that you specify, and produces an executable output file. The executable output can be either an executable program file or a shared object.

Linker inputs:

- Options
- Object files (*.o)
- Static library files (*.a)
- Dynamic library files (*.so)
- Import files (*.imp)

Linker outputs:

- Executable file (a.out by default)
- Shared object
- Return code

Import files: *Import files* are ASCII files that identify the external symbols to resolve at run time. An import file identifies the shared object that defines the imported symbols. The system loader finds and resolves those symbols. If the first line of an import file begins with `#!` (pound sign, exclamation point), you can specify the file on the command line as an ordinary input file. Otherwise, you must use the `-bI` or `-bimport` option to specify the import file.

Export files: *Export files* are ASCII files that identify external symbols that are made available for another executable object file to import. The file format of an export file is the same as the file format of an import file.

Library files: *Libraries* are files that have suffix .a or .so. To designate a library, you can specify an absolute or relative path name or use the `-l` (lowercase letter L) option in the form `-l`*`name`*. The last form designates file lib*name*.a, or in dynamic mode, file lib*name*.so, to be searched for in several directories. These search directories include directories that you specify by using `-L` options, and the standard library directories /usr/lib and /lib.

The environment variable LIBPATH is not used to search for libraries that you specify on the command line either explicitly (for example, `libc.a`) or by using the `-l` option (for example, `-lc`). You must use `-L`*`dir`* options to indicate the directories to be searched for libraries that you specified with a `-l` option.

If you specify a shared object, or an archive file that contains a shared object, with an absolute or relative path name instead of with the `-l`*`Name`* option, the path name is included in the import file ID string in the loader section of the output file. You can override this behavior with the `-bnoipath` option.

You can create library files by combining one or more files into a single archive file by using the AIX `ar` command.

"Example: creating a sample shared library" on page 498

RELATED TASKS
"Passing options to the linker" on page 230

RELATED REFERENCES
"Linker search rules"
"Linker file-name defaults"
AIX Commands Reference, Volume 1 (ar command)

Linker search rules

When searching for an object (.o), library (.a), or import file (.imp), the linker looks in several locations until the search is satisfied.

The linker searches these locations:

1. The directory that you specify for the file

   If you specify a path with the file, the linker searches only that path and stops linking if the file cannot be found there.
2. The current directory, if you did not specify a path
3. The value of the environment variable LIBPATH, if defined

The search for files depends on several `ld` options. If you specify `-brtl`, runtime linking is enabled. When it is used in conjunction with either `-bdynamic` or `-bshared`, the search for libraries that you specify with the `-l` option is satisfied by the suffix .so or .a. For each directory searched, a file with the suffix .so is looked for. If that search fails, a file with the suffix .a is looked for. If neither file is found, the search continues with the next directory.

If you use libraries other than the default ones in /usr/lib, you can specify one or more `-L` options that point to the locations of the other libraries. You can also set the LIBPATH environment variable, which lets you specify a search path for libraries at run time.

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

RELATED TASKS
"Passing options to the linker" on page 230

RELATED REFERENCES
"Linker file-name defaults"
AIX General Programming Concepts: Writing and Debugging Programs

Linker file-name defaults

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

Table 31. **Default file-names assumed by the linker**

File	Default file-name	Default suffix
Object files	None. You must enter at least one object file name.	.o
Output file	a.out	None
Map file	The base name of the output file	map
Library files	The default libraries defined in the object files. Use compiler options to define the default libraries. Any additional libraries that you specify are searched before the default libraries.	.a or .so
Import files	None. You must explicitly enter the names of import files.	.imp

Table 31. **Default file-names assumed by the linker** *(continued)*

File	Default file-name	Default suffix
Export files	None. You must explicitly enter the names of export files.	.exp

Correcting errors in linking

When you use the `PGMNAME(UPPER)` compiler option, the names of subprograms referenced in `CALL` statements are translated to uppercase. This change affects the linker, which recognizes case-sensitive names.

For example, the compiler translates `Call "RexxStart"` to `Call "REXXSTART"`. If the real name of the called program is RexxStart, the linker will not find the program, and will produce an error message that indicates that REXXSTART is an unresolved external reference.

This type of error typically occurs when you call API routines that are supplied by another software product. If the API routines have mixed-case names, you must take both of the following actions:

- Use the `PGMNAME(MIXED)` compiler option.
- Ensure that `CALL` statements specify the correct mix of uppercase and lowercase in the names of the API routines.

Running programs

To run a COBOL program, set environment variables, run the program, and correct any runtime errors.

1. Make sure that any needed environment variables are set.

   For example, if the program uses an environment variable name to assign a value to a system file-name, set that environment variable.
2. Run the program: At the command line, either enter the name of the executable module or run a command file that invokes that module.

   For example, if the command `cob2 alpha.cbl beta.cbl-o gamma` is successful, you can run the program by entering `gamma`.
3. Correct runtime errors.

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

Tip: If runtime messages are abbreviated or incomplete, the environment variables LANG or NLSPATH or both might be incorrectly set.

RELATED TASKS
"Setting environment variables" on page 209
"Using the debugger" on page 316

RELATED REFERENCES
"cob2, cob2_j, and cob2_r options" on page 227
"Runtime environment variables" on page 211
Appendix A, "Migration and compatibility," on page 569
Appendix J, "Runtime messages," on page 725

Redistributing COBOL for AIX library components

You can redistribute COBOL for AIX library components with a developed application. You can install the components on any number of AIX systems that you use for production deployment of your COBOL application.

You must install a licensed copy of the full COBOL for AIX product on the AIX system that you use to develop the application that is to be redistributed. You can redistribute the following COBOL for AIX filesets without needing any additional licenses:

- cobol.rte (runtime libraries)
- cobol.msg.Ja_JP (compiler and runtime messages in Japanese)
- cobol.msg.ja_JP (compiler and runtime messages in Japanese)
- cobol.msg.en_US (compiler and runtime messages in English)
- SdU.vsam (SMARTdata UTILITIES VSAM)
- SdU.sfs (SMARTdata UTILITIES SFS)

You can find these filesets on the product CD.

To see the file names that are associated with any of the filesets listed above, issue the `lslpp` command, for example: `lslpp -f cobol.rte`.

Chapter 13. Compiling, linking, and running OO applications

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

RELATED CONCEPTS
"Java SDK support" on page 241

RELATED TASKS
"Compiling OO applications"
"Preparing OO applications" on page 238
"Running OO applications" on page 239

Compiling OO applications

When you compile OO applications, use the `cob2_j` command to compile COBOL client programs and class definitions, and the `javac` command to compile Java class definitions to produce *bytecode* (suffix .class).

When you compile using the `cob2_j` command, the `THREAD` compiler option is automatically included as a default invocation option, and the Java Native Interface (JNI) libraries are included in the list of libraries passed to the linker. (You might need to update the libraries, however, for the release level of Java that you
| are using. See the related concept about Java SDK support and the related task about modifying the default compiler configuration.)

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_j` command or by setting the SYSLIB environment variable.

RELATED CONCEPTS
"Java SDK support" on page 241

RELATED TASKS
"Setting environment variables" on page 209
"Compiling programs" on page 218
"Modifying the default compiler configuration" on page 221

"Preparing OO applications"
"Running OO applications" on page 239
"Accessing JNI services" on page 443

RELATED REFERENCES
"Compiler environment variables" on page 210
"cob2, cob2_j, and cob2_r options" on page 227
"THREAD" on page 287

Preparing OO applications

Use the `cob2_j` command to link OO COBOL applications. When you link using `cob2_j`, thread-enabled libraries are used for linking the application.

To prepare an OO COBOL client program for execution, link the object file to create a shared library.

To prepare a COBOL class definition for execution:

1. Link the object file to create a shared library.

   You must name the resulting shared library lib*Classname*.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 shared library 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 shared library 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:

Table 32. **Commands for compiling and linking a class definition**

Command	Input	Output
`cob2_j -c Classname.cbl`	*Classname*.cbl	*Classname*.o, *Classname*.java
`cob2_j -o libClassname.so -bnoentry -bexpall -bM:SRE Classname.o`	*Classname*.o	lib*Classname*.so
`javac Classname.java`	*Classname*.java	*Classname*.class

The `cob2_j` command also causes the library libjava.a to be linked with the object file.

After you issue the `cob2_j` and `javac` commands successfully, you have the executable components for the program: the shared library lib*Classname*.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" on page 239

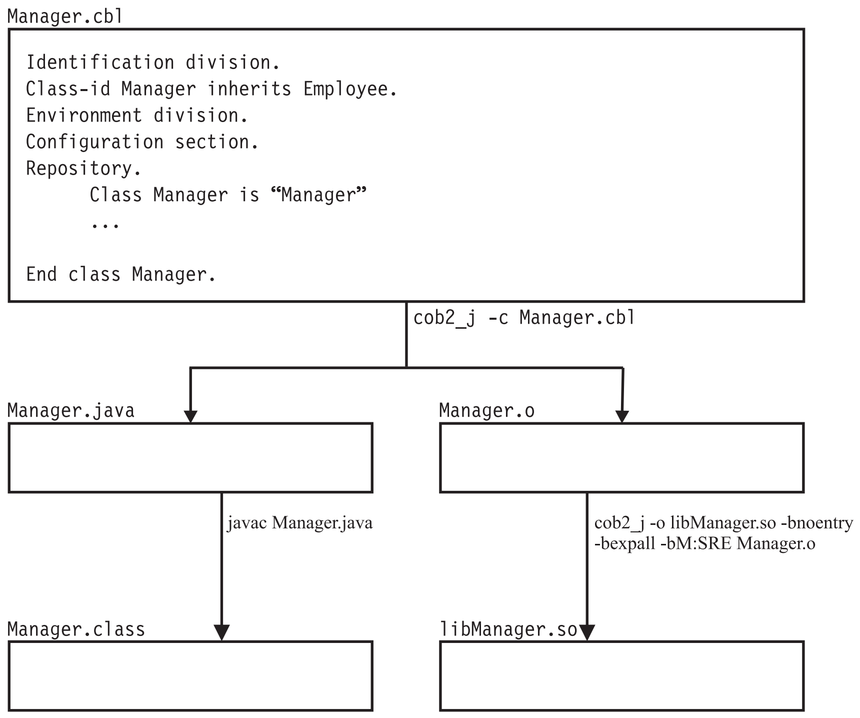
RELATED TASKS
"Compiling programs" on page 218

"REPOSITORY paragraph for defining a class" on page 402
"Passing options to the linker" on page 230
Chapter 28, "Building shared libraries," on page 497

RELATED REFERENCES
"cob2, cob2_j, and cob2_r options" on page 227
"Object-oriented syntax, and Java 5 or Java 6 SDKs" on page 569

Example: compiling and linking a COBOL class definition

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



```
Identification division.
Class-id Manager inherits Employee.
Environment division.
Configuration section.
Repository.
      Class Manager is "Manager"
      ...

End class Manager.
```

The class file Manager.class and the shared library libManager.so are the executable components of the application, and are generated in the current working directory.

RELATED TASKS
"Passing options to the linker" on page 230

Running OO applications

You can run object-oriented COBOL applications from the command line.

Specify the directory that contains the shared libraries 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.

RELATED TASKS
"Running OO applications that start with a main method"
"Running OO applications that start with a COBOL program"
"Running programs" on page 234
"Setting environment variables" on page 209
Chapter 23, "Writing object-oriented programs," on page 397
"Structuring OO applications" on page 439

Running OO applications that start with a main method

If the first routine of a mixed COBOL and Java application is the `main` method of a Java class or the `main` factory method of a COBOL class, run the application by using the `java` command and by specifying the name of the class that contains the `main` method.

The `java` command initializes the Java virtual machine (JVM). To customize the initialization of the JVM, specify options on the `java` command as in the following examples:

Table 33. **java command options for customizing the JVM**

Purpose	Option
To set a system property	`-D`*name*`=`*value*
To request that the JVM generate verbose messages about garbage collection	`-verbose:gc`
To request that the JVM generate verbose messages about class loading	`-verbose:class`
To request that the JVM generate verbose messages about native methods and other Java Native Interface activity	`-verbose:jni`
To set the initial Java heap size to *value* bytes	`-Xms`*value*
To set the maximum Java heap size to *value* bytes	`-Xmx`*value*

See the output from the `java -h` command or the related references for details about the options that the JVM supports.

RELATED REFERENCES
Java Technology Edition SDK User Guides

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
"Running programs" on page 234
"Setting environment variables" on page 209

RELATED REFERENCES
Java Technology Edition SDK User Guides

Java SDK support

The `cob2_j` command for compiling an application that has object-oriented syntax links the application with the Java SDK 1.4 libraries by default.

To use a later Java SDK, such as Java 5 or Java 6, modify the cob2_j stanza of the default compiler configuration file cob2.cfg to indicate the path names of the higher-level Java libraries.

64-bit COBOL applications that use object-oriented syntax require a 64-bit Java SDK. 32-bit COBOL applications that use object-oriented syntax require a 32-bit SDK. Specify the `-q64` option of the `cob2_j` command to obtain 64-bit SDK libraries. The `-q32` option selects 32-bit SDK libraries.

RELATED TASKS
"Modifying the default compiler configuration" on page 221

RELATED REFERENCES
"cob2, cob2_j, and cob2_r options" on page 227
"ADDR" on page 246
"Object-oriented syntax, and Java 5 or Java 6 SDKs" on page 569

Chapter 14. Compiler options

You can direct and control your compilation by using compiler options or by using compiler-directing statements (compiler directives).

Compiler options affect the aspects of your program that are listed in the table below. The linked-to information for each option provides the syntax for specifying the option and describes the option, its parameters, and its interaction with other parameters.

Table 34. **Compiler options**

Aspect of your program	Compiler option	Default	Option abbreviations
Source language	"ARITH" on page 248	`ARITH(COMPAT)`	`AR(C|E)`
	"CICS" on page 252	`NOCICS`	None
	"CURRENCY" on page 255	`NOCURRENCY`	`CURR|NOCURR`
	"LIB" on page 268	`LIB`	None
	"NSYMBOL" on page 273	`NSYMBOL(NATIONAL)`	`NS(NAT|DBCS)`
	"NUMBER" on page 274	`NONUMBER`	`NUM|NONUM`
	"QUOTE/APOST" on page 278	`QUOTE`	`Q|APOST`
	"SEQUENCE" on page 280	`SEQUENCE`	`SEQ|NOSEQ`
	"SOSI" on page 281	`NOSOSI`	None
	"SQL" on page 284	`SQL("")`	None
Date processing	"DATEPROC" on page 256	`NODATEPROC`, or `DATEPROC(FLAG)`if only `DATEPROC` is specified	`DP|NODP`
	"YEARWINDOW" on page 292	`YEARWINDOW(1900)`	`YW`
Maps and listings	"LINECOUNT" on page 268	`LINECOUNT(60)`	`LC`
	"LIST" on page 269	`NOLIST`	None
	"LSTFILE" on page 269	`LSTFILE(LOCALE)`	`LST`
	"MAP" on page 270	`NOMAP`	None
	"SOURCE" on page 282	`SOURCE`	`S|NOS`
	"SPACE" on page 283	`SPACE(1)`	None
	"TERMINAL" on page 286	`TERMINAL`	`TERM|NOTERM`
	"VBREF" on page 290	`NOVBREF`	None
	"XREF" on page 291	`XREF(FULL)`	`X|NOX`
Object module generation	"COMPILE" on page 255	`NOCOMPILE(S)`	`C|NOC`
	"OPTIMIZE" on page 275	`NOOPTIMIZE`	`OPT|NOOPT`
	"PGMNAME" on page 276	`PGMNAME(UPPER)`	`PGMN(LU|LM)`
	"SEPOBJ" on page 278	`SEPOBJ`	None

Table 34. **Compiler options** *(continued)*

Aspect of your program	Compiler option	Default	Option abbreviations
Object code control	"ADDR" on page 246	`ADDR(32)`	None
	"BINARY" on page 249	`BINARY(NATIVE)`	None
	"CHAR" on page 250	`CHAR(NATIVE)`	None
	"COLLSEQ" on page 253	`COLLSEQ(BIN)`	None
	"DIAGTRUNC" on page 257	`NODIAGTRUNC`	`DTR|NODTR`
	"FLOAT" on page 267	`FLOAT(NATIVE)`	None
	"NCOLLSEQ" on page 273	`NCOLLSEQ(BINARY)`	`NCS(L|BIN|B)`
	"TRUNC" on page 288	`TRUNC(STD)`	None
	"ZWB" on page 293	`ZWB`	None
`CALL` statement behavior	"DYNAM" on page 258	`NODYNAM`	`DYN|NODYN`
Debugging and diagnostics	"FLAG" on page 264	`FLAG(I,I)`	`F|NOF`
	"FLAGSTD" on page 265	`NOFLAGSTD`	None
	"SSRANGE" on page 285	`NOSSRANGE`	`SSR|NOSSR`
	"TEST" on page 286	`NOTEST`	None
Other	"ADATA" on page 246	`NOADATA`	None
	"CALLINT" on page 249	`CALLINT(SYSTEM,NODESC)`	None
	"ENTRYINT" on page 259	`ENTRYINT(SYSTEM)`	None
	"EXIT" on page 259	`NOEXIT`	`EX(INX,LIBX,PRTX,ADX)`
	"MAXMEM" on page 271	`MAXMEM(2000)`	None
	"MDECK" on page 272	`NOMDECK`	`NOMD|MD|MD(C)|MD(NOC)`
	"PROFILE" on page 277	`PROFILE`	None
	"SIZE" on page 280	`SIZE(8388608)`	`SZ`
	"SPILL" on page 284	`SPILL(512)`	None
	"THREAD" on page 287	`NOTHREAD`	None
	"WSCLEAR" on page 291	`NOWSCLEAR`	None

Installation defaults: The defaults listed with the options above are the defaults shipped with the product.

Performance considerations: The `ADDR`, `ARITH`, `CHAR`, `DYNAM`, `FLOAT`, `OPTIMIZE`, `SSRANGE`, `TEST`, and `TRUNC` compiler options can affect runtime performance.

RELATED TASKS
"Compiling programs" on page 218
Chapter 32, "Tuning your program," on page 543

RELATED REFERENCES
Chapter 15, "Compiler-directing statements," on page 295
"Performance-related compiler options" on page 552

Conflicting compiler options

The COBOL for AIX compiler can encounter conflicting compiler options in either of two ways: both the positive and negative form of an option are specified at the same level in the hierarchy of precedence of options, or mutually exclusive options are specified at the same level.

The compiler recognizes options in the following order of precedence from highest to lowest:

1. Options specified in the `PROCESS` (or `CBL`) statement
2. Options specified in the `cob2`, `cob2_j`, or `cob2_r` command invocation
3. Options set in the COBOPT environment variable
4. Default options

If you specify conflicting options at the same level in the hierarchy, the option specified last takes effect.

If you specify mutually exclusive compiler options at the same level, the compiler forces one of the options to a nonconflicting value, and generates an error message. For example, if you specify both `OPTIMIZE` and `TEST` in the `PROCESS` statement in any order, `TEST` takes effect and `OPTIMIZE` is ignored, as shown in the following table.

Table 35. **Mutually exclusive compiler options**

Specified	Ignored	Forced on
CICS	ADDR(64)	ADDR(32)
	DYNAM	NODYNAM
	NOLIB	LIB
	THREAD	NOTHREAD
MDECK	NOLIB	LIB
SQL	NOLIB	LIB
TEST	OPTIMIZE	NOOPTIMIZE

However, options specified at a higher level of precedence override options specified at a lower level of precedence. For example, if you code `TEST` in the COBOPT environment variable but `OPTIMIZE` in the `PROCESS` statement, `OPTIMIZE` takes effect because the options coded in the `PROCESS` statement and any options forced on by an option coded in the `PROCESS` statement have higher precedence.

RELATED TASKS
"Compiling programs" on page 218

RELATED REFERENCES
"Compiler environment variables" on page 210
"cob2, cob2_j, and cob2_r options" on page 227
Chapter 15, "Compiler-directing statements," on page 295

ADATA

Use ADATA when you want the compiler to create a SYSADATA file that contains records of additional compilation information.

ADATA option syntax

```
   ┌─NOADATA─┐
►►─┴─ADATA───┴───────────────────────────────────────────────────►◄
```

Default is: NOADATA

Abbreviations are: None

SYSADATA information is used by other tools, which will set ADATA ON for their use.

The size of the SYSADATA file generally grows with the size of the associated program.

Option specification: You cannot specify the ADATA option in a PROCESS (or CBL) statement. You can specify it only in one of the following ways:

- As an option in the cob2 command or one of its variants
- In the COBOPT environment variable

RELATED REFERENCES
Appendix I, "COBOL SYSADATA file contents," on page 657
"Compiler environment variables" on page 210
"cob2, cob2_j, and cob2_r options" on page 227

ADDR

Use the ADDR compiler option to indicate whether a 32-bit or 64-bit object program should be generated.

ADDR option syntax

```
           ┌─32─┐
►►─ADDR(──┴─64─┴──)──────────────────────────────────────────────►◄
```

Default is: ADDR(32)

Abbreviations are: None

Option specification:

You can specify the ADDR option in any of the ways that you specify other compiler options, as described in the related task below about compiling programs. However, if you specify ADDR in a PROCESS (or CBL) statement, you can specify it only for the first program in a batch compilation. You cannot change the value of the ADDR option for subsequent programs in the batch.

When you specify compiler options using the -q option of the cob2 command (or its variants cob2_j or cob2_r), you can abbreviate ADDR(32) as 32 or ADDR(64) as 64. For example:

```
cob2 -q64 prog64.cbl
```

Storage allocation:

The storage allocation for the following COBOL data types depends on the setting of the ADDR compiler option:

- USAGE POINTER (also the ADDRESS OF and JNIEnvPtr special registers, which implicitly have this usage)
- USAGE PROCEDURE-POINTER
- USAGE FUNCTION-POINTER
- USAGE OBJECT REFERENCE
- USAGE INDEX

If ADDR(32) is in effect, 4 bytes are allocated for each item in your program that has one of the usages listed above; if ADDR(64) is in effect, 8 bytes are allocated for each of the items.

If the SYNCHRONIZED clause is specified for a data item that has one of the usages shown above, the item is aligned on a fullword boundary if ADDR(32) is in effect, or on a doubleword boundary if ADDR(64) is in effect.

The setting of the ADDR option affects several compiler limits. For details, see the related reference below about compiler limits.

LENGTH OF special register:

If ADDR(32) is in effect, the LENGTH OF special register has this implicit definition:

```
PICTURE 9(9) USAGE IS BINARY
```

If ADDR(64) is in effect, the LENGTH OF special register has this implicit definition:

```
PICTURE 9(18) USAGE IS BINARY
```

LENGTH intrinsic function:

If ADDR(32) is in effect, the returned value of the LENGTH intrinic function is a 9-digit integer. If ADDR(64) is in effect, the returned value is an 18-digit integer.

Programming requirements and restrictions:

- All program components within an application must be compiled using the same setting of the ADDR option. You cannot mix 32-bit programs and 64-bit programs in an application.

- **Interlanguage communication:** In multilanguage applications, 64-bit COBOL programs can be combined with 64-bit C/C++ programs, and 32-bit COBOL programs can be combined with 32-bit C/C++ or PL/I programs.
- **Object-oriented applications:** 64-bit COBOL applications that use object-oriented syntax require a 64-bit Java SDK. 32-bit COBOL applications that use object-oriented syntax require a 32-bit SDK.
- **CICS:** COBOL programs that will run in the CICS TXSeries environment must be 32 bit.

RELATED CONCEPTS
"Java SDK support" on page 241

RELATED TASKS
"Finding the length of data items" on page 110
"Compiling programs" on page 218
"Coding COBOL programs to run under CICS" on page 337
"Calling between COBOL and C/C++ programs" on page 471

RELATED REFERENCES
"cob2, cob2_j, and cob2_r options" on page 227
"Conflicting compiler options" on page 245
"Migration of 32-bit applications to 64-bit mode" on page 570
Compiler limits (*COBOL for AIX Language Reference*)

ARITH

`ARITH` affects the maximum number of digits that you can code for integers, and the number of digits used in fixed-point intermediate results.

ARITH option syntax

```
           ┌─COMPAT─┐
►►──ARITH(─┴─EXTEND─┴─)────────────────────────────────────────────►◄
```

Default is: `ARITH(COMPAT)`

Abbreviations are: `AR(C)`, `AR(E)`

When you specify `ARITH(EXTEND)`:
- The maximum number of digit positions that you can specify in the `PICTURE` clause for packed-decimal, external-decimal, and numeric-edited data items is raised from 18 to 31.
- The maximum number of digits that you can specify in a fixed-point numeric literal is raised from 18 to 31. You can use numeric literals with large precision anywhere that numeric literals are currently allowed, including:
  - Operands of `PROCEDURE DIVISION` statements
  - `VALUE` clauses (for numeric data items with large-precision `PICTURE`)
  - Condition-name values (on numeric data items with large-precision `PICTURE`)

- The maximum number of digits that you can specify in the arguments to NUMVAL and NUMVAL-C is raised from 18 to 31.
- The maximum value of the integer argument to the FACTORIAL function is 29.
- Intermediate results in arithmetic statements use *extended mode*.

When you specify ARITH(COMPAT):
- The maximum number of digit positions in the PICTURE clause for packed-decimal, external-decimal, and numeric-edited data items is 18.
- The maximum number of digits in a fixed-point numeric literal is 18.
- The maximum number of digits in the arguments to NUMVAL and NUMVAL-C is 18.
- The maximum value of the integer argument to the FACTORIAL function is 28.
- Intermediate results in arithmetic statements use *compatibility mode*.

RELATED CONCEPTS
Appendix D, "Intermediate results and arithmetic precision," on page 583

BINARY

BINARY is treated as a comment.

BINARY option syntax

```
             ┌─NATIVE─┐
►►──BINARY(──┴─S390───┴─)──────────────────────────────────────────►◄
```

Default is: BINARY(NATIVE)

Abbreviations are: None

CALLINT

Use CALLINT to indicate the call interface convention applicable to calls made with the CALL statement, and to indicate whether argument descriptors are to be generated.

CALLINT option syntax

```
              ┌─SYSTEM──┐
►►──CALLINT(──┼─OPTLINK─┼──┬──────────────────────────┬─)──────────►◄
              └─CDECL───┘  │    ┌─NODESC───────┐      │
                           └─,──┼─DESC─────────┼──────┘
                                ├─DESCRIPTOR───┤
                                └─NODESCRIPTOR─┘
```

Default is: CALLINT(SYSTEM,NODESC)

Abbreviations are: None

You can override this option for specific CALL statements by using the compiler directive >>CALLINT.

CALLINT has two sets of suboptions:

- Selecting a call interface convention:

  SYSTEM The SYSTEM suboption specifies that the call convention is that of the standard system linkage convention of the platform.

  SYSTEM is the only call interface convention supported on AIX.

  OPTLINK
  If you code the OPTLINK suboption, the compiler generates an I-level diagnostic, and the entire directive (not just the first keyword) is ignored.

  CDECL If you code the CDECL suboption, the compiler generates an I-level diagnostic, and the entire directive (not just the first keyword) is ignored.

- Specifying whether the argument descriptors are to be generated:

  DESC The DESC suboption specifies that an argument descriptor is passed for each argument on a CALL statement. For more information about argument descriptors, see the related references below.

  Attention: Do not specify the DESC suboption in object-oriented programs.

  DESCRIPTOR
  The DESCRIPTOR suboption is synonymous with the DESC suboption.

  NODESC The NODESC suboption specifies that no argument descriptors are passed for any arguments in a CALL statement.

  NODESCRIPTOR
  The NODESCRIPTOR suboption is synonymous with the NODESC suboption.

RELATED REFERENCES
Chapter 15, “Compiler-directing statements,” on page 295

CHAR

CHAR affects the representation and runtime treatment of USAGE DISPLAY and USAGE DISPLAY-1 data items.

CHAR option syntax

```
          ┌─NATIVE─┐
►►──CHAR(─┼─EBCDIC─┼─)──────────────────────────────────────────►◄
          └─S390───┘
```

Default is: CHAR(NATIVE)

Abbreviations are: None

Specify CHAR(NATIVE) to use the native character representation (the native format) of the platform. For COBOL for AIX, the native format is defined by the code page that is indicated by the locale in effect at run time. The code page can be a single-byte ASCII code page or an ASCII-based multibyte code page (UTF-8, EUC, or ASCII DBCS).

CHAR(EBCDIC) and CHAR(S390) are synonymous and indicate that DISPLAY and DISPLAY-1 data items are in the character representation of zSeries, that is, in EBCDIC.

However, DISPLAY and DISPLAY-1 data items defined with the NATIVE phrase in the USAGE clause are not affected by the CHAR(EBCDIC) option. They are always stored in the native format of the platform.

The CHAR(EBCDIC) compiler option has the following effects on runtime processing:

- **USAGE DISPLAY and USAGE DISPLAY-1 items:** Characters in data items that are described with USAGE DISPLAY are treated as single-byte EBCDIC format. Characters in data items that are described with USAGE DISPLAY-1 are treated as EBCDIC DBCS format. (In the bullets that follow, the term *EBCDIC* refers to single-byte EBCDIC format for USAGE DISPLAY and to EBCDIC DBCS format for USAGE DISPLAY-1.)
  - Data that is encoded in the native format is converted to EBCDIC format upon ACCEPT from the terminal.
  - EBCDIC data is converted to the native format upon DISPLAY to the terminal.
  - The content of alphanumeric literals and DBCS literals is converted to EBCDIC format for assignment to data items that are encoded in EBCDIC. For the rules about the comparison of character data when the CHAR(EBCDIC) option is in effect, see the related reference below about the COLLSEQ option.
  - Editing is done with EBCDIC characters.
  - Padding is done with EBCDIC spaces. Group items that are used in alphanumeric operations (such as assignments and comparisons) are padded with single-byte EBCDIC spaces regardless of the definition of the elementary items within the group.
  - Figurative constant SPACE or SPACES used in a VALUE clause for an assignment to, or in a relation condition with, a USAGE DISPLAY item is treated as a single-byte EBCDIC space (that is, X'40').
  - Figurative constant SPACE or SPACES used in a VALUE clause for an assignment to, or in a relation condition with, a DISPLAY-1 item is treated as an EBCDIC DBCS space (that is, X'4040').
  - Class tests are performed based on EBCDIC value ranges.
- **USAGE DISPLAY items:**
  - The program-name in CALL *identifier*, CANCEL *identifier*, or in a format-6 SET statement is converted to the native format if the data item referenced by *identifier* is encoded in EBCDIC.
  - The file-name in the data item referenced by *data-name* in ASSIGN USING *data-name* is converted to the native format if the data item is encoded in EBCDIC.

  - The file-name in the `SORT-CONTROL` special register is converted to native format before being passed to a sort or merge function. (`SORT-CONTROL` has the implicit definition `USAGE DISPLAY`.)
  - Zoned decimal data (numeric `PICTURE` clause with `USAGE DISPLAY`) and display floating-point data are treated as EBCDIC format. For example, the value of `PIC S9 value "1"` is `X'F1'` instead of `X'31'`.
- **Group items:** Alphanumeric group items are treated similarly to `USAGE DISPLAY` items. (Note that a `USAGE` clause for an alphanumeric group item applies to the elementary items within the group and not to the group itself.)

Hexadecimal literals are assumed to represent EBCDIC characters if the literals are assigned to, or compared with, character data. For example, `X'C1'` compares equal to an alphanumeric item that has the value ′A′.

Figurative constants `HIGH-VALUE` or `HIGH-VALUES`, `LOW-VALUE` or `LOW-VALUES`, `SPACE` or `SPACES`, `ZERO` or `ZEROS`, and `QUOTE` or `QUOTES` are treated logically as their EBCDIC character representations for assignments to or comparisons with data items that are encoded in EBCDIC.

In comparisons between alphanumeric `USAGE DISPLAY` items, the collating sequence used is the ordinal sequence of the characters based on their binary (hexadecimal) values as modified by an alternate collating sequence for single-byte characters, if specified.

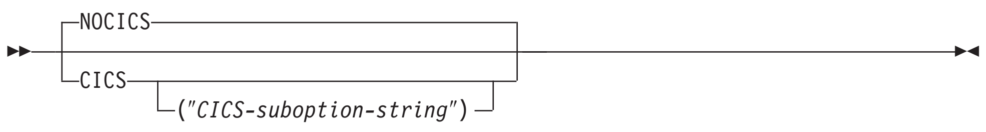
RELATED TASKS
"Specifying the code page for character data" on page 194

RELATED REFERENCES
"COLLSEQ" on page 253
"The encoding of XML documents" on page 371
Appendix C, "zSeries host data format considerations," on page 581

CICS

The `CICS` compiler option enables the integrated CICS translator and lets you specify CICS suboptions. You must use the `CICS` option if your COBOL source program contains `EXEC CICS` statements and the program has not been processed by the separate CICS translator.

CICS option syntax



Default is: `NOCICS`

Abbreviations are: None

Use the `CICS` option to compile CICS programs only. 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 single quotation marks to delimit the string of CICS suboptions.

You can use the syntax shown above in either the `CBL` or `PROCESS` statement. If you use the `CICS` option in the `cob2` or `cob2_r` command, only the single quotation mark (') can be used as the string delimiter: `-q"CICS('`*`options`*`')"`.

You can partition a long CICS suboption string into multiple suboption strings in multiple `CBL` or `PROCESS` statements. The CICS suboptions are concatenated in the order of their appearance. For example, suppose that your source file mypgm.cbl has the following code:

```
cbl . . . CICS("string2") . . .
cbl . . . CICS("string3") . . .
```

When you issue the command `cob2_r mypgm.cbl -q"CICS('string1')"`, the compiler passes the following suboption string to the integrated CICS translator:

```
"string1 string2 string3"
```

The concatenated strings are delimited with single spaces as shown. If multiple instances of the same `CICS` suboption are found, the last specification of that suboption in the concatenated string prevails. The compiler limits the length of the concatenated suboption string to 4 KB.

RELATED CONCEPTS
"Integrated CICS translator" on page 342

RELATED TASKS
Chapter 19, "Developing COBOL programs for CICS," on page 337

RELATED REFERENCES
"Conflicting compiler options" on page 245

COLLSEQ

`COLLSEQ` specifies the collating sequence for comparison of alphanumeric and DBCS operands.

COLLSEQ option syntax

```
            ┌─BINARY─┐
►►──COLLSEQ(─┼─LOCALE─┼─)──────────────────────────────────────►◄
            └─EBCDIC─┘
```

Default is: `COLLSEQ(BINARY)`

Abbreviations are: CS(L), CS(E), CS(BIN), CS(B)

You can specify the following suboptions for COLLSEQ:

- COLLSEQ(EBCDIC): Use the EBCDIC collating sequence rather than the ASCII collating sequence.
- COLLSEQ(LOCALE): Use locale-based collation (consistent with the cultural conventions for collation for the locale).
- COLLSEQ(BIN): Use the hexadecimal values of the characters; the locale setting has no effect. This setting will give better runtime performance.

If you use the PROGRAM COLLATING SEQUENCE clause in your source with an alphabet-name of STANDARD-1, STANDARD-2, or EBCDIC, the COLLSEQ option is ignored
| for comparison of alphanumeric operands. If you specify PROGRAM COLLATING
| SEQUENCE is NATIVE, the COLLSEQ option applies. Otherwise, when the alphabet-name specified in the PROGRAM COLLATING SEQUENCE clause is defined with literals, the collating sequence used is that given by the COLLSEQ option, modified by the user-defined sequence given by the alphabet-name. (For details, see the related reference about the ALPHABET clause.)

The PROGRAM COLLATING SEQUENCE clause has no effect on DBCS comparisons.

The former suboption NATIVE is deprecated. If you specify the NATIVE suboption, COLLSEQ(LOCALE) is assumed.

The following table summarizes the conversion and the collating sequence that are applicable, based on the types of data (ASCII or EBCDIC) used in a comparison and the COLLSEQ option in effect when the PROGRAM COLLATING SEQUENCE clause is not specified. If it is specified, the source specification has precedence over the compiler option specification. The CHAR option affects whether data is ASCII or EBCDIC.

Table 36. **Effect of comparand data type and collating sequence on comparisons**

Comparands	COLLSEQ(BIN)	COLLSEQ(LOCALE)	COLLSEQ(EBCDIC)
Both ASCII	No conversion is performed. The comparison is based on the binary value (ASCII).	No conversion is performed. The comparison is based on the current locale.	Both comparands are converted to EBCDIC. The comparison is based on the binary value (EBCDIC).
Mixed ASCII and EBCDIC	The EBCDIC comparand is converted to ASCII. The comparison is based on the binary value (ASCII).	The EBCDIC comparand is converted to ASCII. The comparison is based on the current locale.	The ASCII comparand is converted to EBCDIC. The comparison is based on the binary value (EBCDIC).
Both EBCDIC	No conversion is performed. The comparison is based on the binary value (EBCDIC).	The comparands are converted to ASCII. The comparison is based on the current locale.	No conversion is performed. The comparison is based on the binary value (EBCDIC).

RELATED TASKS
"Specifying the collating sequence" on page 8
"Controlling the collating sequence with a locale" on page 199

RELATED REFERENCES
"CHAR" on page 250
ALPHABET clause (*COBOL for AIX Language Reference*)

COMPILE

Use the `COMPILE` option only if you want to force full compilation even in the presence of serious errors. All diagnostics and object code will be generated. Do not try to run the object code if the compilation resulted in serious errors: the results could be unpredictable or an abnormal termination could occur.

COMPILE option syntax

```
            ┌S┐
  ┌NOCOMPILE(┼E┼)┐
  │         └W┘  │
►►┼──────────────┼──────────────────────────────────►◄
  ├COMPILE───────┤
  └NOCOMPILE─────┘
```

Default is: `NOCOMPILE(S)`

Abbreviations are: `C|NOC`

Use `NOCOMPILE` without any suboption to request a syntax check (only diagnostics produced, no object code). If you use `NOCOMPILE` without any suboption, several compiler options will have no effect because no object code will be produced, for example: `LIST`, `OPTIMIZE`, `SSRANGE`, and `TEST`.

Use `NOCOMPILE` with suboption `W`, `E`, or `S` for conditional full compilation. Full compilation (diagnosis and object code) will stop when the compiler finds an error of the level you specify (or higher), and only syntax checking will continue.

RELATED TASKS
"Finding coding errors" on page 312

RELATED REFERENCES
"Messages and listings for compiler-detected errors" on page 225

CURRENCY

You can use the `CURRENCY` option to provide an alternate default currency symbol to be used for a COBOL program. (The default currency symbol is the dollar sign ($).)

CURRENCY option syntax

```
   ┌─NOCURRENCY───────┐
►►─┤                  ├──────────────────────────────────────────────────►◄
   └─CURRENCY(literal)┘
```

Default is: NOCURRENCY

Abbreviations are: CURR|NOCURR

NOCURRENCY specifies that no alternate default currency symbol will be used.

To change the default currency symbol, specify CURRENCY(*literal*), where *literal* is a valid COBOL alphanumeric literal (optionally a hexadecimal literal) that represents a single character. The literal must not be from the following list:

- Digits zero (0) through nine (9)
- Uppercase alphabetic characters A B C D E G N P R S V X Z or their lowercase equivalents
- The space
- Special characters * + - / , . ; ( ) ″ = ′
- A figurative constant
- A null-terminated literal
- A DBCS literal
- A national literal

If your program processes only one currency type, you can use the CURRENCY option as an alternative to the CURRENCY SIGN clause for indicating the currency symbol you will use in the PICTURE clause of your program. If your program processes more than one currency type, you should use the CURRENCY SIGN clause with the WITH PICTURE SYMBOL phrase to specify the different currency sign types.

If you use both the CURRENCY option and the CURRENCY SIGN clause in a program, the CURRENCY option is ignored. Currency symbols specified in the CURRENCY SIGN clause or clauses can be used in PICTURE clauses.

When the NOCURRENCY option is in effect and you omit the CURRENCY SIGN clause, the dollar sign ($) is used as the PICTURE symbol for the currency sign.

Delimiter: You can delimit the CURRENCY option literal with either quotation marks or single quotation marks, regardless of the QUOTE|APOST compiler option setting.

RELATED TASKS
"Using currency signs" on page 58

DATEPROC

Use the DATEPROC option to enable the millennium language extensions of the COBOL compiler.

DATEPROC option syntax

```
     ┌─NODATEPROC──────────────────┐
►►───┼─────────────────────────────┼──────────────────────────────────────►◄
     └─DATEPROC─┬──────────────────┤
                │    ┌─FLAG───┐    │
                └─(──┴─NOFLAG─┴──)─┘
```

Default is: NODATEPROC, or DATEPROC(FLAG) if only DATEPROC is specified

Abbreviations are: DP|NODP

DATEPROC(FLAG)
: With DATEPROC(FLAG), the millennium language extensions are enabled, and the compiler produces a diagnostic message wherever a language element uses or is affected by the extensions. The message is usually an information-level or warning-level message that identifies statements that involve date-sensitive processing. Additional messages that identify errors or possible inconsistencies in the date constructs might be generated.

  Production of diagnostic messages, and their appearance in or after the source listing, is subject to the setting of the FLAG compiler option.

DATEPROC(NOFLAG)
: With DATEPROC(NOFLAG), the millennium language extensions are in effect, but the compiler does not produce any related messages unless there are errors or inconsistencies in the COBOL source.

NODATEPROC
: NODATEPROC indicates that the extensions are not enabled for this compilation unit. This option affects date-related program constructs as follows:

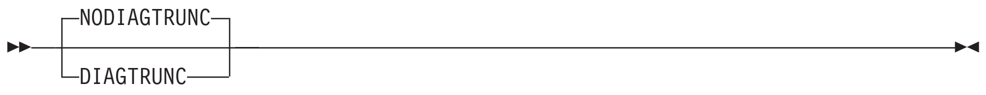
  - The DATE FORMAT clause is syntax-checked, but has no effect on the execution of the program.
  - The DATEVAL and UNDATE intrinsic functions have no effect. That is, the value returned by the intrinsic function is exactly the same as the value of the argument.
  - The YEARWINDOW intrinsic function returns a value of zero.

RELATED REFERENCES
"FLAG" on page 264
"YEARWINDOW" on page 292

DIAGTRUNC

DIAGTRUNC causes the compiler to issue a severity-4 (Warning) diagnostic message for MOVE statements with numeric receivers when the receiving data item has fewer integer positions than the sending data item or literal. In statements with multiple receivers, the message is issued separately for each receiver that could be truncated.

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 is also issued for moves to numeric receivers from alphanumeric data-names or literal senders, except when the sending field is reference modified.

There is no diagnostic for `COMP-5` receivers, nor for binary receivers when you specify the `TRUNC(BIN)` option.

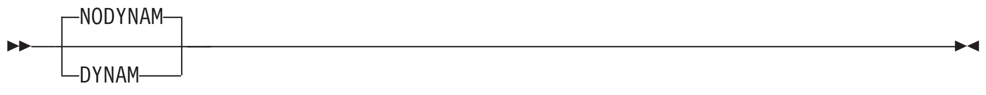
RELATED CONCEPTS
"Formats for numeric data" on page 43
"Reference modifiers" on page 101

RELATED REFERENCES
"TRUNC" on page 288

DYNAM

Use `DYNAM` to cause nonnested, separately compiled programs invoked through the `CALL` *literal* statement to be loaded for `CALL`, and deleted for `CANCEL`, dynamically at run time. (`CALL` *identifier* statements always result in a runtime load of the target program and are not affected by this option.)

DYNAM option syntax



Default is: `NODYNAM`

Abbreviations are: `DYN|NODYN`

The condition for the `ON EXCEPTION` phrase can occur for a `CALL` *literal* statement only if the `DYNAM` option is in effect.

Restriction: The `DYNAM` compiler option must not be used by programs that are translated by the separate or integrated CICS translator.

With `NODYNAM`, the target program-name is resolved through the linker.

With the `DYNAM` option, the following statement:

```
CALL "myprogram" . . .
```

has the identical behavior to these statements:

```
MOVE "myprogram" to id-1
CALL id-1 ...
```

RELATED CONCEPTS
"CALL identifier and CALL literal" on page 470

RELATED REFERENCES
"Conflicting compiler options" on page 245

ENTRYINT

This option is treated as a comment.

ENTRYINT option syntax

```
              ┌─SYSTEM──┐
►►──ENTRYINT(─┼─OPTLINK─┼─)──────────────────────────────────►◄
              └─CDECL───┘
```

Default is: `ENTRYINT(SYSTEM)`

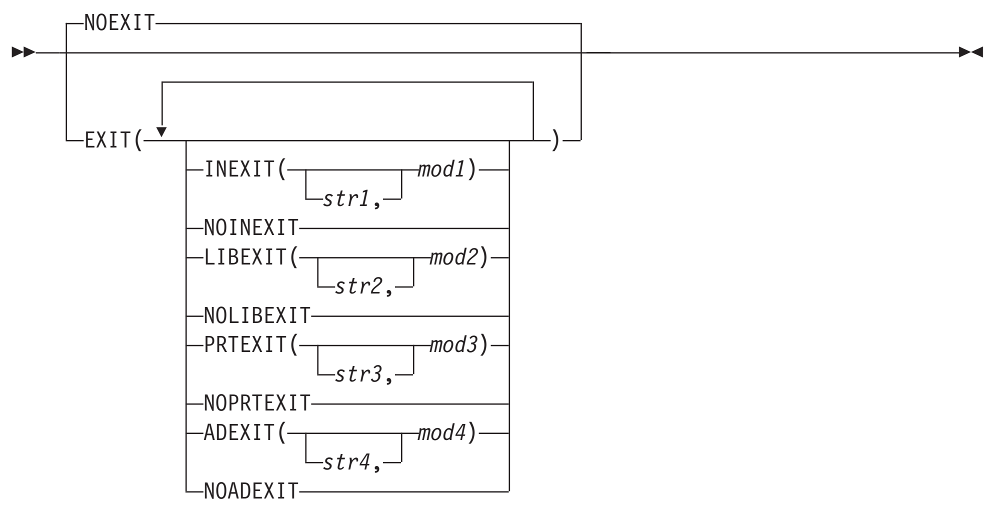
Abbreviations are: None

EXIT

Use the `EXIT` option to have the compiler accept user-supplied modules in place of SYSIN, SYSLIB (or copy library), SYSPRINT, and SYSADATA.

When creating your EXIT module, ensure that the module is linked as a shared library module before you run it with the COBOL compiler. EXIT modules are invoked with the system linkage convention of the platform.

EXIT option syntax



Default is: `NOEXIT`

Abbreviations are: `EX(INX|NOINX, LIBX|NOLIBX, PRTX|NOPRTX, ADX|NOADX)`

Option specification: You cannot specify the `EXIT` option in a `PROCESS` (or `CBL`) statement. You can specify it only in one of the following ways:

- As an option in the `cob2`, `cob2_j`, or `cob2_r` command
- In the COBOPT environment variable

If you specify the `EXIT` option without providing at least one suboption (that is, you specify `EXIT()`), `NOEXIT` will be in effect. You can specify the suboptions in any order, separated by either commas or spaces. If you specify both the positive and negative form of a suboption (`INEXIT|NOINEXT`, `LIBEXIT|NOLIBEXIT`, `PRTEXIT|NOPRTEXIT`, or `ADEXIT|NOADEXIT`), the form specified last takes effect. If you specify the same suboption more than once, the suboption specified last takes effect.

`INEXIT`([*'str1'*,]*mod1*)
: The compiler reads source code from a user-supplied load module (where *mod1* is the module name), instead of SYSIN.

`LIBEXIT`([*'str2'*,]*mod2*)
: The compiler obtains copybooks from a user-supplied load module (where *mod2* is the module name), instead of *library-name* or SYSLIB. For use with either `COPY` or `BASIS` statements.

`PRTEXIT`([*'str3'*,]*mod3*)
: The compiler passes printer-destined output to the user-supplied load module (where *mod3* is the module name), instead of SYSPRINT.

`ADEXIT`([*'str4'*,]*mod4*)
: The compiler passes the SYSADATA output to the user-supplied load module (where *mod4* is the module name).

The module names *mod1*, *mod2*, *mod3*, and *mod4* can refer to the same module.

The suboptions *'str1'*, *'str2'*, *'str3'*, and *'str4'* are character strings that are passed to the load module. These strings are optional; if you use them, they can be up to 64 characters in length and must be enclosed in single quotation marks. Any character can be used, but included single quotation marks must be doubled, and lowercase characters are folded to uppercase.

Character string formats

If *'str1'*, *'str2'*, *'str3'*, or *'str4'* is specified, the string is passed to the appropriate user-exit module with the following format, where `LL` is a halfword (on a halfword boundary) that contains the length of the string. The table shows the location of the character string in the parameter list.

LL	string

User-exit work area

When an exit is used, the compiler provides a user-exit work area that can be used to save the address of storage allocated by the exit module. This lets the module be reentrant.

The user-exit work area is 4 fullwords, residing on a fullword boundary, that is initialized to binary zeroes before the first exit routine is invoked. The address of the work area is passed to the exit module in a parameter list. After initialization, the compiler makes no further reference to the work area. So you will need to establish your own conventions for using the work area if more than one exit is active during the compilation. For example, the INEXIT module uses the first word in the work area, the LIBEXIT module uses the second word, and the PRTEXIT module uses the third word.

Parameter list for exit modules

The following table shows the format of the parameter list used by the compiler to communicate with the exit module.

Table 37. **Parameter list for exit modules**

Offset	Contains	Description of item
00	User-exit type	Halfword identifying which user exit is to perform the operation. 1=INEXIT; 2=LIBEXIT; 3=PRTEXIT; 4=ADEXIT
02	Operation code	Halfword indicating the type of operation. 0=OPEN; 1=CLOSE; 2=GET; 3=PUT; 4=FIND
04	Return code	Fullword, placed by the exit module, indicating status of the requested operation. 0=Successful; 4=End-of-data; 12=Failed
08	Data length	Fullword, placed by the exit module, specifying the length of the record being returned by the GET operation.

Table 37. **Parameter list for exit modules** (continued)

Offset	Contains	Description of item
12	Data or ’str2’	Data is a fullword, placed by the exit module, that contains the address of the record in a user-owned buffer, for the GET operation. ’str2’ applies only to OPEN. The first halfword (on a halfword boundary) contains the length of the string; the string follows.
16	User-exit work area	4-fullword work area provided by the compiler for use by user-exit module.
32	Text-name	Fullword that contains the address of a null-terminated string that contains the fully qualified text-name. Applies only to FIND.
36	User exit parameter string	Fullword that contains the address of a four-element array, each element of which is a structure that contains a 2-byte length field followed by a 64-character string that contains the exit parameter string.

Only the second element of the parameter string array is used for `LIBEXIT`, to store the length of the `LIBEXIT` parameter string followed by the parameter string.

Using INEXIT

When `INEXIT` is specified, the compiler loads the exit module (*mod1*) during initialization, and invokes the module using the OPEN operation code (op code). This lets the module prepare its source for processing and then pass the status of the OPEN request back to the compiler. Subsequently, each time the compiler requires a source statement, the exit module is invoked with the GET op code. The exit module then returns either the address and length of the next statement or the end-of-data indication (if no more source statements exist). When end-of-data is presented, the compiler invokes the exit module with the CLOSE op code so that the module can release any resources that are related to its input.

The compiler uses a parameter list to communicate with the exit module. The parameter list consists of 10 fullwords. The return code, data length, and data parameters are placed by the exit module for return to the compiler; and the other items are passed from the compiler to the exit module.

The table shows the contents of the parameter list and a description of each item.

Using LIBEXIT

When `LIBEXIT` is specified, the compiler loads the exit module (*mod2*) during initialization. Calls are made to the module by the compiler to obtain copybooks whenever `COPY` or `BASIS` statements are encountered.

Use LIB: If `LIBEXIT` is specified, the `LIB` compiler option must be in effect.

The first call invokes the module with an OPEN op code. This lets the module prepare the specified library-name for processing. The OPEN op code is also issued the first time a new library-name is specified. The exit module returns the status of the OPEN request to the compiler by passing a return code.

When the exit invoked with the OPEN op code returns, the exit module is then invoked with a FIND op code. The exit module establishes positioning at the requested text-name (or basis-name) in the specified library-name. This becomes the “active copybook.” When positioning is complete, the exit module passes an appropriate return code to the compiler.

The compiler then invokes the exit module with a GET op code, and the exit module passes the compiler the length and address of the record to be copied from the active copybook. The GET operation is repeated until the end-of-data indicator is passed to the compiler.

When end-of-data is presented, the compiler will issue a CLOSE request so that the exit module can release any resources related to its input.

Nested COPY statements: Any record from the active copybook can contain a `COPY` statement. (However, nested `COPY` statements cannot contain the `REPLACING` phrase, and a `COPY` statement with the `REPLACING` phrase cannot contain nested `COPY` statements.) When a valid nested `COPY` statement is encountered, the compiler issues a request:

- If the requested library-name from the nested `COPY` statement was not previously opened, the compiler invokes the exit module with an OPEN op code, followed by a FIND for the new text-name.
- If the requested library-name is already open, the compiler issues the FIND op code for the new requested text-name (an OPEN is not issued here).

The compiler does not permit recursive calls to text-name. That is, a copybook can be named only once in a set of nested `COPY` statements until the end-of-data for that copybook is reached.

When the exit module receives the OPEN or FIND request, it should push its control information concerning the active copybook onto a stack and then complete the requested action (OPEN or FIND). The newly requested text-name (or basis-name) now becomes the active copybook.

Processing continues in the normal manner with a series of GET requests until the end-of-data indicator is passed to the compiler.

At end-of-data for the nested active copybook, the exit module should pop its control information from the stack. The next request from the compiler will be a FIND, so that the exit module can reestablish positioning at the previous active copybook.

The compiler now invokes the exit module with a GET request, and the exit module must pass the same record that was passed previously from this copybook. The compiler verifies that the same record was passed, and then the processing continues with GET requests until the end-of-data indicator is passed.

The table shows the contents of the parameter list used for `LIBEXIT` and a description of each item.

Using PRTEXIT

When `PRTEXIT` is specified, the compiler loads the exit module (*mod3*) during initialization. The exit module is used in place of the SYSPRINT data set.

The compiler invokes the module using the OPEN operation code (op code). This lets the module prepare its output destination for processing and then pass the status of the OPEN request back to the compiler. Subsequently, each time the compiler has a line to be printed, the exit module is invoked with the PUT op code. The compiler supplies the address and length of the record that is to be printed, and the exit module returns the status of the PUT request to the compiler by a return code. The first byte of the record to be printed contains an ANSI printer control character.

Before the compilation is ended, the compiler invokes the exit module with the CLOSE op code so that the module can release any resources that are related to its output destination.

The table shows the contents of the parameter list used for `PRTEXIT` and a description of each item.

Using ADEXIT

When `ADEXIT` is specified, the compiler loads the exit module (*mod4*) during initialization. The exit module is called for each record written to the SYSADATA data set immediately after the record has been written to the file.

The compiler invokes the module using the OPEN operation code (op code). This lets the module prepare for processing and then pass the status of the OPEN request back to the compiler. Subsequently, each time the compiler has written a SYSADATA record, the exit module is invoked with the PUT op code. The compiler supplies the address and length of the SYSADATA record, and the exit module returns the status of the PUT request to the compiler by a return code.

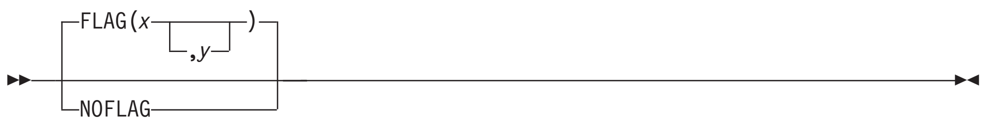
Before the compilation is ended, the compiler invokes the exit module with the CLOSE op code so that the module can release any resources.

The table shows the contents of the parameter list used for `ADEXIT` and a description of each item.

FLAG

Use `FLAG(`*x*`)` to produce diagnostic messages at the end of the source listing for errors of a severity level *x* or above.

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, messages of severity *y* and above are embedded in the listing after the line that caused the message. (See the related reference below for information about messages for exceptions.)

Use NOFLAG to suppress error flagging. NOFLAG does not suppress error messages for compiler options.

Embedded messages
- Embedding level-U messages is not recommended. The specification of embedded level-U messages is accepted, but does not produce any messages in the source.
- The FLAG option does not affect diagnostic messages that are produced before the compiler options are processed.
- Diagnostic messages that are produced during processing of compiler options, CBL or PROCESS statements, or BASIS, COPY, or REPLACE statements are not embedded in the source listing. All such messages appear at the beginning of the compiler output.
- Messages that are produced during processing of the *CONTROL or *CBL statement are not embedded in the source listing.

RELATED REFERENCES
"Messages and listings for compiler-detected errors" on page 225

FLAGSTD

Use FLAGSTD to specify the level or subset of Standard COBOL 85 to be regarded as conforming, and to get informational messages about Standard COBOL 85 elements that are included in your program.

You can specify any of the following items for flagging:
- A selected Federal Information Processing Standard (FIPS) COBOL subset
- Any of the optional modules
- Obsolete language elements
- Any combination of subset and optional modules
- Any combination of subset and obsolete elements

- IBM extensions (these are flagged any time that FLAGSTD is specified, and identified as ″nonconforming nonstandard″)

FLAGSTD option syntax

```
    ┌─NOFLAGSTD─────────────────┐
►►──┼───────────────────────────┼──────────────────────────────────────►◄
    └─FLAGSTD(x─┬────┬─┬────┬─)─┘
                └─yy─┘ └─,O─┘
```

Default is: NOFLAGSTD

Abbreviations are: None

x specifies the subset of Standard COBOL 85 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.

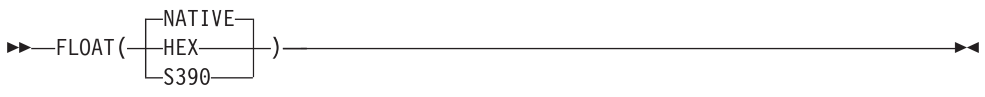
In the following example, the line number and column where a flagged clause, statement, or header occurred are shown with the associated message code and text. At the bottom is a summary of the total number of flagged items and their type.

```
  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
```

FLOAT

Specify FLOAT(NATIVE) to use the native floating-point representation format of the platform. For COBOL for AIX, the native format is the IEEE format.

FLOAT option syntax



Default is: FLOAT(NATIVE)

Abbreviations are: None

FLOAT(HEX) and FLOAT(S390) are synonymous and indicate that COMP-1 and COMP-2 data items are represented consistently with zSeries, that is, in hexadecimal floating-point format:

- Hexadecimal floating-point values are converted to IEEE format before any arithmetic operations (computations or comparisons).
- IEEE floating-point values are converted to hexadecimal format before being stored in floating-point data fields.
- Assignment to a floating-point item is done by converting the source floating-point data (for example, external floating point) to hexadecimal floating point as necessary.

However, COMP-1 and COMP-2 data items defined with the NATIVE keyword on the USAGE clause are not affected by the FLOAT(S390) option. They are always stored in the native format of the platform.

RELATED REFERENCES
Appendix C, "zSeries host data format considerations," on page 581

LIB

If your program uses `COPY`, `BASIS`, or `REPLACE` statements, the `LIB` compiler option must be in effect.

LIB option syntax

```
    ┌─NOLIB─┐
►►──┴─LIB───┴──────────────────────────────────────────────►◄
```

Default is: `NOLIB`

Abbreviations are: None

For `COPY` and `BASIS` statements, you need additionally to define the library or libraries from which the compiler can take the copied code:

- If the library-name is specified with a user-defined word (not a literal), you must set the corresponding environment variable to point to the desired directory and path for the copybook.
- If the library-name is omitted for a `COPY` statement, you can specify the path to be searched by using the `-I`*`xxx`* option of the `cob2`, `cob2_j`, or `cob2_r` command.
- If the library-name is specified with a literal, the literal value is treated as the actual path name.

RELATED REFERENCES
Chapter 15, "Compiler-directing statements," on page 295
"Conflicting compiler options" on page 245

LINECOUNT

Use `LINECOUNT(`*nnn*`)` to specify the number of lines to be printed on each page of the compilation listing, or use `LINECOUNT(0)` to suppress pagination.

LINECOUNT option syntax

```
►►──LINECOUNT(nnn)─────────────────────────────────────────►◄
```

Default is: `LINECOUNT(60)`

Abbreviations are: `LC`

nnn must be an integer between 10 and 255, or 0.

If you specify `LINECOUNT(0)`, no page ejects are generated in the compilation listing.

The compiler uses three lines of *nnn* for titles. For example, if you specify `LINECOUNT(60)`, 57 lines of source code are printed on each page of the output listing.

LIST

Use the `LIST` compiler option to produce a listing of the assembler-language expansion of your source code.

LIST option syntax

```
      ┌─NOLIST─┐
►►────┤        ├──────────────────────────────────────────────────►◄
      └─LIST───┘
```

Default is: `NOLIST`

Abbreviations are: None

Any `*CONTROL` (or `*CBL`) `LIST` or `NOLIST` statements that you code in the `PROCEDURE DIVISION` have no effect. They are treated as comments.

The assembler listing is written to a file that has the same name as the source program but has the suffix .wlist.

RELATED TASKS
"Getting listings" on page 316

RELATED REFERENCES
*CONTROL (*CBL) statement (*COBOL for AIX Language Reference*)

LSTFILE

Specify `LSTFILE(LOCALE)` to have your generated compiler listing encoded in the code page specified by the locale in effect. Specify `LSTFILE(UTF-8)` to have your generated compiler listing encoded in UTF-8.

LSTFILE option syntax

```
                ┌─LOCALE─┐
►►──LSTFILE(────┴─UTF-8──┴──)─────────────────────────────────────►◄
```

Default is: LSTFILE(LOCALE)

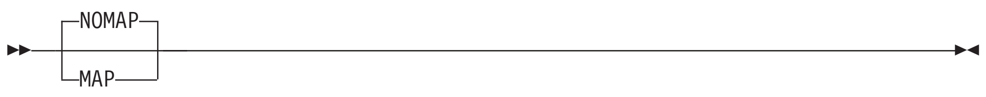
Abbreviations are: LST

RELATED REFERENCES
Chapter 11, "Setting the locale," on page 193

MAP

Use MAP to produce a listing of the items defined in the DATA DIVISION.

MAP option syntax



Default is: NOMAP

Abbreviations are: None

The output includes the following items:
- DATA DIVISION map
- Global tables
- Literal pools
- Nested program structure map, and program attributes
- Size of the program's WORKING-STORAGE and LOCAL-STORAGE

If you want to limit the MAP output, use *CONTROL MAP or NOMAP statements in the DATA DIVISION. Source statements that follow *CONTROL NOMAP are not included in the listing until a *CONTROL MAP statement switches the output back to normal MAP format. For example:

```
*CONTROL NOMAP          *CBL NOMAP
    01  A                   01  A
    02  B                   02  B
*CONTROL MAP            *CBL MAP
```

By selecting the MAP option, you can also print an embedded MAP report in the source code listing. The condensed MAP information is printed to the right of data-name definitions in the FILE SECTION, LOCAL-STORAGE SECTION, and LINKAGE SECTION of the DATA DIVISION. When both XREF data and an embedded MAP summary are on the same line, the embedded summary is printed first.

"Example: MAP output" on page 320

RELATED CONCEPTS
Chapter 17, "Debugging," on page 307

RELATED TASKS
"Getting listings" on page 316

RELATED REFERENCES
*CONTROL (*CBL) statement (*COBOL for AIX Language Reference*)

MAXMEM

Use `MAXMEM` in conjunction with `OPTIMIZE` to limit the amount of memory used by the compiler for local tables of specific, memory-intensive optimizations to *size* KB. If that memory is insufficient for a particular optimization, the scope of the optimization is reduced.

MAXMEM option syntax

```
►►──MAXMEM(size)──────────────────────────────────────────────────►◄
```

| Default is: `MAXMEM(2048)`

Abbreviations are: None

A value of -1 permits each optimization to take as much memory as it needs without checking for limits. Depending on the source file being compiled, the size of subprograms in the source, the machine configuration, and the workload on the system, this amount might exceed available system resources.

Usage notes

- The limit set by `MAXMEM` is the amount of memory for specific optimizations, not for the compiler as a whole. Tables required during the entire compilation process are not affected or included in this limit.
- Setting a large limit has no negative effect on the compilation of source files where the compiler needs less memory.
- Limiting the scope of optimization does not necessarily mean that the resulting program will be slower, only that the compiler might finish before finding all opportunities to improve performance.
- Increasing the limit does not necessarily mean that the resulting program will be faster, only that the compiler is better able to find opportunities to increase performance if they exist.

Depending on the source file being compiled, the size of the subprograms in the source, the machine configuration, and the workload on the system, setting the limit too high might lead to page-space exhaustion. In particular, specifying `MAXMEM(-1)` lets the compiler try to use an unlimited amount of storage, which in the worst case could exhaust the resources of the machine.

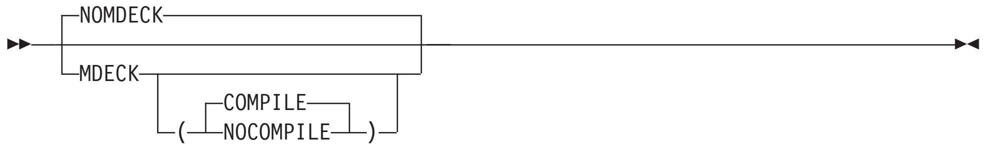
RELATED REFERENCES
"OPTIMIZE" on page 275

MDECK

The MDECK compiler option specifies that output from library processing (that is, expansion of COPY, BASIS, REPLACE, EXEC CICS, and EXEC SQL INCLUDE statements) is written to a file.

The MDECK output is written in the current directory to a file that has the same name as the COBOL source file and a suffix of .dek.

MDECK option syntax



Default is: NOMDECK

Abbreviations are: NOMD, MD, MD(C), MD(NOC)

Option specification:

You cannot specify the MDECK option in a PROCESS (or CBL) statement. You can specify it only in one of the following ways:
- As an option in the cob2, cob2_j, or cob2_r command
- 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 setting of the COMPILE|NOCOMPILE option.
- 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 setting of the COMPILE option.

When you specify MDECK with no suboption, MDECK(COMPILE) is implied.

Contents of the MDECK output file:

When you use the MDECK option with the CICS compiler option (integrated CICS translator) or the SQL compiler option (DB2 coprocessor), in general, EXEC CICS and EXEC SQL statements in the COBOL source program are included in the MDECK output as is. However, EXEC SQL INCLUDE statements are expanded in the MDECK output in the same manner as COPY statements.

CBL, PROCESS, *CONTROL, and *CBL card images are passed to the MDECK output file in the proper locations.

For a batch compilation (multiple COBOL source programs in a single input file), a single `MDECK` output file that contains the complete expanded source is created.

Any `SEQUENCE` compiler-option processing is reflected in the `MDECK` file.

`COPY` statements are included in the `MDECK` file as comments.

RELATED REFERENCES
"Conflicting compiler options" on page 245
Chapter 15, "Compiler-directing statements," on page 295

NCOLLSEQ

NCOLLSEQ option syntax

```
            ┌─BINARY─┐
►►──NCOLLSEQ(──┴─LOCALE─┴──)──────────────────────────────────────►◄
```

`NCOLLSEQ` specifies the collating sequence for comparison of class national operands.

Default is: `NCOLLSEQ(BINARY)`

Abbreviations are: `NCS(L)`, `NCS(BIN)`, `NCS(B)`

`NCOLLSEQ(BIN)` uses the hexadecimal values of the character pairs.

`NCOLLSEQ(LOCALE)` uses the algorithm for collation order that is associated with the locale value in effect.

RELATED TASKS
"Comparing two class national operands" on page 187
"Controlling the collating sequence with a locale" on page 199

NSYMBOL

The `NSYMBOL` option controls the interpretation of the `N` symbol used in literals and `PICTURE` clauses, indicating whether national or DBCS processing is assumed.

NSYMBOL option syntax

```
           ┌─NATIONAL─┐
►►──NSYMBOL(──┴─DBCS─────┴──)─────────────────────────────────────►◄
```

Default is: `NSYMBOL(NATIONAL)`

Abbreviations are: NS(NAT|DBCS)

With NSYMBOL(NATIONAL):

- Data items defined with a PICTURE clause that consists only of the symbol N without the USAGE clause are treated as if the USAGE NATIONAL clause is specified.
- Literals of the form N". . ." or N'. . .' are treated as national literals.

With NSYMBOL(DBCS):

- Data items defined with a PICTURE clause that consists only of the symbol N without the USAGE clause are treated as if the USAGE DISPLAY-1 clause is specified.
- Literals of the form N". . ." or N'. . .' are treated as DBCS literals.

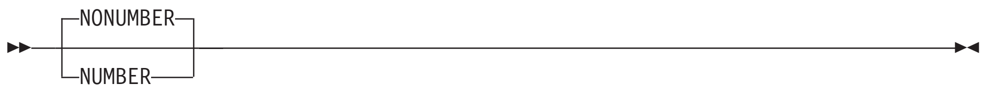
The NSYMBOL(DBCS) option provides compatibility with previous releases of IBM COBOL, and the NSYMBOL(NATIONAL) option makes the handling of the above language elements consistent with Standard COBOL 2002 in this regard.

NSYMBOL(NATIONAL) is recommended for applications that use Unicode data or object-oriented syntax for Java interoperability.

NUMBER

Use the NUMBER compiler option if you have line numbers in your source code and want those numbers to be used in error messages and SOURCE, MAP, LIST, and XREF listings.

NUMBER option syntax



Default is: NONUMBER

Abbreviations are: NUM|NONUM

If you request NUMBER, the compiler checks columns 1 through 6 to make sure that they contain only numbers and that the numbers are in numeric collating sequence. (In contrast, SEQUENCE checks the characters in these columns according to EBCDIC collating sequence.) When a line number is found to be out of sequence, the compiler assigns to it a line number with a value one higher than the line number of the preceding statement. The compiler flags the new value with two asterisks and includes in the listing a message indicating an out-of-sequence error. Sequence-checking continues with the next statement, based on the newly assigned value of the previous line.

If you use COPY statements and NUMBER is in effect, be sure that your source program line numbers and the copybook line numbers are coordinated.

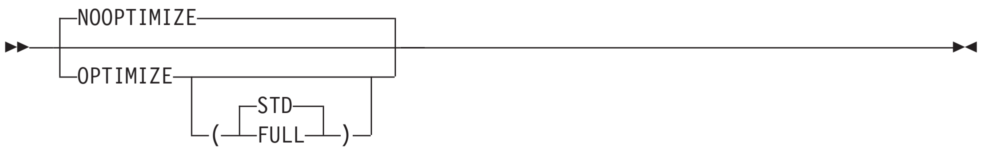
Use NONUMBER if you do not have line numbers in your source code, or if you want the compiler to ignore the line numbers you do have in your source code. With

NONUMBER in effect, the compiler generates line numbers for your source statements and uses those numbers as references in listings.

OPTIMIZE

Use OPTIMIZE to reduce the run time of your object program. Optimization might also reduce the amount of storage your object program uses. Optimizations performed include the propagation of constants, instruction scheduling, and the elimination of computations whose results are never used.

OPTIMIZE option syntax



Default is: NOOPTIMIZE

Abbreviations are: OPT|NOOPT

If OPTIMIZE is specified without any suboptions, OPTIMIZE(STD) will be in effect.

The FULL suboption requests that, in addition to the optimizations performed with OPT(STD), the compiler discard unreferenced data items from the DATA DIVISION and suppress generation of code to initialize these data items to the values in their VALUE clauses. When OPT(FULL) is in effect, all unreferenced level-77 items and elementary level-01 items are discarded. In addition, level-01 group items are discarded if none of their subordinate items are referenced. The deleted items are shown in the listing. If the MAP option is in effect, a BL number of XXXXX in the data map information indicates that the data item was discarded.

Recommendation: Use OPTIMIZE(FULL) for database applications. It can make a huge performance improvement, because unused constants included by the associated COPY statements are eliminated. However, if your database application depends on unused data items, see the recommendations below.

Unused data items: Do not use OPT(FULL) if your programs depend on making use of unused data items. In the past, this was commonly done in two ways:

- A technique sometimes used in old OS/VS COBOL programs was to place an unreferenced table after a referenced table and use out-of-range subscripts on the first table to access the second table. To determine whether your programs use this technique, use the SSRANGE compiler option with the CHECK(ON) runtime option. To work around this problem, use the ability of newer COBOL to code large tables and use just one table.
- Place eye-catcher data items in the WORKING-STORAGE SECTION to identify the beginning and end of the program data or to mark a copy of a program for a library tool that uses the data to identify the version of a program. To solve this

problem, initialize these items with PROCEDURE DIVISION statements rather than VALUE clauses. With this method, the compiler will consider these items used and will not delete them.

The OPTIMIZE option is turned off in the case of a severe-level error or higher.

RELATED CONCEPTS
"Optimization" on page 551

RELATED REFERENCES
"Conflicting compiler options" on page 245
"MAXMEM" on page 271

PGMNAME

The PGMNAME option controls the handling of program-names and entry-point names.

PGMNAME option syntax

```
              ┌─UPPER─┐
►►──PGMNAME(──┴─MIXED─┴──)─────────────────────────────────────────────►◄
```

Default is: PGMNAME(UPPER)

Abbreviations are: PGMN(LU|LM)

For compatibility with COBOL for OS/390® & VM, LONGMIXED and LONGUPPER are also supported.

LONGUPPER can be abbreviated as UPPER, LU, or U. LONGMIXED can be abbreviated as MIXED, LM, or M.

COMPAT: If you specify PGMNAME(COMPAT), PGMNAME(UPPER) will be set, and you will receive a warning message.

PGMNAME controls the handling of names used in the following contexts:
- Program-names defined in the PROGRAM-ID paragraph
- Program entry-point names in the ENTRY statement
- Program-name references in:
  - CALL statements
  - CANCEL statements
  - SET *procedure-pointer* TO ENTRY statements
  - SET *function-pointer* TO ENTRY statements

PGMNAME(UPPER)

With `PGMNAME(UPPER)`, program-names that are specified in the `PROGRAM-ID` paragraph as COBOL user-defined words must follow the normal COBOL rules for forming a user-defined word:

- The program-name can be up to 30 characters in length.
- All the characters used in the name must be alphabetic, digits, or the hyphen.
- At least one character must be alphabetic.
- The hyphen cannot be used as the first or last character.

When a program-name is specified as a literal, in either a definition or a reference, then:

- The program-name can be up to 160 characters in length.
- All the characters used in the name must be alphabetic, digits, or the hyphen.
- At least one character must be alphabetic.
- The hyphen cannot be used as the first or last character.

External program-names are processed with alphabetic characters folded to uppercase.

PGMNAME(MIXED)

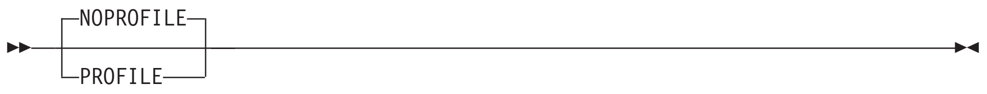
With `PGMNAME(MIXED)`, program-names are processed as is, without truncation, translation, or folding to uppercase.

With `PGMNAME(MIXED)`, all program-name definitions must be specified using the literal format of the program-name in the `PROGRAM-ID` paragraph or `ENTRY` statement.

PROFILE

`PROFILE` instructs the compiler to prepare the object file for AIX profiling (performance analysis with the `prof` utility). Use this compiler option with the `-p` option of the `cob2` command or one of its variants.

PROFILE option syntax



Default is: `NOPROFILE`

Abbreviations are: None

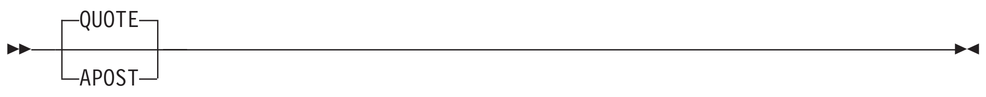
RELATED TASKS
"Compiling from the command line" on page 219

RELATED REFERENCES
"cob2, cob2_j, and cob2_r options" on page 227

QUOTE/APOST

Use QUOTE if you want the figurative constant [ALL] QUOTE or [ALL] QUOTES to represent one or more quotation mark (") characters. Use APOST if you want the figurative constant [ALL] QUOTE or [ALL] QUOTES to represent one or more single quotation mark (') characters.

QUOTE/APOST option syntax



Default is: QUOTE

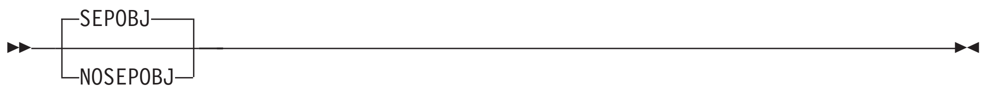
Abbreviations are: Q|APOST

Delimiters: You can use either quotation marks or single quotation marks as literal delimiters regardless of whether the APOST or QUOTE option is in effect. The delimiter character used as the opening delimiter for a literal must be used as the closing delimiter for that literal.

SEPOBJ

SEPOBJ specifies whether each of the outermost COBOL programs in a batch compilation is to be generated as a separate object file rather than as a single object file.

SEPOBJ option syntax



Default is: SEPOBJ

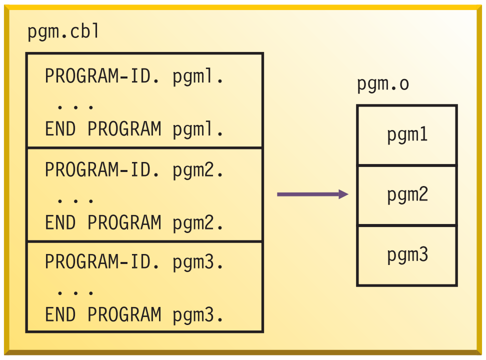
Abbreviations are: None

Batch compilation

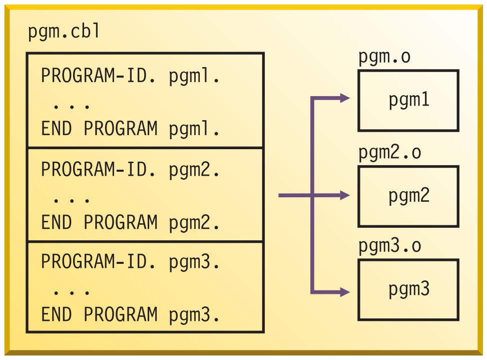
When multiple outermost programs (nonnested programs) are compiled with a single batch invocation of the compiler, the number of files produced for the object program output of the batch compilation depends on the compiler option `SEPOBJ`.

Assume that the COBOL source file pgm.cbl contains three outermost COBOL programs named `pgm1`, `pgm2`, and `pgm3`. The following figures illustrate whether the object program output is generated as one file (with `NOSEPOBJ`) or three files (with `SEPOBJ`).

Batch compilation with `NOSEPOBJ`



Batch compilation with `SEPOBJ`



Usage notes

- The `SEPOBJ` option is required to conform to Standard COBOL 85 where `pgm2` or `pgm3` in the above example is called using `CALL` *identifier* from another program.
- If `NOSEPOBJ` is in effect, the object files are given the name of the source file but with suffix .o. If `SEPOBJ` is in effect, the names of the object files are based on the `PROGRAM-ID` name with suffix .o.
- The programs called using `CALL` *identifier* must be referred to by the names of the object files (rather than the `PROGRAM-ID` names) where `PROGRAM-ID` and the object file name do not match.

You must give the object file a valid file-name for the platform and the file system.

SEQUENCE

When you use SEQUENCE, the compiler examines columns 1 through 6 to check that the source statements are arranged in ascending order according to their ASCII collating sequence. The compiler issues a diagnostic message if any statements are not in ascending order.

Source statements with blanks in columns 1 through 6 do not participate in this sequence check and do not result in messages.

SEQUENCE option syntax

```
    ┌─SEQUENCE───┐
►►──┴─NOSEQUENCE─┴─────────────────────────────────────────────►◄
```

Default is: SEQUENCE

Abbreviations are: SEQ|NOSEQ

If you use COPY statements with the SEQUENCE option in effect, be sure that your source program's sequence fields and the copybook sequence fields are coordinated.

If you use NUMBER and SEQUENCE, the sequence is checked according to numeric, rather than ASCII, collating sequence.

Use NOSEQUENCE to suppress this checking and the diagnostic messages.

RELATED TASKS
"Finding line sequence problems" on page 312

SIZE

Use SIZE to indicate the amount of main storage to be made available for compilation.

SIZE option syntax

```
►►──SIZE(──┬─nnnnn─┬──)────────────────────────────────────────►◄
           └─nnnK──┘
```

Default is: 8388608 bytes (approximately 8 MB)

Abbreviations are: SZ

nnnnn specifies a decimal number, which must be at least 800768.

*nnn*K specifies a decimal number in 1-KB increments, where 1 KB = 1024 bytes. The minimum acceptable value is 782K.

SOSI

The `SOSI` option affects the treatment of values `X'1E'` and `X'1F'` in alphanumeric literals, comments, and DBCS user-defined words.

SOSI option syntax

```
      ┌─NOSOSI─┐
►►────┤        ├──────────────────────────────────────────────────────────►◄
      └─SOSI───┘
```

Default is: `NOSOSI`

Abbreviations are: None

`NOSOSI` With `NOSOSI`, character positions that have values `X'1E'` and `X'1F'` are treated as data characters.

`NOSOSI` conforms to Standard COBOL 85.

`SOSI` With `SOSI`, COBOL for AIX shift-out (SO) and shift-in (SI) control characters delimit ASCII DBCS character strings in COBOL source programs. The SO and SI characters have the encoded values of `X'1E'` and `X'1F'`, respectively.

SO and SI characters have no effect on COBOL for AIX source code, except to act as placeholders for host DBCS SO and SI characters to ensure proper data handling when remote files are converted from EBCDIC to ASCII.

When the `SOSI` option is in effect, in addition to existing rules for COBOL for AIX, the following rules apply:

- All DBCS character strings (in user-defined words, DBCS literals, alphanumeric literals, national literals, and in comments) must be delimited by the SO and SI characters.
- User-defined words cannot contain both DBCS and SBCS characters.
- The maximum length of a DBCS user-defined word is 14 DBCS characters.
- Double-byte uppercase alphabetic letters are equivalent to the corresponding double-byte lowercase letters when used in user-defined words.
- A DBCS user-defined word must contain at least one letter that does not have its counterpart in a single-byte representation.
- Double-byte representations of single-byte characters for A-Z, a-z, 0-9, and the hyphen (-) cannot be included within a DBCS user-defined word. Rules applicable to these characters in single-byte representation apply to those in double-byte representation. For example, the hyphen (-) cannot appear as the first or the last character in a user-defined word.

- For alphanumeric literals that contain `X'1E'` or `X'1F'` values, the following rules apply when the `SOSI` compiler option is in effect:
  - Character positions with `X'1E'` and `X'1F'` are treated as SO and SI characters.
  - Character positions with `X'1E'` and `X'1F'` are included in the character string value of the literal unless it is part of a DBCS or alphanumeric literal that is not represented in hexadecimal notation.
  - The SO and SI characters must be paired (with each pair starting with the SO character) with zero or an even number of intervening bytes.
  - The pairs of SO and SI characters cannot be nested.
  - The alphanumeric literals cannot be continued. Their maximum length is limited by the available positions in Area B on a single source line.
  - The alphanumeric literals cannot be used in any of the following language elements:
    - `ALPHABET` clause
    - `ASSIGN` clause
    - `CLASS` clause
    - `CURRENCY SIGN` clause
    - `END METHOD` marker
    - `END PROGRAM` marker
    - `METHOD-ID` paragraph
    - `PADDING CHARACTER` clause
    - `PROGRAM-ID` paragraph
    - `RERUN` clause
    - `STOP` statement
- To embed DBCS quotation marks within an `N`-literal delimited by quotation marks, use two consecutive DBCS quotation marks to represent a single DBCS quotation mark. Do not include a single DBCS quotation mark in an `N`-literal if the literal is delimited by quotation marks. The same rule applies to single quotation marks.
- The `SHIFT-OUT` and `SHIFT-IN` special registers are defined with `X'0E'` and `X'0F'` regardless of whether the `SOSI` option is in effect.

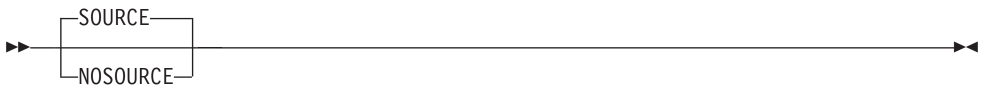
In general, host COBOL programs that are sensitive to the encoded values for the SO and SI characters will not have the same behavior on the AIX workstation.

RELATED TASKS
"Handling differences in ASCII multibyte and EBCDIC DBCS strings" on page 460

SOURCE

Use `SOURCE` to get a listing of your source program. This listing will include any statements embedded by `PROCESS` or `COPY` statements.

SOURCE option syntax



Default is: SOURCE

Abbreviations are: S|NOS

You must specify SOURCE if you want embedded messages in the source listing.

Use NOSOURCE to suppress the source code from the compiler output listing.

If you want to limit the SOURCE output, use *CONTROL SOURCE or NOSOURCE statements in your PROCEDURE DIVISION. Source statements that follow a *CONTROL NOSOURCE statement are not included in the listing until a subsequent *CONTROL SOURCE statement switches the output back to normal SOURCE format.

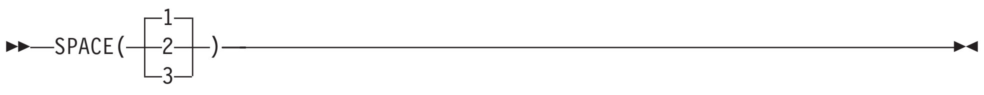
"Example: MAP output" on page 320

RELATED REFERENCES
*CONTROL (*CBL) statement (*COBOL for AIX 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 282

SPILL

This option specifies the number of KB set aside for the register spill area. If the program being compiled is very complex or large, this option might be required.

SPILL option syntax

```
►►──SPILL(n)──────────────────────────────────────────────►◄
```

Default is: `SPILL(512)`

Abbreviations are: None

The spill size, *n*, is any integer between 96 and 32704.

SQL

Use the `SQL` compiler option to enable the DB2 coprocessor and to specify DB2 suboptions. You must specify the `SQL` option if your COBOL source program contains SQL statements and the program has not been processed by the DB2 precompiler.

SQL option syntax

```
     ┌─NOSQL───────────────────────────────┐
►►───┼─────────────────────────────────────┼──────────────►◄
     └─SQL─┬─────────────────────────────┬─┘
           └─("DB2-suboption-string")────┘
```

Default is: `NOSQL`

Abbreviations are: None

If `NOSQL` is in effect, any SQL statements found in the source program are diagnosed and discarded.

Use either quotation marks or single quotation marks to delimit the string of DB2 suboptions.

You can use the syntax shown above in either the `CBL` or `PROCESS` statement. If you use the `SQL` option in the `cob2` command or its variants `cob2_j` or `cob2_r`, only the single quotation mark (') can be used as the suboption string delimiter: `-q"SQL('`*`suboptions`*`')"`.

RELATED TASKS
Chapter 18, "Programming for a DB2 environment," on page 331

"Compiling with the SQL option" on page 333
"Separating DB2 suboptions" on page 334

RELATED REFERENCES
"Conflicting compiler options" on page 245

SSRANGE

Use SSRANGE to generate code that checks whether subscripts (including ALL subscripts) or indexes try to reference an area outside the region of the table. Each subscript or index is not individually checked for validity; rather, the effective address is checked to ensure that it does not cause a reference outside the region of the table.

Variable-length items are also checked to ensure that the reference is within their maximum defined length.

SSRANGE option syntax

```
   ┌─NOSSRANGE─┐
►►─┴─SSRANGE───┴──────────────────────────────────────────────────►◄
```

Default is: NOSSRANGE

Abbreviations are: SSR|NOSSR

Reference modification expressions are checked to ensure that:
- The starting position is greater than or equal to 1.
- The starting position is not greater than the current length of the subject data item.
- The length value (if specified) is greater than or equal to 1.
- The starting position and length value (if specified) do not reference an area beyond the end of the subject data item.

If SSRANGE is in effect at compile time, range-checking code is generated. You can inhibit range checking by specifying the CHECK(OFF) runtime option. Doing so leaves range-checking code dormant in the object code. Optionally, the range-checking code can be used to aid in resolving unexpected errors without recompilation.

If an out-of-range condition is detected, an error message is generated and the program is terminated.

Attention: Range checking is done only if you compile a program with the SSRANGE option and run it with the CHECK(ON) option.

RELATED CONCEPTS
"Reference modifiers" on page 101

RELATED TASKS
"Checking for valid ranges" on page 312

RELATED REFERENCES
"CHECK" on page 301

TERMINAL

Use `TERMINAL` to send progress and diagnostic messages to the display device.

TERMINAL option syntax

```
      ┌─TERMINAL───┐
►►────┤            ├──────────────────────────────────────────────►◄
      └─NOTERMINAL─┘
```

Default is: `TERMINAL`

Abbreviations are: `TERM|NOTERM`

Use `NOTERMINAL` if you do not want this additional output.

TEST

Use `TEST` to produce object code that contains symbol and statement information that enables the debugger to perform symbolic source-level debugging.

TEST option syntax

```
      ┌─NOTEST─┐
►►────┤        ├──────────────────────────────────────────────────►◄
      └─TEST───┘
```

Default is: `NOTEST`

Abbreviations are: None

Use `NOTEST` if you do not want to generate object code that has debugging information. Programs compiled with `NOTEST` run with the debugger, but have limited debugging support.

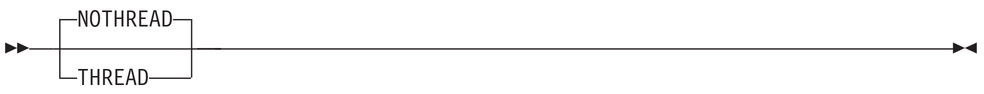
If you use the `WITH DEBUGGING MODE` clause, the `TEST` option is turned off. `TEST` will appear in the list of options, but a diagnostic message is issued to advise you that because of the conflict, `TEST` was not in effect.

RELATED REFERENCES
"Conflicting compiler options" on page 245

THREAD

THREAD indicates that a COBOL application is to be enabled for execution in a run unit that has multiple threads.

THREAD option syntax



Default is: NOTHREAD

Abbreviations are: None

All programs within a run unit must be compiled with the same option: either THREAD or NOTHREAD.

When the THREAD option is in effect, the following 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
- RERUN
- Segmentation module
- STOP *literal* statement
- STOP RUN
- USE FOR DEBUGGING statement

Usage notes
- RERUN is flagged as an error with THREAD, but is accepted as a comment with NOTHREAD.
- If you use the THREAD option, compile by using the command:
  - cob2_j for object-oriented (OO) clients and classes
  - cob2_r for non-CICS programs

  cob2_j causes the THREAD option to automatically be included as a default invocation option when you compile COBOL programs.
- **CICS:** Compile CICS programs by using the cob2_r command and the NOTHREAD 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.

RELATED TASKS
"Compiling from the command line" on page 219
Chapter 29, "Preparing COBOL programs for multithreading," on page 505

TRUNC

TRUNC affects the way that binary data is truncated during moves and arithmetic operations.

TRUNC option syntax

```
        ┌─STD─┐
►►──TRUNC(─┼─OPT─┼─)─────────────────────────────────────────────────►◄
        └─BIN─┘
```

Default is: TRUNC(STD)

Abbreviations are: None

TRUNC has no effect on COMP-5 data items; COMP-5 items are handled as if TRUNC(BIN) were in effect regardless of the TRUNC suboption specified.

TRUNC(STD)
TRUNC(STD) applies only to USAGE BINARY receiving fields in MOVE statements and arithmetic expressions. When TRUNC(STD) is in effect, the final result of an arithmetic expression, or the sending field in the MOVE statement, is truncated to the number of digits in the PICTURE clause of the BINARY receiving field.

TRUNC(OPT)
TRUNC(OPT) is a performance option. When TRUNC(OPT) is in effect, the compiler assumes that data conforms to PICTURE specifications in USAGE BINARY receiving fields in MOVE statements and arithmetic expressions. The results are manipulated in the most optimal way, either truncating to the number of digits in the PICTURE clause, or to the size of the binary field in storage (halfword, fullword, or doubleword).

Tip: Use the TRUNC(OPT) option only if you are sure that the data being moved into the binary areas will not have a value with larger precision than that defined by the PICTURE clause for the binary item. Otherwise, unpredictable results could occur. This truncation is performed in the most efficient manner possible; therefore, the results are dependent on the particular code sequence generated. It is not possible to predict the truncation without seeing the code sequence generated for a particular statement.

TRUNC(BIN)
The TRUNC(BIN) option applies to all COBOL language that processes USAGE BINARY data. When TRUNC(BIN) is in effect, all binary items (USAGE COMP, COMP-4, or BINARY) are handled as native hardware binary items, that is, as if they were each individually declared USAGE COMP-5:

- BINARY receiving fields are truncated only at halfword, fullword, or doubleword boundaries.

- BINARY sending fields are handled as halfwords, fullwords, or doublewords when the receiver is numeric; TRUNC(BIN) has no effect when the receiver is not numeric.
- The full binary content of fields is significant.
- DISPLAY will convert the entire content of binary fields with no truncation.

Recommendations: TRUNC(BIN) is the recommended option for programs that use binary values set by other products. Other products, such as DB2, C/C++, and PL/I, might place values in COBOL binary data items that do not conform to the PICTURE clause of the data items. You can use TRUNC(OPT) with CICS programs 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.

TRUNC example 1

|

```
01  BIN-VAR     PIC S99 USAGE BINARY.
. . .
    MOVE 123451 to BIN-VAR
```

The following table shows values of the data items after the MOVE:

Data item	Decimal	Hex	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 gained by not doing a decimal truncation.

TRUNC example 2

```
01  BIN-VAR     PIC 9(6)  USAGE BINARY
. . .
    MOVE 1234567891 to BIN-VAR
```

The following table shows values of the data items after the MOVE:

Data item	Decimal	Hex	Display
Sender	1234567891	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.

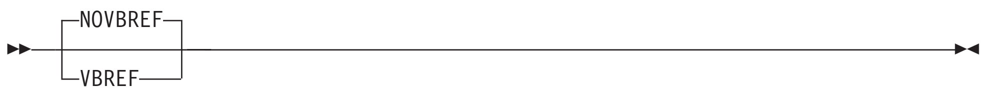
RELATED CONCEPTS
"Formats for numeric data" on page 43

RELATED REFERENCES
VALUE clause (*COBOL for AIX Language Reference*)

VBREF

Use VBREF to get a cross-reference among all verb used in the source program and the line numbers in which they are used. VBREF also produces a summary of how many times each verb was used in the program.

VBREF option syntax



Default is: NOVBREF

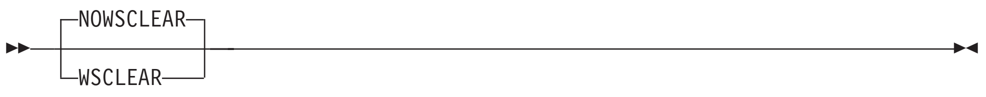
Abbreviations are: None

Use NOVBREF for more efficient compilation.

WSCLEAR

Use WSCLEAR to clear a program's WORKING-STORAGE to binary zeros when the program is initialized. The storage is cleared before any VALUE clauses are applied.

WSCLEAR option syntax



Default is: NOWSCLEAR

Abbreviations are: None

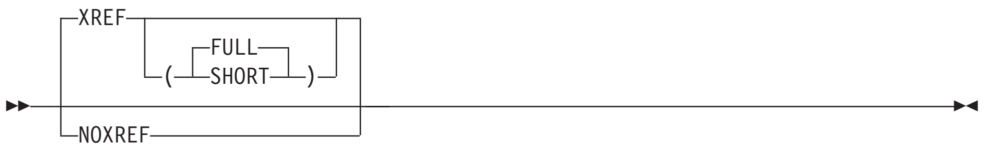
Use NOWSCLEAR to bypass the storage-clearing process.

Performance considerations: If you use WSCLEAR and you are concerned about the size or performance of the object program, then you should also use OPTIMIZE(FULL). Doing so instructs the compiler to eliminate all unreferenced data items from the DATA DIVISION, which will speed up initialization.

XREF

Use XREF to get a sorted cross-reference listing.

XREF option syntax



Default is: XREF(FULL)

Abbreviations are: X|NOX

You can choose XREF, XREF(FULL), or XREF(SHORT). If you specify XREF without any suboptions, XREF(FULL) is in effect.

A section is included in the listing that lists all the program-names, data-names, and procedure-names that are referenced in your program and the line numbers where they are defined. External program-names are identified.

A section is also included that cross-references COPY or BASIS statements in the program with the files from which associated copybooks were obtained.

Names are listed in the order of the collating sequence that is indicated by the locale setting. This order is used whether the names are in single-byte characters or contain multibyte characters (such as DBCS).

If you use XREF and SOURCE, data-name and procedure-name cross-reference information is printed on the same line as the original source. Line-number references or other information appears on the right-hand side of the listing page. On the right of source lines that reference an intrinsic function, the letters IFN are printed with the line number of the locations where the function arguments are defined. Information included in the embedded references lets you know if an identifier is undefined (UND) or defined more than once (DUP), if items are implicitly defined (IMP) (such as special registers or figurative constants), or if a program-name is external (EXT).

If you use XREF and NOSOURCE, you get only the sorted cross-reference listing.

XREF(SHORT) prints only the explicitly referenced data items in the cross-reference listing. XREF(SHORT) applies to multibyte data-names and procedure-names as well as to single-byte names.

NOXREF suppresses this listing.

Usage notes

- Group names used in a MOVE CORRESPONDING statement are in the XREF listing. The elementary names in those groups are also listed.
- In the data-name XREF listing, line numbers that are preceded by the letter M indicate that the data item is explicitly modified by a statement on that line.
- XREF listings take additional storage.

RELATED CONCEPTS
Chapter 17, “Debugging,” on page 307

RELATED TASKS
“Getting listings” on page 316

YEARWINDOW

Use YEARWINDOW to specify the first year of the 100-year window (the *century window*) to be applied to windowed date field processing by the COBOL compiler.

YEARWINDOW option syntax

```
►►──YEARWINDOW(base-year)────────────────────────────────────────►◄
```

Default is: YEARWINDOW(1900)

Abbreviations are: YW

base-year represents the first year of the 100-year window. You must specify it with one of the following values:

- An unsigned decimal number between 1900 and 1999.

  This specifies the starting year of a fixed window. For example, `YEARWINDOW(1930)` indicates a century window of 1930-2029.
- A negative integer from -1 through -99.

  This indicates a sliding window. The first year of the window is calculated by adding the negative integer to the current year. For example, `YEARWINDOW(-80)` indicates that the first year of the century window is 80 years before the year at the time the program is run.

Usage notes

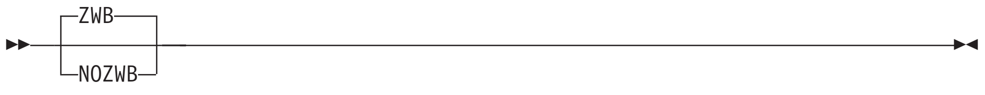
- The `YEARWINDOW` option has no effect unless the `DATEPROC` option is also in effect.
- At run time, two conditions must be true:
  - The century window must have its beginning year in the 1900s.
  - The current year must lie within the century window for the compilation unit.

  For example, if the current year were 2008, the `DATEPROC` option were in effect, and you used the `YEARWINDOW(1900)` option, the program would terminate with an error message.

ZWB

If you compile with `ZWB`, the compiler removes the sign from a signed zoned decimal (`DISPLAY`) field before comparing this field to an alphanumeric elementary field during execution.

ZWB option syntax



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), its use in comparisons is not affected by `ZWB`. Such items always have their sign removed before the comparison is made to an alphanumeric field.

`ZWB` affects how a program runs. The same COBOL source program can give different results, depending on this option setting.

Use `NOZWB` if you want to test input numeric fields for `SPACES`.

Chapter 15. Compiler-directing statements

Several compiler-directing statements and one compiler directive help you to direct the compilation of your program.

These are the compiler-directing statements and directive:

`*CONTROL (*CBL)` statement
: This compiler-directing statement selectively suppresses output or causes output to be produced. The keywords `*CONTROL` and `*CBL` are synonymous.

`>>CALLINTERFACE` directive
: This compiler directive specifies the interface convention for calls, including whether argument descriptors are to be generated. The convention specified with `>>CALLINTERFACE` is in effect until another `>>CALLINTERFACE` specification is made. `>>CALLINT` is an abbreviation for `>>CALLINTERFACE`.

  `>>CALLINTERFACE` can be used only in the `PROCEDURE DIVISION`.

  The syntax and usage of the `>>CALLINTERFACE` directive are similar to that of the `CALLINT` compiler option. Exceptions are:
  - The directive syntax does not include parentheses.
  - The directive can be applied to selected calls as described below.
  - The directive syntax includes the keyword `DESCRIPTOR` and its variants.

  If you specify `>>CALLINT` with no suboptions, the call convention used is determined by the `CALLINT` compiler option.

  `DESCRIPTOR` only: The `>>CALLINT` directive is treated as a comment except for these forms:
  - `>>CALLINT SYSTEM DESCRIPTOR`, or equivalently `>>CALLINT DESCRIPTOR`
  - `>>CALLINT SYSTEM NODESCRIPTOR`, or equivalently `>>CALLINT NODESCRIPTOR`

  These directives turn `DESCRIPTOR` on or off; `SYSTEM` is ignored.

  If you code `OPTLINK` or `CDECL` (valid COBOL for Windows values) instead of `SYSTEM`, the compiler generates an I-level diagnostic, and the entire directive (not just the first keyword) is ignored.

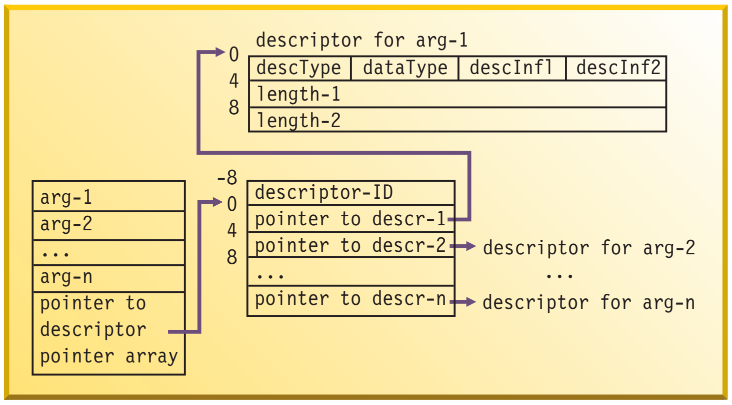
  The `>>CALLINT` directive can be specified anywhere that a COBOL procedure statement can be specified. For example, this is valid syntax:

  ```
  MOVE 3 TO
  >>CALLINTERFACE SYSTEM
  RETURN-CODE.
  ```

  The effect of `>>CALLINT` is limited to the current program. A nested program or a program compiled in the same batch inherits the calling convention specified with the `CALLINT` compiler option, but not a convention specified with the `>>CALLINT` compiler directive.

  If you are writing a routine that is to be called with `>>CALLINT SYSTEM DESCRIPTOR`, this is the argument-passing mechanism:

CALL "PROGRAM1" USING arg-1, arg-2, . . . arg-n



pointer to descr-n
: Points to the descriptor for the specific argument; 0 if no descriptor exists for the argument.

descriptor-ID
: Set to `COBDESC0` to identify this version of the descriptor, allowing for a possible change to the descriptor entry format in the future.

descType
: Set to X'02' (descElmt) for an elementary data item of `USAGE DISPLAY` with `PICTURE X(`*n*`)` or `USAGE DISPLAY-1` with `PICTURE G(`*n*`)` or `N(`*n*`)`. For all others (numeric fields, structures, tables), set to X'00'.

dataType
: Set as follows:

- descType = X'00': dataType = X'00'
- descType = X'02' and the `USAGE` is `DISPLAY`: dataType = X'02' (typeChar)
- descType = X'02' and the `USAGE` is `DISPLAY-1`: dataType = X'09' (typeGChar)

descInf1
: Always set to X'00'.

descInf2
: Set as follows:

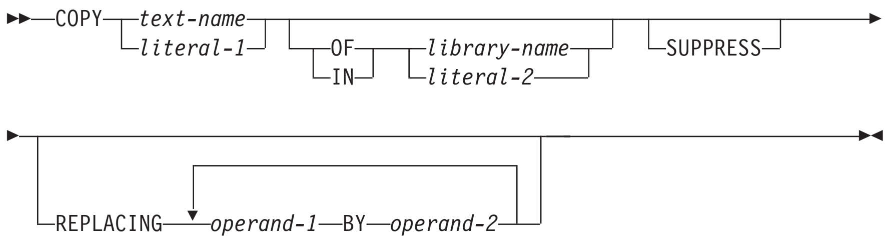
- If descType = X'00'; descInf2 = X'00'
- If descType = X'02':
  - If the `CHAR(EBCDIC)` option is in effect and the argument is not defined with the `NATIVE` option in the `USAGE` clause: descInf2 = X'40'
  - Else: descInf2 = X'00'

length-1
: In the argument descriptor is the length of the argument for a fixed-length argument or the current length for a variable-length item.

length-2
: The maximum length of the argument if the argument is a variable-length item. For a fixed-length argument, length-2 is equal to length-1.

COPY statement

COPY statement syntax



This compiler-directing statement places prewritten text into a COBOL program. You must specify a *text-name* (the name of a copybook) that contains the prewritten text; for example, `COPY` *`my-text`*. You can qualify *text-name* with a *library-name*; for example, `COPY` *`my-text`* `of` *`inventory-lib`*. If *text-name* is not qualified, a *library-name* of `SYSLIB` is assumed. The following affects *library-name* and *text-name*:

library-name

If you specify *library-name* as a literal, the content of the literal is treated as the actual path. If you specify *library-name* as a user-defined word, the name is used as an environment variable and the value of the environment variable is used for the path to locate the copybook. To specify multiple path names, delimit them with a colon (:) or a semicolon (;).

If you do not specify *library-name*, the path used is as described under *text-name*.

text-name

If you specify *text-name* as a user-defined word, processing depends on whether the environment variable that corresponds to *text-name* is set. If the environment variable is set, the value of the environment variable is used as the file-name, and possibly the path name, for the copybook.

A *text-name* is treated as an absolute path if all three of these conditions are met:

- *library-name* is not used.
- *text-name* is a literal or an environment variable.
- The first character is '/'.

For example, this is treated as an absolute path:

```
COPY "/mycpylib/mytext.cpy"
```

If the environment variable that corresponds to *text-name* is not set, the search for the copybook uses the following names:

1. *text-name* with suffix .cpy
2. *text-name* with suffix .cbl
3. *text-name* with suffix .cob
4. *text-name* with no suffix

For example, `COPY MyCopy` searches in the following order:

1. MYCOPY.cpy (in all the specified paths, as described above)
2. MYCOPY.cbl (in all the specified paths, as described above)
3. MYCOPY.cob (in all the specified paths, as described above)
4. MYCOPY (in all the specified paths, as described above)

COBOL defaults *library-name* and *text-name* to uppercase unless the name is contained in a literal ("MyCopy"). In this example, MyCopy is not the same as MYCOPY. If your file name is mixed case (as in MyCopy.cbl), define *text-name* as a literal in the COPY statement.

-I option

For other cases (when neither a *library-name* nor *text-name* indicates the path), the search path is dependent on the -I option.

To have COPY A be equivalent to COPY A OF MYLIB, specify -I$MYLIB.

Based on the above rules, COPY "/X/Y" will be searched in the root directory, and COPY "X/Y" will be searched in the current directory.

COPY A OF SYSLIB is equivalent to COPY A. The -I option does not affect COPY statements with explicit *library-name* qualifications besides those with the library name of SYSLIB.

If both *library-name* and *text-name* are specified, the compiler inserts a path separator (/) between the two values if *library-name* does not end in a /. For example, COPY MYCOPY OF MYLIB with the settings of:

```
export MYCOPY=MYPDS(MYMEMBER)
export MYLIB=MYFILE
```

results in MYFILE/MYPDS(MYMEMBER).

If you specify *text-name* as a user-defined word, you can access local files and also access PDS members on z/OS without changing your mainframe source. For example:

```
COPY mycopybook
```

In this example, when the environment variable *mycopybook* is set to h/mypds(mycopy):

- h is assigned to the specific host.
- mypds is the z/OS PDS data-set name.
- mycopy is the PDS member name.

You can access z/OS files from AIX using NFS (Network File System), which allows z/OS files to be accessed using an AIX path name. However, note that NFS converts the path separator to "." to follow z/OS naming conventions. Keep this in mind when assigning values to your environment variables to ensure proper name formation. For example, these settings:

```
export MYCOPY=(MYMEMBER)
export MYLIB=M/MYFILE/MYPDS
```

do not work because the resulting path is:

```
M/MYFILE/MYPDS/(MYMEMBER)
```

which becomes:

```
M.MYFILE.MYPDS.(MYMEMBER)
```

`PROCESS (CBL)` statement

This compiler-directing statement, which you can place before the `IDENTIFICATION DIVISION` header of an outermost program, specifies compiler options that are to be used during compilation of the program.

RELATED TASKS
"Changing the header of a source listing" on page 6
"Compiling from the command line" on page 219
"Specifying compiler options with the PROCESS (CBL) statement" on page 221

RELATED REFERENCES
"cob2, cob2_j, and cob2_r options" on page 227
CALLINTERFACE directive (*COBOL for AIX Language Reference*)
CBL (PROCESS) statement (*COBOL for AIX Language Reference*)
*CONTROL (*CBL) statement (*COBOL for AIX Language Reference*)
COPY statement (*COBOL for AIX Language Reference*)

Chapter 16. Runtime options

The runtime options shown in the table below are supported.

Table 38. **Runtime options**

Option	Description	Default	Abbreviation
"CHECK"	Flags checking errors	`CHECK(ON)`	`CH`
"DEBUG" on page 302	Specifies whether the COBOL debugging sections specified by the `USE FOR DEBUGGING` declarative are active	`NODEBUG`	None
"ERRCOUNT" on page 302	Specifies how many conditions of severity 1 (W-level) can occur before the run unit terminates abnormally	`ERRCOUNT(20)`	None
"FILEMODE" on page 303	Specifies whether STL and VSAM files created by COBOL for AIX programs have large-format or small-format internal representation, which determines the maximum size limit of STL and VSAM files	`FILEMODE(LARGE)`	`FM(L|S)`
"FILESYS" on page 303	Specifies the file system to use for files for which no explicit file system selections are made, either through `ASSIGN` or an environment variable	`FILESYS(VSA)`	None
"TRAP" on page 304	Indicates whether COBOL intercepts exceptions	`TRAP(ON)`	None
"UPSI" on page 305	Sets the eight `UPSI` switches on or off for applications that use COBOL routines	`UPSI(00000000)`	None

CHECK

`CHECK` causes checking errors to be flagged. In COBOL, index, subscript, and reference-modification ranges can cause checking errors.

CHECK option syntax

```
              ┌─ON──┐
►►──CHECK(──┴─OFF─┴─)──────────────────────────────────────────►◄
```

Default is: `CHECK(ON)`

Abbreviation is: `CH`

`ON` Specifies that runtime checking is performed

`OFF` Specifies that runtime checking is not performed

Usage note: `CHECK(ON)` has no effect if `NOSSRANGE` was in effect during compilation.

Performance consideration: If you compiled a COBOL program with SSRANGE, and you are not testing or debugging an application, performance improves if you specify CHECK(OFF).

DEBUG

DEBUG specifies whether the COBOL debugging sections specified by the USE FOR DEBUGGING declarative are active.

DEBUG option syntax

```
        ┌─NODEBUG─┐
►►──────┼─────────┼──────────────────────────────────────────────►◄
        └─DEBUG───┘
```

Default is: NODEBUG

DEBUG Activates the debugging sections

NODEBUG
Suppresses the debugging sections

Performance consideration: To improve performance, use this option only while debugging.

ERRCOUNT

ERRCOUNT specifies how many severity 1 (W-level) conditions can occur before the run unit terminates abnormally.

ERRCOUNT option syntax

```
►►──ERRCOUNT(─┬────────┬─)───────────────────────────────────────►◄
              └─number─┘
```

Default: ERRCOUNT(20)

number is the number of severity-1 conditions that can occur while this run unit is running. If the number of conditions exceeds *number*, the run unit terminates abnormally.

Any severity-2 (E-level) or higher condition results in termination of the run unit regardless of the value of the ERRCOUNT option.

FILEMODE

FILEMODE specifies whether STL and VSAM files created using OPEN OUTPUT statements will have large-format or small-format metadata (internal control information), which determines their maximum size.

FILEMODE option syntax

```
           ┌─LARGE─┐
►►──FILEMODE(─┴─SMALL─┴─)──────────────────────────────────────►◄
```

Default is: FILEMODE(LARGE)

Abbreviation is: FM(L|S)

LARGE Specifies that OPEN OUTPUT statements for STL or VSAM files create large-format files. These files are essentially unlimited in size.

SMALL Specifies that OPEN OUTPUT statements for STL or VSAM files create small-format files. These files have a maximum size of 2 GB.

You can process large-format and small-format files with either 32-bit or 64-bit COBOL for AIX programs. File processing is not sensitive to the value of the ADDR compiler option.

COBOL for AIX Version 2 programs can create only small-format files. The FILEMODE runtime option is ignored by Version 2 programs even if they are run with a later runtime library.

FILEMODE does not affect the processing of line-sequential files or RSD files. These files do not have a maximum size limit.

RELATED CONCEPTS
"STL and VSAM files that are larger than 2 GB" on page 121

RELATED REFERENCES
"VSAM file system" on page 116
"STL file system" on page 118
"File-system summary" on page 121
"ADDR" on page 246
Compiler limits (*COBOL for AIX Language Reference*)

FILESYS

FILESYS specifies the file system to be used for files for which no explicit file-system selection is made either through an ASSIGN statement or an environment variable. The option applies to sequential, relative, and indexed files.

FILESYS option syntax

```
               ┌─VSA─┐
►►──FILESYS(──┼─STL─┼──)──────────────────────────────────────────────►◄
               └─RSD─┘
```

Default is: `FILESYS(VSA)`

VSA The file system is VSAM. Specify the VSAM file system when you want to use Encina SFS files.

STL The file system is STL.

RSD The file system is RSD.

Only the first three characters of the file-system identifier are used, and the identifier is case insensitive. For example, the following examples are all valid specifications for VSAM:

- `FILESYS(VSA)`
- `FILESYS(vSAM)`
- `FILESYS(vsa)`

RELATED TASKS
"Identifying files" on page 113

RELATED REFERENCES
"Runtime environment variables" on page 211

TRAP

`TRAP` indicates whether COBOL intercepts exceptions.

TRAP option syntax

```
           ┌─ON──┐
►►──TRAP(──┴─OFF─┴──)────────────────────────────────────────────────►◄
```

Default is: `TRAP(ON)`

If `TRAP(OFF)` is in effect, and you do not supply your own trap handler to handle exceptional conditions, the conditions result in a default action by the operating system. For example, if your program attempts to store into an illegal location, the default system action is to issue a message and terminate the process.

ON Activates COBOL interception of exceptions

OFF Deactivates COBOL interception of exceptions

Usage notes

- Use `TRAP(OFF)` only when you need to analyze a program exception before COBOL handles it.
- When you specify `TRAP(OFF)` in a non-CICS environment, no exception handlers are established.
- Running with `TRAP(OFF)` (for exception diagnosis purposes) can cause many side effects, because COBOL requires `TRAP(ON)`. When you run with `TRAP(OFF)`, you can get side effects even if you do not encounter a software-raised condition, program check, or abend. If you do encounter a program check or an abend with `TRAP(OFF)` in effect, the following side effects can occur:
  - Resources obtained by COBOL are not freed.
  - Files opened by COBOL are not closed, so records might be lost.
  - No messages or dump output are generated.

  The run unit terminates abnormally if such conditions are raised.

UPSI

`UPSI` sets the eight UPSI switches on or off for applications that use COBOL routines.

UPSI option syntax

```
►►──UPSI(─┬──────────┬─)──────────────────────────────────────────►◄
          └─nnnnnnnn─┘
```

Default is: `UPSI(00000000)`

Each *n* represents one UPSI switch (between 0 and 7); the leftmost *n* represents the first switch. Each *n* can be either 0 (off) or 1 (on).

Chapter 17. Debugging

You can choose from two approaches to determine the cause of problems in program behavior of your application: source-language debugging or interactive debugging.

For source-language debugging, COBOL provides several language elements, compiler options, and listing outputs that make debugging easier.

For interactive debugging, you can use the Debugger for AIX.

RELATED TASKS
"Debugging with source language"
"Debugging using compiler options" on page 311
"Using the debugger" on page 316
"Getting listings" on page 316
"Debugging with messages that have offset information" on page 327
"Debugging user exits" on page 328
"Debugging assembler routines" on page 328

Debugging with source language

You can use several COBOL language features to pinpoint the cause of a failure in a program.

If a failing program is part of a large application that is already in production (precluding source updates), write a small test case to simulate the failing part of the program. Code debugging features in the test case to help detect these problems:
- Errors in program logic
- Input-output errors
- Mismatches of data types
- Uninitialized data
- Problems with procedures

RELATED TASKS
"Tracing program logic"
"Finding and handling input-output errors" on page 308
"Validating data" on page 309
"Finding uninitialized data" on page 309
"Generating information about procedures" on page 309

RELATED REFERENCES
Source language debugging (*COBOL for AIX Language Reference*)

Tracing program logic

Trace the logic of your program by adding `DISPLAY` statements.

For example, if you determine that the problem is in an `EVALUATE` statement or in a set of nested `IF` statements, use `DISPLAY` statements in each path to see the logic flow. If you determine that the calculation of a numeric value is causing the problem, use DISPLAY statements to check the value of some interim results.

If you use explicit scope terminators to end statements in your program, the logic is more apparent and therefore easier to trace.

To determine whether a particular routine started and finished, you might insert code like this into your program:

```
DISPLAY "ENTER CHECK PROCEDURE"
    .
    .  (checking procedure routine)
    .
DISPLAY "FINISHED CHECK PROCEDURE"
```

After you are sure that the routine works correctly, disable the DISPLAY statements in one of two ways:

- Put an asterisk in column 7 of each DISPLAY statement line to convert it to a comment line.
- Put a D in column 7 of each DISPLAY statement to convert it to a comment line. When you want to reactivate these statements, include a WITH DEBUGGING MODE clause in the ENVIRONMENT DIVISION; the D in column 7 is ignored and the DISPLAY statements are implemented.

Before you put the program into production, delete or disable the debugging aids you used and recompile the program. The program will run more efficiently and use less storage.

RELATED CONCEPTS
"Scope terminators" on page 19

RELATED REFERENCES
DISPLAY statement (*COBOL for AIX 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 160
"Using file status keys" on page 161

RELATED REFERENCES
Status key (*COBOL for AIX Language Reference*)

Validating data

If you suspect that your program is trying to perform arithmetic on nonnumeric data or is receiving the wrong type of data on an input record, use the class test (the class condition) to validate the type of data.

You can use the class test to check whether the content of a data item is
| `ALPHABETIC`, `ALPHABETIC-LOWER`, `ALPHABETIC-UPPER`, `DBCS`, `KANJI`, or `NUMERIC`. If the
| data item is described implicitly or explicitly as `USAGE NATIONAL`, the class test
| checks the national character representation of the characters associated with the
| specified character class.

RELATED TASKS
"Coding conditional expressions" on page 86
"Testing for valid DBCS characters" on page 191

RELATED REFERENCES
Class condition (*COBOL for AIX Language Reference*)

Finding uninitialized data

Use an `INITIALIZE` or `SET` statement to initialize a table or data item when you suspect that a problem might be caused by residual data in those fields.

If the problem happens intermittently and not always with the same data, it could be that a switch was not initialized but is generally set to the right value (0 or 1) by chance. By using a `SET` statement to ensure that the switch is initialized, you can determine that the uninitialized switch is the cause of the problem or remove it as a possible cause.

RELATED REFERENCES
INITIALIZE statement (*COBOL for AIX Language Reference*)
SET statement (*COBOL for AIX Language Reference*)

Generating information about procedures

Generate information about your program or test case and how it is running by coding the `USE FOR DEBUGGING` declarative. This declarative lets you include statements in the program and indicate when they should be performed when you run your program.

For example, to determine how many times a procedure is run, you could include a debugging procedure in the `USE FOR DEBUGGING` declarative and use a counter to keep track of the number of times that control passes to that procedure. You can use the counter technique to check items such as these:

- How many times a `PERFORM` statement runs, and thus whether a particular routine is being used and whether the control structure is correct
- How many times a loop routine runs, and thus whether the loop is executing and whether the number for the loop is accurate

You can use debugging lines or debugging statements or both in your program.

Debugging lines are statements that are identified by a D in column 7. To make debugging lines in your program active, code the WITH DEBUGGING MODE clause on the SOURCE-COMPUTER line in the ENVIRONMENT DIVISION. Otherwise debugging lines are treated as comments.

Debugging statements are the statements that are coded in the DECLARATIVES section of the PROCEDURE DIVISION. Code each USE FOR DEBUGGING declarative in a separate section. Code the debugging statements as follows:

- Only in a DECLARATIVES section.
- Following the header USE FOR DEBUGGING.
- Only in the outermost program; they are not valid in nested programs. Debugging statements are also never triggered by procedures that are contained in nested programs.

To use debugging statements in your program, you must include the WITH DEBUGGING MODE clause and use the DEBUG runtime option.

Options restrictions:

- You cannot use the USE FOR DEBUGGING declarative in a program that you compile with the THREAD option.
- | USE FOR DEBUGGING declaratives, if the WITH DEBUGGING MODE clause has been specified, are mutually exclusive with the TEST compiler option. If USE FOR DEBUGGING declaratives and the WITH DEBUGGING MODE clause are present, the TEST option is cancelled.

"Example: USE FOR DEBUGGING"

RELATED REFERENCES
SOURCE-COMPUTER paragraph (*COBOL for AIX Language Reference*)
Debugging lines (*COBOL for AIX Language Reference*)
Debugging sections (*COBOL for AIX Language Reference*)
DEBUGGING declarative (*COBOL for AIX Language Reference*)

Example: USE FOR DEBUGGING

This example shows the kind of statements that are needed to use a DISPLAY statement and a USE FOR DEBUGGING declarative to test a program.

The DISPLAY statement writes information to the terminal or to an output file. The USE FOR DEBUGGING declarative is used with a counter to show how many times a routine runs.

```
Environment Division.
. . .
Data Division.
. . .
Working-Storage Section.
. . . (other entries your program needs)
01  Trace-Msg    PIC X(30) Value "  Trace for Procedure-Name : ".
01  Total        PIC 9(9)  Value 1.
. . .
Procedure Division.
Declaratives.
Debug-Declaratives Section.
    Use For Debugging On Some-Routine.
Debug-Declaratives-Paragraph.
    Display Trace-Msg, Debug-Name, Total.
End Declaratives.
```

```
Main-Program Section.
    . . . (source program statements)
    Perform Some-Routine.
    . . . (source program statements)
    Stop Run.
Some-Routine.
    . . . (whatever statements you need in this paragraph)
    Add 1 To Total.
Some-Routine-End.
```

The DISPLAY statement in the DECLARATIVES SECTION issues this message every time the procedure Some-Routine runs:

```
  Trace For Procedure-Name : Some-Routine 22
```

The number at the end of the message, 22, is the value accumulated in the data item Total; it indicates the number of times Some-Routine has run. The statements in the debugging declarative are performed before the named procedure runs.

You can also use the DISPLAY statement to trace program execution and show the flow through the program. You do this by dropping Total from the DISPLAY statement and changing the USE FOR DEBUGGING declarative in the DECLARATIVES SECTION to:

```
USE FOR DEBUGGING ON ALL PROCEDURES.
```

As a result, a message is displayed before each nondebugging procedure in the outermost program runs.

Debugging using compiler options

You can use certain compiler options to help you find errors in your program, find various elements in your program, obtain listings, and prepare your program for debugging.

You can find the following errors by using compiler options (the options are shown in parentheses):

- Syntax errors such as duplicate data-names (NOCOMPILE)
- Missing sections (SEQUENCE)
- Invalid subscript values (SSRANGE)

You can find the following elements in your program by using compiler options:

- Error messages and locations of the associated errors (FLAG)
- Program entity definitions and references (XREF)
- Data items in the DATA DIVISION (MAP)
- Verb references (VBREF)

You can get a copy of your source (SOURCE) or a listing of generated code (LIST).

You prepare your program for debugging by using the TEST compiler option.

RELATED TASKS
"Finding coding errors" on page 312
"Finding line sequence problems" on page 312
"Checking for valid ranges" on page 312
"Selecting the level of error to be diagnosed" on page 313

"Finding program entity definitions and references" on page 315
"Listing data items" on page 316
"Getting listings" on page 316

RELATED REFERENCES
Chapter 14, "Compiler options," on page 243

Finding coding errors

Use the NOCOMPILE option to compile conditionally or to only check syntax. When used with the SOURCE option, NOCOMPILE produces a listing that will help you find coding mistakes such as missing definitions, improperly defined data items, and duplicate data-names.

Checking syntax only: To only check the syntax of your program, and not produce object code, use NOCOMPILE without a suboption. If you also specify the SOURCE option, the compiler produces a listing.

When you specify NOCOMPILE, several compiler options are suppressed. See the related reference below about the COMPILE option for details.

Compiling conditionally: To compile conditionally, use NOCOMPILE(*x*), where *x* is one of the severity levels of errors. Your program is compiled if all the errors are of a lower severity than *x*. The severity levels that you can use, from highest to lowest, are S (severe), E (error), and W (warning).

If an error of level *x* or higher occurs, the compilation stops and your program is only checked for syntax.

RELATED REFERENCES
"COMPILE" on page 255

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 280

Checking for valid ranges

Use the SSRANGE compiler option to check whether addresses fall within proper ranges.

SSRANGE causes the following addresses to be checked:

- Subscripted or indexed data references: Is the effective address of the desired element within the maximum boundary of the specified table?

- Variable-length data references (a reference to a data item that contains an `OCCURS DEPENDING ON` clause): Is the actual length positive and within the maximum defined length for the group data item?
- Reference-modified data references: Are the offset and length positive? Is the sum of the offset and length within the maximum length for the data item?

If the `SSRANGE` option is in effect, checking is performed at run time if both of the following conditions are true:
- The COBOL statement that contains the indexed, subscripted, variable-length, or reference-modified data item is performed.
- The `CHECK` runtime option is `ON`.

If an address is generated outside the range of the data item that contains the referenced data, an error message is generated and the program stops. The message identifies the table or identifier that was referenced and the line number where the error occurred. Additional information is provided depending on the type of reference that caused the error.

If all subscripts, indices, and reference modifiers in a given data reference are literals and they result in a reference outside the data item, the error is diagnosed at compile time regardless of the setting of the `SSRANGE` option.

Performance consideration: `SSRANGE` can somewhat degrade performance because of the extra overhead to check each subscripted or indexed item.

RELATED REFERENCES
"SSRANGE" on page 285
"Performance-related compiler options" on page 552

Selecting the level of error to be diagnosed

Use the `FLAG` compiler option to specify the level of error to be diagnosed during compilation and to indicate whether error messages are to be embedded in the listing. Use `FLAG(I)` or `FLAG(I,I)` to be notified of all errors.

Specify as the first parameter the lowest severity level of the syntax-error messages to be issued. Optionally specify the second parameter as the lowest level of the syntax-error messages to be embedded in the source listing. This severity level must be the same or higher than the level for the first parameter. If you specify both parameters, you must also specify the `SOURCE` compiler option.

Table 39. **Severity levels of compiler messages**

Severity level	Resulting messages
U (unrecoverable)	U messages only
S (severe)	All S and U messages
E (error)	All E, S, and U messages
W (warning)	All W, E, S, and U messages
I (informational)	All messages

When you specify the second parameter, each syntax-error message (except a U-level message) is embedded in the source listing at the point where the compiler had enough information to detect that error. All embedded messages (except those issued by the library compiler phase) directly follow the statement to which they refer. The number of the statement that had the error is also included with the message. Embedded messages are repeated with the rest of the diagnostic messages at the end of the source listing.

When you specify the NOSOURCE compiler option, the syntax-error messages are included only at the end of the listing. Messages for unrecoverable errors are not embedded in the source listing, because an error of this severity terminates the compilation.

"Example: embedded messages"

RELATED TASKS
"Generating a list of compiler error messages" on page 225

RELATED REFERENCES
"Severity codes for compiler error messages" on page 225
"Messages and listings for compiler-detected errors" on page 225
"FLAG" on page 264

Example: embedded messages

The following example shows the embedded messages generated by specifying a second parameter to the FLAG option. Some messages in the summary apply to more than one COBOL statement.

```
  LineID  PL SL  ----+-*A-1-B--+----2----+----3----+----4----+----5----+----6----+----7--+--8 Map and Cross Reference
...
  000977              /
  000978              ******************************************************************
  000979              ***         I N I T I A L I Z E   P A R A G R A P H          **
  000980              ***  Open files. Accept date, time and format header lines.    **
  000981        IA4690***   Load location-table.                                   **
  000982              ******************************************************************
  000983                 100-initialize-paragraph.
  000984                   move spaces to ws-transaction-record                              IMP 339
  000985                   move spaces to ws-commuter-record                                 IMP 315
  000986                   move zeroes to commuter-zipcode                                   IMP 326
  000987                   move zeroes to commuter-home-phone                                IMP 327
  000988                   move zeroes to commuter-work-phone                                IMP 328
  000989                   move zeroes to commuter-update-date                               IMP 332
  000990                   open input update-transaction-file                                203
  ==000990==> IGYPS2052-S An error was found in the definition of file "LOCATION-FILE".  The
                     reference to this file was discarded.
    000991                       location-file                                                 192
  000992                         i-o commuter-file                                           180
  000993                         output print-file                                           216
  000994                   if loccode-file-status not = "00" or                              248
  000995                      update-file-status not = "00" or                               247
  000996                      updprint-file-status not = "00"                                249
  000997      1               display "Open Error ..."
  000998      1               display "  Location File Status = " loccode-file-status        248
  000999      1               display "  Update   File Status = " update-file-status         247
  001000      1               display "  Print    File Status = " updprint-file-status       249
  001001      1              perform 900-abnormal-termination                                1433
  001002                   end-if
  001003        IA4760     if commuter-file-status not = "00" and not = "97"                 240
  001004      1              display "100-OPEN"
  001005      1              move 100 to comp-code                                           230
  001006      1              perform 500-vsam-error                                          1387
  001007      1              display "Commuter File Status (OPEN) = "
  001008      1                       commuter-file-status                                   240
  001009      1              perform 900-abnormal-termination                                1433
  001010        IA4790     end-if
  001011                   accept ws-date from date                                          UND
  ==001011==> IGYPS2121-S "WS-DATE" was not defined as a data-name.  The statement was discarded.
    001012        IA4810     move corr ws-date to header-date                                  UND 463
  ==001012==> IGYPS2121-S "WS-DATE" was not defined as a data-name.  The statement was discarded.
    001013                   accept ws-time from time                                          UND
  ==001013==> IGYPS2121-S "WS-TIME" was not defined as a data-name.  The statement was discarded.
    001014         IA4830     move corr ws-time to header-time                                       UND 457
  ==001014==> IGYPS2121-S "WS-TIME" was not defined as a data-name.  The statement was discarded.
    001015         IA4840     read location-file                                                     192
...
LineID  Message code  Message text
     192  IGYDS1050-E   File "LOCATION-FILE" contained no data record descriptions.
                        The file definition was discarded.
     899  IGYPS2052-S   An error was found in the definition of file "LOCATION-FILE".
                        The reference to this file was discarded.
                        Same message on line:    990
    1011  IGYPS2121-S   "WS-DATE" was not defined as a data-name.  The statement was discarded.
                        Same message on line:   1012
    1013  IGYPS2121-S   "WS-TIME" was not defined as a data-name.  The statement was discarded.
                        Same message on line:   1014
    1015  IGYPS2053-S   An error was found in the definition of file "LOCATION-FILE".
                        This input/output statement was discarded.
                        Same message on line:   1027
    1026  IGYPS2121-S   "LOC-CODE" was not defined as a data-name.  The statement was discarded.
    1209  IGYPS2121-S   "COMMUTER-SHIFT" was not defined as a data-name.  The statement was discarded.
                        Same message on line:   1230
    1210  IGYPS2121-S   "COMMUTER-HOME-CODE" was not defined as a data-name.  The statement was discarded.
                        Same message on line:   1231
    1212  IGYPS2121-S   "COMMUTER-NAME" was not defined as a data-name.  The statement was discarded.
                        Same message on line:   1233
    1213  IGYPS2121-S   "COMMUTER-INITIALS" was not defined as a data-name.  The statement was discarded.
                        Same message on line:   1234
    1223  IGYPS2121-S   "WS-NUMERIC-DATE" was not defined as a data-name.  The statement was discarded.
Messages    Total    Informational    Warning    Error    Severe    Terminating
Printed:      19                                   1        18
* Statistics for COBOL program FLAGOUT:
*    Source records = 1755
*    Data Division statements = 279
*    Procedure Division statements = 479
Locale = en_US.ISO8859-1                   (1)
End of compilation 1,  program FLAGOUT,  highest severity: Severe.
Return code 12
```

(1) The locale that the compiler used

Finding program entity definitions and references

Use the XREF(FULL) compiler option to find out where a data-name, procedure-name, or program-name is defined and referenced. Use it also to produce a cross-reference of COPY or BASIS statements to the files from which copybooks were obtained.

A sorted cross-reference includes the line number where the data-name, procedure-name, or program-name was defined and the line numbers of all references to it.

To include only the explicitly referenced data items, use the XREF(SHORT) option.

Use both the XREF (either FULL or SHORT) and the SOURCE options to print a modified cross-reference to the right of the source listing. This embedded cross-reference shows the line number where the data-name or procedure-name was defined.

For further details, see the related reference below about the XREF compiler option.

“Example: XREF output: data-name cross-references” on page 324
“Example: XREF output: program-name cross-references” on page 325
“Example: XREF output: COPY/BASIS cross-references” on page 325
“Example: embedded cross-reference” on page 326

RELATED TASKS
“Getting listings” on page 316

RELATED REFERENCES
"XREF" on page 291

Listing data items

Use the MAP compiler option to produce a listing of the DATA DIVISION items and all implicitly declared items.

When you use the MAP option, an embedded MAP summary that contains condensed MAP information is generated to the right of the COBOL source data declaration. When both XREF data and an embedded MAP summary are on the same line, the embedded summary is printed first.

You can select or inhibit parts of the MAP listing and embedded MAP summary by using *CONTROL MAP|NOMAP (or *CBL MAP|NOMAP) statements throughout the source. For example:

```
*CONTROL NOMAP
   01  A
   02  B
*CONTROL MAP
```

"Example: MAP output" on page 320

RELATED TASKS
"Getting listings"

RELATED REFERENCES
"MAP" on page 270

Using the debugger

COBOL for AIX provides a debugger. Use the TEST compiler option to prepare your COBOL program so that you can step through the executable program with the debugger.

Alternatively, you can use the -g option of the cob2 command (or one of its variants) to prepare your program to use the debugger.

To invoke the debugger to examine an executable file, issue the command `idebug` *filename*, where *filename* is the name of your program. If the program was compiled with the -g option, the debugger will show symbolic and line-number information.

RELATED TASKS
"Compiling from the command line" on page 219

RELATED REFERENCES
"cob2, cob2_j, and cob2_r options" on page 227
"TEST" on page 286

Getting listings

Get the information that you need for debugging by requesting the appropriate compiler listing with the use of compiler options.

Attention: The listings produced by the compiler are not a programming interface and are subject to change.

Table 40. **Using compiler options to get listings**

Use	Listing	Contents	Compiler option
To check a list of the options in effect for the program, statistics about the content of the program, and diagnostic messages about the compilation To check the locale in effect during compilation	Short listing	• List of options in effect for the program • Statistics about the content of the program • Diagnostic messages about the compilation[1] Locale line that shows the locale in effect	`NOSOURCE`, `NOXREF`, `NOVBREF`, `NOMAP`, `NOLIST`
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 282
To find certain data items; to see the final storage allocation after reentrancy or optimization has been accounted for; to see where programs are defined and check their attributes	Map of `DATA DIVISION` items	All `DATA DIVISION` items and all implicitly declared items Embedded map summary (in the right margin of the listing for lines in the `DATA DIVISION` that contain data declarations) Nested program map (if the program contains nested programs)	"MAP" on page 270[2]
To find where a name is defined, referenced, or modified; to determine the context (such as whether a verb was used in a `PERFORM` block) in which a procedure is referenced; to determine the file from which a copybook was obtained	Sorted cross-reference listing of names; sorted cross-reference listing of `COPY/BASIS` statements and copybook files	Data-names, procedure-names, and program-names; references to these names `COPY/BASIS` text-names and library names, and the files from which associated copybooks were obtained Embedded modified cross-reference provides line numbers where data-names and procedure-names were defined	"XREF" on page 291[2,3]
To find the failing verb in a program or the address in storage of a data item that was moved during the program	`PROCEDURE DIVISION` code and assembler code produced by the compiler[3]	Generated code	"LIST" on page 269[2,4]
To find an instance of a certain verb	Alphabetic listing of verbs	Each verb used, number of times each verb was used, line numbers where each verb was used	"VBREF" on page 290

Table 40. **Using compiler options to get listings** *(continued)*

Use	Listing	Contents	Compiler option

1. To eliminate messages, turn off the options (such as FLAG) that govern the level of compile diagnostic information.
2. To use your line numbers in the compiled program, use the NUMBER compiler option. The compiler checks the sequence of your source statement line numbers in columns 1 through 6 as the statements are read in. When it finds a line number out of sequence, the compiler assigns to it a number with a value one higher than the line number of the preceding statement. The new value is flagged with two asterisks. A diagnostic message indicating an out-of-sequence error is included in the compilation listing.
3. The context of the procedure reference is indicated by the characters preceding the line number.
4. The assembler listing is written to the listing file (a file that has the same name as the source program but with the suffix .wlist).

"Example: short listing"
"Example: SOURCE and NUMBER output" on page 320
"Example: MAP output" on page 320
"Example: embedded map summary" on page 321
"Example: nested program map" on page 323
"Example: XREF output: data-name cross-references" on page 324
"Example: XREF output: program-name cross-references" on page 325
"Example: XREF output: COPY/BASIS cross-references" on page 325
"Example: embedded cross-reference" on page 326
"Example: VBREF compiler output" on page 326

RELATED TASKS
"Generating a list of compiler error messages" on page 225

RELATED REFERENCES
"Messages and listings for compiler-detected errors" on page 225

Example: short listing

The parenthetical numbers shown in the listing below correspond to numbered explanations that follow the listing. For illustrative purposes, some errors that cause diagnostic messages were deliberately introduced.

```
PROCESS(CBL) statements:   (1)
 CBL    NOSOURCE,NOXREF,NOVBREF,NOMAP,NOLIST    (2)
Options in effect:    (3)
 NOADATA
   ADDR(32)
   QUOTE
   ARITH(COMPAT)
   CALLINT(NODESCRIPTOR)
   CHAR(NATIVE)
 NOCICS
   COLLSEQ(BINARY)
 NOCOMPILE(S)
 NOCURRENCY
 NODATEPROC
 NODIAGTRUNC
 NODYNAM
 NOEXIT
   FLAG(I,I)
 NOFLAGSTD
   FLOAT(NATIVE)
 NOLIB
   LINECOUNT(60)
 NOLIST
   LSTFILE(LOCALE)
 NOMAP
   MAXMEM(2048K)
 NOMDECK
   NCOLLSEQ(BINARY)
   NSYMBOL(NATIONAL)
 NONUMBER
 NOOPTIMIZE
   PGMNAME(LONGUPPER)
 NOPROFILE
   SEPOBJ
   SEQUENCE
   SIZE(8388608)
 NOSOSI
 NOSOURCE
   SPACE(1)
   SPILL(512)
 NOSQL
 NOSSRANGE
   TERM
 NOTEST
 NOTHREAD
   TRUNC(STD)
 NOVBREF
 NOWSCLEAR
 NOXREF
   YEARWINDOW(1900)
   ZWB
LineID  Message code  Message text    (4)
        IGYDS0139-W   Diagnostic messages were issued during processing of compiler options.  These messages are
                    located at the beginning of the listing.
   193  IGYDS1050-E   File "LOCATION-FILE" contained no data record descriptions.  The file definition was discarded.
   889  IGYPS2052-S   An error was found in the definition of file "LOCATION-FILE".  The reference to this file
                    was discarded.
                      Same message on line:    983
   993  IGYPS2121-S   "WS-DATE" was not defined as a data-name.  The statement was discarded.
                      Same message on line:    994
   995  IGYPS2121-S   "WS-TIME" was not defined as a data-name.  The statement was discarded.
                      Same message on line:    996
   997  IGYPS2053-S   An error was found in the definition of file "LOCATION-FILE".  This input/output statement
                    was discarded.
                      Same message on line:   1009
  1008  IGYPS2121-S   "LOC-CODE" was not defined as a data-name.  The statement was discarded.
  1219  IGYPS2121-S   "COMMUTER-SHIFT" was not defined as a data-name.  The statement was discarded.
                      Same message on line:   1240
  1220  IGYPS2121-S   "COMMUTER-HOME-CODE" was not defined as a data-name.  The statement was discarded.
                      Same message on line:   1241
  1222  IGYPS2121-S   "COMMUTER-NAME" was not defined as a data-name.  The statement was discarded.
                      Same message on line:   1243
  1223  IGYPS2121-S   "COMMUTER-INITIALS" was not defined as a data-name.  The statement was discarded.
                      Same message on line:   1244
  1233  IGYPS2121-S   "WS-NUMERIC-DATE" was not defined as a data-name.  The statement was discarded.
Messages    Total    Informational    Warning    Error   Severe   Terminating    (5)
Printed:      21                         2         1       18
  * Statistics for COBOL program SLISTING:    (6)
* Source records = 1765
* Data Division statements = 277
* Procedure Division statements = 513
Locale = en_US.ISO8859-1                    (7)
End of compilation 1,  program SLISTING,  highest severity: Severe.   (8)
Return code 12
```

(1) Message about options specified in a PROCESS (or CBL) statement. This message does not appear if no options were specified.

(2) Options coded in the PROCESS (or CBL) statement.

(3) Status of options at the start of this compilation.

(4) Program diagnostics. The first message refers you to the library phase diagnostics, if there were any. Diagnostics for the library phase are always presented at the beginning of the listing.

(5) Count of diagnostic messages in this program, grouped by severity level.

(6) Program statistics for the program SLISTING.

(7) The locale that the compiler used.

(8) Program statistics for the compilation unit. When you perform a batch compilation (multiple outermost COBOL programs in a single compilation), the return code is the highest message severity level for the entire compilation.

Example: SOURCE and NUMBER output

In the portion of the listing shown below, the programmer numbered two of the statements out of sequence. The note numbers in the listing correspond to numbered explanations that follow the listing.

```
                     (1)
LineID  PL SL  ----+-*A-1-B--+----2----+----3----+----4----+----5----+----6----+----7-|--+----8  Cross-Reference
  (2)    (3)    (4)
               087000/****************************************************************
               087100***                 D O   M A I N   L O G I C                  **
               087200***                                                            **
               087300*** Initialization. Read and process update transactions until **
               087400*** EOE. Close files and stop run.                             **
               087500*****************************************************************
               087600 procedure division.
               087700   000-do-main-logic.
               087800     display "PROGRAM SRCOUT - Beginning"
               087900     perform 050-create-vsam-master-file.
               088150     display "perform 050-create-vsam-master finished".
088151**       088125     perform 100-initialize-paragraph
               088200     display "perform 100-initialize-paragraph finished"
               088300     read update-transaction-file into ws-transaction-record
               088400         at end
            1  088500       set transaction-eof to true
               088600     end-read
               088700     display "READ completed"
               088800     perform until transaction-eof
            1  088900       display "inside perform until loop"
            1  089000       perform 200-edit-update-transaction
            1  089100       display "After perform 200-edit   "
            1  089200       if no-errors
            2  089300         perform 300-update-commuter-record
            2  089400         display "After perform 300-update "
            1  089650       else
089651**    2  089600         perform 400-print-transaction-errors
            2  089700         display "After perform 400-errors "
            1  089800       end-if
            1  089900       perform 410-re-initialize-fields
            1  090000       display "After perform 410-reinitialize"
            1  090100       read update-transaction-file into ws-transaction-record
            1  090200           at end
            2  090300         set transaction-eof to true
            1  090400       end-read
            1  090500       display "After '2nd READ'   "
               090600     end-perform
```

(1) Scale line labels Area A, Area B, and source-code column numbers

(2) Source-code line number assigned by the compiler

(3) Program (PL) and statement (SL) nesting level

(4) Columns 1 through 6 of program (the sequence number area)

Example: MAP output

The following example shows output from the `MAP` option. The numbers used in the explanation below correspond to the numbers that annotate the output.

```
Data Division Map
(1)
Data Definition Attribute codes (rightmost column) have the following meanings:
    D = Object of OCCURS DEPENDING    G = GLOBAL                              LSEQ= ORGANIZATION LINE SEQUENTIAL
    E = EXTERNAL                      O = Has OCCURS clause                   SEQ= ORGANIZATION SEQUENTIAL
    VLO=Variably Located Origin       OG= Group has own length definition     INDX= ORGANIZATION INDEXED
    VL= Variably Located              R = REDEFINES                           REL= ORGANIZATION RELATIVE

(2)      (3) (4)                                             (5)      (6)         (7)           (8)
Source   Hierarchy and                                                                         Data Def
LineID   Data Name                                         Length(Displacement) Data Type      Attributes
     4  PROGRAM-ID IGYTCARA------------------------------------------------------------------------------------*
   180   FD COMMUTER-FILE                                                      File            INDX
   182   1  COMMUTER-RECORD                                       80           Group
   183     2  COMMUTER-KEY                                        16(0000000)  Display
   184     2  FILLER                                              64(0000016)  Display
   186   FD COMMUTER-FILE-MST                                                  File            INDX
   188   1  COMMUTER-RECORD-MST                                   80           Group
   189     2  COMMUTER-KEY-MST                                    16(0000000)  Display
   190     2  FILLER                                              64(0000016)  Display
   192   FD LOCATION-FILE                                                      File            SEQ
   203   FD UPDATE-TRANSACTION-FILE                                            File            SEQ
   208   1  UPDATE-TRANSACTION-RECORD                             80           Display
   216   FD PRINT-FILE                                                 File           SEQ
   221   1  PRINT-RECORD                                     121          Display
   228   1  WORKING-STORAGE-FOR-IGYCARA                        1          Display
```

(1) Explanations of the data definition attribute codes.

(2) Source line number where the data item was defined.

(3) Level definition or number. The compiler generates this number in the following way:

- First level of any hierarchy is always 01. Increase 1 for each level (any item you coded as level 02 through 49).
- Level-numbers 66, 77, and 88, and the indicators `FD` and `SD`, are not changed.

(4) Data-name that is used in the source module in source order.

(5) Length of data item. Base locator value.

(6) Hexadecimal displacement from the beginning of the containing structure.

(7) Data type and usage.

(8) Data definition attribute codes. The definitions are explained at the top of the `DATA DIVISION` map.

"Example: embedded map summary"
"Example: nested program map" on page 323

RELATED REFERENCES
"Terms and symbols used in MAP output" on page 322

Example: embedded map summary

The following example shows an embedded map summary from specifying the `MAP` option. The summary appears in the right margin of the listing for lines in the `DATA DIVISION` that contain data declarations.

```
000002                Identification Division.
000003
000004                Program-id.    EMBMAP.
   . . .
000176                Data division.
000177                File section.
000178
000179
000180                FD  COMMUTER-FILE
000181                    record 80 characters.                                                 (1)   (2)
000182                  01 commuter-record.                                                      80
000183                    05 commuter-key               PIC x(16).                               16(0000000)
000184                    05 filler                     PIC x(64).                               64(0000016)
   . . .
000221         IA1620   01 print-record                 pic x(121).                             121
   . . .
000227                Working-storage section.
000228                  01 Working-storage-for-EMBMAP     pic x.                                   1
000229
000230                  77 comp-code                    pic S9999 comp.                          2
000231                  77 ws-type                      pic x(3)   value spaces.                 3
000232
000233
000234                  01 i-f-status-area.                                                      2
000235                    05 i-f-file-status            pic x(2).                                2(0000000)
000236                      88 i-o-successful           value zeroes.                   IMP
000237
000238
000239                  01 status-area.                                                          8
000240                    05 commuter-file-status       pic x(2).                     (3)       2(0000000)
000241                      88 i-o-okay                 value zeroes.                   IMP
000242                    05 commuter-vsam-status.                                               6(0000002)
000243                      10 vsam-r15-return-code     pic 9(2) comp.                           2(0000002)
000244                      10 vsam-function-code       pic 9(1) comp.                           2(0000004)
000245                      10 vsam-feedback-code       pic 9(3) comp.                           2(0000006)
000246
000247                  77 update-file-status           pic xx.                                  2
000248                  77 loccode-file-status          pic xx.                                  2
```

```
000249                77 updprint-file-status          pic xx.                               2
    . . .
000877              procedure division.
000878                000-do-main-logic.
000879                  display "PROGRAM EMBMAP - Beginning".
000880                  perform 050-create-vsam-master-file.                          931
    . . .
```

(1) Decimal length of data item

(2) Hexadecimal displacement from the beginning of the base locator value

(3) Special definition symbols:

UND The user name is undefined.

DUP The user name is defined more than once.

IMP An implicitly defined name, such as special registers or figurative constants.

IFN An intrinsic function reference.

EXT An external reference.

* The program-name is unresolved because the NOCOMPILE option is in effect.

Terms and symbols used in MAP output

The following table describes the terms and symbols used in the listings produced by the MAP compiler option.

Table 41. **Terms and symbols used in MAP output**

Term	Description
ALPHABETIC	Alphabetic (PICTURE A)
ALPHA-EDIT	Alphabetic-edited
AN-EDIT	Alphanumeric-edited
BINARY	Binary (USAGE BINARY, COMPUTATIONAL, or COMPUTATIONAL-5)
COMP-1	Single-precision internal floating point (USAGE COMPUTATIONAL-1)
COMP-2	Double-precision internal floating point (USAGE COMPUTATIONAL-2)
DBCS	DBCS (USAGE DISPLAY-1)
DBCS-EDIT	DBCS edited
DISP-FLOAT	Display floating point (USAGE DISPLAY)
DISPLAY	Alphanumeric (PICTURE X)
DISP-NUM	Zoned decimal (USAGE DISPLAY)
DISP-NUM-EDIT	Numeric-edited (USAGE DISPLAY)
FD	File definition
FUNCTION-PTR	Pointer to an externally callable function (USAGE FUNCTION-POINTER)
GROUP	Alphanumeric fixed-length group
GRP-VARLEN	Alphanumeric variable-length group
INDEX	Index (USAGE INDEX)
INDEX-NAME	Index-name

Table 41. **Terms and symbols used in MAP output** *(continued)*

Term	Description
NATIONAL	Category national (USAGE NATIONAL)
NAT-EDIT	National-edited (USAGE NATIONAL)
NAT-FLOAT	National floating point (USAGE NATIONAL)
NAT-GROUP	National group (GROUP-USAGE NATIONAL)
NAT-GRP-VARLEN	National variable-length group (GROUP-USAGE NATIONAL)
NAT-NUM	National decimal (USAGE NATIONAL)
NAT-NUM-EDIT	National numeric-edited (USAGE NATIONAL)
OBJECT-REF	Object reference (USAGE OBJECT REFERENCE)
PACKED-DEC	Internal decimal (USAGE PACKED-DECIMAL or COMPUTATIONAL-3)
POINTER	Pointer (USAGE POINTER)
PROCEDURE-PTR	Pointer to an externally callable program (USAGE PROCEDURE-POINTER)
SD	Sort file definition
01-49, 77	Level-numbers for data descriptions
66	Level-number for RENAMES
88	Level-number for condition-names

Example: nested program map

This example shows a map of nested procedures produced by specifying the MAP compiler option. Numbers in parentheses refer to notes that follow the example.

```
Nested Program Map

    (1)
Program Attribute codes (rightmost column) have the following meanings:
   C = COMMON
   I = INITIAL
   U = PROCEDURE DIVISION USING...

  (2)   (3)      (4)                                       (5)
Source Nesting                                             Program
LineID Level   Program Name from PROGRAM-ID paragraph      Attributes
     2         NESTED. . . . . . . . . . . . . . . . . . .
    12     1     X1. . . . . . . . . . . . . . . . . . . .
    20     2       X11 . . . . . . . . . . . . . . . . . .
    27     2       X12 . . . . . . . . . . . . . . . . . .
    35     1     X2. . . . . . . . . . . . . . . . . . . .
```

(1) Explanations of the program attribute codes

(2) Source line number where the program was defined

(3) Depth of program nesting

(4) Program-name

(5) Program attribute codes

Example: XREF output: data-name cross-references

The following example shows a sorted cross-reference of data-names that is produced by the XREF compiler option. Numbers in parentheses refer to notes after the example.

```
An "M" preceding a data-name reference indicates that the
data-name is modified by this reference.

    (1)       (2)                           (3)
  Defined   Cross-reference of data-names   References

      264   ABEND-ITEM1
      265   ABEND-ITEM2
      347   ADD-CODE . . . . . . . . . . .  1126 1192
      381   ADDRESS-ERROR. . . . . . . . .  M1156
      280   AREA-CODE. . . . . . . . . . .  1266 1291 1354 1375
      382   CITY-ERROR . . . . . . . . . .  M1159

       (4)
 Context usage is indicated by the letter preceding a procedure-name
 reference. These letters and their meanings are:
     A = ALTER (procedure-name)
     D = GO TO (procedure-name) DEPENDING ON
     E = End of range of (PERFORM) through (procedure-name)
     G = GO TO (procedure-name)
     P = PERFORM (procedure-name)
     T = (ALTER) TO PROCEED TO (procedure-name)
     U = USE FOR DEBUGGING (procedure-name)

    (5)      (6)                            (7)
  Defined   Cross-reference of procedures   References

      877   000-DO-MAIN-LOGIC
      943   050-CREATE-STL-MASTER-FILE . .  P879
      995   100-INITIALIZE-PARAGRAPH . . .  P881
     1471   1100-PRINT-I-F-HEADINGS. . . .  P926
     1511   1200-PRINT-I-F-DATA. . . . . .  P928
     1573   1210-GET-MILES-TIME. . . . . .  P1540
     1666   1220-STORE-MILES-TIME. . . . .  P1541
     1682   1230-PRINT-SUB-I-F-DATA. . . .  P1562
     1706   1240-COMPUTE-SUMMARY . . . . .  P1563
     1052   200-EDIT-UPDATE-TRANSACTION. .  P890
     1154   210-EDIT-THE-REST. . . . . . .  P1145
     1189   300-UPDATE-COMMUTER-RECORD . .  P893
     1237   310-FORMAT-COMMUTER-RECORD . .  P1194 P1209
     1258   320-PRINT-COMMUTER-RECORD. . .  P1195 P1206 P1212 P1222
     1318   330-PRINT-REPORT . . . . . . .  P1208 P1232 P1286 P1310 P1370
     1342   400-PRINT-TRANSACTION-ERRORS .  P896
```

Cross-reference of data-names:

(1) Line number where the name was defined.

(2) Data-name.

(3) Line numbers where the name was used. If M precedes the line number, the data item was explicitly modified at the location.

Cross-reference of procedure references:

(4) Explanations of the context usage codes for procedure references.

(5) Line number where the procedure-name is defined.

(6) Procedure-name.

(7) Line numbers where the procedure is referenced, and the context usage code for the procedure.

"Example: XREF output: program-name cross-references"
"Example: XREF output: COPY/BASIS cross-references"
"Example: embedded cross-reference" on page 326

Example: XREF output: program-name cross-references

The following example shows a sorted cross-reference of program-names produced by the XREF compiler option. Numbers in parentheses refer to notes that follow the example.

```
   (1)        (2)                             (3)
Defined   Cross-reference of programs     References

EXTERNAL   EXTERNAL1. . . . . . . . . . . 25
      2   X. . . . . . . . . . . . . . .  41
     12   X1 . . . . . . . . . . . . . .  33 7
     20   X11. . . . . . . . . . . . . .  25 16
     27   X12. . . . . . . . . . . . . .  32 17
     35   X2 . . . . . . . . . . . . . .  40 8
```

(1) Line number where the program-name was defined. If the program is external, the word EXTERNAL is displayed instead of a definition line number.

(2) Program-name.

(3) Line numbers where the program is referenced.

Example: XREF output: COPY/BASIS cross-references

The following example shows a sorted cross-reference of copybooks to the library-names and file names of the associated copybooks, produced by the XREF compiler option. Numbers in parentheses refer to notes after the example.

```
        COPY/BASIS cross-reference of text-names, library names and file names

   (1)                            (1)                 (2)
Text-name                     Library-name          File name
                     (3)                        (3)  (4)
"realxrealyrealzlongxlo>    'thisislongdirecto>    <toryname/realxrealyrealzlongxname.cpy
"realxrealyrealzlongxlo>    SYSLIB (default)    (5)./cbldir1/realxrealyrealzlongxname.cpy
"copyA.cpy"                 SYSLIB (default)       ./cbldir1/copyA.cpy
'./copydir2/copyM.cbl'      SYSLIB (default)       ./copydir2/copyM.cbl
'/copyB.cpy'                SYSLIB                 ./cbldir1/copyB.cpy
'/copydir/copyM.cbl'        SYSLIB                 ./cbldir1/copydir/copyM.cbl
'cbldir1/copyC.cpy'         ALTDD2                 ./cbldir1/copyC.cpy
'copydir/copyM.cbl'         SYSLIB                 ./cbldir1/copydir/copyM.cbl
'copydir2/copyM.cbl'        SYSLIB (default)       ./copydir2/copyM.cbl
'copydir3/stuff.cpy'        ALTDD2                 ./copydir3/stuff.cpy
'stuff.cpy'                 ALTDD                  ./copydir3/stuff.cpy
OTHERDD                     ALTDD2                 .//cbldir1/other.cob
REALXLONGXLONGYNAMEX        SYSLIB (default)       ./REALXLONGXLONGYNAMEX
. . .

  (5)
./ = /afs/stllp.sanjose.ibm.com/usr1/cobdev/tmross/stuff/subdir
Note: Some names were truncated.   > = truncated on right   < = truncated on left
```

(1) From the COPY statement in the source; for example the COPY statement corresponding to the fifth item in the cross-reference above would be:

```
COPY '/copyB.cpy' Of SYSLIB
```

(2) The fully qualified path of the file from which the COPY member was copied

(3)	Truncation of a long text-name or library-name on the right is marked by a greater-than sign (>).
(4)	Truncation of a long file name on the left is marked by a less-than sign (<).
(5)	The current working directory portion of the file name is indicated by `./` in the cross-reference. The expansion of the current working directory name is shown in full beneath the cross-reference.

Example: embedded cross-reference

The following example shows a modified cross-reference that is embedded in the source listing. The cross-reference is produced by the XREF compiler option.

```
LineID  PL SL  ----+-*A-1-B--+----2----+----3----+----4----+----5----+----6----+----7-|--+----8  Map and Cross Reference
  . . .
000878                procedure division.
000879                  000-do-main-logic.
000880                    display "PROGRAM IGYTCARA - Beginning".
000881                    perform 050-create-vsam-master-file.                                   932 (1)
000882                    perform 100-initialize-paragraph.                                      984
000883                    perform read update-transaction-file into ws-transaction-record
```

```
LineID  PL SL  ----+-*A-1-B--+----2----+----3----+----4----+----5----+----6----+----7-|--+----8  Map and Cross Reference
  . . .
000878                procedure division.
000879                  000-do-main-logic.
000880                    display "PROGRAM IGYTCARA - Beginning".
000881                    perform 050-create-vsam-master-file.                                   932 (1)
000882                    perform 100-initialize-paragraph.                                      984
000883                    read update-transaction-file into ws-transaction-record                204 340
000884                        at end
000885      1                    set transaction-eof to true                                     254
000886                    end-read.
  . . .
000984                  100-initialize-paragraph.
000985                    move spaces to ws-transaction-record                                   IMP 340 (2)
000986                    move spaces to ws-commuter-record                                      IMP 316
000987                    move zeroes to commuter-zipcode                                        IMP 327
000988                    move zeroes to commuter-home-phone                                     IMP 328
000989                    move zeroes to commuter-work-phone                                     IMP 329
000990                    move zeroes to commuter-update-date                                    IMP 333
000991                    open input update-transaction-file                                     204
000992                        location-file                                                      193
000993                        i-o commuter-file                                                  181
000994                        output print-file                                                  217
  . . .
001442                1100-print-i-f-headings.
001443
001444                    open output print-file.                                                217
001445
001446                    move function when-compiled to when-comp.                              IFN 698 (2)
001447                    move when-comp (5:2) to compile-month.                                 698 640
001448                    move when-comp (7:2) to compile-day.                                   698 642
001449                    move when-comp (3:2) to compile-year.                                  698 644
001450
001451                    move function current-date (5:2) to current-month.                     IFN 649
001452                    move function current-date (7:2) to current-day.                       IFN 651
001453                    move function current-date (3:2) to current-year.                      IFN 653
001454
001455                    write print-record from i-f-header-line-1                              222 635
001456                          after new-page.                                                  138
  . . .
```

(1) Line number of the definition of the data-name or procedure-name in the program

(2) Special definition symbols:

UND	The user name is undefined.
DUP	The user name is defined more than once.
IMP	Implicitly defined name, such as special registers and figurative constants.
IFN	Intrinsic function reference.
EXT	External reference.
*	The program-name is unresolved because the NOCOMPILE option is in effect.

Example: VBREF compiler output

The following example shows an alphabetic listing of all the verbs in a program, and shows where each is referenced. The listing is produced by the VBREF compiler option.

```
 (1)      (2)                (3)
2        ACCEPT . . . . . . .  101  101
2        ADD. . . . . . . . .  129  130
1        CALL . . . . . . . .  140
5        CLOSE. . . . . . . .  90  94  97  152  153
20       COMPUTE. . . . . . . .150  164  164  165  166  166  166  166  167  168  168  169  169  170  171  171
                               171  172  172  173
2        CONTINUE . . . . . . .106  107
2        DELETE . . . . . . . .96  119
47       DISPLAY. . . . . . . .88  90  91  92  92  93  94  94  94  95  96  96  97  99  99  100  100  100  100
                               103  109  117  117  118  119  138  139  139  139  139  139  139  140  140  140
                               140  143  148  148  149  149  149  152  152  152  153  162
2        EVALUATE . . . . . . .116  155
47       IF . . . . . . . . . .88  90  93  94  94  95  96  96  97  99  100  103  105  105  107  107  107  109
                               110  111  111  112  113  113  113  113  114  114  115  115  116  118  119  124
                               124  126  127  129  132  133  134  135  136  148  149  152  152
183      MOVE . . . . . . . . .90  93  95  98  98  98  98  98  99  100  101  101  102  104  105  105  106  106
                               107  107  108  108  108  108  108  108  109  110  111  112  113  113  113  114
                               114  114  115  115  116  116  117  117  117  118  118  118  119  119  120  121
                               121  121  121  121  121  121  121  121  121  122  122  122  122  122  123  123
                               123  123  123  123  123  124  124  124  125  125  125  125  125  125  125  126
                               126  126  126  126  127  127  127  127  128  128  129  129  130  130  130  130
                               131  131  131  131  131  132  132  132  132  132  132  133  133  133  133  133
                               134  134  134  134  134  135  135  135  135  135  135  136  136  137  137  137
                               137  137  138  138  138  138  141  141  142  142  144  144  144  144  145  145
                               145  145  146  149  150  150  150  151  151  155  156  156  157  157  158  158
                               159  159  160  160  161  161  162  162  162  168  168  168  169  169  170  171
                               171  172  172  173  173
5        OPEN . . . . . . . . .93  95  99  144  148
62       PERFORM. . . . . . . .88  88  88  88  89  89  89  91  91  91  91  93  93  94  94  95  95  95  95  96
                               96  96  97  97  97  100  100  101  102  104  109  109  111  116  116  117  117
                               117  118  118  118  118  119  119  119  120  120  124  125  127  128  133  134
                               135  136  136  137  150  151  151  153  153
8        READ . . . . . . . . .88  89  96  101  102  108  149  151
1        REWRITE. . . . . . . .118
4        SEARCH . . . . . . . .106  106  141  142
46       SET. . . . . . . . . .88  89  101  103  104  105  106  108  108  136  141  142  149  150  151  152  154
                               155  156  156  156  156  157  157  157  157  158  158  158  158  159  159  159
                               159  160  160  160  160  161  161  161  161  162  162  164  164
2        STOP . . . . . . . . .92  143
4        STRING . . . . . . . .123  126  132  134
33       WRITE. . . . . . . . .94  116  129  129  129  129  129  130  130  130  130  145  146  146  146  146  147
                               147  151  165  165  166  166  167  174  174  174  174  174  174  174  175  175
```

(1)	Number of times the verb is used in the program
(2)	Verb
(3)	Line numbers where the verb is used

Debugging with messages that have offset information

Some IWZ messages include offset information that you can use to identify the particular line of a program that failed.

To use this information:

1. Compile the program with the `LIST` option. This step produces an assembler listing file, which has a suffix of .wlist.
2. When you get a message that includes offset information, find the offset information for the COBOL program. In the following example, the program is IWZ903, and the corresponding hexadecimal offset is 0x14c (highlighted in bold):

```
IWZ903S  The system detected a data exception.
         Message routine called from offset 0x38 of routine iwzWriteERRmsg.
         iwzWriteERRmsg called from offset 0xa0 of routine _iwzcBCD_ADD_Pckd.
         _iwzcBCD_ADD_Pckd called from offset 0x14c of routine IWZ903.
IWZ901S  Program exits due to severe or critical error.
```

3. Look in the .wlist file for the hexadecimal offset. To the left of the hexadecimal offset is the COBOL statement number. In the following example, the statement number corresponding to 0x14c is 35 (highlighted in bold):

```
35| 000144 addi     39000004   1     LI       gr8=4
35| 000148 bl       4BFFFEB9   0     CALL     gr3=_iwzcBCD_ADD_Pckd
35| 00014C ori      60000000   1
35| 000150 fmr.     FC000091   0
```

4. Look in your program listing for the statement.

RELATED REFERENCES
"LIST" on page 269

Debugging user exits

To debug a user-exit routine, use the debugger on the main compiler module, not on cob2. (The main compiler module is a separate process started by `cob2`, and the debugger can debug only one process.)

Do these steps:

1. Use `cob2` with the `-#` option to see how `cob2` calls the main compiler module and what options it passes. For example, the following command compiles *pgmname*.cbl with the IWZRMGUX user exit:

   ```
   cob2 -c -q"EXIT(ADEXIT(IWZRMGUX))" pgmname.cbl
   ```

   Modify this command as follows:

   ```
   cob2 -# -c -q"EXIT(ADEXIT(IWZRMGUX))" pgmname.cbl
   ```

   As a result, you will see this (`IGYCCOB2` calls your user exit):

   ```
   IGYCCOB2 -q"EXIT(ADEXIT(IWZRMGUX))" pgmname.cbl
   ```

2. Debug the user exit as follows:

   ```
   idebug IGYCCOB2 -q"EXIT(ADEXIT(IWZRMGUX))" pgmname.cbl
   ```

The debugger automatically stops at the beginning of the user exit, provided that you built the exit with debug information.

Debugging assembler routines

Use the Disassembly view to debug assembler routines. Because assembler routines have no debug information, the debugger automatically goes to this view.

Set a breakpoint at a disassembled statement in the Disassembly view by double-clicking in the prefix area. By default, the debugger when it starts runs until it hits the first debuggable statement. To cause the debugger to stop at the very first instruction in the application (debuggable or not), you must use the `-i` option. For example:

```
idebug -i progname
```

Part 4. Accessing databases

Chapter 18. Programming for a DB2 environment 331
DB2 coprocessor 331
Coding SQL statements 332
  Using SQL INCLUDE with the DB2 coprocessor 332
  Using binary items in SQL statements 333
  Determining the success of SQL statements . . 333
Starting DB2 before compiling. 333
Compiling with the SQL option 333
  Separating DB2 suboptions 334
  Using package and bind file-names 334

Chapter 19. Developing COBOL programs for CICS 337
Coding COBOL programs to run under CICS . . 337
  Getting the system date under CICS. 338
  Making dynamic calls under CICS 339
  Accessing SFS data 340
  Calling between COBOL and C/C++ under CICS 341
Compiling and running CICS programs 341
| Integrated CICS translator 342
Debugging CICS programs 342

Chapter 20. Open Database Connectivity (ODBC) 345
Comparison of ODBC and embedded SQL . . . 345
Background 346
Installing and configuring software for ODBC . . 346
Coding ODBC calls from COBOL: overview . . . 346
  Using data types appropriate for ODBC . . . 346
  Passing pointers as arguments in ODBC calls 347
    Example: passing pointers as arguments in ODBC calls 348
  Accessing function return values in ODBC calls 349
  Testing bits in ODBC calls 349
Using COBOL copybooks for ODBC APIs 350
  Example: sample program using ODBC copybooks 351
  Example: copybook for ODBC procedures. . . 352
  Example: copybook for ODBC data definitions 355
  ODBC names truncated or abbreviated for COBOL 355
Compiling and linking programs that make ODBC calls 357

Chapter 18. Programming for a DB2 environment

In general, the coding for your COBOL program will be the same if you want the program to access a DB2 database. However, to retrieve, update, insert, and delete DB2 data and use other DB2 services, you must use SQL statements.

To communicate with DB2, do these steps:

- Code any SQL statements that you need, delimiting them with `EXEC SQL` and `END-EXEC` statements.
- Start DB2 if it is not already started.
- Compile with the `SQL` compiler option.
- Compile with the `NODYNAM` compiler option if the application was compiled using the DB2 stand-alone precompiler.

  If `EXEC SQL` statements are used in COBOL libraries that are loaded by a COBOL dynamic call, then one or more `EXEC SQL` statements must be in the main program. (Called DB2 APIs cannot be loaded using a COBOL dynamic call.)

RELATED CONCEPTS
"DB2 coprocessor"

RELATED TASKS
"Coding SQL statements" on page 332
"Starting DB2 before compiling" on page 333
"Compiling with the SQL option" on page 333

RELATED REFERENCES
"DYNAM" on page 258
DB2 SQL Reference Volume 1
DB2 SQL Reference Volume 2

DB2 coprocessor

When you use the DB2 coprocessor, the compiler handles your source program that contains embedded SQL statements without your having to use a separate precompiler.

When the compiler encounters SQL statements in the source program, it interfaces with the DB2 coprocessor. This coprocessor takes appropriate actions for the SQL statements and indicates to the compiler which native COBOL statements to generate for them.

Certain restrictions on the use of COBOL language that apply with the precompiler do not apply when you use the DB2 coprocessor:

- You can identify host variables used in SQL statements without using `EXEC SQL BEGIN DECLARE SECTION` and `EXEC SQL END DECLARE SECTION` statements.
- You can compile in batch a source file that contains multiple nonnested COBOL programs.
- The source program can contain nested programs.
- The source program can contain object-oriented COBOL language extensions.

You must specify the SQL compiler option to compile programs that use the DB2 coprocessor.

RELATED TASKS
"Compiling with the SQL option" on page 333

RELATED REFERENCES
"SQL" on page 284

Coding SQL statements

| Delimit SQL statements with EXEC SQL and END-EXEC. The EXEC SQL and END-EXEC
| delimiters must each be complete on one line. You cannot continue them across
| multiple lines.

You also need to do these special steps:

| • Code an EXEC SQL INCLUDE statement to include an SQL communication area (SQLCA) in the WORKING-STORAGE SECTION or LOCAL-STORAGE SECTION of the outermost program. LOCAL-STORAGE is recommended for recursive programs and programs that use the THREAD compiler option.
• Declare all host variables that you use in SQL statements in the WORKING-STORAGE SECTION, LOCAL-STORAGE SECTION, or LINKAGE SECTION. However, you do not need to identify them with EXEC SQL BEGIN DECLARE SECTION and EXEC SQL END DECLARE SECTION.

Restriction: You cannot use SQL statements in object-oriented classes or methods.

You can use SQL statements even for large objects (such as BLOB and CLOB) and compound SQL.

RELATED TASKS
"Using SQL INCLUDE with the DB2 coprocessor"
"Using binary items in SQL statements" on page 333
"Determining the success of SQL statements" on page 333

Using SQL INCLUDE with the DB2 coprocessor

An SQL INCLUDE statement is treated identically to a native COBOL COPY statement (including the search path and the file suffixes used) when you use the SQL compiler option.

| The following two lines are therefore treated the same way. (The period that ends
| the EXEC SQL INCLUDE statement is required.)

```
EXEC SQL INCLUDE name END-EXEC.
COPY name.
```

The *name* in an SQL INCLUDE statement follows the same rules as those for COPY *text-name* and is processed identically to a COPY *text-name* statement that does not have a REPLACING phrase.

COBOL does not use the DB2 environment variable DB2INCLUDE for SQL INCLUDE processing. If you use the DB2INCLUDE environment variable for SQL

INCLUDE processing, you can concatenate it with the setting of the COBOL SYSLIB environment variable in the .profile file in your home directory or at the prompt in an AIX command shell. For example:

```
export SYSLIB=$DB2INCLUDE:$SYSLIB
```

RELATED REFERENCES
Chapter 15, "Compiler-directing statements," on page 295
COPY statement (*COBOL for AIX Language Reference*)

Using binary items in SQL statements

For binary data items that you specify in an EXEC SQL statement, you can declare the data items as either USAGE COMP-5 or as USAGE BINARY, COMP, or COMP-4.

If you declare the binary data items as USAGE BINARY, COMP, or COMP-4, use the TRUNC(BIN) option. (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 43

RELATED REFERENCES
"TRUNC" on page 288

Determining the success of SQL statements

When DB2 finishes executing an SQL statement, DB2 sends a return code in the SQLCODE and SQLSTATE fields of the SQLCA structure to indicate whether the operation succeeded or failed. Your program should test the return code and take any necessary action.

Starting DB2 before compiling

Because the compiler works in conjunction with the DB2 coprocessor, you must start DB2 before you compile your program.

You can connect to the target database for the compilation before you start the compilation, or can have the compiler make the connection. To have the compiler connect, specify the database by either of these means:
- Use the DATABASE suboption in the SQL option.
- Name the database in the DB2DBDFT environment variable.

Compiling with the SQL option

The option string that you provide in the SQL compiler option is made available to the DB2 coprocessor. Only the DB2 coprocessor views the content of the string.

For example, the following cob2 command passes the database name SAMPLE and the DB2 options USER and USING to the coprocessor:

```
cob2 -q"sql('database sample user myname using mypassword')" mysql.cbl. . .
```

The following options, which were meaningful to and used by the DB2 precompiler, are ignored by the coprocessor:

- MESSAGES
- NOLINEMACRO
- OPTLEVEL
- OUTPUT
- SQLCA
- TARGET
- WCHARTYPE

RELATED TASKS
"Separating DB2 suboptions"
"Using package and bind file-names"

RELATED REFERENCES
"SQL" on page 284
DB2 Command Reference (Precompile)

Separating DB2 suboptions

Because of the concatenation of multiple SQL option specifications, you can separate DB2 suboptions (which might not fit in one CBL statement) into multiple CBL statements.

The options that you include in the suboption string are cumulative. The compiler concatenates these suboptions from multiple sources in the order that they are specified. For example, suppose that your source file mypgm.cbl has the following code:

```
cbl . . . SQL("string2") . . .
cbl . . . SQL("string3") . . .
```

| When you issue the command `cob2 mypgm.cbl -q:"SQL('string1')"`, the compiler passes the following suboption string to the DB2 coprocessor:

```
"string1 string2 string3"
```

The concatenated strings are delimited with single spaces. If the compiler finds multiple instances of the same SQL suboption, the last specification of that suboption in the concatenated string takes effect. The compiler limits the length of the concatenated DB2 suboption string to 4 KB.

Using package and bind file-names

Two of the suboptions that you can specify with the SQL option are `package name` and `bind file name`. If you do not specify these names, default names are constructed based on the source file-name for a nonbatch compilation or on the first program for a batch compilation.

For subsequent nonnested programs of a batch compilation, the names are based on the PROGRAM-ID of each program.

For the package name, the base name (the source file-name or the PROGRAM-ID) is modified as follows:

- Names longer than eight characters are truncated to eight characters.

- Lowercase letters are folded to uppercase.
- Any character other than A-Z, 0-9, or _ (underscore) is changed to 0.
- If the first character is not alphabetic, it is changed to A.

Thus if the base name is `9123aB-cd`, the package name is `A123AB0C`.

For the bind file-name, the suffix .bnd is added to the base name. Unless explicitly specified, the file-name is relative to the current directory.

Chapter 19. Developing COBOL programs for CICS

You can write CICS applications in COBOL and run them on the AIX workstation using TXSeries.

To prepare COBOL applications to run under CICS, do these steps:

1. Ensure that your CICS administrator modified the region's environment file to set the environment variables COBPATH, COBLIB, and LIBPATH to include the runtime directory. Also, ensure that the CICS region was granted access to the runtime directory.

   The environment file is /var/cics_regions/*xxxxxxxx*/environment (where *xxxxxxxx* is the name of the region).
2. Create the application by using an editor to do the following tasks:
   - Code your program using COBOL statements and CICS commands.
   - Create COBOL copybooks.
   - Create the CICS screen maps that your program uses.
3. Use the command `cicsmap` to process the screen maps.
4. Use one of the forms of the `cicstcl` command to translate the CICS commands and to compile and link the program:
   - Use `cicstcl -p` to use the integrated CICS translator.
   - Use `cicstcl` without the `-p` flag to use the separate CICS translator.
5. Define the resources for your application, such as transactions, application programs, and files, to the CICS region. CICS administrator authority is required to perform these actions.
6. Access the CICS region, for example by using the `cicsterm` command.
7. Run the application by entering the four-character transaction ID that is associated with the application.

RELATED CONCEPTS
"Integrated CICS translator" on page 342

RELATED TASKS
"Coding COBOL programs to run under CICS"
"Compiling and running CICS programs" on page 341
"Debugging CICS programs" on page 342
TXSeries for Multiplatforms: CICS Application Programming Guide

Coding COBOL programs to run under CICS

In general, the COBOL language is supported in a CICS environment. However, there are restrictions and considerations that you should be aware of when you code COBOL programs to run under TXSeries.

Restrictions:

- Object-oriented programming and interoperability with Java are not supported. COBOL class definitions and methods cannot be run in a CICS environment.
- The source program must not contain any nested programs.
- COBOL programs that will run under TXSeries must be 32 bit.

Do not use EXEC, CICS, or END-EXEC as variable names, and do not use user-specified parameters to the main program. In addition, it is recommended that you not use any of the following COBOL language elements:

- FILE-CONTROL entry in the ENVIRONMENT DIVISION
- FILE SECTION in the DATA DIVISION
- USE declaratives, except USE FOR DEBUGGING

The following COBOL statements are also not recommended for use in a CICS environment:

- ACCEPT format 1
- CLOSE
- DELETE
- DISPLAY UPON CONSOLE, DISPLAY UPON SYSPUNCH
- MERGE
- OPEN
- READ
- REWRITE
- SORT
- START
- STOP *literal*
- WRITE

Apart from some forms of the ACCEPT statement, mainframe CICS does not support any of the COBOL language elements in the preceding list. If you use any of those elements, be aware of the following limitations:

- The program is not completely portable to the mainframe CICS environment.
- In the case of a CICS failure, a backout (restoring the resources that are associated with the failed task) for resources that were updated by using the above statements will not be possible.

Restriction: There is no zSeries host data format support for COBOL programs that are translated by the separate or integrated CICS translator and run on TXSeries.

RELATED TASKS
"Getting the system date under CICS"
"Making dynamic calls under CICS" on page 339
"Accessing SFS data" on page 340
"Calling between COBOL and C/C++ under CICS" on page 341

RELATED REFERENCES
"ADDR" on page 246
Appendix C, "zSeries host data format considerations," on page 581

Getting the system date under CICS

To retrieve the system date in a CICS program, use a format-2 ACCEPT statement or the CURRENT-DATE intrinsic function.

You can use any of these format-2 ACCEPT statements in the CICS environment to get the system date:

- ACCEPT *identifier-2* FROM DATE (two-digit year)

- ACCEPT *identifier-2* FROM DATE YYYYMMDD
- ACCEPT *identifier-2* FROM DAY (two-digit year)
- ACCEPT *identifier-2* FROM DAY YYYYDDD
- ACCEPT *identifier-2* FROM DAY-OF-WEEK (one-digit integer, where 1 represents Monday)

You can use this format-2 ACCEPT statement in the CICS environment to get the system time:

- ACCEPT *identifier-2* FROM TIME

Alternatively, you can use the CURRENT-DATE intrinsic function, which can also provide the time.

These methods work in both CICS and non-CICS environments.

Do not use a format-1 ACCEPT statement in a CICS program.

RELATED TASKS
"Assigning input from a screen or file (ACCEPT)" on page 34

RELATED REFERENCES
CURRENT-DATE (*COBOL for AIX Language Reference*)

Making dynamic calls under CICS

You can use CALL *identifier* statements to make dynamic calls in the CICS environment. However, you must set the COBPATH environment variable correctly. You also must make sure that the called module has the correct name.

Consider the following example, in which alpha is a COBOL program that contains CICS statements:

```
WORKING-STORAGE SECTION.
01  WS-COMMAREA   PIC 9    VALUE ZERO.
77   SUBPNAME     PIC X(8) VALUE SPACES
. . .
PROCEDURE DIVISION.
    MOVE 'alpha' TO SUBPNAME.
    CALL SUBPNAME USING DFHEIBLK, DFHCOMMAREA, WS-COMMAREA.
```

You must pass the CICS control blocks DFHEIBLK and DFHCOMMAREA (as shown above) to alpha.

The source for alpha is in file alpha.ccp. Use the command cicstcl to translate, compile, and link alpha.ccp. COBOL defaults to uppercase names. Therefore, unless you change this default by using the PGMNAME(MIXED)option, you need to name the source file ALPHA.ccp (not alpha.cpp) to produce ALPHA.ibmcob (not alpha.ibmcob).

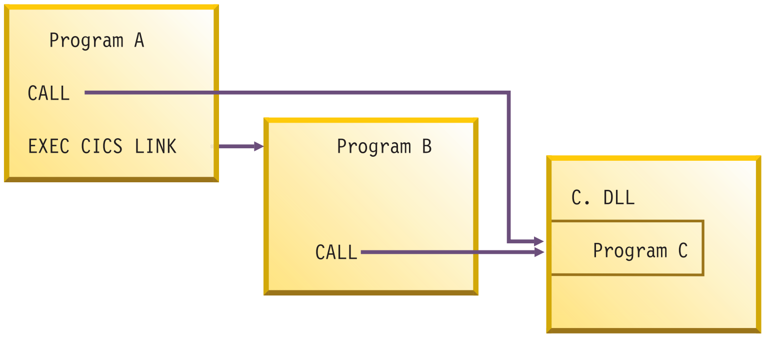
Suppose that the CICS region is called green. Then file ALPHA.ibmcob must be copied to /var/cics_regions/green/bin, and the AIX SMIT tool must be used to define ALPHA as a CICS program. Your installation staff must add the following line to the file /var/cics_regions/green/environment:

```
COBPATH=/var/cics_regions/green/bin
```

Then the staff must shut down the CICS `green` region and restart it. If you put dynamically called programs in some other directory, make sure that your installation staff adds that directory to COBPATH and that the CICS servers have permission to access that directory.

Shared library considerations: If you have a shared library that contains one or more COBOL programs, do not use it in more than one run unit within the same CICS transaction; otherwise the results are unpredictable. The following figure shows a CICS transaction in which the same subprogram is called from two different run units:

- Program A calls Program C (in C.so).
- Program A links to Program B using an `EXEC CICS LINK` command. This combination becomes a new run unit within the same transaction.
- Program B calls Program C (in C.so).



Programs A and B share the same copy of Program C, and any changes to its state affect both programs.

In the CICS environment, programs in a shared library are initialized (both the `WSCLEAR` compiler option and `VALUE` clause initialization) only on the first call within a run unit. If a COBOL subprogram is called more than once from either the same or different main programs, the subprogram is initialized on only the first call. If you need the subprogram to be initialized on the first call from each main program, statically link a separate copy of the subprogram with each calling program. If you need the subprogram to be initialized on every call, use one of the following methods:

- Put data to be reinitialized in the `LOCAL-STORAGE SECTION` of the subprogram rather than in the `WORKING-STORAGE SECTION`. This placement affects initialization only by the `VALUE` clause, not by the `WSCLEAR` compiler option.
- Use `CANCEL` to cancel the subprogram after each use so that the next call will be to the program in its initial state.
- Add the `INITIAL` attribute to the subprogram.

Accessing SFS data

Your program can access SFS (the default file system used by TXSeries) through the VSAM file system when it is not running under CICS.

RELATED TASKS
"Identifying Encina SFS files" on page 114

Calling between COBOL and C/C++ under CICS

You can make a call under CICS from a COBOL to a C/C++ program or from a C/C++ program to a COBOL program only if the called program does not contain any CICS commands. (The calling program can contain CICS commands.)

COBOL programs can issue an EXEC CICS LINK or EXEC CICS XCTL command to a C/C++ program regardless of whether the C/C++ program contains CICS commands. Therefore, if your COBOL program calls a C/C++ program that contains CICS commands, use EXEC CICS LINK or EXEC CICS XCTL rather than the COBOL CALL statement.

RELATED TASKS
"Calling between COBOL and C/C++ programs" on page 471

Compiling and running CICS programs

COBOL for AIX TXSeries programs must use the thread-safe version of the COBOL runtime library. Compile these programs with the cob2_r command and the NOTHREAD compiler option (the default).

TRUNC(BIN) is a recommended compiler option for a COBOL program that will run under CICS. However, if you are certain that the nontruncated values of BINARY, COMP, and COMP-4 data items conform to PICTURE specifications, you might improve program performance by using TRUNC(OPT).

You can use a COMP-5 data item instead of a BINARY, COMP, or COMP-4 data item as an EXEC CICS command argument. COMP-5 data items are treated like BINARY, COMP, or COMP-4 data items if TRUNC(BIN) is in effect.

You must use the PGMNAME(MIXED) compiler option for programs that use CICS Client.

| Do not use the DYNAM, NOLIB, THREAD, or ADDR(64) compiler options when you
| translate COBOL programs with the separate or integrated CICS translator. All other COBOL compiler options are supported.

Runtime options: Use the FILESYS runtime option to specify the file system to use if no specific file system has been selected by means of an ASSIGN clause.

RELATED CONCEPTS
"Integrated CICS translator" on page 342

RELATED TASKS
"Compiling from the command line" on page 219
TXSeries for Multiplatforms: CICS Application Programming Guide

RELATED REFERENCES
Chapter 14, "Compiler options," on page 243
"Conflicting compiler options" on page 245
"FILESYS" on page 303

Integrated CICS translator

When you compile a COBOL program using the `CICS` compiler option, the COBOL compiler works with the integrated CICS translator to handle both native COBOL and embedded CICS statements in the source program.

When the compiler encounters CICS statements, and at other significant points in the source program, the compiler interfaces with the integrated CICS translator. The translator takes appropriate actions and then returns to the compiler, typically indicating which native language statements to generate.

If you compile the COBOL program by using the `cicstcl` command, specify the `-p` flag to use the integrated CICS translator. The `cicstcl -p` command invokes the compiler with the appropriate suboptions of the `CICS` compiler option.

Although you can still translate embedded CICS statements separately, using the integrated CICS translator is recommended. Certain restrictions that apply when you use the separate translator do not apply when you use the integrated translator, and using the integrated translator provides several advantages:

- You can use the Debugger for AIX to debug the original source instead of the expanded source that the separate CICS translator provides.
- You do not need to separately translate the `EXEC CICS` statements that are in copybooks.
- There is no intermediate file for the translated but not compiled version of the source program.
- Only one output listing instead of two is produced.
- `REPLACE` statements can affect `EXEC CICS` statements.
- You can compile programs that contain CICS statements in batch.

RELATED TASKS
"Coding COBOL programs to run under CICS" on page 337
"Compiling and running CICS programs" on page 341

RELATED REFERENCES
"CICS" on page 252

Debugging CICS programs

If you compile CICS programs using the integrated translator, you can debug the programs at the original source level instead of debugging the expanded source that the separate CICS translator provides.

If you use the separate CICS translator, first translate your CICS programs into COBOL. Then you can debug the resulting COBOL programs the same way you would debug any other COBOL programs.

You can debug CICS programs by using the Debugger for AIX that is shipped with COBOL for AIX. Be sure to instruct the compiler to produce the symbolic information that the debugger uses.

RELATED CONCEPTS
Chapter 17, "Debugging," on page 307
"Integrated CICS translator"

RELATED TASKS

"Compiling from the command line" on page 219

TXSeries for Multiplatforms: CICS Application Programming Guide

Chapter 20. Open Database Connectivity (ODBC)

Open Database Connectivity (OBDC) is a specification for an application programming interface (API) that enables applications to access multiple database management systems using Structured Query Language (SQL). Using the ODBC interface in your COBOL applications, you can dynamically access databases and file systems that support the ODBC interface.

ODBC permits maximum interoperability: a single application can access many different database management systems. This interoperability enables you to develop, compile, and ship an application without targeting a specific type of data source. Users can then add the database drivers, which link the application to the database management systems of their choice.

When you use the ODBC interface, your application makes calls through a driver manager. The driver manager dynamically loads the necessary driver for the database server to which the application connects. The driver, in turn, accepts the call, sends the SQL to the specified data source (database), and returns any result.

Comparison of ODBC and embedded SQL

Your COBOL applications that use embedded SQL for database access must be processed by a precompiler or coprocessor for a particular database and need to be recompiled if the target database changes. Because ODBC is a call interface, there is no compile-time designation of the target database as there is with embedded SQL. Not only can you avoid having multiple versions of your application for multiple databases, but your application can dynamically determine which database to target.

Some of the advantages of ODBC are:

- ODBC provides a consistent interface regardless of the kind of database server used.
- You can have more than one concurrent connection.
- Applications do not have to be bound to each database on which they will run. Although COBOL for AIX does this bind for you automatically, it binds automatically to only one database. If you want to choose which database to connect to dynamically at run time, you must take extra steps to bind to a different database.

Embedded SQL also has some advantages:

- Static SQL usually provides better performance than dynamic SQL. It does not have to be prepared at run time, thus reducing both processing and network traffic.
- With static SQL, database administrators have to grant users access to a package only rather than access to each table or view that will be used.

Background

The X/Open Company and the SQL Access Group jointly developed a specification for a callable SQL interface, referred to as the X/Open Call Level Interface. The goal of this interface is to increase portability of applications by enabling them to become independent of any one database vendor's programming interface.

ODBC was originally developed by Microsoft® for Microsoft operating systems based on a preliminary draft of X/Open CLI. Since then, other vendors have provided ODBC drivers that run on other platforms, such as UNIX systems.

Installing and configuring software for ODBC

To enable ODBC for data access in COBOL for AIX, you must install an ODBC driver manager and drivers, add the ODBC database drivers necessary for your installation, and install the RDBMS client (for example, DB2 UDB, Oracle 7 SQL*NET).

Coding ODBC calls from COBOL: overview

To access ODBC from COBOL programs, you need to use data types that are appropriate for ODBC, and understand how to pass arguments, access return values for functions, and test bits. IBM COBOL for AIX provides copybooks that help you make ODBC calls.

RELATED TASKS
"Using data types appropriate for ODBC"
"Passing pointers as arguments in ODBC calls" on page 347
"Accessing function return values in ODBC calls" on page 349
"Testing bits in ODBC calls" on page 349
"Using COBOL copybooks for ODBC APIs" on page 350
"Compiling and linking programs that make ODBC calls" on page 357

Using data types appropriate for ODBC

The data types specified in ODBC APIs are defined in terms of ODBC C types in the API definitions. Use the COBOL data declarations that correspond to the indicated ODBC C types of the arguments.

Table 42. **ODBC C types and corresponding COBOL declarations**

ODBC C type	COBOL form	Description
SQLSMALLINT	`COMP-5 PIC S9(4)`	Signed short integer (2-byte binary)
SQLUSMALLINT	`COMP-5 PIC 9(4)`	Unsigned short integer (2-byte binary)
SQLINTEGER	`COMP-5 PIC S9(9)`	Signed long integer (4-byte binary)
SQLUINTEGER	`COMP-5 PIC 9(9)`	Unsigned long integer (4-byte binary)
SQLREAL	`COMP-1`	Floating point (4 bytes)
SQLFLOAT	`COMP-2`	Floating point (8 bytes)
SQLDOUBLE	`COMP-2`	Floating point (8 bytes)
SQLCHAR	`POINTER`	Pointer to unsigned character
SQLHDBC	`POINTER`	Connection handle

Table 42. **ODBC C types and corresponding COBOL declarations** *(continued)*

ODBC C type	COBOL form	Description
SQLHENV	POINTER	Environment handle
SQLHSTMT	POINTER	Statement handle
SQLHWND	POINTER	Window handle

The pointer to an unsigned character in COBOL is a pointer to a null-terminated string. You define the target of the pointer item as `PIC X(`*n*`)`, where *n* is large enough to represent the null-terminated field. Some ODBC APIs require that you pass null-terminated character strings as arguments.

Do not use zSeries host data formats. ODBC APIs expect parameters to be in native format.

RELATED TASKS
"Passing pointers as arguments in ODBC calls"
"Accessing function return values in ODBC calls" on page 349

Passing pointers as arguments in ODBC calls

If you specify a pointer argument to one of the data types acceptable to ODBC, then you must either pass the target of the pointer `BY REFERENCE`, define a pointer item that points to the target item and pass that `BY VALUE`, or pass the `ADDRESS OF` the target `BY VALUE`.

To illustrate, assume the function is defined as:

```
RETCODE SQLSomeFunction(PSomeArgument)
```

Here `PSomeArgument` is defined as an argument that points to `SomeArgument`. You can pass the argument to `SQLSomeFunction` in one of the following ways:

- Pass `SomeArgument BY REFERENCE`:
  ```
  CALL "SQLSomeFunction" USING BY REFERENCE SomeArgument
  ```
  You can use `USING BY CONTENT SomeArgument` instead if `SomeArgument` is an input argument.
- Define a pointer data item `PSomeArgument` to point to `SomeArgument`:
  ```
  SET PSomeArgument TO ADDRESS OF SomeArgument
  CALL "SQLSomeFunction" USING BY VALUE PSomeArgument
  ```
- Pass `ADDRESS OF SomeArgument BY VALUE`:
  ```
  CALL "SQLSomeFunction" USING BY VALUE ADDRESS OF SomeArgument
  ```

You can use the last approach only if the target argument, `SomeArgument`, is a level-01 item in the `LINKAGE SECTION`. If so, you can set the addressability to `SomeArgument` in one of the following ways:

- Explicitly by using either a pointer or an identifier, as in the following examples:
  ```
  SET ADDRESS OF SomeArgument TO a-pointer-data-item
  SET ADDRESS OF SomeArgument to ADDRESS OF an-identifier
  ```
- Implicitly by having `SomeArgument` passed in as an argument to the program from which the ODBC function call is being made.

"Example: passing pointers as arguments in ODBC calls" on page 348

Example: passing pointers as arguments in ODBC calls

The following example program fragment shows how to invoke the SQLAllocHandle function.

```
. . .
WORKING-STORAGE SECTION.
    COPY "odbc3.cpy".
    . . .
01  SQL-RC     COMP-5    PIC S9(4).
01  Henv       POINTER.
. . .
PROCEDURE DIVISION.
. . .
    CALL "SQLAllocHandle"
        USING
        By VALUE     sql=handle-env
                     sql-null-handle
        By REFERENCE Henv
        RETURNING      SQL-RC
    IF SQL-RC NOT = (SQL-SUCCESS or SQL-SUCCESS-WITH-INFO)
        THEN
            DISPLAY "SQLAllocHandle failed."
            . . .
        ELSE
        . . .
```

You can use any one of the following examples for calling the SQLConnect function.

Example 1:

```
. . .
CALL "SQLConnect" USING BY VALUE     ConnectionHandle
                        BY REFERENCE ServerName
                        BY VALUE     SQL-NTS
                        BY REFERENCE UserIdentifier
                        BY VALUE     SQL-NTS
                        BY REFERENCE AuthentificationString
                        BY VALUE     SQL-NTS
                  RETURNING          SQL-RC
. . .
```

Example 2:

```
. . .
SET Ptr-to-ServerName             TO ADDRESS OF ServerName
SET Ptr-to-UserIdentifier         TO ADDRESS OF UserIdentifier
SET Ptr-to-AuthentificationString TO ADDRESS OF AuthentificationString
CALL "SQLConnect" USING BY VALUE     ConnectionHandle
                                     Ptr-to-ServerName
                                     SQL-NTS
                                     Ptr-to-UserIdentifier
                                     SQL-NTS
                                     Ptr-to-AuthentificationString
                                     SQL-NTS
                  RETURNING          SQL-RC
. . .
```

Example 3:

```
. . .
CALL "SQLConnect" USING BY VALUE     ConnectionHandle
                                     ADDRESS OF ServerName
                                     SQL-NTS
                                     ADDRESS OF UserIdentifier
                                     SQL-NTS
```

```
                                   ADDRESS OF AuthentificationString
                                   SQL-NTS
                  RETURNING        SQL-RC
. . .
```

In Example 3, you must define Servername, UserIdentifier, and AuthentificationString as level-01 items in the LINKAGE SECTION.

The BY REFERENCE or BY VALUE phrase applies to all arguments until another BY REFERENCE, BY VALUE, or BY CONTENT phrase overrides it.

Accessing function return values in ODBC calls

Specify the function return values for an ODBC call by using the RETURNING phrase of the CALL statement.

```
CALL "SQLAllocEnv" USING BY VALUE Phenv RETURNING SQL-RC
    IF SQL-RC NOT = SQL-SUCCESS
        THEN
            DISPLAY "SQLAllocEnv failed."
            . . .
        ELSE
            . . .
    END-IF
```

Testing bits in ODBC calls

For some of the ODBC APIs, you must set bit masks and must query bits. You can use the library routine iwzODBCTestBits to query bits.

Call this routine as follows:

```
CALL "iwzODBCTestBits" USING identifier-1,
identifier-2 RETURNING identifier-3
```

identifier-1
: The field being tested. It must be a 2-byte or 4-byte binary number field, that is, USAGE COMP-5 PIC 9(4) or PIC 9(9).

identifier-2
: The bit mask field to select the bits to be tested. It must be defined with the same USAGE and PICTURE as *identifier-1*.

identifier-3
: The return value for the test (define it as USAGE COMP-5 PIC S9(4)):

0	No bits selected by *identifier-2* are ON in *identifier-1*.
1	All bits selected by *identifier-2* are ON in *identifier-1*.
-1	One or more bits are ON and one or more bits are OFF among the bits selected by *identifier-2* for *identifier-1*.
-100	Invalid input-argument found (such as an 8-byte binary number field used as *identifier-1*).

| You can set multiple bits in a field with COBOL arithmetic expressions that use the bit masks defined in the odbc2.cpy and odbc3.cpy copybooks. For example, you can set bits for SQL-CVT-CHAR, SQL-CVT-NUMERIC, and SQL-CVT-DECIMAL in the InfoValue field with this statement:

```
COMPUTE InfoValue = SQL-CVT-CHAR + SQL-CVT-NUMERIC + SQL-CVT-DECIMAL
```

After you set InfoValue, you can pass it to the iwzTestBits function as the second argument.

The operands in the arithmetic expression above represent disjoint bits as defined in the odbc2.cpy or odbc3.cpy copybook. As a result, addition sets the intended bits on. Be careful to not repeat bits in such an arithmetic expression, however, because the operation is not logical OR. For example, the following code results in InfoValue not having the intended SQL-CVT-CHAR bit on:

```
COMPUTE InfoValue = SQL-CVT-CHAR + SQL-CVT-NUMERIC + SQL-CVT-DECIMAL + SQL-CVT-CHAR
```

Using COBOL copybooks for ODBC APIs

IBM COBOL for AIX provides copybooks that make it easier for you to access databases with ODBC drivers by using ODBC calls from COBOL programs. You can use these copybooks with or without modification.

The copybooks described below are for ODBC Version 3.0. However, Version 2.*x* copybooks are also supplied, and you can substitute them for the Version 3.0 copybooks if you need to develop applications for ODBC Version 2.*x*.

Table 43. **ODBC copybooks**

Copybook for Version 3.0 of ODBC	Copybook for Version 2.*x* of ODBC	Description	Location
odbc3.cpy	odbc2.cpy	Symbols and constants	include folder for COBOL
odbc3d.cpy	odbc2d.cpy	DATA DIVISION definitions	odbc folder in the samples folder for COBOL
odbc3p.cpy	odbc2p.cpy	PROCEDURE DIVISION statements	odbc folder in the samples folder for COBOL

Include the paths for the include and odbc folders in the SYSLIB environment variable to ensure that the copybooks are available to the compiler.

odbc3.cpy defines the symbols for constant values described for ODBC APIs. It maps constants used in calls to ODBC APIs to symbols specified in ODBC guides. You can use this copybook to specify and test arguments and function return values.

odbc3p.cpy lets you use prepared COBOL statements for commonly used functions for initializing ODBC, handling errors, and cleaning up (SQLAllocEnv, SQLAllocConnect, SQLAllocStmt, SQLFreeStmt, SQLDisconnect, SQLFreeConnect, and SQLFreeEnv).

odbc3d.cpy contains data declarations used by odbc3.cpy in the WORKING-STORAGE SECTION (or LOCAL-STORAGE SECTION).

Some COBOL-specific adaptations were made in these copybooks:
- Underscores (_) were replaced with hyphens (-). For example, SQL_SUCCESS is specified as SQL-SUCCESS.
- Names longer than 30 characters are truncated or abbreviated to 30 characters, as shown in the related reference below.

To include the copybook odbc3.cpy, specify a COPY statement in the DATA DIVISION as follows:

- For a program, specify the COPY statement in the WORKING-STORAGE SECTION (in the outermost program if programs are nested).
- For each method that makes ODBC calls, specify the COPY statement in the WORKING-STORAGE SECTION of the method (not the WORKING-STORAGE SECTION of the class definition).

"Example: sample program using ODBC copybooks"
"Example: copybook for ODBC data definitions" on page 355
"Example: copybook for ODBC procedures" on page 352

RELATED REFERENCES
"ODBC names truncated or abbreviated for COBOL" on page 355

Example: sample program using ODBC copybooks

The following example illustrates the use of three ODBC copybooks.

```
cbl  pgmname(mixed)
******************************************************************
* ODBC3EG.CBL                                                    *
*----------------------------------------------------------------*
* Sample program using ODBC3, ODBC3D, and ODBC3P copybooks       *
******************************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. "ODBC3EG".
 DATA DIVISION.

 WORKING-STORAGE SECTION.
*  copy ODBC API constant definitions
     COPY "odbc3.cpy" SUPPRESS.
*  copy additional definitions used by ODBC3P procedures
     COPY "odbc3d.cpy".
*  arguments used for SQLConnect
 01  ServerName                    PIC X(10) VALUE Z"Oracle7".
 01  ServerNameLength       COMP-5 PIC S9(4) VALUE 10.
 01  UserId                        PIC X(10) VALUE Z"TEST123".
 01  UserIdLength           COMP-5 PIC S9(4) VALUE 10.
 01  Authentification              PIC X(10) VALUE Z"TEST123".
 01  AuthentificationLength COMP-5 PIC S9(4) VALUE 10.
 PROCEDURE DIVISION.
 Do-ODBC SECTION.
  Start-ODBC.
     DISPLAY "Sample ODBC 3.0 program starts"
*  allocate henv & hdbc
     PERFORM ODBC-Initialization
*  connect to data source
     CALL "SQLConnect" USING BY VALUE     Hdbc
                             BY REFERENCE ServerName
                             BY VALUE     ServerNameLength
                             BY REFERENCE UserId
                             BY VALUE     UserIdLength
                             BY REFERENCE Authentification
                             BY VALUE     AuthentificationLength
                       RETURNING SQL-RC
     IF SQL-RC NOT = SQL-SUCCESS
       MOVE "SQLConnect" to SQL-stmt
       MOVE SQL-HANDLE-DBC to DiagHandleType
       SET DiagHandle to Hdbc
       PERFORM SQLDiag-Function
     END-IF
*  allocate hstmt
```

```
     PERFORM Allocate-Statement-Handle
*****************************************
*  add application specific logic here  *
*****************************************
*  clean-up environment
     PERFORM ODBC-Clean-Up.
*  End of sample program execution
     DISPLAY "Sample COBOL ODBC program ended"
     GOBACK.
*  copy predefined COBOL ODBC calls which are performed
     COPY "odbc3p.cpy".
*******************************************************
* End of ODBC3EG.CBL:  Sample program for ODBC 3.0    *
*******************************************************
```

Example: copybook for ODBC procedures

This example shows procedures for initializing ODBC, cleaning up, and handling errors.

```
*****************************************************************
* ODBC3P.CPY                                                    *
*---------------------------------------------------------------*
*  Sample ODBC initialization, clean-up and error handling      *
*    procedures    (ODBC Ver 3.0)                               *
*****************************************************************
*** Initialization functions SECTION ****************************
 ODBC-Initialization SECTION.
*
  Allocate-Environment-Handle.
     CALL "SQLAllocHandle" USING
                             BY VALUE     SQL-HANDLE-ENV
                             BY VALUE     SQL-NULL-HANDLE
                             BY REFERENCE Henv
                           RETURNING      SQL-RC
      IF SQL-RC NOT = SQL-SUCCESS
       MOVE "SQLAllocHandle for Env" TO SQL-stmt
       MOVE SQL-HANDLE-ENV to DiagHandleType
       SET DiagHandle to Henv
       PERFORM SQLDiag-Function
     END-IF.
*
  Set-Env-Attr-to-Ver30-Behavior.
     CALL "SQLSetEnvAttr" USING
                            BY VALUE      Henv
                            BY VALUE      SQL-ATTR-ODBC-VERSION
                            BY VALUE      SQL-OV-ODBC3
*                                      or SQL-OV-ODBC2         *
*                                         for Ver 2.x behavior *
                            BY VALUE      SQL-IS-UINTEGER
                         RETURNING        SQL-RC
     IF SQL-RC NOT = SQL-SUCCESS
       MOVE "SQLSetEnvAttr" TO SQL-stmt
       MOVE SQL-HANDLE-ENV to DiagHandleType
       SET DiagHandle to Henv
       PERFORM SQLDiag-Function
     END-IF.
*
  Allocate-Connection-Handle.
     CALL "SQLAllocHandle" USING
                             By VALUE     SQL-HANDLE-DBC
                             BY VALUE     Henv
                             BY REFERENCE Hdbc
                           RETURNING      SQL-RC
     IF SQL-RC NOT = SQL-SUCCESS
         MOVE "SQLAllocHandle for Connection" to SQL-stmt
         MOVE SQL-HANDLE-ENV to DiagHandleType
         SET DiagHandle to Henv
         PERFORM SQLDiag-Function
     END-IF.
*** SQLAllocHandle for statement function SECTION ***************
 Allocate-Statement-Handle SECTION.
  Allocate-Stmt-Handle.
     CALL "SQLAllocHandle" USING
                            By VALUE     SQL-HANDLE-STMT
                            BY VALUE     Hdbc
                            BY REFERENCE Hstmt
                          RETURNING      SQL-RC
     IF SQL-RC NOT = SQL-SUCCESS
         MOVE "SQLAllocHandle for Stmt" TO SQL-stmt
         MOVE SQL-HANDLE-DBC to DiagHandleType
         SET DiagHandle to Hdbc
         PERFORM SQLDiag-Function
     END-IF.
*** Cleanup Functions SECTION ***********************************
 ODBC-Clean-Up SECTION.
*
  Free-Statement-Handle.
     CALL "SQLFreeHandle" USING
                           BY VALUE SQL-HANDLE-STMT
                           BY VALUE Hstmt
                         RETURNING  SQL-RC
     IF SQL-RC NOT = SQL-SUCCESS
         MOVE "SQLFreeHandle for Stmt" TO SQL-stmt
         MOVE SQL-HANDLE-STMT to DiagHandleType
         SET DiagHandle to Hstmt
         PERFORM SQLDiag-Function
     END-IF.
*
  SQLDisconnect-Function.
     CALL "SQLDisconnect" USING
                           BY VALUE Hdbc
                         RETURNING  SQL-RC
     IF SQL-RC NOT = SQL-SUCCESS
         MOVE "SQLDisconnect" TO SQL-stmt
         MOVE SQL-HANDLE-DBC to DiagHandleType
         SET DiagHandle to Hdbc
         PERFORM SQLDiag-Function
     END-IF.
*
  Free-Connection-Handle.
     CALL "SQLFreeHandle" USING
                           BY VALUE SQL-HANDLE-DBC
                           BY VALUE Hdbc
                         RETURNING  SQL-RC
     IF SQL-RC NOT = SQL-SUCCESS
        MOVE "SQLFreeHandle for DBC" TO SQL-stmt
         MOVE SQL-HANDLE-DBC to DiagHandleType
         SET DiagHandle to Hdbc
         PERFORM SQLDiag-Function
      END-IF.
*        Free-Environment-Handle.
     CALL "SQLFreeHandle" USING
                           BY VALUE SQL-HANDLE-ENV
                           BY VALUE Henv
                         RETURNING  SQL-RC
     IF SQL-RC NOT = SQL-SUCCESS
         MOVE "SQLFreeHandle for Env" TO SQL-stmt
         MOVE SQL-HANDLE-ENV to DiagHandleType
         SET DiagHandle to Henv
         PERFORM SQLDiag-Function
     END-IF.
*** SQLDiag function SECTION ************************************
```

```
 SQLDiag-Function SECTION.
  SQLDiag.
     MOVE SQL-RC TO SAVED-SQL-RC
     DISPLAY "Return Value = " SQL-RC
       IF SQL-RC = SQL-SUCCESS-WITH-INFO
         THEN
            DISPLAY SQL-stmt " successful with information"
         ELSE
            DISPLAY SQL-stmt " failed"
       END-IF
*    - get number of diagnostic records - *
     CALL "SQLGetDiagField"
             USING
               BY VALUE     DiagHandleType
                            DiagHandle
                            0
                            SQL-DIAG-NUMBER
               BY REFERENCE DiagRecNumber
               BY VALUE     SQL-IS-SMALLINT
               BY REFERENCE OMITTED
             RETURNING      SQL-RC
     IF SQL-RC = SQL-SUCCESS or SQL-SUCCESS-WITH-INFO
       THEN
*        - get each diagnostic record - *
         PERFORM WITH TEST AFTER
           VARYING DiagRecNumber-Index FROM 1 BY 1
             UNTIL DiagRecNumber-Index > DiagRecNumber
              or   SQL-RC NOT =
                     (SQL-SUCCESS or SQL-SUCCESS-WITH-INFO)
*          - get a diagnostic record - *
           CALL "SQLGetDiagRec"
                   USING
                     BY VALUE     DiagHandleType
                                  DiagHandle
                                  DiagRecNumber-Index
                     BY REFERENCE DiagSQLState
                                  DiagNativeError
                                  DiagMessageText
                     BY VALUE     DiagMessageBufferLength
                     BY REFERENCE DiagMessageTextLength
                   RETURNING          SQL-RC
           IF SQL-RC = SQL-SUCCESS OR SQL-SUCCESS-WITH-INFO
             THEN
               DISPLAY "Information from diagnostic record number"
                       " " DiagRecNumber-Index " for "
                       SQL-stmt ":"
               DISPLAY "  SQL-State = " DiagSQLState-Chars
               DISPLAY "  Native error code = " DiagNativeError
               DISPLAY "  Diagnostic message = "
                        DiagMessageText(1:DiagMessageTextLength)
             ELSE
               DISPLAY "SQLGetDiagRec request for " SQL-stmt
                       " failed with return code of: " SQL-RC
                       " from SQLError"
               PERFORM Termination
           END-IF
         END-PERFORM
       ELSE
*        - indicate SQLGetDiagField failed - *
         DISPLAY "SQLGetDiagField failed with return code of: "
                 SQL-RC
     END-IF
     MOVE Saved-SQL-RC to SQL-RC
     IF Saved-SQL-RC NOT = SQL-SUCCESS-WITH-INFO
       PERFORM Termination
     END-IF.
*** Termination Section******************************************
```

```
 Termination Section.
  Termination-Function.
     DISPLAY "Application being terminated with rollback"
     CALL "SQLTransact" USING BY VALUE henv
                                       hdbc
                                       SQL-ROLLBACK
                        RETURNING      SQL-RC
     IF SQL-RC = SQL-SUCCESS
       THEN
         DISPLAY "Rollback successful"
       ELSE
         DISPLAY "Rollback failed with return code of: "
                 SQL-RC
     END-IF
     STOP RUN.
*************************
* End of ODBC3P.CPY     *
*************************
```

Example: copybook for ODBC data definitions

The following example shows ODBC data definitions for items such as message text and return codes.

```
******************************************************************
* ODBC3D.CPY                 (ODBC Ver 3.0)                      *
*----------------------------------------------------------------*
* Data definitions to be used with sample ODBC function calls    *
*  and included in Working-Storage or Local-Storage Section      *
******************************************************************
* ODBC Handles
 01  Henv                       POINTER           VALUE NULL.
 01  Hdbc                       POINTER           VALUE NULL.
 01  Hstmt                      POINTER           VALUE NULL.
* Arguments used for GetDiagRec calls
 01  DiagHandleType             COMP-5  PIC 9(4).
 01  DiagHandle                 POINTER.
 01  DiagRecNumber              COMP-5  PIC 9(4).
 01  DiagRecNumber-Index        COMP-5  PIC 9(4).
 01  DiagSQLState.
     02 DiagSQLState-Chars              PIC X(5).
     02 DiagSQLState-Null               PIC X.
 01  DiagNativeError            COMP-5  PIC S9(9).
 01  DiagMessageText                    PIC X(511) VALUE SPACES.
 01  DiagMessageBufferLength    COMP-5  PIC S9(4)  VALUE 511.
 01  DiagMessageTextLength      COMP-5  PIC S9(4).
* Misc declarations used in sample function calls
 01  SQL-RC                     COMP-5  PIC S9(4)  VALUE 0.
 01  Saved-SQL-RC               COMP-5  PIC S9(4)  VALUE 0.
 01  SQL-stmt                           PIC X(30).
*************************
* End of ODBC3D.CPY     *
*************************
```

ODBC names truncated or abbreviated for COBOL

The following table shows ODBC names that are longer than 30 characters and shows their corresponding COBOL names.

Table 44. **ODBC names truncated or abbreviated for COBOL**

ODBC C `#define` symbol > 30 characters	Corresponding COBOL name
SQL_AD_ADD_CONSTRAINT_DEFERRABLE	SQL-AD-ADD-CONSTRAINT-DEFER
SQL_AD_ADD_CONSTRAINT_INITIALLY_DEFERRED	SQL-AD-ADD-CONSTRAINT-INIT-DEF

Table 44. **ODBC names truncated or abbreviated for COBOL** *(continued)*

ODBC C #define symbol > 30 characters	Corresponding COBOL name
SQL_AD_ADD_CONSTRAINT_INITIALLY_IMMEDIATE	SQL-AD-ADD-CONSTRAINT-INIT-IMM
SQL_AD_ADD_CONSTRAINT_NON_DEFERRABLE	SQL-AD-ADD-CONSTRAINT-NON-DEFE
SQL_AD_CONSTRAINT_NAME_DEFINITION	SQL-AD-CONSTRAINT-NAME-DEFINIT
SQL_AT_CONSTRAINT_INITIALLY_DEFERRED	SQL-AT-CONSTRAINT-INITIALLY-DE
SQL_AT_CONSTRAINT_INITIALLY_IMMEDIATE	SQL-AT-CONSTRAINT-INITIALLY-IM
SQL_AT_CONSTRAINT_NAME_DEFINITION	SQL-AT-CONSTRAINT-NAME-DEFINIT
SQL_AT_CONSTRAINT_NON_DEFERRABLE	SQL-AT-CONSTRAINT-NON-DEFERRAB
SQL_AT_DROP_TABLE_CONSTRAINT_CASCADE	SQL-AT-DROP-TABLE-CONSTRAINT-C
SQL_AT_DROP_TABLE_CONSTRAINT_RESTRICT	SQL-AT-DROP-TABLE-CONSTRAINT-R
SQL_C_INTERVAL_MINUTE_TO_SECOND	SQL-C-INTERVAL-MINUTE-TO-SECON
SQL_CA_CONSTRAINT_INITIALLY_DEFERRED	SQL-CA-CONSTRAINT-INIT-DEFER
SQL_CA_CONSTRAINT_INITIALLY_IMMEDIATE	SQL-CA-CONSTRAINT-INIT-IMMED
SQL_CA_CONSTRAINT_NON_DEFERRABLE	SQL-CA-CONSTRAINT-NON-DEFERRAB
SQL_CA1_BULK_DELETE_BY_BOOKMARK	SQL-CA1-BULK-DELETE-BY-BOOKMAR
SQL_CA1_BULK_UPDATE_BY_BOOKMARK	SQL-CA1-BULK-UPDATE-BY-BOOKMAR
SQL_CDO_CONSTRAINT_NAME_DEFINITION	SQL-CDO-CONSTRAINT-NAME-DEFINI
SQL_CDO_CONSTRAINT_INITIALLY_DEFERRED	SQL-CDO-CONSTRAINT-INITIALLY-D
SQL_CDO_CONSTRAINT_INITIALLY_IMMEDIATE	SQL-CDO-CONSTRAINT-INITIALLY-I
SQL_CDO_CONSTRAINT_NON_DEFERRABLE	SQL-CDO-CONSTRAINT-NON-DEFERRA
SQL_CONVERT_INTERVAL_YEAR_MONTH	SQL-CONVERT-INTERVAL-YEAR-MONT
SQL_CT_CONSTRAINT_INITIALLY_DEFERRED	SQL-CT-CONSTRAINT-INITIALLY-DE
SQL_CT_CONSTRAINT_INITIALLY_IMMEDIATE	SQL-CT-CONSTRAINT-INITIALLY-IM
SQL_CT_CONSTRAINT_NON_DEFERRABLE	SQL-CT-CONSTRAINT-NON-DEFERRAB
SQL_CT_CONSTRAINT_NAME_DEFINITION	SQL-CT-CONSTRAINT-NAME-DEFINIT
SQL_DESC_DATETIME_INTERVAL_CODE	SQL-DESC-DATETIME-INTERVAL-COD
SQL_DESC_DATETIME_INTERVAL_PRECISION	SQL-DESC-DATETIME-INTERVAL-PRE
SQL_DL_SQL92_INTERVAL_DAY_TO_HOUR	SQL-DL-SQL92-INTERVAL-DAY-TO-H
SQL_DL_SQL92_INTERVAL_DAY_TO_MINUTE	SQL-DL-SQL92-INTERVAL-DAY-TO-M
SQL_DL_SQL92_INTERVAL_DAY_TO_SECOND	SQL-DL-SQL92-INTERVAL-DAY-TO-S
SQL_DL_SQL92_INTERVAL_HOUR_TO_MINUTE	SQL-DL-SQL92-INTERVAL-HR-TO-M
SQL_DL_SQL92_INTERVAL_HOUR_TO_SECOND	SQL-DL-SQL92-INTERVAL-HR-TO-S
SQL_DL_SQL92_INTERVAL_MINUTE_TO_SECOND	SQL-DL-SQL92-INTERVAL-MIN-TO-S
SQL_DL_SQL92_INTERVAL_YEAR_TO_MONTH	SQL-DL-SQL92-INTERVAL-YR-TO-MO
SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1	SQL-FORWARD-ONLY-CURSOR-ATTR1
SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2	SQL-FORWARD-ONLY-CURSOR-ATTR2
SQL_GB_GROUP_BY_CONTAINS_SELECT	SQL-GB-GROUP-BY-CONTAINS-SELEC
SQL_ISV_CONSTRAINT_COLUMN_USAGE	SQL-ISV-CONSTRAINT-COLUMN-USAG
SQL_ISV_REFERENTIAL_CONSTRAINTS	SQL-ISV-REFERENTIAL-CONSTRAINT
SQL_MAXIMUM_CATALOG_NAME_LENGTH	SQL-MAXIMUM-CATALOG-NAME-LENGT
SQL_MAXIMUM_COLUMN_IN_GROUP_BY	SQL-MAXIMUM-COLUMN-IN-GROUP-B

Table 44. **ODBC names truncated or abbreviated for COBOL** *(continued)*

ODBC C #define symbol > 30 characters	Corresponding COBOL name
SQL_MAXIMUM_COLUMN_IN_ORDER_BY	SQL-MAXIMUM-COLUMN-IN-ORDER-B
SQL_MAXIMUM_CONCURRENT_ACTIVITIES	SQL-MAXIMUM-CONCURRENT-ACTIVIT
SQL_MAXIMUM_CONCURRENT_STATEMENTS	SQL-MAXIMUM-CONCURRENT-STAT
SQL_SQL92_FOREIGN_KEY_DELETE_RULE	SQL-SQL92-FOREIGN-KEY-DELETE-R
SQL_SQL92_FOREIGN_KEY_UPDATE_RULE	SQL-SQL92-FOREIGN-KEY-UPDATE-R
SQL_SQL92_NUMERIC_VALUE_FUNCTIONS	SQL-SQL92-NUMERIC-VALUE-FUNCTI
SQL_SQL92_RELATIONAL_JOIN_OPERATORS	SQL-SQL92-RELATIONAL-JOIN-OPER
SQL_SQL92_ROW_VALUE_CONSTRUCTOR	SQL-SQL92-ROW-VALUE-CONSTRUCTO
SQL_TRANSACTION_ISOLATION_OPTION	SQL-TRANSACTION-ISOLATION-OPTI

Compiling and linking programs that make ODBC calls

You must compile programs that make ODBC calls with the compiler option PGMNAME(MIXED) in effect. (The ODBC entry points are case sensitive.)

RELATED REFERENCES
"PGMNAME" on page 276

Part 5. Using XML and COBOL together

Chapter 21. Processing XML input 361
XML parser in COBOL 361
Accessing XML documents 363
Parsing XML documents 363
    Writing procedures to process XML 365
| XML-EVENT 366
    XML-CODE 366
    XML-TEXT and XML-NTEXT 367
    Transforming XML text to COBOL data items 367
    Example: program for processing XML 368
        Output from parse example 370
The encoding of XML documents. 371
XML input document encoding 371
    Specifying the code page 372
| EBCDIC code-page-sensitive characters in XML
| markup 373
| Parsing XML documents encoded in UTF-8 . . . 374
Handling XML PARSE exceptions 374
    How the XML parser handles errors. 376
    Handling conflicts in code pages 376
Terminating XML parsing 377

Chapter 22. Producing XML output 379
Generating XML output 379
Controlling the encoding of generated XML output 383
Handling XML GENERATE exceptions 384
Example: generating XML 385
    Program XGFX 385
    Program Pretty 386
    Output from program XGFX 389
Enhancing XML output 389
    Example: enhancing XML output 390
    Example: converting hyphens in element or
    attribute names to underscores 393

Chapter 21. Processing XML input

You can process XML input in your COBOL program by using the XML PARSE statement.

The XML PARSE statement is the COBOL language interface to the high-speed XML parser that is part of the COBOL run time.

Processing XML input involves passing control to, and receiving control from, the XML parser. You start this exchange of control by using the XML PARSE statement, which specifies a processing procedure that receives control from the XML parser to handle the parser events.

You use special registers in your processing procedure to exchange information with the parser.

Use the following COBOL facilities to process XML input:

- The XML PARSE statement to begin XML parsing and to identify the document and your processing procedure
- Your processing procedure to control the parsing: receive and process XML events and associated document fragments, and return to the parser for continued processing
- Special registers to receive and pass information:
  - XML-CODE to receive the status of XML parsing and, in some cases, to return information to the parser
  - XML-EVENT to receive the name of each XML event from the parser
  - XML-NTEXT to receive XML document fragments that are returned as national character data
  - XML-TEXT to receive document fragments that are returned as alphanumeric data

RELATED CONCEPTS
"XML parser in COBOL"
"XML input document encoding" on page 371

RELATED TASKS
"Accessing XML documents" on page 363
"Parsing XML documents" on page 363
"Parsing XML documents encoded in UTF-8" on page 374
"Handling XML PARSE exceptions" on page 374
"Terminating XML parsing" on page 377

RELATED REFERENCES
Appendix G, "XML reference material," on page 639
Extensible Markup Language (XML)

XML parser in COBOL

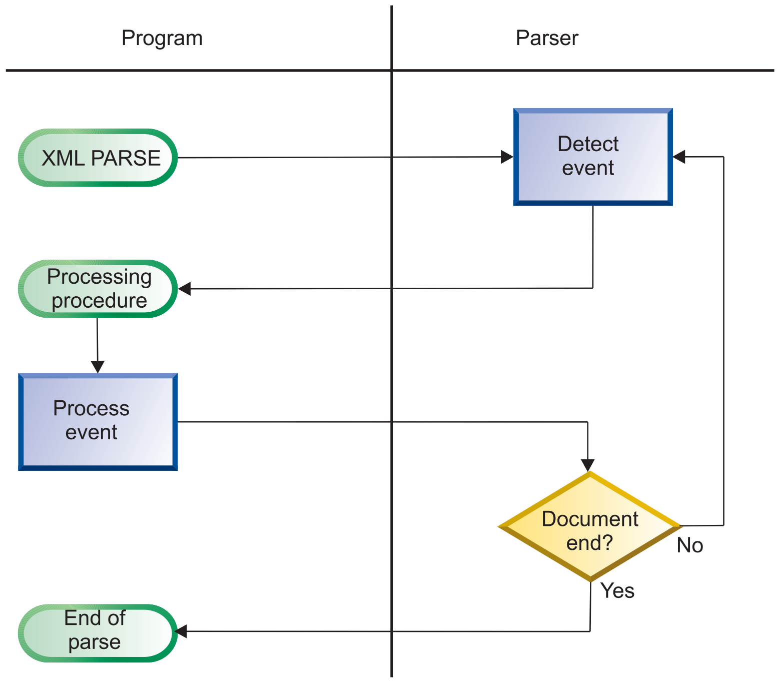
COBOL for AIX provides an event-based interface that lets you parse XML documents and transform them to COBOL data structures.

The XML parser finds fragments within the document, and your processing procedure acts on those fragments. The fragments are associated with XML events. You code your processing procedure to handle each XML event. Throughout this operation, control passes back and forth between the parser and the processing procedure.

You start this exchange with the parser by using the XML PARSE statement, in which you designate your processing procedure. 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.

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 program.



Normally, parsing continues until the entire XML document has been parsed.

When the XML parser parses XML documents, it checks them for most aspects of well formedness. A document is *well formed* if it adheres to the XML syntax in the *XML specification* and follows some additional rules such as proper use of end tags and uniqueness of attribute names.

RELATED CONCEPTS
"XML input document encoding" on page 371

RELATED TASKS
"Accessing XML documents" on page 363

"Parsing XML documents"
"Writing procedures to process XML" on page 365
"Handling XML PARSE exceptions" on page 374
"Terminating XML parsing" on page 377

RELATED REFERENCES
"XML conformance" on page 646
XML specification

Accessing XML documents

Before you can parse an XML document using an XML PARSE statement, you must make the document available to your program. Common methods of acquiring an XML document are from a parameter to your program or by reading the document from a file.

If the XML document that you want to parse is held in a file, use ordinary COBOL facilities to place the document into a data item in your program:

- A FILE-CONTROL entry to define the file to your program.
- An OPEN statement to open the file.
- READ statements to read all the records from the file into a data item (either an elementary item of category alphanumeric or national, or an alphanumeric or national group). You can define the data item in the WORKING-STORAGE SECTION or the LOCAL-STORAGE SECTION.
- Optionally, the STRING statement to string all of the separate records together into one continuous stream, to remove extraneous blanks, and to handle variable-length records.

Parsing XML documents

To parse XML documents, use the XML PARSE statement, specifying the XML document that is to be parsed and the processing procedure for handling XML events that occur during parsing, as shown in the following code fragment.

```
XML PARSE xml-document
    PROCESSING PROCEDURE xml-event-handler
  ON EXCEPTION
     DISPLAY 'XML document error ' XML-CODE
     STOP RUN
  NOT ON EXCEPTION
     DISPLAY 'XML document was successfully parsed.'
END-XML
```

In the XML PARSE statement, you first identify the *parse data item* (xml-document in the example above) that contains the XML document character stream. In the DATA DIVISION, define the parse data item as an elementary data item of category national or as a national group item if the encoding of the document is Unicode UTF-16; otherwise, define the parse data item as an alphanumeric group item or an elementary alphanumeric data item:

- If the parse data item is national, the XML document must be encoded in UTF-16 in big-endian format.
- If the parse data item is alphanumeric, its content must be encoded in one of the supported code pages described in the related reference below about the encoding of XML documents.

For more information about code pages, see the related concept below about XML input document encoding.

Next you specify the name of the processing procedure (`xml-event-handler` in the example above) that is to handle the XML events from the document.

In addition, you can optionally specify either or both of the following 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 has been parsed, as indicated by the `END-OF-DOCUMENT` event.
- The parser detects an error in the document and signals an `EXCEPTION` event, and the processing procedure does not reset the special register `XML-CODE` to zero before returning to the parser.
- You terminate the parsing process deliberately by setting the `XML-CODE` special register to -1 before returning to the parser.

Special registers:

Use the `XML-EVENT` special register to determine which event the parser passed to your processing procedure. `XML-EVENT` contains an event name, such as `'START-OF-ELEMENT'`. The parser passes the content for the event in special register `XML-TEXT` or `XML-NTEXT`.

RELATED CONCEPTS
"XML-EVENT" on page 366
"XML-CODE" on page 366
"XML input document encoding" on page 371

RELATED TASKS
"Specifying the code page for character data" on page 194
"Writing procedures to process XML" on page 365
"Parsing XML documents encoded in UTF-8" on page 374

RELATED REFERENCES
"The encoding of XML documents" on page 371
"XML PARSE exceptions that allow continuation" on page 639
"XML PARSE exceptions that do not allow continuation" on page 643
XML PARSE statement (*COBOL for AIX 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, as shown in the following table. Use the content of these special registers to populate COBOL data structures and to control the processing.

When used in nested programs, these special registers are implicitly defined as `GLOBAL` in the outermost program.

Table 45. **Special registers used by the XML parser**

Special register	Implicit definition and usage	Content
`XML-EVENT`[1, 3]	`PICTURE X(30) USAGE DISPLAY VALUE SPACE`	The name of the XML event
`XML-CODE`[2]	`PICTURE S9(9) USAGE BINARY VALUE ZERO`	An exception code or zero for each XML event
	`XML-TEXT`[1, 4]	Variable-length elementary category alphanumeric item	Text (corresponding to the event that the parser encountered) from the XML document if you specify an alphanumeric item for the `XML PARSE` identifier
	`XML-NTEXT`[1]	Variable-length elementary category national item	Text (corresponding to the event that the parser encountered) from the XML document if you specify a national item for the `XML PARSE` identifier

1. You cannot use this special register as a receiving data item.
2. The `XML GENERATE` statement also uses `XML-CODE`. Therefore, if you have an `XML GENERATE` statement in the processing procedure, save the value of `XML-CODE` before the `XML GENERATE` statement, and restore the saved value after the `XML GENERATE` statement.
3. The content of this special register is encoded according to the setting of the `CHAR` compiler option (`EBCDIC`, `NATIVE`, or `S390`).
4. | The content of `XML-TEXT` has the encoding of the source XML document: ASCII or UTF-8 if the `CHAR(NATIVE)` compiler option is in effect; EBCDIC if `CHAR(EBCDIC)` is in effect.

Restrictions:

- A processing procedure must not directly execute an `XML PARSE` statement. However, if a processing procedure passes control to 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.

The compiler inserts a return mechanism after the last statement in each processing procedure. You can code a `STOP RUN` statement in a processing procedure to end the run unit.

"Example: program for processing XML" on page 368

RELATED CONCEPTS
"XML-EVENT" on page 366
"XML-CODE" on page 366
"XML-TEXT and XML-NTEXT" on page 367

RELATED TASKS
"Converting to or from national (Unicode) representation" on page 182
"Transforming XML text to COBOL data items" on page 367

RELATED REFERENCES
"CHAR" on page 250

XML-EVENT

For each event that occurs during XML parsing, the parser sets the associated event name in the `XML-EVENT` special register. The parser then passes the `XML-EVENT` special register to your processing procedure.

Depending on the event, the parser passes other special registers that 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.

The parser sets `XML-NTEXT` if the XML document is in a national data item or the parser finds a Unicode character reference; otherwise, the parser sets `XML-TEXT`.

When the parser detects an encoding conflict or a well-formedness error in the document, it sets `XML-EVENT` to `'EXCEPTION'`, and provides additional information about the exception in the `XML-CODE` special register.

For a detailed description of the set of XML events, see the related reference below about `XML-EVENT`.

RELATED CONCEPTS
"XML-CODE"

RELATED TASKS
"Parsing XML documents" on page 363

RELATED REFERENCES
XML-EVENT (*COBOL for AIX Language Reference*)

XML-CODE

When the parser returns control to your `XML PARSE` statement, special register `XML-CODE` contains the most recent value that was set by the parser (or by your processing procedure, if it sets `XML-CODE` to -1).

For each event except the `EXCEPTION` event, the value of `XML-CODE` is zero. If your processing procedure sets `XML-CODE` to -1 before returning control to the parser for an event other than `EXCEPTION`, processing stops with a user-initiated COBOL exception condition.

For the `EXCEPTION` event, special register `XML-CODE` contains the exception code.

For an encoding conflict exception, your processing procedure can, in some cases, reset `XML-CODE` to a valid value before returning to the parser. If you reset `XML-CODE` to any other nonzero value or reset it for any other exception, the parser sets `XML-CODE` to the original exception code.

The `XML GENERATE` statement also uses the `XML-CODE` special register. For details, see the related task about handling `XML GENERATE` exceptions.

RELATED TASKS
"Writing procedures to process XML" on page 365
"Handling XML PARSE exceptions" on page 374
"Handling XML GENERATE exceptions" on page 384

RELATED REFERENCES
Appendix G, "XML reference material," on page 639
XML-CODE (*COBOL for AIX Language Reference*)

XML-TEXT and XML-NTEXT

For most XML events, the parser sets `XML-TEXT` or `XML-NTEXT` to an associated document fragment.

Typically, the parser sets `XML-TEXT` if the XML document is in an alphanumeric data item. The parser sets `XML-NTEXT` if:

- The XML document is in a national data item.
- The XML document is in an alphanumeric data item and the `ATTRIBUTE-NATIONAL-CHARACTER` or `CONTENT-NATIONAL-CHARACTER` event occurs.

The special registers `XML-TEXT` and `XML-NTEXT` are mutually exclusive. When the parser sets `XML-TEXT`, `XML-NTEXT` is empty with length zero. When the parser sets `XML-NTEXT`, `XML-TEXT` is empty with length zero.

To determine the number of national characters 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 national characters 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.

RELATED CONCEPTS
"XML-EVENT" on page 366
"XML-CODE" on page 366

RELATED TASKS
"Writing procedures to process XML" on page 365

RELATED REFERENCES
XML-TEXT (*COBOL for AIX Language Reference*)
XML-NTEXT (*COBOL for AIX Language Reference*)

Transforming XML text to COBOL data items

Because XML data is neither fixed length nor fixed format, you need to use special techniques when you move XML data to COBOL data items.

For alphanumeric items, decide whether the XML data should go at the left (default) end of a COBOL item or at the right end. If the data should go at the right end, specify the `JUSTIFIED RIGHT` clause in the declaration of the COBOL item.

Give special consideration to numeric XML values, particularly "decorated" monetary values such as '$1,234.00' or '$1234'. These two strings mean the same thing in XML, but need quite different declarations 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 is redefined appropriately as a numeric-edited item. Then do the final move to a numeric (operational) item by moving from, and thus de-editing, the numeric-edited item. (A regular format would have the same number of digits after the decimal point, a comma separator for values greater than 999, and so on.)
- For simplicity and vastly increased flexibility, use the following intrinsic functions for alphanumeric XML data:
  - NUMVAL to extract and decode simple numeric values from XML data that represents plain numbers
  - NUMVAL-C to extract and decode numeric values from XML data that represents monetary quantities

  However, using these functions is at the expense of performance.

RELATED TASKS
"Converting to numbers (NUMVAL, NUMVAL-C)" on page 105
"Using national data (Unicode) in COBOL" on page 174
"Writing procedures to process XML" on page 365

Example: program for processing XML

| This example shows the use of an XML PARSE statement and a processing
| procedure.

The XML document is shown in the source so that you can follow the flow of the parsing. The output of the program is shown after the example. Compare the document to the output of the program to follow the interaction of the parser and the processing procedure, and to match events to document fragments.

```
 Identification division.
   Program-id. XMLSAMPL.

 Data division.
  Working-storage section.
******************************************************************
* XML document, encoded as initial values of data items.        *
******************************************************************
   1 xml-document.
|    2 pic x(39) value '<?xml version="1.0" encoding="UTF-8"'.
     2 pic x(19) value ' standalone="yes"?>'.
     2 pic x(39) value '<!--This document is just an example-->'.
     2 pic x(10) value '<sandwich>'.
     2 pic x(35) value '  <bread type="baker&apos;s best"/>'.
     2 pic x(41) value '  <?spread please use real mayonnaise  ?>'.
     2 pic x(31) value '  <meat>Ham &amp; turkey</meat>'.
     2 pic x(40) value '  <filling>Cheese, lettuce, tomato, etc.'.
     2 pic x(10) value '</filling>'.
     2 pic x(35) value '  <![CDATA[We should add a <relish>'.
     2 pic x(22) value ' element in future!]]>'.
     2 pic x(31) value '  <listprice>$4.99 </listprice>'.
     2 pic x(27) value '  <discount>0.10</discount>'.
     2 pic x(11) value '</sandwich>'.
   1 xml-document-length computational pic 999.

******************************************************************
* Sample data definitions for processing numeric XML content.    *
******************************************************************
   1 current-element pic x(30).
   1 xfr-ed pic x(9) justified.
   1 xfr-ed-1 redefines xfr-ed pic 999999.99.
   1 list-price computational pic 9v99 value 0.
   1 discount computational pic 9v99 value 0.
   1 display-price pic $$9.99.

 Procedure division.
  Mainline section.

     XML parse xml-document processing procedure xml-handler
       On exception
         Display 'XML document error ' XML-Code
       Not on exception
         Display 'XML document successfully parsed'
     End-XML

******************************************************************
*    Process the transformed content and calculate promo price.  *
******************************************************************
     Display ' '
     Display '-----+++++***** Using information from XML '
         '*****+++++-----'
     Display ' '
     Move list-price to display-price
     Display '  Sandwich list price: ' display-price
     Compute display-price = list-price * (1 - discount)
     Display '  Promotional price:   ' display-price
     Display '  Get one today!'

     Goback.

  xml-handler section.
     Evaluate XML-Event
* ==> Order XML events most frequent first
       When 'START-OF-ELEMENT'
         Display 'Start element tag: {' XML-Text '}'
         Move XML-Text to current-element
       When 'CONTENT-CHARACTERS'
         Display 'Content characters: {' XML-Text '}'
* ==> Transform XML content to operational COBOL data item...
         evaluate current-element
           When 'listprice'
* ==> Using function NUMVAL-C...
             Compute list-price = function numval-c(XML-Text)
           When 'discount'
* ==> Using de-editing of a numeric edited item...
             Move XML-Text to xfr-ed
             Move xfr-ed-1 to discount
         End-evaluate
       When 'END-OF-ELEMENT'
         Display 'End element tag: {' XML-Text '}'
         Move spaces to current-element
       When 'START-OF-DOCUMENT'
         Display 'Start of document'
       When 'END-OF-DOCUMENT'
         Display 'End of document.'
       When 'VERSION-INFORMATION'
         Display 'Version: {' XML-Text '}'
       When 'ENCODING-DECLARATION'
         Display 'Encoding: {' XML-Text '}'
       When 'STANDALONE-DECLARATION'
         Display 'Standalone: {' XML-Text '}'
       When 'ATTRIBUTE-NAME'
         Display 'Attribute name: {' XML-Text '}'
       When 'ATTRIBUTE-CHARACTERS'
         Display 'Attribute value characters: {' XML-Text '}'
       When 'ATTRIBUTE-CHARACTER'
```

```
            Display 'Attribute value character: {' XML-Text '}'
          When 'START-OF-CDATA-SECTION'
            Display 'Start of CData: {' XML-Text '}'
          When 'END-OF-CDATA-SECTION'
            Display 'End of CData: {' XML-Text '}'
          When 'CONTENT-CHARACTER'
            Display 'Content character: {' XML-Text '}'
          When 'PROCESSING-INSTRUCTION-TARGET'
            Display 'PI target: {' XML-Text '}'
          When 'PROCESSING-INSTRUCTION-DATA'
            Display 'PI data: {' XML-Text '}'
          When 'COMMENT'
            Display 'Comment: {' XML-Text '}'
          When 'EXCEPTION'
            Compute xml-document-length = function length (XML-Text)
            Display 'Exception ' XML-Code ' at offset '
                xml-document-length '.'
          When other
            Display 'Unexpected XML event: ' XML-Event '.'
        End-evaluate
        .
    End program XMLSAMPL.
```

Output from parse example

From the following output you can see which parsing events came from which fragments of the document:

```
Start of document
Version: {1.0}
Encoding: {UTF-8}
Standalone: {yes}
Comment: {This document is just an example}
Start element tag: {sandwich}
Content characters: {  }
Start element tag: {bread}
Attribute name: {type}
Attribute value characters: {baker}
Attribute value character: {'}
Attribute value characters: {s best}
End element tag: {bread}
Content characters: {  }
PI target: {spread}
PI data: {please use real mayonnaise  }
Content characters: {  }
Start element tag: {meat}
Content characters: {Ham }
Content character: {&}
Content characters: { turkey}
End element tag: {meat}
Content characters: {  }
Start element tag: {filling}
Content characters: {Cheese, lettuce, tomato, etc.}
End element tag: {filling}
Content characters: {  }
Start of CData: {<![CDATA[}
Content characters: {We should add a <relish> element in future!}
End of CData: {]]>}
Content characters: {  }
Start element tag: {listprice}
Content characters: {$4.99 }
End element tag: {listprice}
Content characters: {  }
Start element tag: {discount}
Content characters: {0.10}
End element tag: {discount}
End element tag: {sandwich}
End of document.
XML document successfully parsed

-----+++++++++ Using information from XML *********+-----

  Sandwich list price:  $4.99
  Promotional price:    $4.49
  Get one today!
```

The encoding of XML documents

XML documents must be encoded in one of the supported code pages described below.

XML documents that you parse using `XML PARSE` statements must be encoded, and XML documents that you create using `XML GENERATE` statements are encoded, as follows:

- Documents in national data items: Unicode UTF-16 in big-endian format
- Documents in native alphanumeric data items: Unicode UTF-8 or a single-byte ASCII code page that is supported by International Components for Unicode (ICU) conversion libraries

  A *native alphanumeric data item* is a category alphanumeric data item that is compiled with the `CHAR(NATIVE)` compiler option in effect or whose data description entry contains the `NATIVE` phrase.
- Documents in host alphanumeric data items: a single-byte EBCDIC code page that is supported by ICU conversion libraries

  A *host alphanumeric data item* is a category alphanumeric data item that is compiled with the `CHAR(EBCDIC)` compiler option in effect and whose data description entry does not contain the `NATIVE` phrase.

The encodings supported by the ICU conversion libraries are shown in the related reference below about the ICU converter explorer.

RELATED CONCEPTS
"XML input document encoding"

RELATED TASKS
"Specifying the code page" on page 372

RELATED REFERENCES
"CHAR" on page 250
International Components for Unicode: Converter Explorer

XML input document encoding

To parse an XML document using the `XML PARSE` statement, the document must be encoded in a supported encoding.

The supported encodings for a given parse operation depend on the type of the data item that contains the XML document. The parser supports the following types of data items and encodings:

- Category national data items with content that is encoded in Unicode UTF-16 in big-endian format
- Native alphanumeric data items with content that is encoded in Unicode UTF-8 or one of the supported single-byte ASCII code pages

- Host alphanumeric data items with content that is encoded in one of the supported single-byte EBCDIC code pages

The supported code pages are described in the related reference about the encoding of XML documents.

The parser determines the *actual document encoding* by examining the first few bytes of the XML document. If the actual document encoding is ASCII or EBCDIC, the parser needs specific code-page information to be able to parse correctly. This additional code-page information is acquired from the document encoding declaration or from the external code-page information.

The document encoding declaration is an optional part of the XML declaration at the beginning of the document. For details, see the related task about specifying the code page.

The *external code page* for ASCII XML documents (the *external ASCII code page*) is the code page indicated by the current runtime locale. The external code page for EBCDIC XML documents (the *external EBCDIC code page*) is one of these:

- The code page that you specified in the EBCDIC_CODEPAGE environment variable
- The default EBCDIC code page selected for the current runtime locale if you did not set the EBCDIC_CODEPAGE environment variable

| If the specified encoding is not one of the supported coded character sets, the parser signals an XML exception event before beginning the parse operation. If the actual document encoding does not match the specified encoding, the parser signals an appropriate XML exception after beginning the parse operation.

To parse an XML document that is encoded in an unsupported code page, first convert the document to national character data (UTF-16) by using the `NATIONAL-OF` intrinsic function. You can convert the individual pieces of document text that are passed to the processing procedure in special register `XML-NTEXT` back to the original code page by using the `DISPLAY-OF` intrinsic function.

RELATED TASKS
"Converting to or from national (Unicode) representation" on page 182
"Specifying the code page"
"Parsing XML documents encoded in UTF-8" on page 374
"Handling XML PARSE exceptions" on page 374

RELATED REFERENCES
"Locales and code pages that are supported" on page 197
"The encoding of XML documents" on page 371
"XML PARSE exceptions that allow continuation" on page 639
"XML PARSE exceptions that do not allow continuation" on page 643

Specifying the code page

| You can choose the way to specify the code page for parsing an XML document that is in an alphanumeric data item.

| The preferred way is to omit the encoding declaration from the document and to rely instead on the external code-page specification.

Omitting the encoding declaration makes it possible to more easily transmit an XML document between heterogeneous systems. (If you included an encoding declaration, you would need to update it to reflect any code-page translation imposed by the transmission process.)

The code page used for parsing an alphanumeric XML document that does not have an encoding declaration is the runtime code page.

You can instead specify an encoding declaration in the XML declaration with which most XML documents begin. For example:

```
<?xml version="1.0" encoding="ibm-1140"?>
```

Note that the XML parser generates an exception if it encounters an XML declaration that does not begin in the first byte of an XML document.

If you specify an encoding declaration, use one of the primary or alias code-page names that are supported by the ICU conversion libraries. The code-page names are shown in the related reference below about the ICU converter explorer.

For more information about the code pages that are supported for XML parsing, see the related reference below about the encoding of XML documents.

RELATED CONCEPTS
"XML input document encoding" on page 371

RELATED TASKS
"Parsing XML documents encoded in UTF-8" on page 374
"Handling conflicts in code pages" on page 376

RELATED REFERENCES
"Locales and code pages that are supported" on page 197
"The encoding of XML documents" on page 371
International Components for Unicode: Converter Explorer

EBCDIC code-page-sensitive characters in XML markup

Several special characters that are used in XML markup have different hexadecimal representations in different EBCDIC code pages.

The following table shows the special characters and their hexadecimal values for various EBCDIC code-page CCSIDs.

Table 46. **Hexadecimal values of special characters for various EBCDIC CCSIDs**

Character	1047	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149
[	X'AD'	X'BA'	X'63'	X'9E'	X'B5'	X'90'	X'4A'	X'B1'	X'90'	X'4A'	X'AE'
]	X'BD'	X'BB'	X'FC'	X'9F'	X'9F'	X'51'	X'5A'	X'BB'	X'B5'	X'5A'	X'9E'
!	X'5A'	X'5A'	X'4F'	X'4F'	X'4F'	X'4F'	X'BB'	X'5A'	X'4F'	X'4F'	X'4F'
|	X'4F'	X'4F'	X'BB'	X'BB'	X'BB'	X'BB'	X'4F'	X'4F'	X'BB'	X'BB'	X'BB'
#	X'7B'	X'7B'	X'7B'	X'4A'	X'63'	X'B1'	X'69'	X'7B'	X'B1'	X'7B'	X'7B'

Parsing XML documents encoded in UTF-8

You can parse XML documents that are encoded in UTF-8 in a manner similar to parsing other XML documents, except that some additional requirements apply.

To parse an XML document that is encoded in UTF-8, code the `XML PARSE` statement as you would normally for parsing XML documents:

```
XML PARSE xml-document
    PROCESSING PROCEDURE xml-event-handler
    . . .
END-XML
```

Observe the following additional requirements though:

- The parse data item (`xml-document` in the example above) must be category alphanumeric, and the `CHAR(EBCDIC)` compiler option must not be in effect.
- So that the XML document will be parsed as UTF-8 rather than ASCII, ensure that at least one of the following conditions applies:
  - The runtime locale is a UTF-8 locale.
  - The document contains an XML encoding declaration that specifies UTF-8 (`encoding="UTF-8"`).
  - The document starts with a UTF-8 byte order mark.
- The document must not contain any characters that have a Unicode scalar value that is greater than x'FFFF'. Use a character reference (`"&#x`*`hhhhh`*`;"`) for such characters.

The parser returns the XML document fragments in the alphanumeric special register `XML-TEXT`.

UTF-8 characters are encoded using a variable number of bytes per character. Most COBOL operations on alphanumeric data assume a single-byte encoding, in which each character is encoded in 1 byte. When you operate on UTF-8 characters as alphanumeric data, you must ensure that the data is processed correctly. Avoid operations (such as reference modification and moves that involve truncation) that can split a multibyte character between bytes. You cannot reliably use statements such as `INSPECT` to process multibyte characters in alphanumeric data.

RELATED CONCEPTS
"XML-TEXT and XML-NTEXT" on page 367

RELATED TASKS
"Processing UTF-8 data" on page 185
"Parsing XML documents" on page 363
"Specifying the code page" on page 372

RELATED REFERENCES
"CHAR" on page 250
"The encoding of XML documents" on page 371
XML PARSE statement (*COBOL for AIX Language Reference*)

Handling XML PARSE exceptions

The parser sets an exception code in the `XML-CODE` special register when it encounters an anomaly or error during parsing.

If the exception code that the XML parser passes in XML-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 event in your processing procedure, do these steps:
1. Check the contents of XML-CODE.
2. Handle the exception as appropriate.
3. Set XML-CODE to zero to indicate that you handled the exception.
4. Return control to the parser. The exception condition no longer exists.

You can handle exceptions this way only if the exception code that is passed in XML-CODE is within one of the following ranges, which indicate that an encoding conflict was detected:
- 50-99
- 100,001-165,535
- 200,001-265,535

You can do limited handling of exceptions for which the exception code passed in XML-CODE is within the range 1-49. After an exception in this range occurs, the parser does not signal any further normal events, except the END-OF-DOCUMENT event, even if you set XML-CODE to zero before returning. If you set XML-CODE to zero, the parser continues parsing the document and signals any exceptions that it finds. (Doing so can provide a useful way to discover multiple errors in the document.)

At the end of parsing after an exception in this range, control is passed to the statement that you specify in the ON EXCEPTION phrase, if any, otherwise to the end of the XML PARSE statement. The special register XML-CODE contains the code for the most recent exception set by the parser.

For all other exceptions, the parser signals no further events, and passes control to the statement that you specify in the ON EXCEPTION phrase. In this case, XML-CODE contains the original exception number even if you reset 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 that you specify in the ON EXCEPTION phrase. If you do not code an ON EXCEPTION phrase, control is transferred to the end of the XML PARSE statement.

If no unhandled exceptions occur before the end of parsing, control is passed to the statement that you specify in the NOT ON EXCEPTION phrase (normal end of parsing). If you do not code a NOT ON EXCEPTION phrase, control is passed to the end of the XML PARSE statement. The special register XML-CODE contains zero.

RELATED CONCEPTS
"XML-CODE" on page 366
"XML input document encoding" on page 371
"How the XML parser handles errors" on page 376

RELATED TASKS
"Writing procedures to process XML" on page 365
"Handling conflicts in code pages" on page 376

How the XML parser handles errors

When the XML parser detects an error in an XML document, it generates an XML exception event and passes control to your processing procedure.

The parser provides the following information in special registers:

- XML-EVENT contains 'EXCEPTION'.
- XML-CODE contains a numeric exception code. The exception codes are described in the related references below about XML PARSE exceptions.
- XML-TEXT or XML-NTEXT contains the document text up to and including the point where the exception was detected.

All other XML special registers are empty with length zero.

You might be able to handle the exception in your processing procedure and continue parsing if the numeric exception code is within one of the following ranges:

- 1-99
- 100,001-165,535
- 200,001-265,535

If the exception code has any other nonzero value, parsing cannot continue. The exceptions for encoding conflicts (50-99 and 300-399) are signaled before the parsing of the document begins. For these exceptions, XML-TEXT or XML-NTEXT is either of length zero or contains only the encoding declaration value from the document.

Exceptions in the range 1-49 are fatal errors according to the XML specification. Therefore the parser does not continue normal parsing even if you handle the exception. However, the parser does continue scanning for further errors until it reaches the end of the document or encounters an error that does not allow continuation. For these exceptions, the parser does not signal any further normal events except the END-OF-DOCUMENT event.

RELATED CONCEPTS
"XML-CODE" on page 366
"XML input document encoding" on page 371

RELATED TASKS
"Handling XML PARSE exceptions" on page 374
"Handling conflicts in code pages"
"Terminating XML parsing" on page 377

RELATED REFERENCES
"XML PARSE exceptions that allow continuation" on page 639
"XML PARSE exceptions that do not allow continuation" on page 643

Handling conflicts in code pages

Your processing procedure might be able to handle exceptions for certain document encoding conflicts.

Exception events in which the document item is alphanumeric and the exception code in XML-CODE is between 100,001 and 165,535 or between 200,001 and 265,535 indicate that the code page of the document (as specified by its encoding declaration) conflicts with the external code-page information.

In this special case, you can choose to parse using the code page of the document by subtracting 100,000 or 200,000 from the value in `XML-CODE` (depending on whether it is an EBCDIC code page or ASCII code page, respectively). For instance, if `XML-CODE` contains 101,140, the code page of the document is 1140. Alternatively, you can choose to parse 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 a code-page conflict exception event:

- If you set `XML-CODE` to zero, the parser uses the external ASCII code page or external EBCDIC code page, depending on whether the document data item is a native alphanumeric or host alphanumeric item, respectively.
- If you set `XML-CODE` to the code page of the document (that is, the original `XML-CODE` value minus 100,000 or 200,000, as appropriate), the parser uses the code page of the document. This is the only case 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 to the exception event.

RELATED CONCEPTS
"XML-CODE" on page 366
"XML input document encoding" on page 371
"How the XML parser handles errors" on page 376

RELATED TASKS
"Handling XML PARSE exceptions" on page 374

RELATED REFERENCES
"XML PARSE exceptions that allow continuation" on page 639
"XML PARSE exceptions that do not allow continuation" on page 643

Terminating XML parsing

| You can terminate parsing immediately, without processing any remaining XML
| document text, by setting `XML-CODE` to -1 in your processing procedure before returning to the parser from any normal XML event (that is, any event other than `EXCEPTION`).

You can use this technique when you have seen enough of the document or have detected some irregularity in the document that precludes further meaningful processing.

| In this case, the parser does not signal any further events, including the XML
| exception event. Control transfers to the `ON EXCEPTION` phrase of the `XML PARSE` statement, if specified. In the imperative statement of the `ON EXCEPTION` phrase, you can test whether `XML-CODE` is -1, which indicates that you terminated parsing deliberately. If you do not specify an `ON EXCEPTION` phrase, control transfers to the end of the `XML PARSE` statement.

You can also terminate parsing after any XML `EXCEPTION` event by returning to the parser without changing `XML-CODE`. The result is similar to the result of deliberate termination except that the parser returns to the `XML PARSE` statement with `XML-CODE` containing the original exception code.

RELATED CONCEPTS
"XML-CODE" on page 366
"How the XML parser handles errors" on page 376

RELATED TASKS
"Handling XML PARSE exceptions" on page 374

Chapter 22. Producing XML output

You can produce XML output from a COBOL program by using the `XML GENERATE` statement.

In the `XML GENERATE` statement, you identify the source and the output data items. You can optionally also identify:

- A field to receive a count of the XML characters generated
- The encoding for the generated XML document
- A *namespace* for the generated document
- A namespace prefix to qualify the start and end tag of each element, if you specify a namespace
- A statement to receive control if an exception occurs

Optionally, you can generate an XML declaration for the document, and can cause eligible source data items to be expressed as attributes in the output rather than as elements.

You can use the `XML-CODE` special register to determine the status of XML generation.

After you transform COBOL data items to XML, you can use the resulting XML output in various ways, such as deploying it in a Web service, writing it to a file, or passing it as a parameter to another program.

RELATED TASKS
"Generating XML output"
"Controlling the encoding of generated XML output" on page 383
"Handling XML GENERATE exceptions" on page 384
"Enhancing XML output" on page 389

RELATED REFERENCES
Extensible Markup Language (XML)
XML GENERATE statement (*COBOL for AIX 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 declare the receiving identifier as alphanumeric (either an alphanumeric group item or an elementary item of category alphanumeric) or as national (either a national group item or an elementary item of category national).

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 each elementary data item included in the generated XML document (provided that such a data item has a name other than FILLER and does not have an OCCURS clause in its data description entry) to be expressed as an attribute of the XML element that corresponds to its immediately superordinate data item, rather than as a child element.

For example, suppose that the XML GENERATE statement in the program excerpt above had instead been coded as follows:

```
XML Generate Doc from G with attributes
```

The code would then generate the following XML document, in which A and E are expressed as attributes of element G, and C and D become attributes of element B:

```
<G A="aaa" E="eee"><B C="ccc" D="ddd"></B></G>
```

Optionally, you can code the ENCODING phrase of the XML GENERATE statement to specify the encoding of the generated XML document. If you do not use the ENCODING phrase, the document encoding is determined by the category of the receiving data item. For further details, see the related task below about controlling the encoding of generated XML output.

Optionally, you can code the XML-DECLARATION phrase to cause the generated XML document to have an XML declaration that includes version information and an encoding declaration. If the receiving data item is of category:

- National: The encoding declaration has the value UTF-16 (encoding="UTF-16").
- Alphanumeric: The encoding declaration is derived from the ENCODING phrase, if specified, or from the runtime locale or EBCDIC_CODEPAGE environment variable if the ENCODING phrase is not specified.

For example, the program excerpt below specifies the XML-DECLARATION phrase of XML GENERATE, and specifies encoding in UTF-8:

```
01  Greeting.
    05 msg  pic x(80)  value 'Hello, world!'.
    . . .
    XML Generate Doc from Greeting
        with Encoding "UTF-8"
        with XML-declaration
    End-XML
```

The code above generates the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?><Greeting><msg>Hello, world!</msg></Greeting>
```

If you do not code the XML-DECLARATION phrase, an XML declaration is not generated.

Optionally, you can code the NAMESPACE phrase to specify a *namespace* for the generated XML document. The namespace value must be a valid *Uniform Resource Identifier (URI)*, for example, a URL (Uniform Resource Locator); for further details, see the related concept about URI syntax below.

Specify the namespace in an identifier or literal of either category national or alphanumeric.

If you specify a namespace, but do not specify a namespace prefix (described below), the namespace becomes the *default namespace* for the document. That is, the namespace declared 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>
```

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 385

RELATED CONCEPTS
Uniform Resource Identifier (URI): Generic Syntax

RELATED TASKS
"Controlling the encoding of generated XML output"
"Handling XML GENERATE exceptions" on page 384
"Processing UTF-8 data" on page 185

RELATED REFERENCES
XML GENERATE statement (*COBOL for AIX Language Reference*)
Extensible Markup Language (XML)
Namespaces in XML 1.0

Controlling the encoding of generated XML output

When you generate XML output by using the XML GENERATE statement, you can control the encoding of the output by the category of the data item that receives the output, and by identifying the document encoding using the WITH ENCODING phrase of the XML GENERATE statement.

If you specify the WITH ENCODING *codepage* phrase, *codepage* must identify one of the code pages supported for COBOL XML processing as described in the related reference below about the encoding of XML documents. If *codepage* is an integer, it must be a valid CCSID number. If *codepage* is of class alphanumeric or national, it must identify a code-page name that is supported by the International Components for Unicode (ICU) conversion libraries as shown in the converter explorer table referenced below.

If you do not code the WITH ENCODING phrase, the generated XML output is encoded as shown in the table below.

Table 47. **Encoding of generated XML if the ENCODING phrase is omitted**

If you define the receiving XML identifier as:	The generated XML output is encoded in:
Native alphanumeric (CHAR(EBCDIC) is not in effect, or the data description contains the NATIVE phrase)	The ASCII or UTF-8 code page indicated by the runtime locale in effect
Host alphanumeric (CHAR(EBCDIC) is in effect, and the data description does not contain the NATIVE phrase)	The EBCDIC code page in effect[1]
National	UTF-16 in big-endian format

1. You can set the EBCDIC code page by using the EBCDIC_CODEPAGE environment variable. If the environment variable is not set, the encoding is in the default EBCDIC code page associated with the current runtime locale.

A byte order mark is not generated.

For details about how data items are converted to XML and how the XML element names and attributes names are formed from the COBOL data-names, see the related reference below about the operation of the XML GENERATE statement.

RELATED TASKS
Chapter 11, "Setting the locale," on page 193
"Setting environment variables" on page 209

RELATED REFERENCES
"CHAR" on page 250
"The encoding of XML documents" on page 371
XML GENERATE statement (*COBOL for AIX Language Reference*)
Operation of XML GENERATE (*COBOL for AIX Language Reference*)
International Components for Unicode: Converter Explorer

Handling XML GENERATE exceptions

When an error is detected during generation of XML output, an exception condition exists. You can write code to check the XML-CODE special register, which contains a numeric exception code that indicates the error type.

To handle errors, use either or both of the following phrases of the XML GENERATE statement:
- ON EXCEPTION
- COUNT IN

If you code the ON EXCEPTION phrase in the XML GENERATE statement, control is transferred to the imperative statement that you specify. You might code an imperative statement, for example, to display the XML-CODE value. If you do not code an ON EXCEPTION phrase, control is transferred to the end of the XML GENERATE statement.

When an error occurs, one problem might be that the data item that receives the XML output is not large enough. In that case, the XML output is not complete, and the XML-CODE special register contains error code 400.

You can examine the generated XML output by doing these steps:
1. Code the COUNT IN phrase in the XML GENERATE statement.

   The count field that you specify holds a count of the XML character encoding units that are filled during XML generation. If you define the XML output as national, the count is in UTF-16 character encoding units; for all other encodings (including for UTF-8), the count is in bytes.
2. Use the count field as a reference modification length to refer to the substring of the receiving data item that contains the XML characters that were generated until the point when the error occurred.

   For example, if XML-OUTPUT is the data item that receives the XML output, and XML-CHAR-COUNT is the count field, then XML-OUTPUT(1:XML-CHAR-COUNT) references the XML output.

Use the contents of XML-CODE to determine what corrective action to take. For a list of the exceptions that can occur during XML generation, see the related reference below.

RELATED TASKS
"Referring to substrings of data items" on page 99

RELATED REFERENCES
"XML generation" on page 570
"XML GENERATE exceptions" on page 649
XML-CODE (*COBOL for AIX Language Reference*)

Example: generating XML

The following example simulates the building of a purchase order in a group data item, and generates an XML version of that purchase order.

Program XGFX uses `XML GENERATE` to produce XML output in elementary data item `xmlPO` from the source record, group data item `purchaseOrder`. Elementary data items in the source record are converted to character format as necessary, and the characters are inserted as the values of XML attributes whose names are derived from the data-names in the source record.

XGFX calls program `Pretty`, which uses the `XML PARSE` statement with processing procedure `p` to format the XML output with new lines and indentation so that the XML content can more easily be verified.

Program XGFX

```
Identification division.
  Program-id. XGFX.
Data division.
 Working-storage section.
   01 numItems pic 99 global.
   01 purchaseOrder global.
     05 orderDate pic x(10).
     05 shipTo.
       10 country pic xx value 'US'.
       10 name pic x(30).
       10 street pic x(30).
       10 city pic x(30).
       10 state pic xx.
       10 zip pic x(10).
     05 billTo.
       10 country pic xx value 'US'.
       10 name pic x(30).
       10 street pic x(30).
       10 city pic x(30).
       10 state pic xx.
       10 zip pic x(10).
     05 orderComment pic x(80).
     05 items occurs 0 to 20 times depending on numItems.
       10 item.
         15 partNum pic x(6).
         15 productName pic x(50).
         15 quantity pic 99.
         15 USPrice pic 999v99.
         15 shipDate pic x(10).
         15 itemComment pic x(40).
   01 numChars comp pic 999.
   01 xmlPO pic x(999).
Procedure division.
  m.
    Move 20 to numItems
    Move spaces to purchaseOrder
    Move '1999-10-20' to orderDate

    Move 'US' to country of shipTo
    Move 'Alice Smith' to name of shipTo
    Move '123 Maple Street' to street of shipTo
    Move 'Mill Valley' to city of shipTo
    Move 'CA' to state of shipTo
    Move '90952' to zip of shipTo

    Move 'US' to country of billTo
    Move 'Robert Smith' to name of billTo
    Move '8 Oak Avenue' to street of billTo
    Move 'Old Town' to city of billTo
    Move 'PA' to state of billTo
    Move '95819' to zip of billTo
    Move 'Hurry, my lawn is going wild!' to orderComment

    Move 0 to numItems
    Call 'addFirstItem'
    Call 'addSecondItem'
    Move space to xmlPO
    Xml generate xmlPO from purchaseOrder  count in numChars
        with xml-declaration  with attributes
        namespace 'http://www.example.com'  namespace-prefix 'po'
    Call 'pretty' using xmlPO value numChars
    Goback
    .

Identification division.
  Program-id. 'addFirstItem'.
Procedure division.
    Add 1 to numItems
    Move '872-AA' to partNum(numItems)
    Move 'Lawnmower' to productName(numItems)
    Move 1 to quantity(numItems)
    Move 148.95 to USPrice(numItems)
    Move 'Confirm this is electric' to itemComment(numItems)
    Goback.
End program 'addFirstItem'.

Identification division.
  Program-id. 'addSecondItem'.
Procedure division.
    Add 1 to numItems
    Move '926-AA' to partNum(numItems)
    Move 'Baby Monitor' to productName(numItems)
    Move 1 to quantity(numItems)
    Move 39.98 to USPrice(numItems)
    Move '1999-05-21' to shipDate(numItems)
    Goback.
End program 'addSecondItem'.

End program XGFX.
```

Program Pretty

```
Identification division.
  Program-id. Pretty.
Data division.
 Working-storage section.
   01 prettyPrint.
     05 pose pic 999.
     05 posd pic 999.
     05 depth pic 99.
     05 inx pic 999.
     05 elementName pic x(30).
     05 indent pic x(40).
```

```
           05 buffer pic x(998).
           05 lastitem pic 9.
             88 unknown value 0.
             88 xml-declaration value 1.
             88 element value 2.
             88 attribute value 3.
             88 charcontent value 4.
       Linkage section.
        1 doc.
         2 pic x occurs 16384 times depending on len.
        1 len comp-5 pic 9(9).
      Procedure division using doc value len.
        m.
          Move space to prettyPrint
          Move 0 to depth
          Move 1 to posd pose
          Xml parse doc processing procedure p
          Goback
          .
        p.
          Evaluate xml-event
            When 'VERSION-INFORMATION'
              String '<?xml version="' xml-text '"' delimited by size
                  into buffer with pointer posd
              Set xml-declaration to true
            When 'ENCODING-DECLARATION'
              String ' encoding="' xml-text '"' delimited by size
                  into buffer with pointer posd
            When 'STANDALONE-DECLARATION'
              String ' standalone="' xml-text '"' delimited by size
                  into buffer with pointer posd
            When 'START-OF-ELEMENT'
              Evaluate true
                When xml-declaration
                  String '?>' delimited by size into buffer
                      with pointer posd
                  Set unknown to true
                  Perform printline
                  Move 1 to posd
                When element
                  String '>' delimited by size into buffer
                      with pointer posd
                When attribute
                  String '">' delimited by size into buffer
                      with pointer posd
              End-evaluate
              If elementName not = space
                Perform printline
              End-if
              Move xml-text to elementName
              Add 1 to depth
              Move 1 to pose
              Set element to true
              String '<' xml-text delimited by size
                  into buffer with pointer pose
              Move pose to posd
            When 'ATTRIBUTE-NAME'
              If element
                String ' ' delimited by size into buffer
                    with pointer posd
              Else
                String '" ' delimited by size into buffer
                    with pointer posd
              End-if
              String xml-text '="' delimited by size into buffer
                  with pointer posd
              Set attribute to true
            When 'ATTRIBUTE-CHARACTERS'
              String xml-text delimited by size into buffer
                  with pointer posd
            When 'ATTRIBUTE-CHARACTER'
              String xml-text delimited by size into buffer
                  with pointer posd
            When 'CONTENT-CHARACTERS'
              Evaluate true
                When element
                  String '>' delimited by size into buffer
                      with pointer posd
                When attribute
                  String '">' delimited by size into buffer
                      with pointer posd
              End-evaluate
              String xml-text delimited by size into buffer
                  with pointer posd
              Set charcontent to true
            When 'CONTENT-CHARACTER'
              Evaluate true
                When element
                  String '>' delimited by size into buffer
                      with pointer posd
                When attribute
                  String '">' delimited by size into buffer
                      with pointer posd
              End-evaluate
              String xml-text delimited by size into buffer
                  with pointer posd
              Set charcontent to true
            When 'END-OF-ELEMENT'
              Move space to elementName
              Evaluate true
                When element
                  String '/>' delimited by size into buffer
                      with pointer posd
                When attribute
                  String '"/>' delimited by size into buffer
                      with pointer posd
                When other
                  String '</' xml-text '>' delimited by size
                      into buffer with pointer posd
              End-evaluate
              Set unknown to true
              Perform printline
              Subtract 1 from depth
              Move 1 to posd
            When other
              Continue
          End-evaluate
          .
        printline.
          Compute inx = function max(0 2 * depth - 2) + posd - 1
          If inx > 120
            compute inx = 117 - function max(0 2 * depth - 2)
            If depth > 1
              Display indent(1:2 * depth - 2) buffer(1:inx) '...'
            Else
              Display buffer(1:inx) '...'
            End-if
          Else
            If depth > 1
              Display indent(1:2 * depth - 2) buffer(1:posd - 1)
            Else
              Display buffer(1:posd - 1)
```

```
              End-if
            End-if
            .
          End program Pretty.
```

Output from program XGFX

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<po:purchaseOrder xmlns:po="http://www.example.com" orderDate="1999-10-20" orderComment="Hurry, my lawn is going wild!">
  <po:shipTo country="US" name="Alice Smith" street="123 Maple Street" city="Mill Valley" state="CA" zip="90952"/>
  <po:billTo country="US" name="Robert Smith" street="8 Oak Avenue" city="Old Town" state="PA" zip="95819"/>
  <po:items>
    <po:item partNum="872-AA" productName="Lawnmower" quantity="1" USPrice="148.95" shipDate=" " itemComment="Confirm...
  </po:items>
  <po:items>
    <po:item partNum="926-AA" productName="Baby Monitor" quantity="1" USPrice="39.98" shipDate="1999-05-21" itemComme...
  </po:items>
</po:purchaseOrder>
```

RELATED TASKS
Chapter 21, "Processing XML input," on page 361

RELATED REFERENCES
Operation of XML GENERATE (*COBOL for AIX Language Reference*)

Enhancing XML output

It might happen that the information that you want to express in XML format already exists in a group item in the DATA DIVISION, but you are unable to use that item directly to generate an XML document because of one or more factors.

For example:

- In addition to the required data, the item has subordinate data items that contain values that are irrelevant to the XML output document.
- The names of the required data items are unsuitable for external presentation, and are possibly meaningful only to programmers.
- The definition of the data is not of the required data type. Perhaps only the redefinitions (which are ignored by the XML GENERATE statement) have the appropriate format.
- The required data items are nested too deeply within irrelevant subordinate groups. The XML output should be "flattened" rather than hierarchical as it would be by default.
- The required data items are broken up into too many components, and should be output as the content of the containing group.
- The group item contains the required information but in the wrong order.

There are various ways that you can deal with such situations. One possible technique is to define a new data item that has the appropriate characteristics, and move the required data to the appropriate fields of this new data item. However, this approach is somewhat laborious and requires careful maintenance to keep the original and new data items synchronized.

An alternative approach that has some advantages is to provide a redefinition of the original group data item, and to generate the XML output from that redefinition. To do so, start from the original set of data descriptions, and make these changes:

- Exclude elementary data items from the generated XML either by renaming them to FILLER or by deleting their names.

- Provide more meaningful and appropriate names for the selected elementary items and for the group items that contain them.
- Remove unneeded intermediate group items to flatten the hierarchy.
- Specify different data types to obtain the desired trimming behavior.
- Choose a different order for the output by using a sequence of XML GENERATE statements.

The safest way to accomplish these changes is to use another copy of the original declarations accompanied by one or more REPLACE compiler-directing statements. The example that is referenced below shows a way to do so.

"Example: enhancing XML output"

| You might also find when you generate an XML document that some of the
| element or attribute names and values contain hyphens. You might want to convert
| the hyphens in the element and attribute names to underscores without changing
| the hyphens that are in the element and attribute values. The example that is
| referenced below shows a way to do so.

"Example: converting hyphens in element or attribute names to underscores" on page 393

RELATED REFERENCES
Operation of XML GENERATE (*COBOL for AIX 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 X(02).
            25  CDR-LIFE-BP-DASH   PIC X(02).
            25  CDR-LIFE-BP-SPACE1 PIC X(02).
        20  CDR-LIFE-BASE-PRICE-ED  REDEFINES
             CDR-LIFE-BASE-PRICE  PIC $$$.$$.
        20  CDR-LIFE-BASE-QTY.
            25  CDR-LIFE-QTY-SPACE   PIC X(08).
            25  CDR-LIFE-QTY-DASH    PIC X(02).
            25  CDR-LIFE-QTY-SPACE1  PIC X(02).
            25  FILLER              PIC X(02).
        20  CDR-LIFE-BASE-QTY-ED    REDEFINES
                CDR-LIFE-BASE-QTY PIC ZZ,ZZZ,ZZZ.ZZZ.
        20  CDR-LIFE-BASE-VALUE   PIC X(15).
        20  CDR-LIFE-BASE-VALUE-ED  REDEFINES
                CDR-LIFE-BASE-VALUE
                                 PIC $(4),$$$,$$9.99.
    15  CDR-LIFE-BASE-TOT-VALUE-LINE.
        20  CDR-LIFE-BASE-TOT-VALUE   PIC X(15).
```

When this data structure is populated with some sample values, and XML is generated directly from it and then formatted using program `Pretty` (shown in "Example: generating XML" on page 385), the result is as follows:

```
<CDR-LIFE-BASE-VALUES-BOX>
  <CDR-LIFE-BASE-VAL-DATE>01/02/03</CDR-LIFE-BASE-VAL-DATE>
  <CDR-LIFE-BASE-VALUE-LINE>
    <CDR-LIFE-BASE-DESC>
      <CDR-LIFE-BASE-DESC1>First</CDR-LIFE-BASE-DESC1>
      <CDR-LIFE-BASE-LIT> </CDR-LIFE-BASE-LIT>
      <CDR-LIFE-BASE-DTE>01/01/01</CDR-LIFE-BASE-DTE>
    </CDR-LIFE-BASE-DESC>
    <CDR-LIFE-BASE-PRICE>
      <CDR-LIFE-BP-SPACE>$2</CDR-LIFE-BP-SPACE>
      <CDR-LIFE-BP-DASH>3.</CDR-LIFE-BP-DASH>
      <CDR-LIFE-BP-SPACE1>00</CDR-LIFE-BP-SPACE1>
    </CDR-LIFE-BASE-PRICE>
    <CDR-LIFE-BASE-QTY>
      <CDR-LIFE-QTY-SPACE>       1</CDR-LIFE-QTY-SPACE>
      <CDR-LIFE-QTY-DASH>23</CDR-LIFE-QTY-DASH>
      <CDR-LIFE-QTY-SPACE1>.0</CDR-LIFE-QTY-SPACE1>
    </CDR-LIFE-BASE-QTY>
    <CDR-LIFE-BASE-VALUE>        $765.00</CDR-LIFE-BASE-VALUE>
  </CDR-LIFE-BASE-VALUE-LINE>
  <CDR-LIFE-BASE-VALUE-LINE>
    <CDR-LIFE-BASE-DESC>
      <CDR-LIFE-BASE-DESC1>Second</CDR-LIFE-BASE-DESC1>
      <CDR-LIFE-BASE-LIT> </CDR-LIFE-BASE-LIT>
      <CDR-LIFE-BASE-DTE>02/02/02</CDR-LIFE-BASE-DTE>
    </CDR-LIFE-BASE-DESC>
    <CDR-LIFE-BASE-PRICE>
      <CDR-LIFE-BP-SPACE>$3</CDR-LIFE-BP-SPACE>
      <CDR-LIFE-BP-DASH>4.</CDR-LIFE-BP-DASH>
      <CDR-LIFE-BP-SPACE1>00</CDR-LIFE-BP-SPACE1>
    </CDR-LIFE-BASE-PRICE>
    <CDR-LIFE-BASE-QTY>
      <CDR-LIFE-QTY-SPACE>       2</CDR-LIFE-QTY-SPACE>
      <CDR-LIFE-QTY-DASH>34</CDR-LIFE-QTY-DASH>
      <CDR-LIFE-QTY-SPACE1>.0</CDR-LIFE-QTY-SPACE1>
    </CDR-LIFE-BASE-QTY>
    <CDR-LIFE-BASE-VALUE>        $654.00</CDR-LIFE-BASE-VALUE>
  </CDR-LIFE-BASE-VALUE-LINE>
  <CDR-LIFE-BASE-TOT-VALUE-LINE>
    <CDR-LIFE-BASE-TOT-VALUE>Very high!</CDR-LIFE-BASE-TOT-VALUE>
  </CDR-LIFE-BASE-TOT-VALUE-LINE>
</CDR-LIFE-BASE-VALUES-BOX>
```

This generated XML suffers from several problems:

- The element names are long and not very meaningful.
- There is unwanted data, for example, `CDR-LIFE-BASE-LIT` and `CDR-LIFE-BASE-DTE`.
- Required data has an unnecessary parent. For example, `CDR-LIFE-BASE-DESC1` has parent `CDR-LIFE-BASE-DESC`.
- Other required fields are split into too many subcomponents. For example, `CDR-LIFE-BASE-PRICE` has three subcomponents for one amount.

These and other characteristics of the XML output can be remedied by redefining the storage as follows:

```
1 BaseValues redefines CDR-LIFE-BASE-VALUES-BOX.
 2 BaseValueDate pic x(8).
 2 BaseValueLine occurs 2 times.
  3 Description pic x(15).
  3 pic x(9).
  3 BaseDate pic x(8).
  3 BasePrice pic x(6) justified.
  3 BaseQuantity pic x(14) justified.
  3 BaseValue pic x(15) justified.
 2 TotalValue pic x(15).
```

The result of generating and formatting XML from the set of definitions of the data values shown above is more usable:

```
<BaseValues>
  <BaseValueDate>01/02/03</BaseValueDate>
  <BaseValueLine>
    <Description>First</Description>
    <BaseDate>01/01/01</BaseDate>
    <BasePrice>$23.00</BasePrice>
    <BaseQuantity>123.000</BaseQuantity>
    <BaseValue>$765.00</BaseValue>
  </BaseValueLine>
  <BaseValueLine>
    <Description>Second</Description>
    <BaseDate>02/02/02</BaseDate>
    <BasePrice>$34.00</BasePrice>
    <BaseQuantity>234.000</BaseQuantity>
    <BaseValue>$654.00</BaseValue>
  </BaseValueLine>
  <TotalValue>Very high!</TotalValue>
</BaseValues>
```

You can redefine the original data definition directly, as shown above. However, it is generally safer to use the original definition but to modify it suitably using the text-manipulation capabilities of the compiler. An example is shown in the REPLACE compiler-directing statement below. This REPLACE statement might appear complicated, but it has the advantage of being self-maintaining if the original data definitions are modified.

```
replace ==CDR-LIFE-BASE-VALUES-BOX== by
            ==BaseValues redefines CDR-LIFE-BASE-VALUES-BOX==
        ==CDR-LIFE-BASE-VAL-DATE== by ==BaseValueDate==
        ==CDR-LIFE-BASE-VALUE-LINE== by ==BaseValueLine==
        ==20  CDR-LIFE-BASE-DESC.== by ====
        ==CDR-LIFE-BASE-DESC1== by ==Description==
        ==CDR-LIFE-BASE-LIT== by ====
        ==CDR-LIFE-BASE-DTE== by ==BaseDate==
        ==20  CDR-LIFE-BASE-PRICE.== by ====
        ==25  CDR-LIFE-BP-SPACE PIC X(02).== by ====
        ==25  CDR-LIFE-BP-DASH PIC X(02).== by ====
        ==25  CDR-LIFE-BP-SPACE1 PIC X(02).== by ====
        ==CDR-LIFE-BASE-PRICE-ED== by ==BasePrice==
        ==REDEFINES CDR-LIFE-BASE-PRICE PIC $$$.$$.== by
            ==pic x(6) justified.==
        ==20  CDR-LIFE-BASE-QTY.
            25  CDR-LIFE-QTY-SPACE PIC X(08).
            25  CDR-LIFE-QTY-DASH PIC X(02).
            25  CDR-LIFE-QTY-SPACE1 PIC X(02).
            25  FILLER PIC X(02).== by ====
        ==CDR-LIFE-BASE-QTY-ED== by ==BaseQuantity==
        ==REDEFINES CDR-LIFE-BASE-QTY PIC ZZ,ZZZ,ZZZ.ZZZ.== by
            ==pic x(14) justified.==
        ==CDR-LIFE-BASE-VALUE-ED== by ==BaseValue==
        ==20  CDR-LIFE-BASE-VALUE PIC X(15).== by ====
        ==REDEFINES CDR-LIFE-BASE-VALUE PIC $(4),$$$,$$9.99.==
            by ==pic x(15) justified.==
        ==CDR-LIFE-BASE-TOT-VALUE-LINE. 20== by ====
        ==CDR-LIFE-BASE-TOT-VALUE== by ==TotalValue==.
```

The result of this REPLACE statement followed by a second instance of the original set of definitions is similar to the suggested redefinition of group item BaseValues shown above. This REPLACE statement illustrates a variety of techniques for eliminating unwanted definitions and for modifying the definitions that should be retained. Use whichever technique is appropriate for your situation.

RELATED REFERENCES
Operation of XML GENERATE (*COBOL for AIX Language Reference*)
REPLACE statement (*COBOL for AIX Language Reference*)

Example: converting hyphens in element or attribute names to underscores

When you generate an XML document from a data structure whose items have data-names that contain hyphens, the generated XML has element or attribute names that contain hyphens. This example shows a way to convert hyphens that occur in element or attribute names to underscores without changing hyphens that occur in element or attribute values.

```
1 Customer-Record.
 2 Customer-Number  pic 9(9).
 2 First-Name       pic x(10).
 2 Last-Name        pic x(20).
```

When the data structure above is populated with some sample values, and XML is generated from it and then formatted using program Pretty (shown in “Example: generating XML” on page 385), the result might be as follows:

```
<Customer-Record>
   <Customer-Number>12345</Customer-Number>
   <First-Name>John</First-Name>
   <Last-Name>Smith-Jones</Last-Name>
</Customer-Record>
```

The element names contain hyphens, but the content of the element Last-Name also contains a hyphen.

Assuming that this XML document is the content of data item xmldoc, and that charcnt has been set to the length of the XML document, you can change all the hyphens in the element names to underscores but leave the element values unchanged by using the following code:

```
1 xmldoc              pic x(16384).
1 charcnt     comp-5  pic 9(5).
1 pos         comp-5  pic 9(5).
1 tagstate    comp-5  pic 9  value zero.
1 quotestate  comp-5  pic 9  value zero.

. . .
dash-to-underscore.
  perform varying pos from 1 by 1
          until pos > charcnt
    if xmldoc(pos:1) = '<'
      move 1 to tagstate
    end-if
    if tagstate = 1
      if xmldoc(pos:1) = '"'
        if quotestate = 0
          move 1 to quotestate
        else
          move 0 to quotestate
        end-if
      end-if
    end-if
    if tagstate = 1 and quotestate = 0 and xmldoc(pos:1) = '-'
      move '_' to xmldoc(pos:1)
    else
      if xmldoc(pos:1) = '>'
        move 0 to tagstate
      end-if
    end-if
  end-perform.
```

The revised XML document in data item `xmldoc` has underscores instead of hyphens in the element names. Hyphens in the element values have been preserved, though, as shown below:

```
<Customer_Record>
  <Customer_Number>12345</Customer_Number>
  <First_Name>John</First_Name>
  <Last_Name>Smith-Jones</Last_Name>
</Customer_Record>
```

Part 6. Developing object-oriented programs

Chapter 23. Writing object-oriented programs 397
Example: accounts 398
    Subclasses 399
Defining a class 400
    CLASS-ID paragraph for defining a class . . . 402
    REPOSITORY paragraph for defining a class 402
        Example: external class-names and Java packages 403
    WORKING-STORAGE SECTION for defining class instance data 404
    Example: defining a class 405
Defining a class instance method 405
    METHOD-ID paragraph for defining a class instance method 406
    INPUT-OUTPUT SECTION for defining a class instance method 407
    DATA DIVISION for defining a class instance method 407
    PROCEDURE DIVISION for defining a class instance method 408
    Overriding an instance method 409
    Overloading an instance method 410
    Coding attribute (get and set) methods 411
        Example: coding a get method. 411
    Example: defining a method 412
        Account class 412
        Check class 413
Defining a client 414
    REPOSITORY paragraph for defining a client 415
    DATA DIVISION for defining a client 416
        Choosing LOCAL-STORAGE or WORKING-STORAGE 416
    Comparing and setting object references . . . 417
    Invoking methods (INVOKE) 418
        USING phrase for passing arguments . . . 419
        Example: passing conforming object-reference arguments from a COBOL client 420
        RETURNING phrase for obtaining a returned value 421
        Invoking overridden superclass methods . . 422
    Creating and initializing instances of classes . . 422
        Instantiating Java classes 423
        Instantiating COBOL classes 423
    Freeing instances of classes 424
    Example: defining a client 424
Defining a subclass 425
    CLASS-ID paragraph for defining a subclass 426
    REPOSITORY paragraph for defining a subclass 427
    WORKING-STORAGE SECTION for defining subclass instance data 427
    Defining a subclass instance method 428
    Example: defining a subclass (with methods) 428
        CheckingAccount class (subclass of Account) 428
Defining a factory section 429
    WORKING-STORAGE SECTION for defining factory data 430
    Defining a factory method 431
        Hiding a factory or static method 432
        Invoking factory or static methods 432
    Example: defining a factory (with methods) . . 433
        Account class 434
        CheckingAccount class (subclass of Account) 435
        Check class 437
        TestAccounts client program 437
        Output produced by the TestAccounts client program 438
Wrapping procedure-oriented COBOL programs 438
Structuring OO applications 439
    Examples: COBOL applications that run using the java command. 439
        Displaying a message 439
        Echoing the input strings 440

Chapter 24. Communicating with Java methods 443
Accessing JNI services 443
    Handling Java exceptions 445
        Example: handling Java exceptions 445
    Managing local and global references 446
        Deleting, saving, and freeing local references 446
    Java access controls 447
Sharing data with Java 448
    Coding interoperable data types in COBOL and Java 448
    Declaring arrays and strings for Java 449
    Manipulating Java arrays 450
        Example: processing a Java int array . . . 452
    Manipulating Java strings 453

Chapter 23. Writing object-oriented programs

When you write an object-oriented (OO) program, you have to determine what classes you need and the methods and data that the classes need to do their work.

OO programs are based on *objects* (entities that encapsulate state and behavior) and their classes, methods, and data. A *class* is a template that defines the state and the capabilities of an object. Usually a program creates and works with multiple *object instances* (or simply, *instances*) of a class, that is, multiple objects that are members of that class. The state of each instance is stored in data known as *instance data*, and the capabilities of each instance are called *instance methods*. A class can define data that is shared by all instances of the class, known as *factory* or *static* data, and methods that are supported independently of any object instance, known as *factory* or *static* methods.

Using COBOL for AIX, you can:
- Define classes, with methods and data implemented in COBOL.
- Create instances of Java and COBOL classes.
- Invoke methods on Java and COBOL objects.
- Write classes that inherit from Java classes or other COBOL classes.
- Define and invoke overloaded methods.

In COBOL for AIX programs, you can call the services provided by the Java Native Interface (JNI) to obtain Java-oriented capabilities in addition to the basic OO capabilities available directly in the COBOL language.

In COBOL for AIX classes, you can code `CALL` statements to interface with procedural COBOL programs. Thus COBOL class definition syntax can be especially useful for writing *wrapper* classes for procedural COBOL logic, enabling existing COBOL code to be accessed from Java.

Java code can create instances of COBOL classes, invoke methods of these classes, and can extend COBOL classes.

Restrictions:
- COBOL class definitions and methods cannot contain `EXEC SQL` statements and cannot be compiled using the `SQL` compiler option.
- COBOL class definitions and methods cannot contain `EXEC CICS` statements, and cannot be run in a CICS environment. They cannot be compiled using the `CICS` compiler option.

"Example: accounts" on page 398

RELATED CONCEPTS
"Java SDK support" on page 241

RELATED TASKS
"Defining a class" on page 400
"Defining a class instance method" on page 405
"Defining a client" on page 414
"Defining a subclass" on page 425

"Defining a factory section" on page 429
Chapter 13, "Compiling, linking, and running OO applications," on page 237

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.

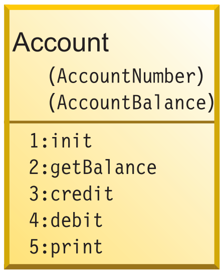
The following methods for an Account class meet those needs:

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

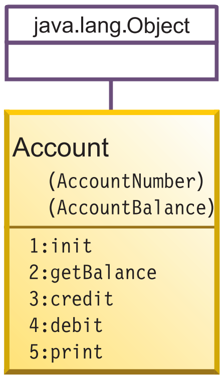
To keep the example simple, however, it is assumed that the account number and account balance are the only instance data that the Account class needs.

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.



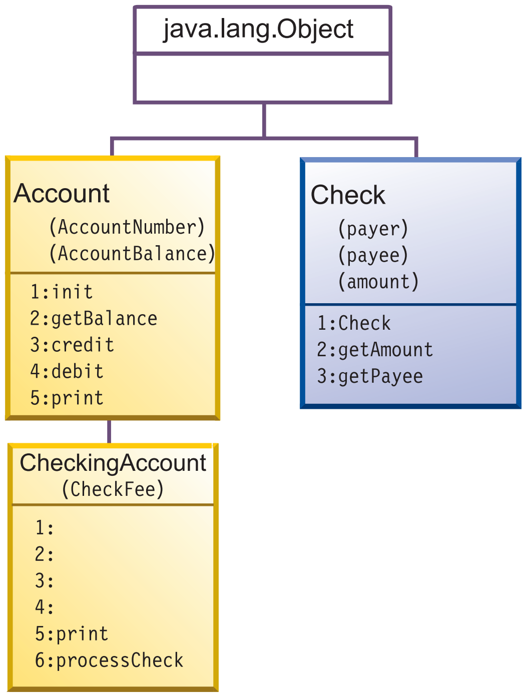
Subclasses

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 412

RELATED TASKS
"Defining a class"
"Defining a class instance method" on page 405
"Defining a subclass" on page 425

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 48. **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 402 (required) `AUTHOR` paragraph (optional) `INSTALLATION` paragraph (optional) `DATE-WRITTEN` paragraph (optional) `DATE-COMPILED` paragraph (optional)

Table 48. **Structure of class definitions** *(continued)*

Section	Purpose	Syntax
ENVIRONMENT DIVISION (required)	Describe the computing environment. Relate class-names used within the class definition to the corresponding external class-names known outside the compilation unit.	CONFIGURATION SECTION (required) "REPOSITORY paragraph for defining a class" on page 402 (required) SOURCE-COMPUTER paragraph (optional) OBJECT-COMPUTER paragraph (optional) SPECIAL-NAMES paragraph (optional)

Factory definition (optional)	Define data to be shared by all instances of the class, and methods supported independently of any object instance.	`IDENTIFICATION DIVISION.`
		`FACTORY.`
		`DATA DIVISION.`
		`WORKING-STORAGE SECTION.`
		`*   (Factory data here)`
		`PROCEDURE DIVISION.`
		`*   (Factory methods here)`
		`END FACTORY.`

Object definition (optional)	Define instance data and instance methods.	`IDENTIFICATION DIVISION.`
		`OBJECT.`
		`DATA DIVISION.`
		`WORKING-STORAGE SECTION.`
		`*   (Instance data here)`
		`PROCEDURE DIVISION.`
		`*   (Instance methods here)`
		`END OBJECT.`

If you specify the SOURCE-COMPUTER, OBJECT-COMPUTER, or SPECIAL-NAMES paragraphs in a class CONFIGURATION SECTION, they apply to the entire class definition including all methods that the class introduces.

A class CONFIGURATION SECTION can consist of the same entries as a program CONFIGURATION SECTION, except that a class CONFIGURATION SECTION cannot contain an INPUT-OUTPUT SECTION. You define an INPUT-OUTPUT SECTION only in the individual methods that require it rather than defining it at the class level.

As shown above, you define instance data and methods in the DATA DIVISION and PROCEDURE DIVISION, respectively, within the OBJECT paragraph of the class definition. In classes that require data and methods that are to be associated with the class itself rather than with individual object instances, define a separate DATA DIVISION and PROCEDURE DIVISION within the FACTORY paragraph of the class definition.

Each COBOL class definition must be in a separate source file.

"Example: defining a class" on page 405

RELATED TASKS
"WORKING-STORAGE SECTION for defining class instance data" on page 404
"Defining a class instance method" on page 405
"Defining a subclass" on page 425
"Defining a factory section" on page 429
"Describing the computing environment" on page 7
Chapter 13, "Compiling, linking, and running OO applications," on page 237

RELATED REFERENCES
COBOL class definition structure (*COBOL for AIX 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"

RELATED REFERENCES
CLASS-ID paragraph (*COBOL for AIX Language Reference*)
User-defined words (*COBOL for AIX 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.

In the example above, class Account is known externally as Account (in mixed case) because the external name is spelled using mixed case.

You can optionally include in the REPOSITORY paragraph an entry for the class that you are defining (Account in this example). You must include an entry for the class that you are defining if the external class-name contains non-COBOL characters, or to specify a fully package-qualified class-name if the class is to be part of a Java package.

"Example: external class-names and Java packages"

RELATED TASKS
"Declaring arrays and strings for Java" on page 449

RELATED REFERENCES
REPOSITORY paragraph (*COBOL for AIX 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 402

RELATED REFERENCES
REPOSITORY paragraph (*COBOL for AIX Language Reference*)

WORKING-STORAGE SECTION for defining class instance data

Use the `WORKING-STORAGE SECTION` in the `DATA DIVISION` of the `OBJECT` paragraph to describe the *instance data* that a COBOL class needs, that is, the data to be allocated for each instance of the class.

The `OBJECT` keyword, which you must immediately precede with an `IDENTIFICATION DIVISION` declaration, indicates the beginning of the definitions of the instance data and instance methods for the class. For example, the definition of the instance data for the Account class might look like this:

```
Identification division.
Object.
  Data division.
  Working-storage section.
  01 AccountNumber  pic 9(6).
  01 AccountBalance pic S9(9) value zero.
  . . .
End Object.
```

The instance data is allocated when an object instance is created, and exists until garbage collection of the instance by the Java run time.

You can initialize simple instance data by using `VALUE` clauses as shown above. You can initialize more complex instance data by coding customized methods to create and initialize instances of classes.

COBOL instance data is equivalent to Java private nonstatic member data. No other class or subclass (nor factory method in the same class, if any) can reference COBOL instance data directly. Instance data is global to all instance methods that the `OBJECT` paragraph defines. If you want to make instance data accessible from outside the `OBJECT` paragraph, define attribute (get or set) instance methods for doing so.

The syntax of the `WORKING-STORAGE SECTION` for instance data declaration is generally the same as in a program, with these exceptions:

- You cannot use the EXTERNAL attribute.
- You can use the GLOBAL attribute, but it has no effect.

RELATED TASKS
"Creating and initializing instances of classes" on page 422
"Freeing instances of classes" on page 424
"Defining a factory method" on page 431
"Coding attribute (get and set) methods" on page 411

Example: defining a class

The following example shows a first attempt at the definition of the Account class, excluding method definitions.

```
 cbl thread,pgmname(longmixed)
 Identification Division.
 Class-id. Account inherits Base.
 Environment Division.
 Configuration section.
 Repository.
     Class Base    is "java.lang.Object"
     Class Account is "Account".
*
 Identification division.
 Object.
  Data division.
  Working-storage section.
  01 AccountNumber  pic 9(6).
  01 AccountBalance pic S9(9) value zero.
*
  Procedure Division.
*
*   (Instance method definitions here)
*
 End Object.
*
 End class Account.
```

RELATED TASKS
Chapter 13, "Compiling, linking, and running OO applications," on page 237
"Defining a client" on page 414

Defining a class instance method

Define COBOL *instance methods* in the PROCEDURE DIVISION of the OBJECT paragraph of a class definition. An instance method defines an operation that is supported for each object instance of a class.

A COBOL instance method definition consists of four divisions (like a COBOL program), followed by an END METHOD marker.

Table 49. **Structure of instance method definitions**

Division	**Purpose**	**Syntax**
`IDENTIFICATION` (required)	Name a method.	"METHOD-ID paragraph for defining a class instance method" (required) `AUTHOR paragraph` (optional) `INSTALLATION` paragraph (optional) `DATE-WRITTEN` paragraph (optional) `DATE-COMPILED` paragraph (optional)
`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 407 (optional)
`DATA` (optional)	Define external files. Allocate a copy of the data.	"DATA DIVISION for defining a class instance method" on page 407 (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 408 (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 412

RELATED TASKS
"PROCEDURE DIVISION for defining a class instance method" on page 408
"Overloading an instance method" on page 410
"Overriding an instance method" on page 409
"Invoking methods (INVOKE)" on page 418
"Defining a subclass instance method" on page 428
"Defining a factory method" on page 431

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 418
"Using national data (Unicode) in COBOL" on page 174

RELATED REFERENCES
The Java Language Specification (Meaning of method names)
The Java Language Specification (Identifiers)
METHOD-ID paragraph (*COBOL for AIX Language Reference*)

INPUT-OUTPUT SECTION for defining a class instance method

The `ENVIRONMENT DIVISION` of an instance method can have only one section, the `INPUT-OUTPUT SECTION`. This section relates the file-names used in a method definition to the corresponding file-names as they are known to the operating system.

For example, if the Account class defined a method that read information from a file, the Account class might have an `INPUT-OUTPUT SECTION` that is coded like this:

```
Environment Division.
Input-Output Section.
File-Control.
    Select account-file Assign AcctFile.
```

The syntax for the `INPUT-OUTPUT SECTION` of a method is the same as the syntax for the `INPUT-OUTPUT SECTION` of a program.

RELATED TASKS
"Describing the computing environment" on page 7

RELATED REFERENCES
INPUT-OUTPUT section (*COBOL for AIX 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 491

RELATED REFERENCES

DATA DIVISION overview (*COBOL for AIX Language Reference*)

PROCEDURE DIVISION for defining a class instance method

Code the executable statements to implement the service that an instance method provides in the PROCEDURE DIVISION of the instance method.

You can code most COBOL statements in the PROCEDURE DIVISION of a method that you can code in the PROCEDURE DIVISION of a program. You cannot, however, code the following statements in a method:

- ENTRY
- EXIT PROGRAM
- The following obsolete elements of Standard COBOL 85:
  - ALTER
  - GOTO without a specified procedure-name
  - SEGMENT-LIMIT
  - USE FOR DEBUGGING

You can code the EXIT METHOD or GOBACK statement in an instance method to return control to the invoking client. Both statements have the same effect. If you specify the RETURNING phrase upon invocation of the method, the EXIT METHOD or GOBACK statement returns the value of the data item to the invoking client.

An implicit EXIT METHOD is generated as the last statement in the PROCEDURE DIVISION of each method.

You can specify STOP RUN in a method; doing so terminates the entire run unit including all threads executing within it.

You must terminate a method definition with an END METHOD marker. For example, the following statement marks the end of the credit method:

```
End method "credit".
```

USING phrase for obtaining passed arguments: Specify the formal parameters to a method, if any, in the `USING` phrase of the method's `PROCEDURE DIVISION` header. You must specify that the arguments are passed `BY VALUE`. Define each parameter as a level-01 or level-77 item in the method's `LINKAGE SECTION`. The data type of each parameter must be one of the types that are interoperable with Java.

RETURNING phrase for returning a value: Specify the data item to be returned as the method result, if any, in the `RETURNING` phrase of the method's `PROCEDURE DIVISION` header. Define the data item as a level-01 or level-77 item in the method's `LINKAGE SECTION`. The data type of the return value must be one of the types that are interoperable with Java.

RELATED TASKS
"Coding interoperable data types in COBOL and Java" on page 448
"Overriding an instance method"
"Overloading an instance method" on page 410
"Comparing and setting object references" on page 417
"Invoking methods (INVOKE)" on page 418
Chapter 13, "Compiling, linking, and running OO applications," on page 237

RELATED REFERENCES
"THREAD" on page 287
The procedure division header (*COBOL for AIX 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 412

RELATED TASKS
"PROCEDURE DIVISION for defining a class instance method" on page 408
"Coding interoperable data types in COBOL and Java" on page 448
"Invoking methods (INVOKE)" on page 418
"Invoking overridden superclass methods" on page 422
"Defining a subclass" on page 425
"Hiding a factory or static method" on page 432

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 desired method.

The presence or absence of a method return value does not have to be consistent in overloaded methods, and the data type of the return value given in the PROCEDURE DIVISION RETURNING phrase (if any) does not have to be identical in overloaded methods.

You can overload factory methods in exactly the same way that you overload instance methods.

The rules for overloaded method definition and resolution of overloaded method invocations are based on the corresponding rules for Java.

RELATED TASKS
"Invoking methods (INVOKE)" on page 418
"Defining a factory method" on page 431

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"

RELATED TASKS
"WORKING-STORAGE SECTION for defining class instance data" on page 404
"Processing the data" on page 16

Example: coding a get method

The following example shows the definition in the Account class of an instance method, getBalance, to return the value of the instance variable AccountBalance to a client. getBalance and AccountBalance are defined in the OBJECT paragraph of the Account class definition.

```
 Identification Division.
 Class-id. Account inherits Base.
*  (ENVIRONMENT DIVISION not shown)
*  (FACTORY paragraph not shown)
*
 Identification division.
 Object.
  Data division.
  Working-storage section.
  01 AccountBalance pic S9(9) value zero.
```

```
*   (Other instance data not shown)
*
  Procedure Division.
*
   Identification Division.
   Method-id. "getBalance".
   Data division.
   Linkage section.
   01 outBalance pic S9(9) binary.
*
   Procedure Division returning outBalance.
     Move AccountBalance to outBalance.
   End method "getBalance".
*
*  (Other instance methods not shown)
 End Object.
*
 End class Account.
```

Example: defining a method

The following example adds to the previous example the instance method definitions of the Account class, and shows the definition of the Java Check class.

(The previous example was “Example: defining a class” on page 405.)

Account class

```
 cbl thread,pgmname(longmixed)
 Identification Division.
 Class-id. Account inherits Base.
 Environment Division.
 Configuration section.
 Repository.
     Class Base    is "java.lang.Object"
     Class Account is "Account".
*
*  (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 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.
```

Check class

```
/**
 * A Java class for check information
 */
public class Check {
  private CheckingAccount payer;
  private Account         payee;
  private int             amount;

  public Check(CheckingAccount inPayer, Account inPayee, int inAmount) {
    payer=inPayer;
    payee=inPayee;
    amount=inAmount;
  }

  public int getAmount() {
    return amount;
  }

  public Account getPayee() {
    return payee;
  }
}
```

RELATED TASKS
Chapter 13, “Compiling, linking, and running OO applications,” on page 237

Defining a client

A program or method that requests services from one or more methods in a class is called a *client* of that class.

In a COBOL or Java client, you can:

- Create object instances of Java and COBOL classes.
- Invoke instance methods on Java and COBOL objects.
- Invoke COBOL factory methods and Java static methods.

In a COBOL client, you can also call services provided by the Java Native Interface (JNI).

A COBOL client program consists of the usual four divisions:

Table 50. **Structure of COBOL clients**

Division	Purpose	Syntax
`IDENTIFICATION` (required)	Name a client.	Code as usual, except that a client program must be: • 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 415 (required)
`DATA` (optional)	Describe the data that the client needs.	"DATA DIVISION for defining a client" on page 416 (optional)
`PROCEDURE` (optional)	Create instances of classes, manipulate object reference data items, and invoke methods.	Code using `INVOKE`, `IF`, and `SET` statements.

"Example: defining a client" on page 424

RELATED TASKS
Chapter 13, "Compiling, linking, and running OO applications," on page 237
Chapter 29, "Preparing COBOL programs for multithreading," on page 505
Chapter 24, "Communicating with Java methods," on page 443
"Coding interoperable data types in COBOL and Java" on page 448
"Creating and initializing instances of classes" on page 422
"Comparing and setting object references" on page 417
"Invoking methods (INVOKE)" on page 418
"Invoking factory or static methods" on page 432

RELATED REFERENCES
"THREAD" on page 287

REPOSITORY paragraph for defining a client

Use the REPOSITORY paragraph to declare to the compiler that the specified words are class-names when you use them in a COBOL client, and to optionally relate the class-names to the corresponding external class-names (the class-names as they are known outside the compilation unit).

External class-names are case sensitive, and must conform to Java rules of formation. For example, in a client program that uses the Account and Check classes you might code this:

```
Environment division.      Required
Configuration section.     Required
  Source-Computer.
  Object-Computer.
Repository.                Required
    Class Account is "Account"
    Class Check   is "Check".
```

The REPOSITORY paragraph entries indicate that the external class-names of the classes referred to as Account and Check within the client are Account and Check, respectively.

In the REPOSITORY paragraph, you must code an entry for each class-name that you explicitly reference in the client. In a REPOSITORY paragraph entry, you must specify the external class-name if the name contains non-COBOL characters.

You must specify the external class-name for any referenced class that is part of a Java package. For such a class, specify the external class-name as the fully qualified name of the package, followed by period (.), followed by the simple name of the Java class.

An external class-name that you specify in the REPOSITORY paragraph must be an alphanumeric literal that conforms to the rules of formation for a fully qualified Java class-name.

If you do not include the external class-name in a REPOSITORY paragraph entry, the external class-name is formed from the class-name in the same manner as it is when an external class-name is not included in a REPOSITORY paragraph entry in a class definition. In the example above, class Account and class Check are known externally as Account and Check (in mixed case), respectively, because the external names are spelled using mixed case.

The SOURCE-COMPUTER, OBJECT-COMPUTER, and SPECIAL-NAMES paragraphs of the CONFIGURATION SECTION are optional.

RELATED TASKS
"REPOSITORY paragraph for defining a class" on page 402

RELATED REFERENCES
REPOSITORY paragraph (*COBOL for AIX 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"
"Coding interoperable data types in COBOL and Java" on page 448
"Invoking methods (INVOKE)" on page 418
"REPOSITORY paragraph for defining a client" on page 415

RELATED REFERENCES
RETURNING phrase (*COBOL for AIX 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 29, "Preparing COBOL programs for multithreading," on page 505

Comparing and setting object references

You can compare object references by coding conditional statements or a call to the JNI service IsSameObject, and you can set object references by using the SET statement.

For example, code either IF statement below to check whether the object reference anAccount refers to no object instance:

```
If anAccount = Null . . .
If anAccount = Nulls . . .
```

You can code a call to IsSameObject to check whether two object references, object1 and object2, refer to the same object instance or whether each refers to no object instance. To ensure that the arguments and return value are interoperable with Java and to establish addressability to the callable service, code the following data definitions and statements before the call to IsSameObject:

```
Local-storage Section.
. . .
01  is-same Pic X.
    88 is-same-false Value X'00'.
    88 is-same-true  Value X'01' Through X'FF'.
Linkage Section.
    Copy JNI.
Procedure Division.
    Set Address Of JNIEnv To JNIEnvPtr
    Set Address Of JNINativeInterface To JNIEnv
    Call IsSameObject Using By Value JNIEnvPtr object1 object2
                      Returning is-same
    If is-same-true . . .
```

Within a method you can check whether an object reference refers to the object instance on which the method was invoked by coding a call to IsSameObject that compares the object reference and SELF.

You can instead invoke the Java equals method (inherited from java.lang.Object) to determine whether two object references refer to the same object instance.

You can make an object reference refer to no object instance by using the SET statement. For example:

```
Set anAccount To Null.
```

You can also make one object reference refer to the same instance as another object reference does by using the SET statement. For example:

```
Set anotherAccount To anAccount.
```

This SET statement causes anotherAccount to refer to the same object instance as anAccount does. If the receiver (anotherAccount) is a universal object reference, the sender (anAccount) can be either a universal or a typed object reference. If the receiver is a typed object reference, the sender must be a typed object reference bound to the same class as the receiver or to one of its subclasses.

Within a method you can make an object reference refer to the object instance on which the method was invoked by setting it to SELF. For example:

```
Set anAccount To Self.
```

RELATED TASKS
"Coding interoperable data types in COBOL and Java" on page 448
"Accessing JNI services" on page 443

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 severe error condition is raised at run time unless you code the ON EXCEPTION phrase in the INVOKE statement.

You can use the optional scope terminator END-INVOKE with the INVOKE statement.

The INVOKE statement does not set the RETURN-CODE special register.

RELATED TASKS
"USING phrase for passing arguments"
"RETURNING phrase for obtaining a returned value" on page 421
"PROCEDURE DIVISION for defining a class instance method" on page 408
"Coding interoperable data types in COBOL and Java" on page 448
"Invoking overridden superclass methods" on page 422
"Invoking factory or static methods" on page 432

RELATED REFERENCES
INVOKE statement (*COBOL for AIX 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 51. **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 420

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 408
"Overloading an instance method" on page 410
"Coding interoperable data types in COBOL and Java" on page 448
"Passing data" on page 481

RELATED REFERENCES
INVOKE statement (*COBOL for AIX Language Reference*)
SET statement (*COBOL for AIX Language Reference*)
REDEFINES clause (*COBOL for AIX Language Reference*)

Example: passing conforming object-reference arguments from a COBOL client

The following example shows a way to make an object-reference argument in a COBOL client conform to the expected class of the corresponding formal parameter in an invoked method.

Class C defines a method M that has one parameter, a reference to an object of class java.lang.Object:

```
 . . .
 Class-id. C inherits Base.
 . . .
 Repository.
     Class Base       is "java.lang.Object"
     Class JavaObject is "java.lang.Object".
 Identification division.
 Factory.
 . . .
  Procedure Division.
   Identification Division.
   Method-id. "M".
   Data division.
   Linkage section.
   01 obj object reference JavaObject.
   Procedure Division using by value obj.
   . . .
```

To invoke method M, a COBOL client must pass an argument that is a reference to an object of class java.lang.Object. The client below defines a data item aString, which cannot be passed as an argument to M because aString is a reference to an object of class java.lang.String. The client first uses a SET statement to assign aString to a data item, anObj, that is a reference to an object of class java.lang.Object. (This SET statement is legal because java.lang.String is a subclass of java.lang.Object.) The client then passes anObj as the argument to M.

```
 . . .
 Repository.
     Class jstring    is "java.lang.String"
     Class JavaObject is "java.lang.Object".
 Data division.
 Local-storage section.
 01  aString object reference jstring.
 01  anObj   object reference JavaObject.
```

```
*
 Procedure division.
     . . . (statements here assign a value to aString)
     Set anObj to aString
     Invoke C "M"
       using by value anObj
```

Instead of using a SET statement to obtain anObj as a reference to an object of class java.lang.Object, the client could define aString and anObj with the REDEFINES clause as follows:

```
 . . .
 01  aString object reference jstring.
 01  anObj   redefines aString object reference JavaObject.
```

After the client assigns a value to data item aString (that is, a valid reference to an object of class java.lang.String), anObj can be passed as the argument to M.

RELATED TASKS
"Coding interoperable data types in COBOL and Java" on page 448
"PROCEDURE DIVISION for defining a class instance method" on page 408

RELATED REFERENCES
INVOKE statement (*COBOL for AIX Language Reference*)
SET statement (*COBOL for AIX Language Reference*)
REDEFINES clause (*COBOL for AIX Language Reference*)

RETURNING phrase for obtaining a returned value

If a data item is to be returned as the method result, specify the item in the RETURNING phrase of the INVOKE statement. Define the returned item in the DATA DIVISION of the client.

The item that you specify in the RETURNING phrase of the INVOKE statement must conform to the type returned by the target method, as shown in the table below.

Table 52. **Conformance of the returned data item in a COBOL client**

Programming language of the target method	Is the returned item an object reference?	Then code the DATA DIVISION definition of the returned item as:
COBOL	No	The same as the definition of the RETURNING item in the target method
Java	No	Interoperable with the returned Java data item
COBOL or Java	Yes	An object reference that is typed to the same class as the object reference that is returned by the target method

In all cases, the data type of the returned value must be one of the types that are interoperable with Java.

RELATED TASKS
"Coding interoperable data types in COBOL and Java" on page 448

RELATED REFERENCES
INVOKE statement (*COBOL for AIX 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 398

RELATED TASKS
"Overriding an instance method" on page 409

RELATED REFERENCES
INVOKE statement (*COBOL for AIX 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 423

"Instantiating COBOL classes"
"Accessing JNI services" on page 443
"Managing local and global references" on page 446
"DATA DIVISION for defining a client" on page 416
"Invoking methods (INVOKE)" on page 418
"Coding interoperable data types in COBOL and Java" on page 448

RELATED REFERENCES
INVOKE statement (*COBOL for AIX 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 418
"Coding interoperable data types in COBOL and Java" on page 448

RELATED REFERENCES
VALUE clause (*COBOL for AIX Language Reference*)
INVOKE statement (*COBOL for AIX Language Reference*)

Instantiating COBOL classes

To instantiate a COBOL class, you can specify either a typed or universal object reference in the `RETURNING` phrase of the `INVOKE . . . NEW` statement. However, you cannot code the `USING` phrase: the instance data is initialized as specified in the `VALUE` clauses in the class definition.

Thus the `INVOKE . . . NEW` statement is useful for instantiating COBOL classes that have only simple instance data. For example, the following statement creates an instance of the Account class, initializes the instance data as specified in `VALUE` clauses in the `WORKING-STORAGE SECTION` of the `OBJECT` paragraph of the Account class definition, and provides reference outAccount to the new instance:

```
Invoke Account New returning outAccount
```

To make it possible to initialize COBOL instance data that cannot be initialized using `VALUE` clauses alone, when designing a COBOL class you must define a parameterized creation method in the `FACTORY` paragraph and a parameterized initialization method in the `OBJECT` paragraph:

1. In the parameterized factory creation method, do these steps:

a. Code `INVOKE` *class-name* `NEW RETURNING` *objectRef* to create an instance of *class-name* and to give initial values to the instance data items that have `VALUE` clauses.

b. Invoke the parameterized initialization method on the instance (*objectRef*), passing `BY VALUE` the arguments that were supplied to the factory method.

2. In the initialization method, code logic to complete the instance data initialization using the values supplied through the formal parameters.

To create an instance of the COBOL class and properly initialize it, the client invokes the parameterized factory method, passing `BY VALUE` the desired arguments. The object reference returned to the client is a local reference. If the client code is within a method, and the use of the returned object reference is not limited to the duration of that method, the client code must convert the returned object reference to a global reference by calling the JNI service NewGlobalRef.

"Example: defining a factory (with methods)" on page 433

RELATED TASKS
"Accessing JNI services" on page 443
"Managing local and global references" on page 446
"Invoking methods (INVOKE)" on page 418
"Defining a factory section" on page 429

RELATED REFERENCES
VALUE clause (*COBOL for AIX Language Reference*)
INVOKE statement (*COBOL for AIX 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 446

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 412.)

```
 cbl 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 433

RELATED TASKS
"Defining a factory method" on page 431
"Invoking factory or static methods" on page 432
Chapter 13, "Compiling, linking, and running OO applications," on page 237

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 398
"Example: defining a subclass (with methods)" on page 428

RELATED TASKS
"Defining a class" on page 400
"Overriding an instance method" on page 409
"Coding attribute (get and set) methods" on page 411
"Defining a subclass instance method" on page 428
"Defining a factory section" on page 429

RELATED REFERENCES
The Java Language Specification (Inheritance, overriding, and hiding)
COBOL class definition structure (*COBOL for AIX 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 402
"Coding attribute (get and set) methods" on page 411
"REPOSITORY paragraph for defining a subclass" on page 427

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 402

RELATED REFERENCES
REPOSITORY paragraph (*COBOL for AIX 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 404

Defining a subclass instance method

A subclass inherits the methods of its superclasses. In a subclass definition, you can override any instance method that the subclass inherits by defining an instance method with the same signature as the inherited method. You can also define new methods that the subclass needs.

The structure and syntax of a subclass instance method are identical to those of a class instance method. Define subclass instance methods in the `PROCEDURE DIVISION` of the `OBJECT` paragraph of the subclass definition.

"Example: defining a subclass (with methods)"

RELATED TASKS
"Defining a class instance method" on page 405
"Overriding an instance method" on page 409
"Overloading an instance method" on page 410

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 412.)

CheckingAccount class (subclass of Account)

```
 cbl thread,pgmname(longmixed)
 Identification Division.
 Class-id. CheckingAccount inherits Account.
 Environment Division.
 Configuration section.
 Repository.
     Class CheckingAccount is "CheckingAccount"
     Class Check           is "Check"
     Class Account         is "Account".
*
*  (FACTORY paragraph not shown)
*
 Identification division.
 Object.
  Data division.
  Working-storage section.
  01 CheckFee pic S9(9) value 1.
  Procedure Division.
*
*    processCheck method to get the check amount and payee,
*    add the check fee, and invoke inherited methods debit
*    to debit the payer and credit to credit the payee:
   Identification Division.
   Method-id. "processCheck".
   Data division.
   Local-storage section.
   01 amount pic S9(9) binary.
   01 payee usage object reference Account.
   Linkage section.
   01 aCheck usage object reference Check.
*
   Procedure Division using by value aCheck.
     Invoke aCheck "getAmount" returning amount
     Invoke aCheck "getPayee" returning payee
     Invoke payee  "credit" using by value amount
     Add checkFee to amount
     Invoke self   "debit"  using by value amount.
   End method "processCheck".
*
*    print method override to display account status:
   Identification Division.
   Method-id. "print".
   Data division.
   Local-storage section.
   01 printableFee pic $$,$$$,$$9.
   Procedure Division.
     Invoke super "print"
     Move CheckFee to printableFee
     Display " Check fee: " printableFee.
   End method "print".
*
 End Object.
*
 End class CheckingAccount.
```

RELATED TASKS
Chapter 13, "Compiling, linking, and running OO applications," on page 237
"Invoking methods (INVOKE)" on page 418
"Overriding an instance method" on page 409
"Invoking overridden superclass methods" on page 422

Defining a factory section

Use the FACTORY paragraph in a class definition to define data and methods that are to be associated with the class itself rather than with individual object instances.

COBOL *factory data* is equivalent to Java private static data. A single copy of the data is instantiated for the class and is shared by all object instances of the class. You most commonly use factory data when you want to gather data from all the instances of a class. For example, you could define a factory data item to keep a running total of the number of instances of the class that are created.

COBOL *factory methods* are equivalent to Java public static methods. The methods are supported by the class independently of any object instance. You most commonly use factory methods to customize object creation when you cannot use VALUE clauses alone to initialize instance data.

By contrast, you use the OBJECT paragraph in a class definition to define data that is created for each object instance of the class, and methods that are supported for each object instance of the class.

A factory definition consists of three divisions, followed by an END FACTORY statement:

Table 53. **Structure of factory definitions**

Division	Purpose	Syntax
IDENTIFICATION (required)	Identify the start of the factory definition.	IDENTIFICATION DIVISION. FACTORY.
DATA (optional)	Describe data that is allocated once for the class (as opposed to data allocated for each instance of a class).	"WORKING-STORAGE SECTION for defining factory data" (optional)
PROCEDURE (optional)	Define factory methods.	Factory method definitions: "Defining a factory method" on page 431

"Example: defining a factory (with methods)" on page 433

RELATED TASKS
"Defining a class" on page 400
"Instantiating COBOL classes" on page 423
"Wrapping procedure-oriented COBOL programs" on page 438
"Structuring OO applications" on page 439

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 411
"Instantiating COBOL classes" on page 423

Defining a factory method

Define COBOL *factory methods* in the PROCEDURE DIVISION of the FACTORY paragraph of a class definition. A factory method defines an operation that is supported by a class independently of any object instance of the class. COBOL factory methods are equivalent to Java public static methods.

You typically define factory methods for classes whose instances require complex initialization, that is, to values that you cannot assign by using VALUE clauses alone. Within a factory method you can invoke instance methods to initialize the instance data. A factory method cannot directly access instance data.

You can code factory attribute (get and set) methods to make factory data accessible from outside the FACTORY paragraph, for example, to make the data accessible from instance methods in the same class or from a client program. For example, the Account class could define a factory method getNumberOfAccounts to return the current tally of the number of accounts.

You can use factory methods to wrap procedure-oriented COBOL programs so that they are accessible from Java programs. You can code a factory method called main to enable you to run an OO application by using the java command, and to structure your applications in keeping with standard Java practice. See the related tasks for details.

In defining factory methods, you use the same syntax that you use to define instance methods. A COBOL factory method definition consists of four divisions (like a COBOL program), followed by an END METHOD marker:

Table 54. **Structure of factory method definitions**

Division	Purpose	Syntax
IDENTIFICATION (required)	Same as for a class instance method	Same as for a class instance method (required)
ENVIRONMENT (optional)	Same as for a class instance method	Same as for a class instance method
DATA (optional)	Same as for a class instance method	Same as for a class instance method
PROCEDURE (optional)	Same as for a class instance method	Same as for a class instance method

Within a class definition, you do not need to make each factory method-name unique, but you do need to give each factory method a unique signature. You can overload factory methods in exactly the same way that you overload instance methods. For example, the CheckingAccount subclass provides two versions of the factory method createCheckingAccount: one that initializes the account to have a default balance of zero, and one that allows the opening balance to be passed in. Clients can invoke either createCheckingAccount method by passing arguments that match the signature of the intended method.

If you define a data item with the same name in both the DATA DIVISION of a factory method and the DATA DIVISION of the FACTORY paragraph, a reference in the method to that data-name refers only to the method data item. The method DATA DIVISION takes precedence.

"Example: defining a factory (with methods)" on page 433

RELATED TASKS
"Structuring OO applications" on page 439
"Wrapping procedure-oriented COBOL programs" on page 438
"Instantiating COBOL classes" on page 423
"Defining a class instance method" on page 405
"Coding attribute (get and set) methods" on page 411
"Overloading an instance method" on page 410
"Hiding a factory or static method"
"Invoking factory or static methods"

Hiding a factory or static method

A factory method defined in a subclass is said to *hide* an inherited COBOL or Java method that would otherwise be accessible in the subclass if the two methods have the same signature.

To hide a superclass factory method `f1` in a COBOL subclass, define a factory method `f1` in the subclass that has the same name and whose `PROCEDURE DIVISION USING` phrase (if any) has the same number and type of formal parameters as the superclass method has. (If the superclass method is implemented in Java, you must code formal parameters that are interoperable with the data types of the corresponding Java parameters.) When a client invokes `f1` using the subclass name, the subclass method rather than the superclass method is invoked.

The presence or absence of a method return value and the data type of the return value used in the `PROCEDURE DIVISION RETURNING` phrase (if any) must be identical in the subclass factory method and the hidden superclass method.

A factory method must not hide an instance method in a Java or COBOL superclass.

"Example: defining a factory (with methods)" on page 433

RELATED TASKS
"Coding interoperable data types in COBOL and Java" on page 448
"Overriding an instance method" on page 409
"Invoking methods (INVOKE)" on page 418

RELATED REFERENCES
The Java Language Specification (Inheritance, overriding, and hiding)
The procedure division header (*COBOL for AIX Language Reference*)

Invoking factory or static methods

To invoke a COBOL factory method or Java static method in a COBOL method or client program, code the class-name as the first operand of the `INVOKE` statement.

For example, a client program could invoke one of the overloaded CheckingAccount factory methods called createCheckingAccount to create a checking account with account number 777777 and an opening balance of $300 by coding this statement:

```
Invoke CheckingAccount "createCheckingAccount"
  using by value 777777 300
  returning aCheckingAccount
```

To invoke a factory method from within the same class in which you define the factory method, you also use the class-name as the first operand in the INVOKE statement.

Code the name of the method to be invoked either as a literal or as an identifier whose value at run time is the method-name. The method-name must be an alphanumeric or national literal or a category alphabetic, alphanumeric, or national data item, and is interpreted in a case-sensitive manner.

If an invoked method is not supported in the class that you name in the INVOKE statement, a severe error condition is raised at run time unless you code the ON EXCEPTION phrase in the INVOKE statement.

The conformance requirements for passing arguments to a COBOL factory method or Java static method in the USING phrase, and receiving a return value in the RETURNING phrase, are the same as those for invoking instance methods.

"Example: defining a factory (with methods)"

RELATED TASKS
"Invoking methods (INVOKE)" on page 418
"Using national data (Unicode) in COBOL" on page 174
"Coding interoperable data types in COBOL and Java" on page 448

RELATED REFERENCES
INVOKE statement (*COBOL for AIX Language Reference*)

Example: defining a factory (with methods)

The following example updates the previous examples to show the definition of factory data and methods.

These updates are shown:
- The Account class adds factory data and a parameterized factory method, createAccount, which allows an Account instance to be created using an account number that is passed in.
- The CheckingAccount subclass adds factory data and an overloaded parameterized factory method, createCheckingAccount. One implementation of createCheckingAccount initializes the account with a default balance of zero, and the other allows the opening balance to be passed in. Clients can invoke either method by passing arguments that match the signature of the desired method.
- The TestAccounts client invokes the services provided by the factory methods of the Account and CheckingAccount classes, and instantiates the Java Check class.
- The output from the TestAccounts client program is shown.

(The previous examples were "Example: defining a method" on page 412, "Example: defining a client" on page 424, and "Example: defining a subclass (with methods)" on page 428.)

You can also find the complete source code for this example in the samples subdirectory of the COBOL install directory, typically /usr/lpp/cobol/samples/oo. You can use the makefile there to compile and link the code.

Account class

```
 cbl thread,pgmname(longmixed),lib
 Identification Division.
 Class-id. Account inherits Base.
 Environment Division.
 Configuration section.
 Repository.
     Class Base    is "java.lang.Object"
     Class Account is "Account".
*
 Identification division.
 Factory.
  Data division.
  Working-storage section.
  01 NumberOfAccounts pic 9(6) value zero.
*
  Procedure Division.
*
*    createAccount method to create a new Account
*    instance, then invoke the OBJECT paragraph's init
*    method on the instance to initialize its instance data:
   Identification Division.
   Method-id. "createAccount".
   Data division.
   Linkage section.
   01 inAccountNumber  pic S9(6) binary.
   01 outAccount object reference Account.
*      Facilitate access to JNI services:
     Copy JNI.
   Procedure Division using by value inAccountNumber
       returning outAccount.
*      Establish addressability to JNI environment structure:
     Set address of JNIEnv to JNIEnvPtr
     Set address of JNINativeInterface to JNIEnv
     Invoke Account New returning outAccount
     Invoke outAccount "init" using by value inAccountNumber
     Add 1 to NumberOfAccounts.
   End method "createAccount".
*
 End Factory.
*
 Identification division.
 Object.
  Data division.
  Working-storage section.
  01 AccountNumber  pic 9(6).
  01 AccountBalance pic S9(9) value zero.
*
  Procedure Division.
*
*    init method to initialize the account:
   Identification Division.
   Method-id. "init".
   Data division.
   Linkage section.
   01 inAccountNumber pic S9(9) binary.
   Procedure Division using by value inAccountNumber.
     Move inAccountNumber to AccountNumber.
   End method "init".
*
*    getBalance method to return the account balance:
   Identification Division.
   Method-id. "getBalance".
   Data division.
   Linkage section.
   01 outBalance pic S9(9) binary.
   Procedure Division returning outBalance.
```

```
     Move AccountBalance to outBalance.
   End method "getBalance".
*
*    credit method to deposit to the account:
   Identification Division.
   Method-id. "credit".
   Data division.
   Linkage section.
   01 inCredit   pic S9(9) binary.
   Procedure Division using by value inCredit.
     Add inCredit to AccountBalance.
   End method "credit".
*
*    debit method to withdraw from the account:
   Identification Division.
   Method-id. "debit".
   Data division.
   Linkage section.
   01 inDebit    pic S9(9) binary.
   Procedure Division using by value inDebit.
     Subtract inDebit from AccountBalance.
   End method "debit".
*
*    print method to display formatted account number and balance:
   Identification Division.
   Method-id. "print".
   Data division.
   Local-storage section.
   01 PrintableAccountNumber  pic 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 thread,pgmname(longmixed),lib
 Identification Division.
 Class-id. CheckingAccount inherits Account.
 Environment Division.
 Configuration section.
 Repository.
     Class CheckingAccount is "CheckingAccount"
     Class Check           is "Check"
     Class Account         is "Account".
*
 Identification division.
 Factory.
  Data division.
  Working-storage section.
  01 NumberOfCheckingAccounts pic 9(6) value zero.
*
  Procedure Division.
*
*    createCheckingAccount overloaded method to create a new
*    CheckingAccount instance with a default balance, invoke
*    inherited instance method init to initialize the account
*    number, and increment factory data tally of checking accounts:
   Identification Division.
   Method-id. "createCheckingAccount".
   Data division.
   Linkage section.
   01 inAccountNumber  pic S9(6) binary.
   01 outCheckingAccount object reference CheckingAccount.
*      Facilitate access to JNI services:
     Copy JNI.
   Procedure Division using by value inAccountNumber
       returning outCheckingAccount.
*      Establish addressability to JNI environment structure:
     Set address of JNIEnv to JNIEnvPtr
     Set address of JNINativeInterface to JNIEnv
     Invoke CheckingAccount New returning outCheckingAccount
     Invoke outCheckingAccount "init"
       using by value inAccountNumber
     Add 1 to NumberOfCheckingAccounts.
   End method "createCheckingAccount".
*
*    createCheckingAccount overloaded method to create a new
*    CheckingAccount instance, invoke inherited instance methods
*    init to initialize the account number and credit to set the
*    balance, and increment factory data tally of checking accounts:
   Identification Division.
   Method-id. "createCheckingAccount".
   Data division.
   Linkage section.
   01 inAccountNumber  pic S9(6) binary.
   01 inInitialBalance pic S9(9) binary.
   01 outCheckingAccount object reference CheckingAccount.
     Copy JNI.
   Procedure Division using by value inAccountNumber
                                     inInitialBalance
       returning outCheckingAccount.
     Set address of JNIEnv to JNIEnvPtr
     Set address of JNINativeInterface to JNIEnv
     Invoke CheckingAccount New returning outCheckingAccount
     Invoke outCheckingAccount "init"
       using by value inAccountNumber
     Invoke outCheckingAccount "credit"
       using by value inInitialBalance
     Add 1 to NumberOfCheckingAccounts.
   End method "createCheckingAccount".
*
 End Factory.
*
 Identification division.
 Object.
  Data division.
  Working-storage section.
  01 CheckFee pic S9(9) value 1.
  Procedure Division.
*
*    processCheck method to get the check amount and payee,
*    add the check fee, and invoke inherited methods debit
*    to debit the payer and credit to credit the payee:
   Identification Division.
   Method-id. "processCheck".
   Data division.
   Local-storage section.
   01 amount pic S9(9) binary.
   01 payee usage object reference Account.
   Linkage section.
   01 aCheck usage object reference Check.
   Procedure Division using by value aCheck.
     Invoke aCheck "getAmount" returning amount
     Invoke aCheck "getPayee" returning payee
     Invoke payee  "credit" using by value amount
     Add checkFee to amount
     Invoke self   "debit"  using by value amount.
   End method "processCheck".
*
*    print method override to display account status:
   Identification Division.
   Method-id. "print".
   Data division.
   Local-storage section.
   01 printableFee pic $$,$$$,$$9.
   Procedure Division.
     Invoke super "print"
     Move CheckFee to printableFee
     Display " Check fee: " printableFee.
   End method "print".
*
 End Object.
*
 End class CheckingAccount.
```

Check class

```
/**
 * A Java class for check information
 */
public class Check {
  private CheckingAccount payer;
  private Account         payee;
  private int             amount;

  public Check(CheckingAccount inPayer, Account inPayee, int inAmount) {
    payer=inPayer;
    payee=inPayee;
    amount=inAmount;
  }

  public int getAmount() {
    return amount;
  }

  public Account getPayee() {
    return payee;
  }
}
```

TestAccounts client program

```
 cbl thread,pgmname(longmixed)
 Identification division.
 Program-id. "TestAccounts" recursive.
 Environment division.
 Configuration section.
 Repository.
     Class Account         is "Account"
     Class CheckingAccount is "CheckingAccount"
     Class Check           is "Check".
 Data Division.
*  Working data is declared in Local-storage
*  so that each thread has its own copy:
 Local-storage section.
 01  anAccount        usage object reference Account.
 01  aCheckingAccount usage object reference CheckingAccount.
 01  aCheck           usage object reference Check.
 01  payee            usage object reference Account.
*
 Procedure division.
 Test-Account-section.
     Display "Test Account class"
*  Create account 123456 with 0 balance:
     Invoke Account "createAccount"
```

```
       using by value 123456
       returning anAccount
*  Deposit 500 to the account:
     Invoke anAccount "credit" using by value 500
     Invoke anAccount "print"
     Display space
*
     Display "Test CheckingAccount class"
*  Create checking account 777777 with balance of 300:
     Invoke CheckingAccount "createCheckingAccount"
       using by value 777777 300
       returning aCheckingAccount
*  Set account 123456 as the payee:
     Set payee to anAccount
*  Initialize check for 125 to be paid by account 777777 to payee:
     Invoke Check New
       using by value aCheckingAccount, payee, 125
       returning aCheck
*  Debit the payer, and credit the payee:
     Invoke aCheckingAccount "processCheck"
       using by value aCheck
     Invoke aCheckingAccount "print"
     Invoke anAccount "print"
*
     Stop Run.
 End program "TestAccounts".
```

Output produced by the TestAccounts client program

```
Test Account class
 Account:      123456
 Balance:        $500

Test CheckingAccount class
 Account:      777777
 Balance:        $174
 Check fee:        $1
 Account:      123456
 Balance:        $625
```

RELATED TASKS

"Creating and initializing instances of classes" on page 422
"Defining a factory method" on page 431
"Invoking factory or static methods" on page 432
Chapter 13, "Compiling, linking, and running OO applications," on page 237

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 400
"Defining a factory section" on page 429
"Defining a factory method" on page 431

Structuring OO applications

You can structure applications that use object-oriented COBOL syntax in one of three ways.

An OO application can begin with:
- A COBOL program, which can have any name.

  You can run the program by specifying the name of the executable module at the command prompt.
- A Java class definition that contains a method called `main`. Declare `main` as `public`, `static`, and `void`, with a single parameter of type String[].

  You can run the application with the `java` command, specifying the name of the class that contains `main` and zero or more strings as command-line arguments.
- A COBOL class definition that contains a factory method called `main`. Declare `main` with no `RETURNING` phrase and a single `USING` parameter, an object reference to a class that is an array with elements of type java.lang.String. (Thus `main` is in effect public, static, and void, with a single parameter of type String[].)

  You can run the application with the `java` command, specifying the name of the class that contains `main` and zero or more strings as command-line arguments.

  Structure an OO application this way if you want to:
  - Run the application by using the `java` command.
  - Run the application in an environment where applications must start with the `main` method of a Java class file.
  - Follow standard Java programming practice.

  "Examples: COBOL applications that run using the java command"

RELATED TASKS
Chapter 13, "Compiling, linking, and running OO applications," on page 237
"Running programs" on page 234
"Defining a factory method" on page 431
"Declaring arrays and strings for Java" on page 449

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 thread
Identification Division.
Class-id. CBLmain inherits Base.
Environment Division.
Configuration section.
```

```
 Repository.
     Class Base is "java.lang.Object"
     Class stringArray is "jobjectArray:java.lang.String"
     Class CBLmain is "CBLmain".
*
 Identification Division.
 Factory.
  Procedure division.
*
   Identification Division.
   Method-id. "main".
   Data division.
   Linkage section.
   01 SA usage object reference stringArray.
   Procedure division using by value SA.
     Display " >> COBOL main method entered"
     .
   End method "main".
 End factory.
 End class CBLmain.
```

Echoing the input strings

```
 cbl thread,lib,ssrange
 Identification Division.
 Class-id. Echo inherits Base.
 Environment Division.
 Configuration section.
 Repository.
     Class Base is "java.lang.Object"
     Class stringArray is "jobjectArray:java.lang.String"
     Class jstring is "java.lang.String"
     Class Echo is "Echo".
*
 Identification Division.
 Factory.
  Procedure division.
*
   Identification Division.
   Method-id. "main".
   Data division.
   Local-storage section.
   01 SAlen        pic s9(9) binary.
   01 I            pic s9(9) binary.
   01 SAelement    object reference jstring.
   01 SAelementlen pic s9(9) binary.
   01 P            pointer.
   Linkage section.
   01 SA           object reference stringArray.
   01 Sbuffer      pic N(65535).
   Copy "JNI.cpy" suppress.
   Procedure division using by value SA.
     Set address of JNIEnv to JNIEnvPtr
     Set address of JNINativeInterface to JNIEnv
     Call GetArrayLength using by value JNIEnvPtr SA
       returning SAlen
     Display "Input string array length: " SAlen
     Display "Input strings:"
     Perform varying I from 0 by 1 until I = SAlen
       Call GetObjectArrayElement
         using by value JNIEnvPtr SA I
         returning SAelement
       Call GetStringLength
         using by value JNIEnvPtr SAelement
         returning SAelementlen
       Call GetStringChars
          using by value JNIEnvPtr SAelement 0
          returning P
```

```
      Set address of Sbuffer to P
      Display function display-of(Sbuffer(1:SAelementlen))
      Call ReleaseStringChars
        using by value JNIEnvPtr SAelement P
    End-perform
    .
  End method "main".
End factory.
End class Echo.
```

RELATED TASKS

Chapter 13, "Compiling, linking, and running OO applications," on page 237
"Defining a factory method" on page 431
Chapter 24, "Communicating with Java methods," on page 443

Chapter 24. Communicating with Java methods

To achieve interlanguage interoperability with Java, you need to follow certain rules and guidelines for using services in the Java Native Interface (JNI), coding data types, and compiling COBOL programs.

You can invoke methods that are written in Java from COBOL programs, and you can invoke methods that are written in COBOL from Java programs. You need to code COBOL object-oriented language for basic Java object capabilities. For additional Java capabilities, you can call JNI services.

Because Java programs might be multithreaded and use asynchronous signals, compile COBOL programs with the `THREAD` option.

RELATED CONCEPTS
"Java SDK support" on page 241

RELATED TASKS
Chapter 13, "Compiling, linking, and running OO applications," on page 237
"Accessing JNI services"
"Sharing data with Java" on page 448
Chapter 23, "Writing object-oriented programs," on page 397
Chapter 29, "Preparing COBOL programs for multithreading," on page 505

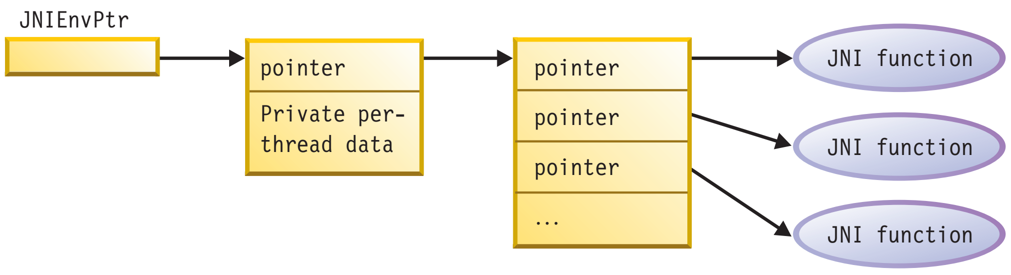
RELATED REFERENCES
Java 2 SDK, Standard Edition Documentation

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. To be interoperable, the arguments must be in native format (for example, by being declared with the NATIVE clause on the data description entry).

Some JNI callable services require a Java class-object reference as an argument. To obtain a reference to the class object that is associated with a class, use one of the following JNI callable services:

- GetObjectClass
- FindClass

Restriction: The JNI environment pointer is thread specific. Do not pass it from one thread to another.

RELATED TASKS
"Managing local and global references" on page 446
"Handling Java exceptions" on page 445
"Coding interoperable data types in COBOL and Java" on page 448
"Defining a client" on page 414

RELATED REFERENCES
"ADDR" on page 246
Appendix H, "JNI.cpy," on page 651
The Java Native Interface

Handling Java exceptions

Use JNI services to throw and catch Java exceptions.

Throwing an exception: Use one of the following services to throw a Java exception from a COBOL method:
- Throw
- ThrowNew

You must make the thrown object an instance of a subclass of java.lang.Throwable.

The Java virtual machine (JVM) does not recognize and process the thrown exception until the method that contains the call has completed and returned to the JVM.

Catching an exception: After you invoke a method that might have thrown a Java exception, you can do these steps:
1. Test whether an exception occurred.
2. If an exception occurred, process the exception.
3. Clear the exception, if clearing is appropriate.

Use the following JNI services:
- ExceptionOccurred
- ExceptionCheck
- ExceptionDescribe
- ExceptionClear

To do error analysis, use the methods supported by the exception object that is returned. This object is an instance of the java.lang.Throwable class.

"Example: handling Java exceptions"

Example: handling Java exceptions

The following example shows the use of JNI services for catching an exception from Java and the use of the printStackTrace method of java.lang.Throwable for error analysis.

```
Repository.
    Class JavaException is "java.lang.Exception".
. . .
Local-storage section.
01 ex usage object reference JavaException.
Linkage section.
COPY "JNI.cpy".
. . .
Procedure division.
    Set address of JNIEnv to JNIEnvPtr
    Set address of JNINativeInterface to JNIEnv
    . . .
    Invoke anObj "someMethod"
    Perform ErrorCheck
. . .
ErrorCheck.
    Call ExceptionOccurred
        using by value JNIEnvPtr
        returning ex
    If ex not = null then
```

```
        Call ExceptionClear using by value JNIEnvPtr
        Display "Caught an unexpected exception"
        Invoke ex "printStackTrace"
        Stop run
    End-if
```

Managing local and global references

The Java virtual machine tracks the object references that you use in native methods, such as COBOL methods. This tracking ensures that the objects are not prematurely released during garbage collection.

There are two classes of such references:

Local references
Local references are valid only while the method that you invoke runs. Automatic freeing of the local references occurs after the native method returns.

Global references
Global references remain valid until you explicitly delete them. You can create global references from local references by using the JNI service NewGlobalRef.

The following object references are always local:

- Object references that are received as method parameters
- Object references that are returned as the method `RETURNING` value from a method invocation
- Object references that are returned by a call to a JNI function
- Object references that you create by using the `INVOKE . . . NEW` statement

You can pass either a local reference or a global reference as an object reference argument to a JNI service.

You can code methods to return either local or global references as `RETURNING` values. However, in either case, the reference that is received by the invoking program is a local reference.

You can pass either local or global references as `USING` arguments in a method invocation. However, in either case, the reference that is received by the invoked method is a local reference.

Local references are valid only in the thread in which you create them. Do not pass them from one thread to another.

RELATED TASKS
"Accessing JNI services" on page 443
"Deleting, saving, and freeing local references"

Deleting, saving, and freeing local references

You can manually delete local references at any point within a method. Save local references only in object references that you define in the `LOCAL-STORAGE SECTION` of a method.

Use a `SET` statement to convert a local reference to a global reference if you want to save a reference in any of these data items:

- An object instance variable
- A factory variable
- A data item in the `WORKING-STORAGE SECTION` of a method

Otherwise, an error occurs. These storage areas persist when a method returns; therefore a local reference is no longer valid.

In most cases you can rely on the automatic freeing of local references that occurs when a method returns. However, in some cases you should explicitly free a local reference within a method by using the JNI service DeleteLocalRef. Here are two situations where explicit freeing is appropriate:

- In a method you access a large object, thereby creating a local reference to the object. After extensive computations, the method returns. Free the large object if you do not need it for the additional computations, because the local reference prevents the object from being released during garbage collection.
- You create a large number of local references in a method, but do not use all of them at the same time. Because the Java virtual machine requires space to keep track of each local reference, you should free those that you no longer need. Freeing the local references helps prevent the system from running out of memory.

  For example, in a COBOL method you loop through a large array of objects, retrieve the elements as local references, and operate on one element at each iteration. You can free the local reference to the array element after each iteration.

Use the following callable services to manage local references and global references.

Table 55. **JNI services for local and global references**

Service	Input arguments	Return value	Purpose
NewGlobalRef	• 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
DeleteGlobalRef	• 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	• 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 443

Java access controls

The Java access modifiers `protected` and `private` are not enforced when you use the Java Native Interface. Therefore a COBOL program could invoke a protected or private Java method that is not invocable from a Java client. This usage is not recommended.

Sharing data with Java

You can share the COBOL data types that have Java equivalents. (Some COBOL data types have Java equivalents, but others do not.)

Share data items with Java in these ways:
- Pass them as arguments in the `USING` phrase of an `INVOKE` statement.
- Receive them as parameters in the `USING` phrase from a Java method.
- Receive them as the `RETURNING` value in an `INVOKE` statement.
- Return them as the value in the `RETURNING` phrase of the `PROCEDURE DIVISION` header in a COBOL method.

To pass or receive arrays and strings, declare them as object references:
- Declare an array as an object reference that contains an instance of one of the special array classes.
- Declare a string as an object reference that contains an instance of the jstring class.

RELATED TASKS
"Coding interoperable data types in COBOL and Java"
"Declaring arrays and strings for Java" on page 449
"Manipulating Java arrays" on page 450
"Manipulating Java strings" on page 453
"Invoking methods (INVOKE)" on page 418
Chapter 27, "Sharing data," on page 481

Coding interoperable data types in COBOL and Java

Your COBOL program can use only certain data types when communicating with Java.

Table 56. **Interoperable data types in COBOL and Java**

Primitive Java data type	Corresponding COBOL data type

boolean[1]	`PIC X` followed by exactly two condition-names of this form:
	 `level-number data-name PIC X.`
	 `88 data-name-false value X'00'.`
	 `88 data-name-true value X'01' through X'FF'.`

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` *class-name*

Table 56. **Interoperable data types in COBOL and Java** *(continued)*

Primitive Java data type	**Corresponding COBOL data type**
1. You must distinguish boolean from byte, because they each correspond to `PIC X`. `PIC X` is interpreted as boolean only when you define an argument or a parameter with the two condition-names as shown. Otherwise, a `PIC X` data item is interpreted as the Java byte type. A single-byte alphanumeric item can contain either EBCDIC or native content. 2. Java floating-point data is represented in IEEE floating point. Floating-point data items that you pass as arguments in an `INVOKE` statement or receive as parameters from a Java method must be in the native format. Therefore if you compile using the `-host` option of the `cob2_j` command or using the `FLOAT(S390)` option, each floating-point data item that you pass to or receive from a Java method must be declared with the `NATIVE` phrase in its data description.	

RELATED TASKS
"Using national data (Unicode) in COBOL" on page 174

RELATED REFERENCES
"cob2, cob2_j, and cob2_r options" on page 227
"CHAR" on page 250
"FLOAT" on page 267

Declaring arrays and strings for Java

When you communicate with Java, declare arrays by using the special array classes, and declare strings by using jstring. Code the COBOL data types shown in the table below.

Table 57. **Interoperable arrays and strings in COBOL and Java**

Java data type	**Corresponding COBOL data type**
boolean[ ]	`object reference jbooleanArray`
byte[ ]	`object reference jbyteArray`
short[ ]	`object reference jshortArray`
int[ ]	`object reference jintArray`
long[ ]	`object reference jlongArray`
char[ ]	`object reference jcharArray`
Object[ ]	`object reference jobjectArray`
String	`object reference jstring`

To use one of these classes for interoperability with Java, you must code an entry in the `REPOSITORY` paragraph. For example:

```
Configuration section.
Repository.
    Class jbooleanArray is "jbooleanArray".
```

The `REPOSITORY` paragraph entry for an object array type must specify an external class-name in one of these forms:

```
"jobjectArray"
"jobjectArray:external-classname-2"
```

In the first case, the REPOSITORY entry specifies an array class in which the elements of the array are objects of type java.lang.Object. In the second case, the REPOSITORY entry specifies an array class in which the elements of the array are objects of type *external-classname-2*. Code a colon as the separator between the specification of the jobjectArray type and the external class-name of the array elements.

The following example shows both cases. In the example, oa defines an array of elements that are objects of type java.lang.Object. aDepartment defines an array of elements that are objects of type com.acme.Employee.

```
Environment Division.
Configuration Section.
Repository.
    Class jobjectArray is "jobjectArray"
    Class Employee     is "com.acme.Employee"
    Class Department   is "jobjectArray:com.acme.Employee".
. . .
Linkage section.
01 oa          usage object reference jobjectArray.
01 aDepartment usage object reference Department.
. . .
Procedure division using by value aDepartment.
. . .
```

"Examples: COBOL applications that run using the java command" on page 439

The following Java array types are currently not supported for interoperation with COBOL programs.

Table 58. **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 402

Manipulating Java arrays

To represent an array in a COBOL program, code a group item that contains a single elementary item that is of the data type that corresponds to the Java type of the array. Specify an OCCURS or OCCURS DEPENDING ON clause that is appropriate for the array.

For example, the following code specifies a structure to receive 500 or fewer integer values from a jlongArray object:

```
01  longArray.
    02 X pic S9(10) comp-5 occurs 1 to 500 times depending on N.
```

To operate on objects of the special Java-array classes, call the services that the JNI provides. You can use services to access and set individual elements of an array and for the following purposes, using the services cited:

Table 59. **JNI array services**

Service	Input arguments	Return value	Purpose
GetArrayLength	• 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	• 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	• 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 Release*xxx*ArrayElements 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	• 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
NewObjectArray	• 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	• 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	• 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

Table 59. **JNI array services** *(continued)*

Service	Input arguments	Return value	Purpose

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 439
"Example: processing a Java int array"

RELATED TASKS
"Coding interoperable data types in COBOL and Java" on page 448
"Declaring arrays and strings for Java" on page 449
"Accessing JNI services" on page 443

Example: processing a Java int array

The following example shows the use of the Java-array classes and JNI services to process a Java array in COBOL.

```
 cbl lib,thread
 Identification division.
 Class-id. OOARRAY inherits Base.
 Environment division.
 Configuration section.
 Repository.
     Class Base is "java.lang.Object"
     Class jintArray is "jintArray".
 Identification division.
 Object.
 Procedure division.
   Identification division.
   Method-id. "ProcessArray".
   Data Division.
   Local-storage section.
   01 intArrayPtr pointer.
   01 intArrayLen pic S9(9) comp-5.
   Linkage section.
      COPY JNI.
   01 inIntArrayObj usage object reference jintArray.
   01 intArrayGroup.
      02 X pic S9(9) comp-5
          occurs 1 to 1000 times depending on intArrayLen.
   Procedure division using by value inIntArrayObj.
       Set address of JNIEnv to JNIEnvPtr
       Set address of 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 national (Unicode) or UTF-8 data from the object.

Services for Unicode: Use the following standard services to convert between jstring object references and COBOL `USAGE NATIONAL` data items. Access these services by using function pointers in the JNINativeInterface environment structure.

Table 60. **Services that convert between jstring references and national data**

Service	Input arguments	Return value
NewString[1]	• 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	• The JNI environment pointer • A jstring object reference	The number of Unicode characters in the jstring object reference; binary fullword
GetStringChars[1]	• The JNI environment pointer • A jstring object reference • A pointer to a boolean data item, or `NULL`	• 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. • When the pointer to the boolean data item is not null, the boolean value is set to true if a copy is made of the string and to false if no copy is made.
ReleaseStringChars	• 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 UTF-8: You can use the following services, an extension of the JNI, to convert between jstring object references and UTF-8 strings. Use these services in programs that do not need to be portable to the mainframe. Access these services by using function pointers in the JNI environment structure JNINativeInterface.

Table 61. **Services that convert between jstring references and UTF-8 data**

Service	Input arguments	Return value
NewStringUTF[1]	• The JNI environment pointer • A pointer to a null-terminated UTF-8 string	jstring object reference, or `NULL` if the string cannot be constructed
GetStringUTFLength	• The JNI environment pointer • A jstring object reference	The number of bytes needed to represent the string in UTF-8 format; binary fullword

Table 61. **Services that convert between jstring references and UTF-8 data** *(continued)*

Service	Input arguments	Return value
GetStringUTFChars[1]	• The JNI environment pointer • A jstring object reference • A pointer to a boolean data item, or NULL	• Pointer to an array of UTF-8 characters extracted from the jstring object, or NULL if the operation fails. The pointer is valid until it is released with ReleaseStringUTFChars. • When the pointer to the boolean data item is not null, the boolean value is set to true if a copy is made of the string and to false if no copy is made.
ReleaseStringUTFChars	• The JNI environment pointer • A jstring object reference • A pointer to the UTF-8 string that was derived from the jstring argument by using GetStringUTFChars	None; the storage for the UTF-8 string is released.
1. This service throws an exception if the system runs out of memory.		

RELATED TASKS
"Accessing JNI services" on page 443
"Coding interoperable data types in COBOL and Java" on page 448
"Declaring arrays and strings for Java" on page 449
"Using national data (Unicode) in COBOL" on page 174
Chapter 13, "Compiling, linking, and running OO applications," on page 237

Part 7. Working with more complex applications

Chapter 25. Porting applications between platforms 457
Getting mainframe applications to compile . . . 457
Getting mainframe applications to run: overview 458
Fixing differences caused by data representations 458
Handling differences in ASCII SBCS and EBCDIC SBCS characters 458
Handling differences in IEEE and hexadecimal data 459
Handling differences in ASCII multibyte and EBCDIC DBCS strings 460
Fixing environment differences that affect portability 460
Fixing differences caused by language elements 461
Writing code to run on the mainframe 462
Writing applications that are portable between AIX and Windows 462

Chapter 26. Using subprograms 465
Main programs, subprograms, and calls 465
Ending and reentering main programs or subprograms 465
Calling nested COBOL programs 466
Nested programs 467
Example: structure of nested programs 468
Scope of names. 469
Local names 469
Global names 469
Searches for name declarations 470
Calling nonnested COBOL programs 470
CALL identifier and CALL literal. 470
Calling between COBOL and C/C++ programs . . 471
Initializing environments 472
Passing data between COBOL and C/C++ . . 472
Collapsing stack frames and terminating run units or processes 473
COBOL and C/C++ data types 473
Example: COBOL program calling C functions 474
Example: C programs that are called by and call COBOL 475
Example: COBOL program called by a C program 476
Example: results of compiling and running examples 476
| Example: COBOL program calling C++ function 477
Making recursive calls 478
Passing return codes 478

Chapter 27. Sharing data 481
Passing data. 481
Describing arguments in the calling program 483
Describing parameters in the called program 483
Testing for OMITTED arguments 484
Coding the LINKAGE SECTION 484
Coding the PROCEDURE DIVISION for passing arguments 485
Grouping data to be passed 485
Handling null-terminated strings 486
Using pointers to process a chained list . . . 486
Example: using pointers to process a chained list 487
Using procedure and function pointers 489
Passing return-code information 490
Understanding the RETURN-CODE special register 490
Using PROCEDURE DIVISION RETURNING 490
Specifying CALL . . . RETURNING 490
Sharing data by using the EXTERNAL clause. . . 491
Sharing files between programs (external files) . . 491
Example: using external files 492
Input-output using external files 492
Using command-line arguments 495
Example: command-line arguments without -host option 495
Example: command-line arguments with -host option 496

Chapter 28. Building shared libraries 497
Static linking and shared libraries 497
How the linker resolves references to shared libraries 498
Example: creating a sample shared library. . . 498
Example 1: alpha.cbl 499
Example 2: beta.cbl 499
Example 3: gamma.cbl 499
Procedure 499
Example: creating a makefile for the sample shared library 501
Example: calling shared libraries using CALL identifier 501
dl1.cbl. 502
dlla.cbl 502
Procedure 502
Example: COBOL program calling COBOL and C shared libraries 503
Example 1: d16.cbl 503
Example 2: d16a.c 504
Example 3: d16b.cbl 504
Procedure 504

Chapter 29. Preparing COBOL programs for multithreading 505
Multithreading 505
Working with language elements with multithreading 506
Working with elements that have run-unit scope 507
Working with elements that have program invocation instance scope 507

Scope of COBOL language elements with multithreading 508
Choosing THREAD to support multithreading . . 508
Transferring control to multithreaded programs 509
Ending multithreaded programs 509
Handling COBOL limitations with multithreading 510
Example: using COBOL in a multithreaded environment. 510
Source code for thrcob.c 511
Source code for subd.cbl. 512
Source code for sube.cbl 512

Chapter 30. Preinitializing the COBOL runtime environment 513
Initializing persistent COBOL environment . . . 513
Terminating preinitialized COBOL environment 514
Example: preinitializing the COBOL environment 515

Chapter 31. Processing two-digit-year dates 519
Millennium language extensions (MLE) 520
Principles and objectives of these extensions . . 520
Resolving date-related logic problems 521
Using a century window 522
Example: century window 523
Using internal bridging 523
Example: internal bridging 524
Moving to full field expansion. 525
Example: converting files to expanded date form 525
Using year-first, year-only, and year-last date fields 527
Compatible dates 527
Example: comparing year-first date fields . . . 528
Using other date formats 528
Example: isolating the year 529
Manipulating literals as dates 529
Assumed century window 530
Treatment of nondates 531
Using sign conditions 532
Performing arithmetic on date fields. 533
Allowing for overflow from windowed date fields 533
Specifying the order of evaluation 534
Controlling date processing explicitly 535
Using DATEVAL 535
Using UNDATE 536
Example: DATEVAL 536
Example: UNDATE 536
Analyzing and avoiding date-related diagnostic messages 537
Avoiding problems in processing dates 538
Avoiding problems with packed-decimal fields 539
Moving from expanded to windowed date fields 539

Chapter 25. Porting applications between platforms

Your AIX workstation has a different hardware and operating-system architecture than IBM mainframes or Windows-based workstations. Because of these differences, some problems can arise as you move COBOL programs between these environments.

The related tasks and reference below describe some of the differences between development platforms, and provide instructions to help you minimize portability problems.

RELATED TASKS
"Getting mainframe applications to compile"
"Getting mainframe applications to run: overview" on page 458
"Writing code to run on the mainframe" on page 462
"Writing applications that are portable between AIX and Windows" on page 462

RELATED REFERENCES
Appendix B, "Summary of differences with host COBOL," on page 573

Getting mainframe applications to compile

If you move programs to the AIX workstation from the mainframe and compile them in the new environment, you need to choose the right compiler options and be aware of language features that differ from mainframe COBOL. You can also use the `COPY` statement to help port programs.

Choosing the right compiler options: The `NOADV` COBOL compiler option, available on the mainframe, is not applicable under COBOL for AIX, and is treated as a comment. It might yield unpredictable results. The compiler flags it with a W-level message:

`NOADV` Programs that require the use of `NOADV` are sensitive to device control characters and almost certainly are not portable. If your program relies on `NOADV`, revise the program such that it does not assume a printer control character as the first character of level-01 records in a file.

| For additional information about host compiler options that affect portability, see
| the related reference below about compiler options.

Allowing for language features of mainframe COBOL: Several language features that are valid in mainframe COBOL programs can create errors or unpredictable
| results when compiled with COBOL for AIX. For details, see the related reference
| below about language elements.

Using the `COPY` statement to help port programs: In many cases, you can avoid potential portability problems by using the `COPY` statement to isolate platform-specific code. For example, you can include platform-specific code in a compilation for a given platform and exclude it from compilation for a different platform. You can also use the `COPY REPLACING` phrase to globally change nonportable source code elements, such as file-names.

RELATED REFERENCES
"Runtime environment variables" on page 211
"Compiler options" on page 573
"Language elements" on page 577
COPY statement (*COBOL for AIX Language Reference*)

Getting mainframe applications to run: overview

After you download a source program and successfully compile it on the AIX workstation, the next step is to run the program. In many cases, you can get the same results as on the mainframe without greatly modifying the source.

To assess whether to modify the source, you need to know how to fix the elements and behavior of the COBOL language that vary due to the underlying hardware or software architecture.

RELATED TASKS
"Fixing differences caused by data representations"
"Fixing environment differences that affect portability" on page 460
"Fixing differences caused by language elements" on page 461

Fixing differences caused by data representations

To ensure the same behavior for your programs, you should understand the differences in certain ways of representing data, and take appropriate action.

COBOL stores signed packed-decimal and binary data in the same manner on both the mainframe and the workstation. However, external-decimal, floating-point, and unsigned packed-decimal data are by default represented differently. Character data might be represented differently, depending on the `USAGE` clause that describes data items and the locale that is in effect at run time.

Most programs behave the same on the workstation and the mainframe regardless of the data representation.

RELATED TASKS
"Handling differences in ASCII SBCS and EBCDIC SBCS characters"
"Handling differences in IEEE and hexadecimal data" on page 459
"Handling differences in ASCII multibyte and EBCDIC DBCS strings" on page 460

RELATED REFERENCES
"Data representation" on page 573

Handling differences in ASCII SBCS and EBCDIC SBCS characters

To avoid problems with the different data representation between ASCII and EBCDIC characters, use the `CHAR(EBCDIC)` compiler option.

The AIX workstation uses the ASCII character set, and the mainframe uses the EBCDIC character set. Therefore, most characters have a different hexadecimal value, as shown in the following table.

Table 62. **ASCII characters contrasted with EBCDIC**

Character	Hexadecimal value if ASCII	Hexadecimal value if EBCDIC
'0' through '9'	X'30' through X'39'	X'F0' through X'F9'
'a'	X'61'	X'81'
'A'	X'41'	X'C1'
blank	X'20'	X'40'

Also, code that depends on the EBCDIC hexadecimal values of character data probably fails when the character data has ASCII values, as shown in the following table.

Table 63. **ASCII comparisons contrasted with EBCDIC**

Comparison	Evaluation if ASCII	Evaluation if EBCDIC
'a' < 'A'	False	True
'A' < '1'	False	True
x >= '0'	If true, does not indicate whether *x* is a digit	If true, *x* is probably a digit
x = X'40'	Does not test whether *x* is a blank	Tests whether *x* is a blank

Because of these differences, the results of sorting character strings are different between EBCDIC and ASCII. For many programs, these differences have no effect, but you should be aware of potential logic errors if your program depends on the exact sequence in which some character strings are sorted. If your program depends on the EBCDIC collating sequence and you are porting it to the workstation, you can obtain the EBCDIC collating sequence by using `PROGRAM COLLATING SEQUENCE IS EBCDIC` or the `COLLSEQ(EBCDIC)` compiler option.

RELATED REFERENCES
"CHAR" on page 250
"COLLSEQ" on page 253

Handling differences in IEEE and hexadecimal data

To avoid most problems with the different representation between IEEE and hexadecimal floating-point data, use the `FLOAT(S390)` compiler option.

The AIX workstation represents floating-point data using the IEEE format. The mainframe uses the zSeries hexadecimal format. The following table summarizes the differences between normalized floating-point IEEE and normalized hexadecimal for `USAGE COMP-1` data and `USAGE COMP-2` data.

Table 64. **IEEE contrasted with hexadecimal**

Specification	IEEE for `COMP-1` data	Hexadecimal for `COMP-1` data	IEEE for `COMP-2` data	Hexadecimal for `COMP-2` data
Range	1.17E-38* to 3.37E+38*	5.4E-79* to 7.2E+75*	2.23E-308* to 1.67E+308*	5.4E-79* to 7.2E+75*
Exponent representation	8 bits	7 bits	11 bits	7 bits
Mantissa representation	23 bits	24 bits	53 bits	56 bits

Table 64. **IEEE contrasted with hexadecimal** *(continued)*

Specification	IEEE for COMP-1 data	Hexadecimal for COMP-1 data	IEEE for COMP-2 data	Hexadecimal for COMP-2 data
Digits of accuracy	6 digits	6 digits	15 digits	16 digits
* Indicates that the value can be positive or negative.				

For most programs, these differences should create no problems. However, use caution when porting if your program depends on hexadecimal representation of data.

Performance consideration: In general, zSeries floating-point representation makes a program run more slowly because the software must simulate the semantics of zSeries hardware instructions. This is a consideration especially if the `FLOAT(S390)` compiler option is in effect and a program has a large number of floating-point calculations.

"Examples: numeric data and internal representation" on page 46

RELATED REFERENCES
"FLOAT" on page 267

Handling differences in ASCII multibyte and EBCDIC DBCS strings

To obtain mainframe behavior for alphanumeric data items that contain DBCS characters, use the `CHAR(EBCDIC)` compiler option and the `SOSI` compiler option. To avoid problems with the different data representation between ASCII DBCS and EBCDIC DBCS characters, use the `CHAR(EBCDIC)` compiler option.

In alphanumeric data items, mainframe double-byte character strings (containing EBCDIC DBCS characters) are enclosed in shift codes, and AIX workstation multibyte character strings (containing ASCII DBCS, UTF-8, or EUC characters) are not enclosed in shift codes. The hexadecimal values used to represent the same characters are also different.

In DBCS data items, mainframe double-byte character strings are not enclosed in shift codes, but the hexadecimal values used to represent characters are different from the hexadecimal values used to represent the same characters in workstation multibyte strings.

For most programs, these differences should not make porting difficult. However, if your program depends on the hexadecimal value of a multibyte string, or expects that an alphanumeric character string contains a mixture of single-byte characters and multibyte characters, use caution in your coding practices.

RELATED REFERENCES
"CHAR" on page 250
"SOSI" on page 281

Fixing environment differences that affect portability

Differences in file-names and control codes between the AIX workstation and mainframe platforms can affect the portability of your programs.

File naming conventions on the AIX workstation are very different from those on the mainframe. This difference can affect portability if you use file-names in your COBOL source programs. The following file-name, for example, is valid on the AIX workstation but not on the mainframe (except perhaps in the HFS):

```
/users/joesmith/programs/cobol/myfile.cbl
```

Case sensitivity: Unlike the mainframe, AIX is case sensitive. Names used in source programs (such as uppercase file-names) should be named appropriately in AIX file directories.

Some characters that have no particular meaning on the mainframe are interpreted as control characters by AIX. This difference can lead to incorrect processing of ASCII text files. Files should not contain any of the following characters:

- X'0A' (LF: line feed)
- X'0D' (CR: carriage return)
- X'1A' (EOF: end-of-file)

If you use device-dependent (platform-specific) control codes in your programs or files, these control codes can cause problems when you try to port the programs or files to platforms that do not support the control codes. As with all other platform-specific code, it is best to isolate such code as much as possible so that you can replace it easily when you move the application to another platform.

Fixing differences caused by language elements

In general, you can expect portable COBOL programs to behave the same way on the AIX workstation as they do on the mainframe. However, be aware of the differences in file-status values used in I/O processing.

If your program responds to status key data items, you should be concerned with two issues, depending on whether the program is written to respond to the first or the second status key:

- If your program responds to the first file status data item (*data-name-1*), be aware that values returned in the 9*n* range depend on the platform. If your program relies on the interpretation of a particular 9*n* value (for example, 97), do not expect the value to have the same meaning on the workstation that it has on the mainframe. Instead, revise your program so that it responds to any 9*n* value as a generic I/O failure.
- If your program responds to the second file status data item (*data-name-8*), be aware that the values returned depend on both the platform and file system. For example, the STL file system returns values with a different record structure on the workstation than the VSAM file system does on the mainframe. If your program relies on the interpretation of the second file status data item, it is probably not portable.

RELATED TASKS
"Using file status keys" on page 161
"Using file system status codes" on page 162

RELATED REFERENCES
FILE STATUS clause (*COBOL for AIX Language Reference*)

Writing code to run on the mainframe

You can use IBM COBOL for AIX to write new applications, taking advantage of the productivity gains and increased flexibility of using your AIX workstation. However, you need to avoid using language features that are not supported by IBM mainframe COBOL.

IBM COBOL for AIX supports several language features not supported by the IBM mainframe COBOL compilers. As you write code on the AIX workstation that is intended to run on the mainframe, avoid using these features:

- Code-page names as arguments to the `DISPLAY-OF` or `NATIONAL-OF` intrinsic function
- `READ` statement using `PREVIOUS` phrase
- `START` statement using <, <=, or `NOT >` in the `KEY` phrase
- `>>CALLINTERFACE` compiler directive

Several compiler options are not available with the mainframe compilers. Do not use any of the following options in your source code if you intend to port the code to the mainframe:

- `BINARY(NATIVE)` (treated as a comment)
- `CALLINT` (treated as a comment)
- `CHAR(NATIVE)`
- `ENTRYINT` (treated as a comment)
- `FLOAT(NATIVE)`
- `PROFILE`

Be aware of the difference in file-naming conventions between AIX and other file systems. Avoid hard-coding the names of files in your source programs. Instead, use mnemonic names that you define on each platform, and map them in turn to mainframe ddnames or environment variables. You can then compile your program to accommodate the changes in file-names without having to change the source code.

Specifically, consider how you refer to files in the following language elements:

- `ACCEPT` or `DISPLAY` target names
- `ASSIGN` clause
- `COPY` statement (*text-name* and *library-name*)

Multithreaded programs on the mainframe must be recursive. Therefore, avoid coding nested programs if you intend to port your programs to the mainframe and enable them for execution in a multithreaded environment.

Writing applications that are portable between AIX and Windows

The Windows and AIX language support is almost identical. However, there are differences between the Windows and the AIX platforms that you should keep in mind.

When developing applications to be portable between the Windows-based and the AIX workstations, consider these items:

- Hard-coded file-names in source programs can lead to problems. Instead of hard-coding the names, use mnemonic names so that you can compile a program without having to change the source code. In particular, consider how you refer to files in the following language elements:
  - `ACCEPT` and `DISPLAY` statement
  - `ASSIGN` clause
  - `COPY` (*text-name* and *library-name*) statement
- Windows represents integers in *little-endian* format. AIX maintains integers in *big-endian* format. Therefore, if your Windows COBOL program depends on the internal representation of an integer, the program is probably not portable to AIX. Avoid writing programs that rely on such internal representation.
- Windows represents class national data items in little-endian format. AIX represents class national data items in big-endian format. Therefore, if your COBOL program depends on the internal representation of such data items, the program is probably not portable between COBOL for Windows and COBOL for AIX. Avoid writing programs that depend on such internal representation.

  If your program does not depend on the internal representation of class national data items, your program is probably portable between COBOL for Windows and COBOL for AIX, but you must convert file data to the representation of the target platform.

Chapter 26. Using subprograms

Many applications consist of several separately compiled programs that are linked together. If the programs call each other, they must be able to communicate. They need to transfer control and usually need access to common data.

COBOL programs that are nested within each other can also communicate. All the required subprograms for an application can be in one source file and thus require only one compilation.

RELATED CONCEPTS
"Main programs, subprograms, and calls"

RELATED TASKS
"Ending and reentering main programs or subprograms"
"Calling nested COBOL programs" on page 466
"Calling nonnested COBOL programs" on page 470
"Calling between COBOL and C/C++ programs" on page 471
"Making recursive calls" on page 478

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"
"Calling nested COBOL programs" on page 466
"Calling nonnested COBOL programs" on page 470
"Calling between COBOL and C/C++ programs" on page 471
"Making recursive calls" on page 478

Ending and reentering main programs or subprograms

Whether a program is left in its last-used state or its initial state, and to what caller it returns, can depend on the termination statements that you use.

To end execution in the main program, you must code a STOP RUN or GOBACK statement in the main program. STOP RUN terminates the run unit and closes all files opened by the main program and its called subprograms. Control is returned to the caller of the main program, which is often the operating system. GOBACK has the same effect in the main program. An EXIT PROGRAM performed in a main program has no effect.

You can end a subprogram with an EXIT PROGRAM, a GOBACK, or a STOP RUN statement. If you use an EXIT PROGRAM or a GOBACK statement, control returns to the immediate caller of the subprogram without the run unit ending. An implicit EXIT PROGRAM statement is generated if there is no next executable statement in a called program. If you end the subprogram with a STOP RUN statement, the effect is the same as it is in a main program: all COBOL programs in the run unit are terminated, and control returns to the caller of the main program.

A subprogram is usually left in its *last-used state* when it terminates with EXIT PROGRAM or GOBACK. The next time the subprogram is called in the run unit, its internal values are as they were left, except that return values for PERFORM statements are reset to their initial values. (In contrast, a main program is initialized each time it is called.)

There are some cases where programs will be in their initial state:
- A subprogram that is dynamically called and then canceled will be in the initial state the next time it is called.
- A program that has the INITIAL attribute will be in the initial state each time it is called.
- Data items defined in the LOCAL-STORAGE SECTION will be reset to the initial state specified by their VALUE clauses each time the program is called.

RELATED CONCEPTS
"Comparison of WORKING-STORAGE and LOCAL-STORAGE" on page 13

RELATED TASKS
"Calling nested COBOL programs"
"Making recursive calls" on page 478

Calling nested COBOL programs

By calling nested programs, you can create applications that use structured programming techniques. You can also call nested programs instead of PERFORM procedures to prevent unintentional modification of data items. Use either CALL *literal* or CALL *identifier* statements to make calls to nested programs.

You can call a nested program only from its directly containing program unless you identify the nested program as COMMON in its PROGRAM-ID paragraph. In that case, you can call the *common program* from any program that is nested (directly or indirectly) in the same program as the common program. Only nested programs can be identified as COMMON. Recursive calls are not allowed.

Follow these guidelines when using nested program structures:
- Code an IDENTIFICATION DIVISION in each program. All other divisions are optional.
- | Optionally make the name of each nested program unique. Although the names
| of nested programs are not required to be unique (as described in the related

reference about scope of names), making the names unique could help make your application more maintainable. You can use any valid user-defined word or an alphanumeric literal as the name of a nested program.

- In the outermost program, code any CONFIGURATION SECTION entries that might be required. Nested programs cannot have a CONFIGURATION SECTION.
- Include each nested program in the containing program immediately before the END PROGRAM marker of the containing program.
- Use an END PROGRAM marker to terminate nested and containing programs.

RELATED CONCEPTS
"Nested programs"

RELATED REFERENCES
"Scope of names" on page 469

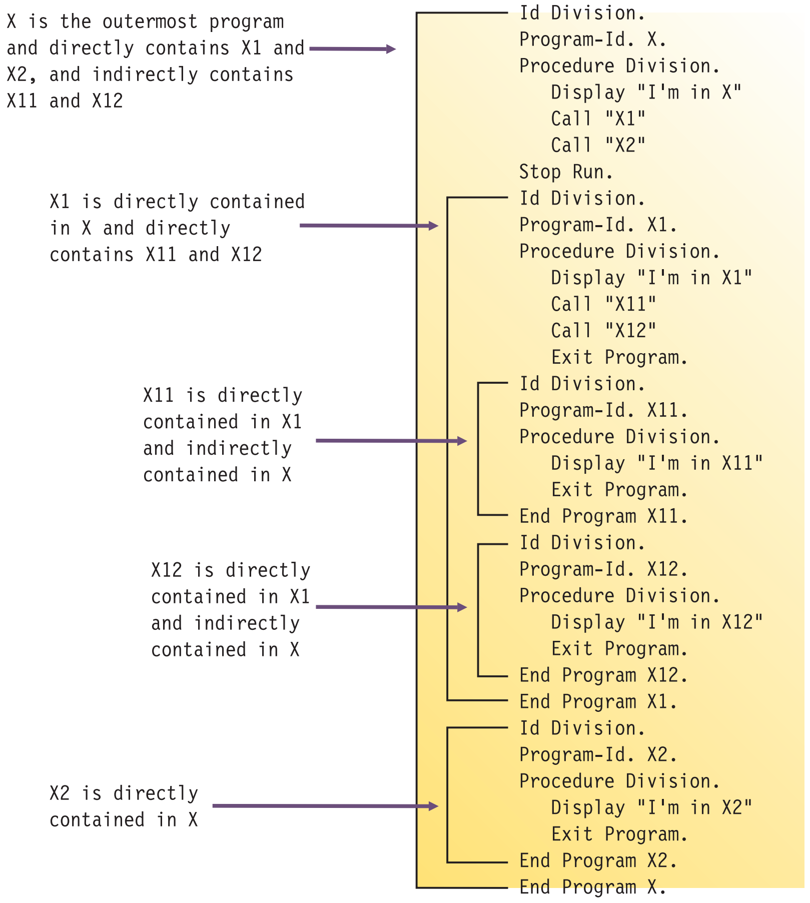
Nested programs

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:



X is the outermost program and directly contains X1 and X2, and indirectly contains X11 and X12

X1 is directly contained in X and directly contains X11 and X12

X11 is directly contained in X1 and indirectly contained in X

X12 is directly contained in X1 and indirectly contained in X

X2 is directly contained in X

```
Id Division.
Program-Id. X.
Procedure Division.
   Display "I'm in X"
   Call "X1"
   Call "X2"
Stop Run.
Id Division.
Program-Id. X1.
Procedure Division.
   Display "I'm in X1"
   Call "X11"
   Call "X12"
   Exit Program.
Id Division.
Program-Id. X11.
Procedure Division.
   Display "I'm in X11"
   Exit Program.
End Program X11.
Id Division.
Program-Id. X12.
Procedure Division.
   Display "I'm in X12"
   Exit Program.
End Program X12.
End Program X1.
Id Division.
Program-Id. X2.
Procedure Division.
   Display "I'm in X2"
   Exit Program.
End Program X2.
End Program X.
```

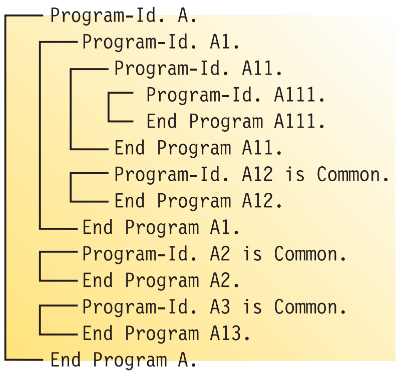
"Example: structure of nested programs"

RELATED TASKS
"Calling nested COBOL programs" on page 466

RELATED REFERENCES
"Scope of names" on page 469

Example: structure of nested programs

The following example shows a nested structure with some nested programs that are identified as COMMON.



```
Program-Id. A.
   Program-Id. A1.
      Program-Id. A11.
         Program-Id. A111.
         End Program A111.
      End Program A11.
      Program-Id. A12 is Common.
      End Program A12.
   End Program A1.
   Program-Id. A2 is Common.
   End Program A2.
   Program-Id. A3 is Common.
   End Program A13.
End Program A.
```

The following table describes the calling hierarchy for the structure that is shown in the example above. Programs A12, A2, and A3 are identified as `COMMON`, and the calls associated with them differ.

This program	Can call these programs	And can be called by these programs
A	A1, A2, A3	None
A1	A11, A12, A2, A3	A
A11	A111, A12, A2, A3	A1
A111	A12, A2, A3	A11
A12	A2, A3	A1, A11, A111
A2	A3	A, A1, A11, A111, A12, A3
A3	A2	A, A1, A11, A111, A12, A2

In this example, note that:
- A2 cannot call A1 because A1 is not common and is not contained in A2.
- A1 can call A2 because A2 is common.

Scope of names

Names in nested structures are divided into two classes: local and global. The class determines whether a name is known beyond the scope of the program that declares it. A specific search sequence locates the declaration of a name after it is referenced in a program.

Local names

Names (except the program-name) are local unless declared to be otherwise. Local names are visible or accessible only within the program in which they are declared. They are not visible or accessible to contained and containing programs.

Global names

A name that is global (indicated by using the `GLOBAL` clause) is visible and accessible to the program in which it is declared and to all the programs that are directly and indirectly contained in that program. Therefore, the contained programs can share common data and files from the containing program simply by referencing the names of the items.

Any item that is subordinate to a global item (including condition-names and indexes) is automatically global.

You can declare the same name with the `GLOBAL` clause more than one time, provided that each declaration occurs in a different program. Be aware that you can mask, or hide, a name in a nested structure by having the same name occur in different programs in the same containing structure. However, such masking could cause problems during a search for a name declaration.

Searches for name declarations

When a name is referenced in a program, a search is made to locate the declaration for that name. The search begins in the program that contains the reference and continues outward to the containing programs until a match is found. The search follows this process:

1. Declarations in the program are searched.
2. If no match is found, only global declarations are searched in successive outer containing programs.
3. The search ends when the first matching name is found. If no match is found, an error exists.

The search is for a global name, not for a particular type of object associated with the name such as a data item or file connector. The search stops when any match is found, regardless of the type of object. If the object declared is of a different type than that expected, an error condition exists.

Calling nonnested COBOL programs

A COBOL program can call a subprogram that is linked into the same executable module as the caller (*static linking*) or that is provided in a shared library (*dynamic linking*). COBOL for AIX also provides for runtime resolution of a target subprogram from a shared library.

If you link a target program statically, it is part of the executable module of the caller and is loaded with the caller. If you link dynamically or resolve a call at run time, the target program is provided in a library and is loaded either when the caller is loaded or when the target program is called.

Either dynamic or static linking of subprograms is done for COBOL `CALL` *literal*. Runtime resolution is always done for COBOL `CALL` *identifier* and is done for `CALL` *literal* if the `DYNAM` option is in effect.

| **Restriction:** You cannot mix 32-bit and 64-bit COBOL programs in an application.
| All program components within an application must be compiled using the same
| setting of the `ADDR` compiler option.

RELATED CONCEPTS
"CALL identifier and CALL literal"
"Static linking and shared libraries" on page 497

RELATED REFERENCES
"ADDR" on page 246
"DYNAM" on page 258
CALL statement (*COBOL for AIX Language Reference*)

CALL identifier and CALL literal

`CALL` *identifier*, where *identifier* is a data item that contains the name of a nonnested subprogram at run time, always results in the target subprogram being loaded when it is called. `CALL` *literal*, where *literal* is the explicit name of a nonnested target subprogram, can be resolved either statically or dynamically.

With `CALL` *identifier*, the name of the shared library must match the name of the target entry point.

With `CALL` *literal*, if the `NODYNAM` compiler option is in effect, either static or dynamic linking can be done. If `DYNAM` is in effect, `CALL` *literal* is resolved in the same way as `CALL` *identifier*: the target subprogram is loaded when it is called, and the name of the shared library must match the name of the target entry point.

These call definitions apply only in the case of a COBOL program calling a nonnested program. When a COBOL program calls a nested program, the call is resolved by the compiler without any system intervention.

Limitation: Two or more separately linked executables in an application must not statically call the same nonnested subprogram.

RELATED CONCEPTS
"Static linking and shared libraries" on page 497

RELATED TASKS
"Calling nonnested COBOL programs" on page 470

RELATED REFERENCES
"DYNAM" on page 258
CALL statement (*COBOL for AIX Language Reference*)

Calling between COBOL and C/C++ programs

You can call functions written in C/C++ from COBOL programs and can call COBOL programs from C/C++ functions.

In an interlanguage application, you can combine 64-bit COBOL programs with 64-bit C/C++ functions, or 32-bit COBOL programs with 32-bit C/C++ functions.

Restriction: You cannot mix 32-bit components and 64-bit components in an application.

Interlanguage communication between COBOL and C++: In an interlanguage application that mixes COBOL and C++, follow these guidelines:
- Specify `extern "C"` in function prototypes for COBOL programs that are called from C++, and in C++ functions that are called from COBOL.
- In COBOL, use `BY VALUE` parameters to match the normal C++ parameter convention.
- In C++, use reference parameters to match the COBOL `BY REFERENCE` convention.

The rules and guidelines referenced below provide further information about how to perform these interlanguage calls.

Unqualified references to ″C/C++″ in the referenced sections are to IBM XL C/C++.

RELATED TASKS
"Calling between COBOL and C/C++ under CICS" on page 341
"Initializing environments" on page 472
"Passing data between COBOL and C/C++" on page 472
"Collapsing stack frames and terminating run units or processes" on page 473
Chapter 27, "Sharing data," on page 481
XL C/C++ for AIX Optimization and Programming Guide

RELATED REFERENCES
"ADDR" on page 246
"COBOL and C/C++ data types" on page 473

Initializing environments

To make a call between C/C++ and COBOL, you must properly initialize the target environment.

If your main program is written in C/C++ and makes multiple calls to a COBOL program, use one of the following approaches:

- Preinitialize the COBOL environment in the C/C++ program before it calls any COBOL program. This approach is recommended because it provides the best performance.
- Put the COBOL program in an executable that is not part of the C/C++ routine that calls COBOL. Then every time that you want to call the main COBOL program, do the following steps in the C/C++ program:
  1. Load the program.
  2. Call the program.
  3. Unload the program.

RELATED CONCEPTS
Chapter 30, "Preinitializing the COBOL runtime environment," on page 513

Passing data between COBOL and C/C++

Some COBOL data types have C/C++ equivalents, but others do not. When you pass data between COBOL programs and C/C++ functions, be sure to limit data exchange to appropriate data types.

By default, COBOL passes arguments `BY REFERENCE`. If you pass an argument `BY REFERENCE`, C/C++ gets a pointer to the argument. If you pass an argument `BY VALUE`, COBOL passes the actual argument. You can use `BY VALUE` only for the following data types:

- An alphanumeric character
- | A `USAGE NATIONAL` character
- `BINARY`
- `COMP`
- `COMP-1`
- `COMP-2`
- `COMP-4`
- `COMP-5`
- `FUNCTION-POINTER`
- `OBJECT REFERENCE`
- `POINTER`
- `PROCEDURE-POINTER`

"Example: COBOL program calling C functions" on page 474
"Example: C programs that are called by and call COBOL" on page 475
"Example: COBOL program calling C++ function" on page 477

RELATED TASKS
Chapter 27, "Sharing data," on page 481

RELATED REFERENCES
"COBOL and C/C++ data types"

Collapsing stack frames and terminating run units or processes

Do not invoke functions in one language that collapse program stack frames of another language.

This guideline includes these situations:

- Collapsing some active stack frames from one language when there are active stack frames written in another language in the to-be-collapsed stack frames (C/C++ longjmp()).
- Terminating a run unit or process from one language while stack frames written in another language are active, such as issuing a COBOL `STOP RUN` or a C/C++ exit() or _exit(). Instead, structure the application in such a way that an invoked program terminates by returning to its invoker.

You can use C/C++ longjmp() or COBOL `STOP RUN` and C/C++ exit() or _exit() calls if doing so does not collapse active stack frames of a language other than the language that initiates that action. For the languages that do not initiate the collapsing and the termination, these adverse effects might otherwise occur:

- Normal cleanup or exit functions of the language might not be performed, such as the closing of files by COBOL during run-unit termination, or the cleanup of dynamically acquired resources by the involuntarily terminated language.
- User-specified exits or functions might not be invoked for the exit or termination, such as destructors and the C/C++ atexit() function.

In general, exceptions incurred during the execution of a stack frame are handled according to the rules of the language that incurs the exception. Because the COBOL implementation does not depend on the interception of exceptions through system services for the support of COBOL language semantics, you can specify the `TRAP(OFF)` runtime option to enable the exception-handling semantics of the non-COBOL language.

COBOL for AIX saves the exception environment at initialization of the COBOL runtime environment and restores it on termination of the COBOL environment. COBOL expects interfacing languages and tools to follow the same convention.

RELATED REFERENCES
"TRAP" on page 304

COBOL and C/C++ data types

The following table shows the correspondence between the data types that are available in COBOL and C/C++.

Table 65. **COBOL and C/C++ data types**

C/C++ data types	COBOL data types
wchar_t	`USAGE NATIONAL (PICTURE N)`

Table 65. **COBOL and C/C++ data types** *(continued)*

C/C++ data types	COBOL data types
char	PIC X
signed char	No appropriate COBOL equivalent
unsigned char	No appropriate COBOL equivalent
short signed int	PIC S9-S9(4) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC(BIN) compiler option.
short unsigned int	PIC 9-9(4) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC(BIN) compiler option.
long int	PIC 9(5)-9(9) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC(BIN) compiler option.
long long int	PIC 9(10)-9(18) COMP-5. Can be COMP, COMP-4, or BINARY if you use the TRUNC(BIN) compiler option.
float	COMP-1
double	COMP-2
enumeration	Analogous to level 88, but not identical
char(*n*)	PICTURE X(*n*)
array pointer (*) to type	No appropriate COBOL equivalent
pointer(*) to function	PROCEDURE-POINTER or FUNCTION-POINTER

RELATED TASKS
"Passing data between COBOL and C/C++" on page 472

RELATED REFERENCES
"TRUNC" on page 288

Example: COBOL program calling C functions

The following example shows a COBOL program that calls C functions by using the CALL statement.

The example illustrates the following concepts:
- The CALL statement does not indicate whether the called program is written in COBOL or C.
- COBOL supports calling programs that have mixed-case names.
- You can pass arguments to C programs in various ways (for example, BY REFERENCE or BY VALUE).
- You must declare a function return value on a CALL statement that calls a non-void C function.
- You must map COBOL data types to appropriate C data types.

```
 CBL PGMNAME(MIXED)
* This compiler option allows for case-sensitive names for called programs.
*
 IDENTIFICATION DIVISION.
 PROGRAM-ID. "COBCALLC".
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 N4             PIC 9(4)  COMP-5.
 01 NS4            PIC S9(4) COMP-5.
 01 N9             PIC 9(9)  COMP-5.
 01 NS9             PIC S9(9) COMP-5.
 01 NS18            USAGE COMP-2.
 01 D1              USAGE COMP-2.
 01 D2              USAGE COMP-2.
 01 R1.
     02 NR1         PIC 9(8) COMP-5.
     02 NR2         PIC 9(8) COMP-5.
     02 NR3         PIC 9(8) COMP-5.
 PROCEDURE DIVISION.
     MOVE 123 TO N4
     MOVE -567 TO NS4
     MOVE 98765432 TO N9
     MOVE -13579456 TO NS9
     MOVE 222.22 TO NS18
     DISPLAY "Call MyFun with n4=" N4 " ns4=" NS4 " N9=" n9
     DISPLAY "                ns9="  NS9" ns18="  NS18
* The following CALL illustrates several ways to pass arguments.
*
     CALL "MyFun" USING N4 BY VALUE NS4 BY REFERENCE N9 NS9 NS18
     MOVE 1024 TO N4
* The following CALL returns the C function return value.
*
     CALL "MyFunR" USING BY VALUE N4 RETURNING NS9
     DISPLAY "n4=" N4 " and ns9= n4 times n4= " NS9
     MOVE -357925680.25 TO D1
     CALL "MyFunD" USING BY VALUE D1 RETURNING D2
     DISPLAY "d1=" D1 " and d2= 2.0 times d2= " D2
     MOVE 11111 TO NR1
     MOVE 22222 TO NR2
     MOVE 33333 TO NR3
     CALL "MyFunV" USING R1
     STOP RUN.
```

RELATED TASKS
"Passing data between COBOL and C/C++" on page 472

RELATED REFERENCES
CALL statement (*COBOL for AIX Language Reference*)

Example: C programs that are called by and call COBOL

The following example illustrates that a called C function receives arguments in the order in which they were passed in a COBOL CALL statement.

The file MyFun.c contains the following source code, which calls the COBOL program tprog1:

```
#include <stdio.h>
extern void TPROG1(double *);
void
MyFun(short *ps1, short s2, long *k1, long *k2, double *m)
{
    double x;
    x = 2.0*(*m);
    printf("MyFun got s1=%d s2=%d k1=%d k2=%d x=%f\n",
             *ps1, s2, *k1,*k2, x);
}

long
MyFunR(short s1)
{
    return(s1 * s1);
}

double
```

```
MyFunD(double d1)
{
    double z;
    /* calling COBOL */
    z = 1122.3344;
    (void) TPROG1(&z);
    /* returning a value to COBOL */
    return(2.0 * d1);
}

void
MyFunV(long *pn)
{
    printf("MyFunV got %d %d %d\n", *pn, *(pn+1), *(pn+2));
}
```

MyFun.c consists of the following functions:

MyFun
Illustrates passing a variety of arguments.

MyFunR
Illustrates how to pass and return a long variable.

MyFunD
Illustrates C calling a COBOL program and illustrates how to pass and return a double variable.

MyFunV
Illustrates passing a pointer to a record and accessing the items of the record in a C program.

Example: COBOL program called by a C program

The following example shows how to write COBOL programs that are called by C programs.

The COBOL program tprog1 is called by the C function MyFunD in program MyFun.c (see "Example: C programs that are called by and call COBOL" on page 475). The called COBOL program contains the following source code:

```
*
 IDENTIFICATION DIVISION.
 PROGRAM-ID. TPROG1.
*
 DATA DIVISION.
 LINKAGE SECTION.
*
 01 X                USAGE COMP-2.
*
 PROCEDURE DIVISION USING X.
     DISPLAY "TPROG1 got x= " X
     GOBACK.
```

RELATED TASKS
"Calling between COBOL and C/C++ programs" on page 471

Example: results of compiling and running examples

This example shows how you can compile, link, and run COBOL programs cobcallc.cbl and tprog.cbl and the C program MyFun.c, and shows the results of running the programs.

Compile and link cobcallc.cbl, tprog.cbl, and MyFun.c by issuing the following commands:

1. `xlc -c MyFun.c`
2. `cob2 cobcallc MyFun.o tprog1.cbl -o cobcallc`

Run the program by issuing the command `cobcallc`. The results are as follows:

```
call MyFun with n4=00123 ns4=-00567 n9=0098765432
            ns9=-0013579456 ns18=.22222000000000000E 03
MyFun got s1=123 s2=-567 k1=98765432 k2=-13579456 x=444.440000
n4=01024 and ns9= n4 times n4= 0001048576
TPROG1 got x=  .11223344000000000E+04
d1=-.35792568025000000E+09 and d2= 2.0 times d2= -.71585136050000000E+09
MyFunV got 11111 22222 33333
```

"Example: COBOL program calling C functions" on page 474
"Example: C programs that are called by and call COBOL" on page 475
"Example: COBOL program called by a C program" on page 476

Example: COBOL program calling C++ function

The following example shows a COBOL program that calls a C++ function by using a `CALL` statement that passes arguments `BY REFERENCE`.

The example illustrates the following concepts:

- The `CALL` statement does not indicate whether the called program is written in COBOL or C++.
- You must declare a function return value on a `CALL` statement that calls a non-void C++ function.
- The COBOL data types must be mapped to appropriate C++ data types.
- The C++ function must be declared `extern "C"`.
- The COBOL arguments are passed `BY REFERENCE`. The C++ function receives them by using reference parameters.

COBOL program driver:

```
cbl pgmname(mixed)
Identification Division.
Program-Id. "driver".
Data division.
Working-storage section.
01 A pic 9(8) binary value 11111.
01 B pic 9(8) binary value 22222.
01 R pic 9(8) binary.
Procedure Division.
    Display "Hello World, from COBOL!"
    Call "sub" using by reference A B
      returning R
    Display R
    Stop Run.
```

C++ function sub:

```
#include <iostream.h>
extern "C" long sub(long& A, long& B) {
  cout << "Hello from C++" << endl;
  return A + B;
}
```

Output:

```
Hello World, from COBOL!
Hello from C++
00033333
```

RELATED TASKS
"Passing data between COBOL and C/C++" on page 472

RELATED REFERENCES
CALL statement (*COBOL for AIX Language Reference*)

Making recursive calls

A called program can directly or indirectly execute its caller. For example, program X calls program Y, program Y calls program Z, and program Z then calls program X. This type of call is *recursive*.

To make a recursive call, you must either code the `RECURSIVE` clause in the `PROGRAM-ID` paragraph of the recursively called program or specify the `THREAD` compiler option. If you try to recursively call a COBOL program that does not either specify the `THREAD` compiler option or have the `RECURSIVE` clause in the `PROGRAM-ID` paragraph, the run unit will end abnormally.

RELATED TASKS
"Identifying a program as recursive" on page 6

RELATED REFERENCES
"THREAD" on page 287
PROGRAM-ID paragraph (*COBOL for AIX Language Reference*)

Passing return codes

You can use the `RETURN-CODE` special register to pass information between separately compiled programs.

You can set `RETURN-CODE` in a called program before returning to the caller, and then test this returned value in the calling program. This technique is typically used to indicate the level of success of the called program. For example, a `RETURN-CODE` of zero can be used to indicate that the called program executed successfully.

With nonthreaded support: When a main program with nonthreaded support ends normally, the value of `RETURN-CODE` is passed to the operating system as a user return code. However, AIX restricts user return code values to 0 through 255. Therefore, if for example `RETURN-CODE` contains 258 when the program ends, AIX wraps the value within the supported range, resulting in a user return code of 2.

When a program with nonthreaded support encounters an unrecoverable exception, the run unit is terminated, and the user return code is set to 128 plus the signal number.

With threaded support: With threaded support, the value of `RETURN-CODE` is never passed to the operating system. This situation occurs either:
- When a main program ends normally.
- When a program encounters an unrecoverable exception. (In this case, the thread in which the program is executing, not the run unit, is terminated.)

RELATED REFERENCES

RETURN-CODE special register (*COBOL for AIX Language Reference*)

Chapter 27. Sharing data

When a run unit consists of several separately compiled programs that call each other, the programs must be able to communicate with each other. They also usually need access to common data.

This information describes how you can write programs that share data with other programs. In this information, a *subprogram* is any program that is called by another program.

RELATED TASKS
"Passing data"
"Coding the LINKAGE SECTION" on page 484
"Coding the PROCEDURE DIVISION for passing arguments" on page 485
"Using procedure and function pointers" on page 489
"Passing return-code information" on page 490
"Specifying CALL . . . RETURNING" on page 490
"Sharing data by using the EXTERNAL clause" on page 491
"Sharing files between programs (external files)" on page 491
"Using command-line arguments" on page 495
"Sharing data with Java" on page 448

Passing data

You can choose among three ways of passing data between programs: `BY REFERENCE`, `BY CONTENT`, or `BY VALUE`.

`BY REFERENCE`
The subprogram refers to and processes the data items in the storage of the
| calling program rather than working on a copy of the data. `BY REFERENCE` is
| the assumed passing mechanism for a parameter if none of the three ways
| is specified or implied for the parameter.

`BY CONTENT`
The calling program passes only the contents of the *literal* or *identifier*. The called program cannot change the value of the *literal* or *identifier* in the calling program, even if it modifies the data item in which it received the *literal* or *identifier*.

`BY VALUE`
The calling program or method passes the value of the *literal* or *identifier*, not a reference to the sending data item. The called program or invoked method can change the parameter. However, because the subprogram or method has access only to a temporary copy of the sending data item, any change does not affect the argument in the calling program.

Determine which of these data-passing methods to use based on what you want your program to do with the data.

Table 66. **Methods for passing data in the CALL statement**

Code	Purpose	Comments
`CALL . . . BY REFERENCE` *identifier*	To have the definition of the argument of the `CALL` statement in the calling program and the definition of the parameter in the called program share the same memory	Any changes made by the subprogram to the parameter affect the argument in the calling program.
`CALL . . . BY REFERENCE ADDRESS OF` *identifier*	To pass the address of *identifier* to a called program, where *identifier* is an item in the `LINKAGE SECTION`	Any changes made by the subprogram to the address affect the address in the calling program.
`CALL . . . BY CONTENT ADDRESS OF` *identifier*	To pass a copy of the address of *identifier* to a called program	Any changes to the copy of the address will not affect the address of *identifier*, but changes to *identifier* using the copy of the address will cause changes to *identifier*.
`CALL . . . BY CONTENT` *identifier*	To pass a copy of the identifier to the subprogram	Changes to the parameter by the subprogram will not affect the caller's identifier.
`CALL . . . BY CONTENT` *literal*	To pass a copy of a literal value to a called program	
`CALL . . . BY CONTENT LENGTH OF` *identifier*	To pass a copy of the length of a data item	The calling program passes the length of the *identifier* from its `LENGTH` special register.

A combination of `BY REFERENCE` and `BY CONTENT` such as:
`CALL 'ERRPROC'`
`   USING BY REFERENCE A`
`   BY CONTENT LENGTH OF A.`
	To pass both a data item and a copy of its length to a subprogram	

`CALL . . . BY VALUE` *identifier*	To pass data to a program, such as a C/C++ program, that uses `BY VALUE` parameter linkage conventions	A copy of the identifier is passed directly in the parameter list.
`CALL . . . BY VALUE` *literal*	To pass data to a program, such as a C/C++ program, that uses `BY VALUE` parameter linkage conventions	A copy of the literal is passed directly in the parameter list.
`CALL . . . BY VALUE ADDRESS OF` *identifier*	To pass the address of *identifier* to a called program. This is the recommended way to pass data to a C/C++ program that expects a pointer to the data.	Any changes to the copy of the address will not affect the address of *identifier*, but changes to *identifier* using the copy of the address will cause changes to *identifier*.
`CALL . . . RETURNING`	To call a C/C++ function with a function return value	

RELATED TASKS
"Describing arguments in the calling program" on page 483
"Describing parameters in the called program" on page 483
"Testing for OMITTED arguments" on page 484
"Specifying CALL . . . RETURNING" on page 490
"Sharing data by using the EXTERNAL clause" on page 491
"Sharing files between programs (external files)" on page 491
"Sharing data with Java" on page 448

RELATED REFERENCES
CALL statement (*COBOL for AIX Language Reference*)

The USING phrase (*COBOL for AIX Language Reference*)
INVOKE statement (*COBOL for AIX Language Reference*)

Describing arguments in the calling program

In the calling program, describe arguments in the DATA DIVISION in the same manner as other data items in the DATA DIVISION.

Storage for arguments is allocated only in the highest outermost program. For example, program A calls program B, which calls program C. Data items are allocated in program A. They are described in the LINKAGE SECTION of programs B and C, making the one set of data available to all three programs.

If you reference data in a file, the file must be open when the data is referenced.

Code the USING phrase of the CALL statement to pass the arguments. If you pass a data item BY VALUE, it must be an elementary item.

RELATED TASKS
"Coding the LINKAGE SECTION" on page 484
"Coding the PROCEDURE DIVISION for passing arguments" on page 485

RELATED REFERENCES
The USING phrase (*COBOL for AIX 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 484

RELATED REFERENCES
The USING phrase (*COBOL for AIX Language Reference*)

Testing for OMITTED arguments

You can specify that one or more BY REFERENCE arguments are not to be passed to a called program by coding the OMITTED keyword in place of those arguments in the CALL statement.

For example, to omit the second argument when calling program sub1, code this statement:

```
Call 'sub1' Using PARM1, OMITTED, PARM3
```

The arguments in the USING phrase of the CALL statement must match the parameters of the called program in number and position.

In a called program, you can test whether an argument was passed as OMITTED by comparing the address of the corresponding parameter to NULL. For example:

```
Program-ID. sub1.
. . .
Procedure Division Using RPARM1, RPARM2, RPARM3.
    If Address Of RPARM2 = Null Then
        Display 'No 2nd argument was passed this time'
    Else
        Perform Process-Parm-2
    End-If
```

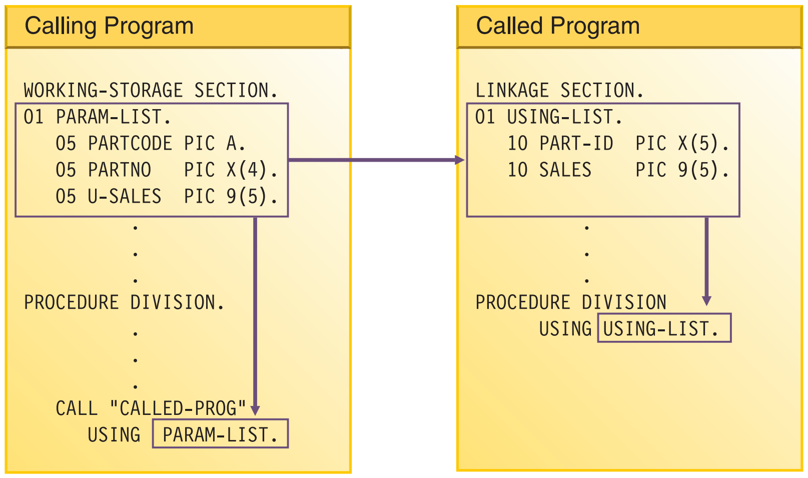
RELATED REFERENCES
CALL statement (*COBOL for AIX Language Reference*)
The USING phrase (*COBOL for AIX Language Reference*)

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):



```
WORKING-STORAGE SECTION.
01 PARAM-LIST.
   05 PARTCODE PIC A.
   05 PARTNO   PIC X(4).
   05 U-SALES  PIC 9(5).
          .
          .
          .
PROCEDURE DIVISION.
          .
          .
          .
   CALL "CALLED-PROG"
      USING  PARAM-LIST.
```

```
LINKAGE SECTION.
01 USING-LIST.
   10 PART-ID  PIC X(5).
   10 SALES    PIC 9(5).
          .
          .
          .
PROCEDURE DIVISION
      USING USING-LIST.
```

In the calling program, the code for parts (PARTCODE) and the part number (PARTNO) are distinct data items. In the called program, by contrast, the code for parts and the part number are combined into one data item (PART-ID). In the called program, a reference to PART-ID is the only valid reference to these items.

Coding the PROCEDURE DIVISION for passing arguments

If you pass an argument BY VALUE, code the USING BY VALUE clause in the PROCEDURE DIVISION header of the subprogram. If you pass an argument BY REFERENCE or BY CONTENT, you do not need to indicate in the header how the argument was passed.

```
PROCEDURE DIVISION USING BY VALUE. . .
PROCEDURE DIVISION USING. . .
PROCEDURE DIVISION USING BY REFERENCE. . .
```

The first header above indicates that the data items are passed BY VALUE; the second or third headers indicate that the items are passed BY REFERENCE or BY CONTENT.

RELATED REFERENCES
The procedure division header (*COBOL for AIX Language Reference*)
The USING phrase (*COBOL for AIX Language Reference*)
CALL statement (*COBOL for AIX 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 484

RELATED REFERENCES
CALL statement (*COBOL for AIX Language Reference*)

Handling null-terminated strings

COBOL supports null-terminated strings when you use string-handling verbs together with null-terminated literals and the hexadecimal literal X'00'.

You can manipulate null-terminated strings (passed from a C program, for example) by using string-handling mechanisms such as those in the following code:

```
01 L        pic X(20) value z'ab'.
01 M        pic X(20) value z'cd'.
01 N        pic X(20).
01 N-Length pic 99    value zero.
01 Y        pic X(13) value 'Hello, World!'.
```

To determine the length of a null-terminated string, and display the value of the string and its length, code:

```
Inspect N tallying N-length for characters before initial X'00'
Display 'N: ' N(1:N-length) ' Length: ' N-length
```

To move a null-terminated string to an alphanumeric string, but delete the null, code:

```
Unstring N  delimited by X'00' into X
```

To create a null-terminated string, code:

```
String Y      delimited by size
       X'00'  delimited by size
       into N.
```

To concatenate two null-terminated strings, code:

```
String L      delimited by x'00'
       M      delimited by x'00'
       X'00'  delimited by size
       into N.
```

RELATED TASKS
"Manipulating null-terminated strings" on page 98

RELATED REFERENCES
Null-terminated alphanumeric literals (*COBOL for AIX 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"

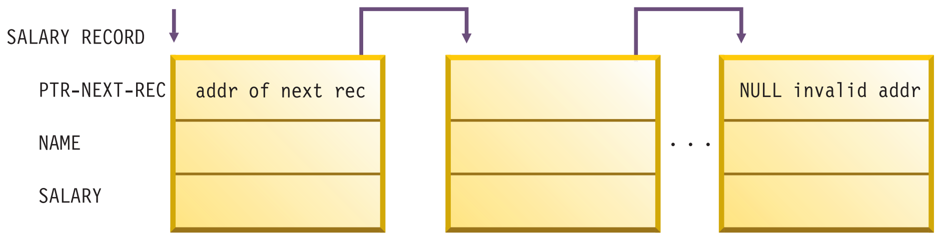
RELATED TASKS
"Coding the LINKAGE SECTION" on page 484
"Coding the PROCEDURE DIVISION for passing arguments" on page 485

RELATED REFERENCES
"ADDR" on page 246
SET statement (*COBOL for AIX Language Reference*)

Example: using pointers to process a chained list

The following example shows how you might process a linked list, that is, a chained list of data items.

For this example, picture a chained list of data that consists of individual salary records. The following figure shows one way to visualize how the records are linked in storage. The first item in each record except the last points to the next record. The first item in the last record contains a null value (instead of a valid address) to indicate that it is the last record.



The high-level pseudocode for an application that processes these records might be:

```
Obtain address of first record in chained list from routine
Check for end of the list
Do until end of the list
   Process record
   Traverse to the next record
End
```

The following code contains an outline of the calling program, LISTS, used in this example of processing a chained list.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. LISTS.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
******
 WORKING-STORAGE SECTION.
 77  PTR-FIRST        POINTER  VALUE IS NULL.                 (1)
 77  DEPT-TOTAL       PIC 9(4) VALUE IS 0.
******
 LINKAGE SECTION.
 01  SALARY-REC.
     02  PTR-NEXT-REC    POINTER.                             (2)
     02  NAME            PIC X(20).
     02  DEPT            PIC 9(4).
     02  SALARY          PIC 9(6).
 01  DEPT-X           PIC 9(4).
******
 PROCEDURE DIVISION USING DEPT-X.
******
* FOR EVERYONE IN THE DEPARTMENT RECEIVED AS DEPT-X,
* GO THROUGH ALL THE RECORDS IN THE CHAINED LIST BASED ON THE
* ADDRESS OBTAINED FROM THE PROGRAM CHAIN-ANCH
* AND CUMULATE THE SALARIES.
* IN EACH RECORD, PTR-NEXT-REC IS A POINTER TO THE NEXT RECORD
* IN THE LIST; IN THE LAST RECORD, PTR-NEXT-REC IS NULL.
* DISPLAY THE TOTAL.
******
     CALL "CHAIN-ANCH" USING PTR-FIRST                        (3)
     SET ADDRESS OF SALARY-REC TO PTR-FIRST                   (4)
******
     PERFORM WITH TEST BEFORE UNTIL ADDRESS OF SALARY-REC = NULL (5)

      IF DEPT = DEPT-X
        THEN ADD SALARY TO DEPT-TOTAL
        ELSE CONTINUE
      END-IF
      SET ADDRESS OF SALARY-REC TO PTR-NEXT-REC               (6)

     END-PERFORM
******
     DISPLAY DEPT-TOTAL
     GOBACK.
```

(1) PTR-FIRST is defined as a pointer data item with an initial value of NULL. On a successful return from the call to CHAIN-ANCH, PTR-FIRST contains the address of the first record in the chained list. If something goes wrong with the call, and PTR-FIRST never receives the value of the address of the first record in the chain, a null value remains in PTR-FIRST and, according to the logic of the program, the records will not be processed.

(2) The LINKAGE SECTION of the calling program contains the description of the records in the chained list. It also contains the description of the department code that is passed, using the USING clause of the CALL statement.

(3) To obtain the address of the first SALARY-REC record area, the LISTS program calls the program CHAIN-ANCH:

(4) The SET statement bases the record description SALARY-REC on the address contained in PTR-FIRST.

(5) The chained list in this example is set up so that the last record contains an address that is not valid. This check for the end of the chained list is accomplished with a do-while structure where the value NULL is assigned to the pointer data item in the last record.

(6) The address of the record in the LINKAGE-SECTION is set equal to the address of the next record by means of the pointer data item sent as the first field in SALARY-REC. The record-processing routine repeats, processing the next record in the chained list.

To increment addresses received from another program, you could set up the LINKAGE SECTION and PROCEDURE DIVISION like this:

```
LINKAGE SECTION.
01  RECORD-A.
    02  HEADER          PIC X(12).
    02  REAL-SALARY-REC PIC X(30).
. . .
01  SALARY-REC.
    02  PTR-NEXT-REC    POINTER.
    02  NAME            PIC X(20).
    02  DEPT            PIC 9(4).
    02  SALARY          PIC 9(6).
. . .
PROCEDURE DIVISION USING DEPT-X.
. . .
    SET ADDRESS OF SALARY-REC TO ADDRESS OF REAL-SALARY-REC
```

The address of SALARY-REC is now based on the address of REAL-SALARY-REC, or RECORD-A + 12.

RELATED TASKS
"Using pointers to process a chained list" on page 486

Using procedure and function pointers

Procedure pointers are data items defined with the USAGE IS PROCEDURE-POINTER clause. *Function pointers* are data items defined with the USAGE IS FUNCTION-POINTER clause.

In this information, "pointer" refers to either a procedure-pointer data item or a function-pointer data item. You can set either of these data items to contain entry addresses of, or pointers to, the following entry points:
- Another COBOL program that is not nested.
- A program written in another language. For example, to receive the entry address of a C function, call the function with the CALL RETURNING format of the CALL statement. It returns a pointer that you can convert to a procedure pointer by using a form of the SET statement.
- An alternate entry point in another COBOL program (as defined in an ENTRY statement).

You can set a pointer data item only by using the SET statement. For example:

```
CALL 'MyCFunc' RETURNING ptr.
SET  proc-ptr  TO ptr.
CALL proc-ptr  USING dataname.
```

Suppose that you set a pointer item to an entry address in a load module that is called by a CALL *identifier* statement, and your program later cancels that called module. The pointer item becomes undefined, and reference to it thereafter is not reliable.

RELATED REFERENCES
"ADDR" on page 246

"Migration of 32-bit applications to 64-bit mode" on page 570
PROCEDURE-POINTER phrase (*COBOL for AIX Language Reference*)
SET statement (*COBOL for AIX Language Reference*)

Passing return-code information

Use the RETURN-CODE special register to pass return codes between programs. (Methods do not return information in the RETURN-CODE special register, but they can check the register after a call to a program.)

You can also use the RETURNING phrase in the PROCEDURE DIVISION header of a method to return information to an invoking program or method. If you use PROCEDURE DIVISION . . . RETURNING with CALL . . . RETURNING, the RETURN-CODE register will not be set.

Understanding the RETURN-CODE special register

When a COBOL program returns to its caller, the contents of the RETURN-CODE special register are set according to the value of the RETURN-CODE special register in the called program.

Setting of the RETURN-CODE by the called program is limited to calls between COBOL programs. Therefore, if your COBOL program calls a C program, you cannot expect the RETURN-CODE special register of the COBOL program to be set.

For equivalent function between COBOL and C programs, have your COBOL program call the C program with the RETURNING phrase. If the C program (function) correctly declares a function value, the RETURNING value of the calling COBOL program will be set.

You cannot set the RETURN-CODE special register by using the INVOKE statement.

Using PROCEDURE DIVISION RETURNING . . .

Use the RETURNING phrase in the PROCEDURE DIVISION header of a program to return information to the calling program.

```
PROCEDURE DIVISION RETURNING dataname2
```

When the called program in the example above successfully returns to its caller, the value in *dataname2* is stored into the identifier that you specified in the RETURNING phrase of the CALL statement:

```
CALL . . . RETURNING dataname2
```

Specifying CALL . . . RETURNING

You can specify the RETURNING phrase of the CALL statement for calls to C/C++ functions or to COBOL subroutines.

The RETURNING phrase has the following format.

```
CALL . . . RETURNING dataname2
```

The return value of the called program is stored into *dataname2*. You must define *dataname2* in the DATA DIVISION of the calling program. The data type of the return value that is declared in the target function must be identical to the data type of *dataname2*.

Sharing data by using the EXTERNAL clause

Use the EXTERNAL clause to allow separately compiled programs and methods (including programs in a batch sequence) to share data items. Code EXTERNAL in the level-01 data description in the WORKING-STORAGE SECTION.

The following rules apply:

- Items that are subordinate to an EXTERNAL group item are themselves EXTERNAL.
- You cannot use the name of an EXTERNAL data item as the name for another EXTERNAL item in the same program.
- You cannot code the VALUE clause for any group item or subordinate item that is EXTERNAL.

In the run unit, any COBOL program or method that has the same data description for the item as the program that contains the item can access and process that item. For example, suppose program A has the following data description:

```
01 EXT-ITEM1     EXTERNAL     PIC 99.
```

Program B can access that data item if it has the identical data description in its WORKING-STORAGE SECTION.

Any program that has access to an EXTERNAL data item can change the value of that item. Therefore do not use this clause for data items that you need to protect.

Sharing files between programs (external files)

To enable separately compiled programs or methods in a run unit to access a file as a common file, use the EXTERNAL clause for the file.

It is recommended that you follow these guidelines:

- Use the same data-name in the FILE STATUS clause of all the programs that check the file status code.
- For each program that checks the same file status field, code the EXTERNAL clause on the level-01 data definition for the file status field.

Using an external file has these benefits:

- Even though the main program does not contain any input or output statements, it can reference the record area of the file.
- Each subprogram can control a single input or output function, such as OPEN or READ.
- Each program has access to the file.

"Example: using external files" on page 492

RELATED TASKS
"Using data in input and output operations" on page 12

RELATED REFERENCES
EXTERNAL clause (*COBOL for AIX Language Reference*)

Example: using external files

The following example shows the use of an external file in several programs. COPY statements ensure that each subprogram contains an identical description of the file.

The table below describes the main program and subprograms.

Name	Function
ef1	The main program, which calls all the subprograms and then verifies the contents of a record area
ef1openo	Opens the external file for output and checks the file status code
ef1write	Writes a record to the external file and checks the file status code
ef1openi	Opens the external file for input and checks the file status code
ef1read	Reads a record from the external file and checks the file status code
ef1close	Closes the external file and checks the file status code

Each program uses three copybooks:
- efselect is placed in the FILE-CONTROL paragraph.
```
Select ef1
Assign To ef1
File Status Is efs1
Organization Is Sequential.
```
- effile is placed in the FILE SECTION.
```
Fd ef1 Is External
          Record Contains 80 Characters
          Recording Mode F.
01  ef-record-1.
    02 ef-item-1  Pic X(80).
```
- efwrkstg is placed in the WORKING-STORAGE SECTION.
```
01  efs1         Pic 99 External.
```

Input-output using external files

```
 Identification Division.
 Program-Id.
     ef1.
*
* This main program controls external file processing.
*
 Environment Division.
 Input-Output Section.
 File-Control.
     Copy efselect.
 Data Division.
 File Section.
     Copy effile.
 Working-Storage Section.
     Copy efwrkstg.
 Procedure Division.
     Call "ef1openo"
     Call "ef1write"
     Call "ef1close"
     Call "ef1openi"
```

```
         Call "ef1read"
         If ef-record-1 = "First record" Then
           Display "First record correct"
         Else
           Display "First record incorrect"
           Display "Expected: " "First record"
           Display "Found   : " ef-record-1
         End-If
         Call "ef1close"
         Goback.
     End Program ef1.
     Identification Division.
     Program-Id.
         ef1openo.
    *
    * This program opens the external file for output.
    *
     Environment Division.
     Input-Output Section.
     File-Control.
         Copy efselect.
     Data Division.
     File Section.
         Copy effile.
     Working-Storage Section.
         Copy efwrkstg.
     Procedure Division.
         Open Output ef1
         If efs1 Not = 0
           Display "file status " efs1 " on open output"
           Stop Run
         End-If
         Goback.
     End Program ef1openo.
     Identification Division.
     Program-Id.
         ef1write.
    *
    * This program writes a record to the external file.
    *
     Environment Division.
     Input-Output Section.
     File-Control.
         Copy efselect.
     Data Division.
     File Section.
         Copy effile.
     Working-Storage Section.
         Copy efwrkstg.
     Procedure Division.
         Move "First record" to ef-record-1
         Write ef-record-1
         If efs1 Not = 0
           Display "file status " efs1 " on write"
           Stop Run
         End-If
         Goback.
     End Program ef1write.
     Identification Division.
     Program-Id.
         ef1openi.
    *
    * This program opens the external file for input.
    *
     Environment Division.
     Input-Output Section.
     File-Control.
         Copy efselect.
     Data Division.
     File Section.
         Copy effile.
     Working-Storage Section.
         Copy efwrkstg.
     Procedure Division.
         Open Input ef1
         If efs1 Not = 0
           Display "file status " efs1 " on open input"
           Stop Run
         End-If
         Goback.
     End Program ef1openi.
     Identification Division.
     Program-Id.
         ef1read.
    *
    * This program reads a record from the external file.
    *
     Environment Division.
     Input-Output Section.
     File-Control.
         Copy efselect.
     Data Division.
     File Section.
         Copy effile.
     Working-Storage Section.
         Copy efwrkstg.
     Procedure Division.
         Read ef1
         If efs1 Not = 0
           Display "file status " efs1 " on read"
           Stop Run
         End-If
         Goback.
     End Program ef1read.
     Identification Division.
     Program-Id.
         ef1close.
    *
    * This program closes the external file.
    *
     Environment Division.
     Input-Output Section.
     File-Control.
         Copy efselect.
     Data Division.
     File Section.
         Copy effile.
     Working-Storage Section.
         Copy efwrkstg.
     Procedure Division.
         Close ef1
         If efs1 Not = 0
           Display "file status " efs1 " on close"
           Stop Run
         End-If
         Goback.
     End Program ef1close.
```

Using command-line arguments

You can pass arguments to a main program on the command line. The operating system calls main programs with null-terminated strings that contain the arguments.

If the arguments that are entered are shorter than the COBOL data-names that receive them, use the technique to isolate them that is shown in the related task below about manipulating null-terminated strings.

How the arguments are treated depends on whether you use the `-host` option of the `cob2` command (or one of its variants).

If you do not specify the `-host` option, command-line arguments are passed in native data format, and AIX calls all main programs with the following arguments:

- Number of command-line arguments plus one
- Pointer to the name of the program
- Pointer to the first argument
- Pointer to the second argument
- . . .
- Pointer to the *n*th argument

If you specify the `-host` option, AIX calls all main programs with an EBCDIC character string that has a halfword prefix that contains the string length. You must enter the command-line arguments as a single string enclosed in quotation marks ("). To pass a quotation-mark character in the string, precede the quotation mark with the backslash (\) escape character.

"Example: command-line arguments without -host option"
"Example: command-line arguments with -host option" on page 496

RELATED TASKS
"Manipulating null-terminated strings" on page 98
"Handling null-terminated strings" on page 486

RELATED REFERENCES
"cob2, cob2_j, and cob2_r options" on page 227

Example: command-line arguments without -host option

This example shows how to read command-line arguments.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. "targaix".
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
*
 LINKAGE SECTION.
 01  PARM-LEN PIC S9(9) COMP.
 01  OS-PARM.
     02 PARMPTR-TABLE OCCURS 1 TO 100 TIMES DEPENDING ON PARM-LEN.
        03 PARMPTR  POINTER.
 01 PARM-STRING PIC XX.
```

```
*
 PROCEDURE DIVISION USING BY VALUE PARM-LEN BY REFERENCE OS-PARM.
     display "parm-len=" parm-len
     SET ADDRESS OF PARM-STRING TO PARMPTR(2).
     display "parm-string= '" PARM-STRING "'";
     EVALUATE PARM-STRING
         when "01"  display "case one"
         when "02"  display "case two"
         when "95"  display "case ninety-five"
         when other display "case unknown"
     END-EVALUATE
     GOBACK.
```

Suppose you compile and run the following program:

```
cob2 targaix.cbl
a.out 95
```

The result is:

```
parm-len=000000002
parm-string= '95'
case ninety-five
```

Example: command-line arguments with -host option

This example shows how to read the command-line arguments.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. "testarg".
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
*
 linkage section.
 01  os-parm.
     05 parm-len         pic s999 comp.
     05 parm-string.
         10 parm-char    pic x occurs 0 to 100 times
                         depending on parm-len.
*
 PROCEDURE DIVISION using os-parm.
     display "parm-len=" parm-len
     display "parm-string='" parm-string "'"
     evaluate parm-string
       when "01"  display "case one"
       when "02"  display "case two"
       when "95"  display "case ninety-five"
       when other display "case unknown"
     end-evaluate
     GOBACK.
```

Suppose you compile and run the program as follows:

```
cob2 -host testarg.cbl
a.out "95"
```

Then the resulting output is:

```
parm-len=002
parm-string='95'
case ninety-five
```

Chapter 28. Building shared libraries

By using linking, you can have a program call another program that is not contained in the source code of the caller. Before or during execution, the object module of the calling program is linked with the object module of the called program.

You can use a shared library as a library of frequently used functions. Even if the shared library holds only one function, the shared library can provide that function to calling programs as needed.

In COBOL, a *shared library* is a collection of outermost programs. Although these outermost programs can contain nested programs, programs external to the shared library can call only the outermost programs (known as *entry points*). Just as you can compile and link several COBOL programs as a single executable file, you link one or more compiled outermost COBOL programs to create a shared library.

Because outermost programs in a shared library are part of a library of programs, each program is referred to as a *subprogram* in the shared library. Even if a shared library provides only one program, that program is considered a subprogram of the shared library.

You can call shared COBOL libraries and call a mix of COBOL and C shared libraries.

"Example: creating a sample shared library" on page 498
"Example: calling shared libraries using CALL identifier" on page 501
"Example: COBOL program calling COBOL and C shared libraries" on page 503

RELATED CONCEPTS
"Static linking and shared libraries"
"How the linker resolves references to shared libraries" on page 498

Static linking and shared libraries

Static linking occurs when a calling program is linked to a called program in a single executable module. When the program is loaded, the operating system places into memory a single file that contains the executable code and data.

The result of statically linking programs is an executable file or subprogram in a shared library that contains the executable code for multiple programs.

The primary advantage of static linking is that you can create self-contained, independent programs. In other words, the executable program consists of one part (the executable file) that you need to keep track of. Static linking has these disadvantages:

- Because linked external programs are built into the executable files, these files become larger.
- You cannot change the behavior of executable files without relinking them.
- External called programs cannot be shared. Therefore, if more than one calling program needs to access the called programs, duplicate copies of the called programs must be loaded in memory.

To overcome these disadvantages, use shared libraries.

With shared libraries, several programs can use a single copy of an executable module. The executable module is completely separate from the programs that use it. You can build several subprograms into a shared library, and calling programs can use these subprograms as if they were part of the executable code of the calling program. You can change the shared library subprograms without recompiling or relinking the calling program.

Shared libraries typically provide common functions that can be used by a number of programs. For example, you can use shared libraries to implement subprogram packages, subsystems, and interfaces in other programs or to create object-oriented class libraries.

You can dynamically link with the supplied runtime shared libraries as well as with your own COBOL shared libraries.

"Example: creating a sample shared library"

RELATED CONCEPTS
"How the linker resolves references to shared libraries"

How the linker resolves references to shared libraries

When you compile a program, the compiler generates an object module for the code in the program. If you use any subprograms (*functions* in C, *subroutines* in other languages) that are in an external object module, the compiler adds an external program reference to the program's object module.

The linker resolves these external references. If it finds a reference to an external subprogram in an import library or in a module definition file of a shared library, the code for the external subprogram is in a shared library. To resolve an external reference to a shared library, the linker adds information to the executable file that tells the loader where to find the shared library code when the executable file is loaded.

The linker does not resolve all references to shared libraries by COBOL `CALL` statements. With the `DYNAM` compiler option in effect, COBOL resolves `CALL` *identifier* and `CALL` *literal* statements when these calls are executed.

"Example: creating a sample shared library"

RELATED CONCEPTS
"Static linking and shared libraries" on page 497

RELATED REFERENCES
"Linker input and output files" on page 232

Example: creating a sample shared library

Each of the following three example files (alpha.cbl, beta.cbl, and gamma.cbl) contains a COBOL program. You can create a sample shared library by using the example programs and the procedure that follows.

Example 1: alpha.cbl

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. alpha.
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 hello1   pic x(30)  value is "message from alpha to beta".
 01 hello2   pic x(30)  value is "message from alpha to gamma".
*
 PROCEDURE DIVISION.
     display "alpha begins"
     call "beta" using hello1
     display "alpha after beta"
     call "gamma" using hello2
     display "alpha after gamma"
     goback.
```

Example 2: beta.cbl

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. beta.
*
 ENVIRONMENT DIVISION.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
*
 Linkage section.
 01 msg   pic x(30).
*
 PROCEDURE DIVISION using msg.
     DISPLAY "beta gets msg=" msg.
     goback.
```

Example 3: gamma.cbl

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. gamma.
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
*
 Linkage section.
 01 msg           pic x(30).
*
PROCEDURE DIVISION using msg.
    DISPLAY "gamma gets msg=" msg.
    goback.
```

Procedure

The simplest method of running these programs is to compile them into a single module by using the following command:

```
cob2 -o m_abg alpha.cbl beta.cbl gamma.cbl
```

You can then run the program by issuing the command `m_abg`, which results in the following output:

```
alpha begins
beta gets msg=message from alpha to beta
alpha after beta
gamma gets msg=message from alpha to gamma
alpha after gamma
```

Instead of compiling the programs into a single module, you can put beta and gamma in a shared library (such as sh_bg), and put alpha.cbl in a module that uses that library. In this case, do these steps:

1. (Optional) Create an export file (such as bg.exp) in which you define the symbols that the shared library exports. The contents of bg.exp are:

   ```
   #!
   * symbols exported by shared library sh_bg (or archive libbg.a)
   BETA
   GAMMA
   ```

   The symbol names in the file are uppercase because the COBOL default is to use uppercase names for external symbols. If you need mixed-case names, use the `PGMNAME(MIXED)` compiler option.

   If you use an export file called bg.exp, you must use the `-bE:bg.exp` option when you create the shared library, as shown in the next step.
2. Combine beta and gamma into a shared library object called sh_bg by using the following command:

   ```
   cob2 -o sh_bg  beta.cbl gamma.cbl {-bE:bg.exp | -bexpall} -bM:SRE -bnoentry
   ```

   This command provides the following information to the compiler and linker:
   - `-o sh_bg beta.cbl gamma.cbl` names the output module sh_bg, and compiles and links beta.cbl and gamma.cbl.
   - `-bE:bg.exp` tells the linker to export the symbols that are named in export file bg.exp. Use this option only if you created an export file as shown in step 1 above.
   - `-bexpall` tells the linker to export most global symbols. Use this option if you are not using an export file.
   - `-bM:SRE` tells the linker to make the output module a shared reentrant object.
   - `-bnoentry` tells the linker that this object does not have a main entry point (it will be entered at BETA and GAMMA).
3. You can use sh_bg immediately or, if you have several shared objects, you might want to put them in an archive library for convenience. To make an archive library, use the following command:

   ```
   ar qv libbg.a sh_bg
   ```
4. Recompile alpha.cbl, this time using the archive library, by issuing *one* of the following commands:

   ```
   cob2 -o m_alpha alpha.cbl -L. libbg.a
   cob2 -o m_alpha alpha.cbl -L. -lbg
   ```

   `-l`*`xx`* is a standard AIX convention for `lib`*`xx`*`.a`. The `-L.` option tells the linker that libbg.a is in the current directory (`.`). Specifying the full directory is safer, as in:

   ```
   cob2 -o m_alpha alpha.cbl -L/usr/jane/libraries -lbg
   ```

   The `-L.` form works only if m_alpha and libbg.a are in the current directory.

As an alternative to the last three steps, you can create a makefile that combines the commands.

"Example: creating a makefile for the sample shared library" on page 501

RELATED TASKS
"Passing options to the linker" on page 230

RELATED REFERENCES
"cob2, cob2_j, and cob2_r options" on page 227
"Linker input and output files" on page 232
AIX General Programming Concepts: Writing and Debugging Programs
AIX Commands Reference, Volume 1 (ar command)

Example: creating a makefile for the sample shared library

The following example shows how you can create a makefile for the sample shared library. (The makefile assumes that the export file bg.exp defines the symbols that the shared library exports.)

```
#
#
all:     m_abg libbg.a m_alpha

#  Create m_abg containing alpha, beta, and gamma

m_abg:   alpha.cbl beta.cbl gamma.cbl
    cob2 -o m_abg alpha.cbl beta.cbl gamma.cbl

#  Create libbg.a containing beta and gamma
#  sh_bg.o is a shared object that exports the symbols defined in bg.exp
#  libbg.a is a shared library that contains one shared object
#  More objects can be added to the library

libbg.a: beta.cbl gamma.cbl bg.exp
    rm -rf libbg.a
    cob2 -o sh_bg  beta.cbl gamma.cbl -bE:bg.exp -bM:SRE -bnoentry
    ar qv libbg.a sh_bg
    rm -f sh_bg

#  Create m_alpha containing alpha and using library libbg.a
m_alpha: alpha.cbl
    cob2 -o m_alpha alpha.cbl -L. libbg.a

clean:
    rm -f m_abg m_alpha sh_bg libbg.a *.lst
```

At the end of the program, the `rm` command removes sh_bg and the old copy of libbg.a.

Executing the command `m_abg` or the command `m_alpha` provides the same output.

Shared libraries are widely used in AIX. The COBOL run time is in several shared libraries, as is the C run time. Shared libraries provide a convenient and efficient way of packaging applications.

Example: calling shared libraries using CALL identifier

The following example shows that you can create and run modules that use `CALL` *identifier* to call shared libraries.

The first program, dl1.cbl, uses `CALL` *identifier* to call the second program, dl1a.cbl.

dl1.cbl

```
* Simple dynamic call to dl1a

 Identification Division.
 Program-id.    dl1.
*
 Environment Division.
 Configuration Section.
 Input-Output Section.
 File-control.
*
 Data Division.
 File Section.
 Working-storage Section.
 01 var pic x(10).
 Linkage Section.
*
 Procedure Division.
     move "Dl1A" to var.
     display "Calling " var.
     call var.
     move "dl1a      " to var.
     display "Calling " var.
     call var.
     stop run.
 End program dl1.
```

dlla.cbl

```
* Called by dl1.cbl by call identifier.

 IDENTIFICATION DIVISION.
 PROGRAM-ID. dl1a.
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 OBJECT-COMPUTER. ANY-THING.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 77 num pic 9(4) binary value is zero.
*
 PROCEDURE DIVISION.
 LA-START.
     display "COBOL DL1A function." upon console.
     add 11 to num.
     display "num = " num
     goback.
```

Procedure

To create and run the example, do these steps:

1. Enter `cob2 dl1.cbl -o dl1` to generate the executable module `dl1`.
2. Enter `cob2 dl1a.cbl -o DL1A` to generate the executable module `DL1A`.
3. Enter the command `export COBPATH=.` to cause the current directory to be searched for a dynamic call.
4. Enter `dl1` to run the program.

`DL1A` is in uppercase characters: unless you use the `PGMNAME(MIXED)` compiler option, the program-name is changed to uppercase characters.

Restrictions on runtime loading: When you make a call to a shared library using `CALL` *identifier* or `CALL` *literal* with the `DYNAM` option:

- You can export only one name.
- The exported name must be the same as the name of the shared library.
- If the shared library is a COBOL shared library, you must link with the same COBOL library as was the shared library: `libcob2s.a` or `libcob2_r.a`.

The default library is determined by the `-lcob2s` and `-lcob2_r` command flags. Flag `-lcob2s` tells the linker to use `libcob2s.a`; `-lcob2_r` tells the linker to use `libcob2_r.a`.

Because the target file-name of a `CALL` *identifier* must match the name of the program, you must generate the executable module in the above example as `DL1A`, not as `dl1a`.

Example: COBOL program calling COBOL and C shared libraries

This example shows that a main COBOL program can dynamically call both a COBOL program and a C function.

The example consists of three programs:
- dl6.cbl: A COBOL main program that uses `CALL` *identifier* to call dl6a and dl6b
- dl6a.c: A C function that dl6 calls
- dl6b.cbl: A COBOL program that dl6 calls

Example 1: d16.cbl

```
 cbl pgmname(longmixed)
* dl6:
* - call C function - dl6a executable
* - call "dl6b" with pgmname(mixed)
 Identification Division.
 Program-id.    dl6.
*
 Environment Division.
 Configuration Section.
 Input-Output Section.
 File-control.
*
 Data Division.
 File Section.
 Working-storage Section.
 01 var pic x(10).
 Linkage Section.
*
 Procedure Division.
     move "dl6a" to var.
     display "Calling " var.
     call var.
     move "dl6a" to var.
     display "Calling " var.
     call var.
     move "dl6b" to var.
     display "Calling " var.
     call var.
     move "dl6b" to var.
     display "Calling " var.
     call var.
     stop run.
 End program dl6.
```

Example 2: dl6a.c

```
void dl6a() {
  static int n = 123;
  printf("C dl6a function.\n");
  n = n * 10;
  printf("n = %d\n", n);
}
```

Example 3: dl6b.cbl

```
cbl pgmname(longmixed)
 IDENTIFICATION DIVISION.
 PROGRAM-ID. "dl6b".
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 OBJECT-COMPUTER. ANY-THING.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 77 num pic 9(4) binary value is zero.
*
 PROCEDURE DIVISION.
 LA-START.
     display "COBOL dl6b function." upon console.
     add 111 to num.
     display "num = " num
     goback.
```

Procedure

To create and run the example, do the following steps:

1. Enter `cob2 dl6.cbl -o dl6` to generate the executable module dl6.
2. Enter `cob2 dl6b.cbl -o dl6b` to generate the executable module dl6b.
3. Enter `xlc -o dl6a -edl6a dl6a.c` to generate the executable module dl6a with an entry point dl6a.
4. Enter the command `export COBPATH=.` to set the directories to be searched for a dynamic call. In this case, the current directory is searched.
5. Enter `dl6` to run the program.

Chapter 29. Preparing COBOL programs for multithreading

Although you cannot initiate or manage program threads in a COBOL program, you can prepare a COBOL program to run in a multithreaded environment. You can run COBOL programs in multiple threads within a process.

There is no explicit COBOL language to use for multithreaded execution; rather, you compile with the `THREAD` compiler option.

After you compile COBOL programs using the `THREAD` compiler option, other applications can call these COBOL programs in such a way that the programs run in multiple threads within a process or as multiple program invocation instances within a thread.

"Example: using COBOL in a multithreaded environment" on page 510

RELATED CONCEPTS
"Multithreading"

RELATED TASKS
"Working with language elements with multithreading" on page 506
"Choosing THREAD to support multithreading" on page 508
"Transferring control to multithreaded programs" on page 509
"Ending multithreaded programs" on page 509
"Handling COBOL limitations with multithreading" on page 510

RELATED REFERENCES
"THREAD" on page 287
PROGRAM-ID paragraph (*COBOL for AIX Language Reference*)

Multithreading

To use COBOL support for multithreading, you need to understand how processes, threads, run units, and program invocation instances relate to each other.

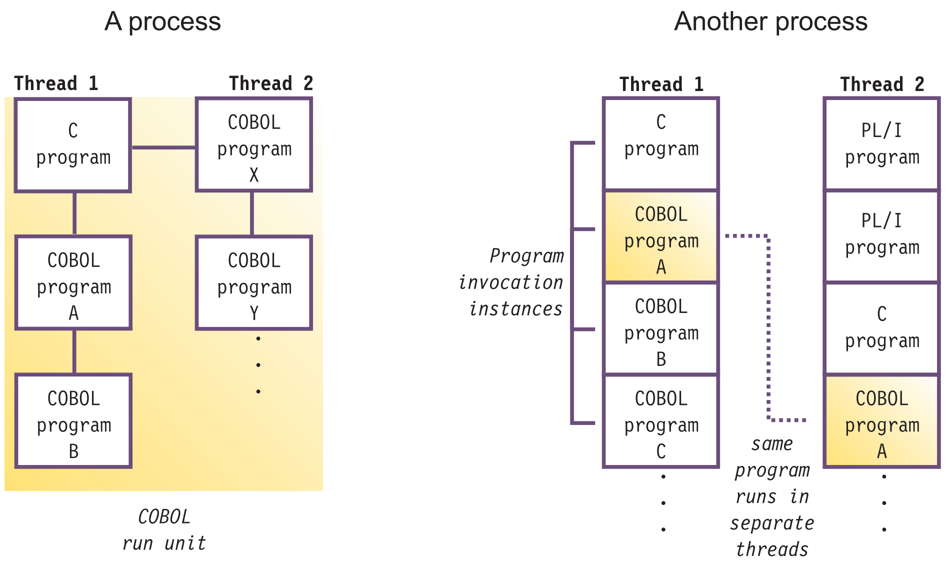
The operating system and multithreaded applications can handle execution flow within a *process*, which is the course of events when all or part of a program runs. Programs that run within a process can share resources. Processes can be manipulated. For example, they can have a high or low priority in terms of the amount of time that the system devotes to running the process.

Within a process, an application can initiate one or more *threads*, each of which is a stream of computer instructions that controls that thread. A multithreaded process begins with one stream of instructions (one thread) and can later create other instruction streams to perform tasks. These multiple threads can run concurrently. Within a thread, control is transferred between executing programs.

In a multithreaded environment, a COBOL *run unit* is the portion of the process that includes threads that have actively executing COBOL programs. The COBOL run unit continues until no COBOL program is active in the execution stack for any of the threads. For example, a called COBOL program contains a `GOBACK` statement and returns control to a C program. Within the run unit, COBOL programs can call non-COBOL programs, and vice versa.

Within a thread, control is transferred between separate COBOL and non-COBOL programs. For example, a COBOL program can call another COBOL program or a C program. Each separately called program is a *program invocation instance*. Program invocation instances of a particular program can exist in multiple threads within a given process.

The following illustration shows these relationships between processes, threads, run units, and program invocation instances.



Do not confuse multiprocessing or multithreading with *multitasking*, a term that is generally used to describe the external behavior of applications. In multitasking, the operating system appears to be running more than one application simultaneously. Multitasking is not related to multithreading as implemented in COBOL.

"Example: using COBOL in a multithreaded environment" on page 510

RELATED TASKS
"Working with language elements with multithreading"
"Choosing THREAD to support multithreading" on page 508
"Transferring control to multithreaded programs" on page 509
"Ending multithreaded programs" on page 509
"Handling COBOL limitations with multithreading" on page 510

RELATED REFERENCES
"THREAD" on page 287

Working with language elements with multithreading

Because your COBOL programs can run as separate threads within a process, a language element can be interpreted in two different scopes: run-unit scope, or program invocation instance scope. These two types of scope are important in determining where an item can be referenced and how long the item persists in storage.

Run-unit scope
: While the COBOL run unit runs, the language element persists and is available to other programs within the thread.

Program invocation instance scope
: The language element persists only within a particular instance of a program invocation.

An item can be referenced from the scope in which it was declared or from its containing scope. For example, if a data item has run-unit scope, any instance of a program invocation in the run unit can reference the data item.

An item persists in storage only as long as the item in which it is declared persists. For example, if a data item has program invocation instance scope, it remains in storage only while that instance is running.

RELATED TASKS
"Working with elements that have run-unit scope"
"Working with elements that have program invocation instance scope"

RELATED REFERENCES
"Scope of COBOL language elements with multithreading" on page 508

Working with elements that have run-unit scope

If you have resources that have run-unit scope (such as `GLOBAL` data declared in `WORKING-STORAGE`), you must synchronize access to that data from multiple threads by using logic in the application.

You can take one or both of the following actions:
- Structure the application so that you do not simultaneously access from multiple threads resources that have run-unit scope.
- If you are going to access resources simultaneously from separate threads, synchronize access by using facilities provided by C or by platform functions.

If you have resources that have run-unit scope and you want them to be isolated within an individual program invocation instance (for example, programs with individual copies of data), define the data in the `LOCAL-STORAGE SECTION`. The data will then have the scope of the program invocation instance.

Working with elements that have program invocation instance scope

With these language elements, storage is allocated for each instance of a program invocation. Therefore, even if a program is called multiple times among multiple threads, each time it is called it is allocated separate storage.

For example, if program X is called in two or more threads, each instance of X that is called gets its own set of resources, such as storage.

Because the storage associated with these language elements has the scope of a program invocation instance, data is protected from access across threads. You do not have to concern yourself with synchronizing access to data. However, this data cannot be shared between invocations of programs unless it is explicitly passed.

Scope of COBOL language elements with multithreading

The following table summarizes the scope of various COBOL language elements.

Table 67. **Scope of COBOL language elements with multithreading**

Language element	Can be referenced from:	Lifetime same as:
ADDRESS-OF special register	Same as associated record	Program invocation instance
Files	Run unit	Run unit
Index data	Program	Program invocation instance
LENGTH of special register	Same as associated identifier	Same as associated identifier
LINAGE-COUNTER special register	Run unit	Run unit
LINKAGE-SECTION data	Run unit	Based on scope of underlying data
LOCAL-STORAGE data	Within the thread	Program invocation instance
RETURN-CODE	Run unit	Program invocation instance
SORT-CONTROL, SORT-CORE-SIZE, SORT-RETURN, TALLY special registers	Run unit	Program invocation instance
WHEN-COMPILED special register	Run unit	Run unit
WORKING-STORAGE data	Run unit	Run unit

Choosing THREAD to support multithreading

Use the THREAD compiler option for multithreading support. Use THREAD if your program will be called in more than one thread in a single process by an application. 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 with the THREAD compiler option. You cannot mix programs compiled with THREAD and programs compiled with NOTHREAD in the same application.

Compile object-oriented (OO) clients and classes by using the cob2_j command. This command causes the THREAD option to automatically be included as a default invocation option.

Language restrictions: When you use the THREAD option, you cannot use certain language elements. For details, see the related reference below.

Recursion: When you compile a program using the THREAD compiler option, you can call the program recursively in a threaded or nonthreaded environment. This recursive capability is available regardless of whether you specified the RECURSIVE phrase in the PROGRAM-ID paragraph.

RELATED TASKS
"Sharing data in recursive or multithreaded programs" on page 16
"Compiling OO applications" on page 237

RELATED REFERENCES
"THREAD" on page 287

Transferring control to multithreaded programs

When you write COBOL programs for a multithreaded environment, choose appropriate program linkage statements.

As in single-threaded environments, a called program is in its initial state when it is first called within a run unit and when it is first called after a `CANCEL` to the called program.

In general, it is recommended that the programs that initiate and manage multiple threads use the COBOL preinitialization interface.

If your program initiates multiple COBOL threads (for example your C program calls COBOL programs to carry out the input and output of data), do not assume that the COBOL programs will clean up their environment, such as releasing storage no longer needed. In particular, do not assume that files will be automatically closed. You should preinitialize the COBOL environment so that your application can control the COBOL cleanup.

RELATED CONCEPTS
Chapter 30, "Preinitializing the COBOL runtime environment," on page 513

RELATED TASKS
"Ending multithreaded programs"
"Ending and reentering main programs or subprograms" on page 465

Ending multithreaded programs

You can end a multithreaded program by using `GOBACK` or `EXIT PROGRAM`.

Use `GOBACK` to return to the caller of the program. When you use `GOBACK` from the first program in a thread, the thread is terminated.

Use `EXIT PROGRAM` as you would `GOBACK`, except from a main program where it has no effect.

When the COBOL environment is not preinitialized, and the COBOL run time can determine that there are no other COBOL programs active in the run unit, the COBOL process for terminating a run unit (including closing all open COBOL files) is performed upon the `GOBACK` from the first program of this thread. This determination can be made if all COBOL programs that are called within the run unit have returned to their callers through `GOBACK` or `EXIT PROGRAM`. This determination cannot be made under certain conditions such as the following:
- A thread that has one or more active COBOL programs was terminated (for example, because of an exception or by means of `pthread_exit`).
- A longjmp was executed and resulted in collapsing active COBOL programs in the invocation stack.

There is no COBOL function that effectively does a `STOP RUN` in a threaded environment. If you need this behavior, consider calling the C exit function from your COBOL program and using `_iwzCOBOLTerm` after the runtime termination exit.

RELATED TASKS
"Ending and reentering main programs or subprograms" on page 465

Handling COBOL limitations with multithreading

Some COBOL applications depend on subsystems or other applications. In a multithreaded environment, these dependencies and others result in some limitations on COBOL programs.

In general, you must synchronize access to resources that are visible to the application within a run unit. Exceptions to this requirement are `DISPLAY` and `ACCEPT`, which you can use from multiple threads; all synchronization is provided for these by the runtime environment.

DB2: You can run a DB2 application in multiple threads. However, you must provide any needed synchronization for accessing DB2 data.

`SORT` and `MERGE`, and VSAM file I/O: `SORT` and `MERGE` should be active in only one thread at a time. Similarly, input and output for VSAM files should be active from only one thread at a time. However, the COBOL runtime environment does not enforce either of these restrictions. The application must therefore do so.

RELATED TASKS
"Making recursive calls" on page 478

Example: using COBOL in a multithreaded environment

This multithreading example consists of a C main program and two COBOL programs.

thrcob.c
A C main program that creates two COBOL threads, waits for them to finish, then exits

subd.cbl
A COBOL program that is run by the thread created by thrcob.c

sube.cbl
A second COBOL program that is run by the thread created by thrcob.c

To create and run the multithreading example, enter the following commands at an AIX shell prompt:

- To compile thrcob.c: `xlc_r -c thrcob.c`
- | To compile subd.cbl: `cob2_r -c -qTHREAD subd.cbl`
- | To compile sube.cbl: `cob2_r -c -qTHREAD sube.cbl`
- To generate the executable thrcob: `cob2_r -o thrcob thrcob.o subd.o sube.o`
- To run the program thrcob: `thrcob`

Source code for thrcob.c

```
/* Creates COBOL threads, SUBD and SUBE.
   Threads must be created with undetached state for join to work. */
typedef int (*PFN)();
#include  <pthread.h>
#include  <stdio.h>
#include  <setjmp.h>

extern int errno;
extern void *SUBD(void *);
extern void *SUBE(void *);

extern void _iwzCOBOLInit(int fcode, PFN StopFun, int *err_code, void *StopArg);
extern void _iwzCOBOLTerm(int fcode, int *err_code);

jmp_buf Jmpbuf;

int StopFun(long *stoparg)
{
        printf("inside StopFun. Got stoparg = %d\n", *stoparg);
        *stoparg = 123;
        longjmp(Jmpbuf,1);
}

long StopArg = 0;

void testrc(int rc, const char *s)
{
        if (rc != 0){
                printf("%s: Fatal error rc=%d\n",s,rc);
                exit(-1);
  }
}

void pgmy()
{
        pthread_t t1, t2;
        int rc;
        long i;
        long j;
        void *r1, *r2;
        pthread_attr_t attr;

        _iwzCOBOLInit(1, StopFun, &rc, &StopArg);
        printf( "_iwzCOBOLinit got %d\n",rc);
        pthread_attr_init(&attr);
        pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_UNDETACHED);

        i = 2000;
|       rc = pthread_create( &t1, &attr, &SUBD, &i);
        testrc(rc,"create 1");
        j = 1000;
        rc = pthread_create( &t2, &attr, &SUBE, &j);
        testrc(rc,"create 2");
        printf("threads are %x and %x\n",t1, t2);

|       rc = pthread_join(t1, &r1);
        printf("join %x got %d\n",t1,rc);
        rc = pthread_join(t2, &r2);
        printf("join %x got %d\n",t2,rc);

        _iwzCOBOLTerm(1, &rc);
        printf( "_iwzCOBOLTerm expects rc=0, got rc=%d\n",rc);
}

main()
```

```
{
        if (setjmp(Jmpbuf) ==0) (
                pgmy();
        }
)
```

Source code for subd.cbl

```
PROCESS PGMNAME(MIXED)
       IDENTIFICATION DIVISION
       PROGRAM-ID. "SUBD".
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SPECIAL NAMES.
           DECIMAL-POINT IS COMMA.
       INPUT-OUTPUT SECTION.
       DATA DIVISION.
       FILE SECTION.
       Working-Storage SECTION.

       Local-Storage Section.
       01 n2 pic 9(8) comp-5 value 0.

       Linkage Section.
       01 n1 pic 9(8) comp-5.

       PROCEDURE DIVISION using by Reference n1.
           Display "In SUBD "

           perform n1 times
              compute n2 = n2 + 1
              Display "From Thread 1: " n2
              CALL "Sleep" Using by value 1000
           end-perform

           GOBACK.
```

Source code for sube.cbl

```
PROCESS PGMNAME(MIXED)
       IDENTIFICATION DIVISION.
       PROGRAM-ID. "SUBE".
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SPECIAL-NAMES.
           DECIMAL-POINT IS COMMA.
       INPUT-OUTPUT SECTION.
       DATA DIVISION.
       FILE SECTION.
       Working-Storage SECTION.

       Local-Storage Section.
       01 n2 pic 9(8) comp-5 value 0.

       Linkage Section.
       01 n1 pic 9(8) comp-5.

       PROCEDURE DIVISION using by reference n1.

           perform n1 times
              compute n2 = n2 + 1
              Display "From Thread 2: " n2
      *Go to sleep for 3/4 sec.
              CALL "Sleep" Using by value 750
           end-perform

           GOBACK.
```

Chapter 30. Preinitializing the COBOL runtime environment

Preinitialization allows an application to initialize the COBOL runtime environment once, perform multiple executions using that environment, and then explicitly terminate the environment.

You can use preinitialization to invoke COBOL programs multiple times from a non-COBOL environment, such as C/C++.

Preinitialization has two primary benefits:

- The COBOL environment stays ready for program calls.

  Because the COBOL run unit is not terminated on return from the first COBOL program in the run unit, the COBOL programs that are invoked from a non-COBOL environment can be invoked in their last-used state.
- Performance is faster.

  Creating and taking down the COBOL runtime environment repeatedly involves overhead and can slow down your application.

Use preinitialization services for multilanguage applications where non-COBOL programs need to use a COBOL program in its last-used state. For example, a file can be opened on the first call to a COBOL program, and the invoking program expects subsequent calls to the program to find the file open.

Restriction: Preinitialization is not supported under CICS.

Use the interfaces described in the related tasks to initialize and terminate a persistent COBOL runtime environment. Any shared library that contains a COBOL program used in a preinitialized environment cannot be deleted until the preinitialized environment is terminated.

"Example: preinitializing the COBOL environment" on page 515

RELATED TASKS
"Initializing persistent COBOL environment"
"Terminating preinitialized COBOL environment" on page 514

Initializing persistent COBOL environment

Use the following interface to initialize a persistent COBOL environment.

CALL init_routine syntax

```
►►──CALL──init_routine(function_code,routine,error_code,token)──────────────►◄
```

CALL Invocation of *init_routine*, using language elements appropriate to the language from which the call is made

init_routine
The name of the initialization routine: `_iwzCOBOLInit` or `IWZCOBOLINIT`

function_code **(input)**
A 4-byte binary number, passed by value. *function_code* can be:

1 The first COBOL program invoked after this function invocation is treated as a subprogram.

routine **(input)**
Address of the routine to be invoked if the run unit terminates. The token argument passed to this function is passed to the run-unit termination exit routine. This routine, when invoked upon run-unit termination, must not return to the invoker of the routine but instead use longjmp() or exit().

If you do not provide an exit routine address, an *error_code* is generated that indicates that preinitialization failed.

error_code **(output)**
A 4-byte binary number. *error_code* can be:

0 Preinitialization was successful.

1 Preinitialization failed.

token **(input)**
A 4-byte token to be passed to the exit *routine* specified above when that routine is invoked upon run-unit termination.

RELATED TASKS
"Terminating preinitialized COBOL environment"

Terminating preinitialized COBOL environment

Use the following interface to terminate the preinitialized persistent COBOL environment.

CALL term_routine syntax

```
►►──CALL──term_routine(function_code,error_code)──────────────────────────►◄
```

`CALL` Invocation of *term_routine*, using language elements appropriate to the language from which the call is made

term_routine
The name of the termination routine: `_iwzCOBOLTerm` or `IWZCOBOLTERM`

function_code **(input)**
A 4-byte binary number, passed by value. *function_code* can be:

1 Clean up the preinitialized COBOL runtime environment as if a COBOL `STOP RUN` statement were performed; for example, all COBOL files are closed. However, the control returns to the caller of this service.

error_code **(output)**
A 4-byte binary number. *error_code* can be:

0	Termination was successful.
1	Termination failed.

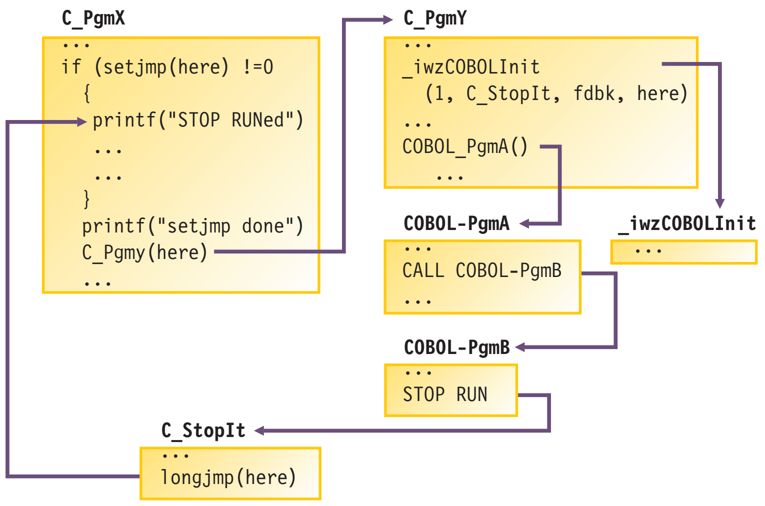
The first COBOL program called after the invocation of the preinitialization routine is treated as a subprogram. Thus a GOBACK from this (initial) program *does not* trigger run-unit termination semantics such as the closing of files. Run-unit termination (such as with STOP RUN) *does* free the preinitialized COBOL environment prior to the invocation of the run-unit exit routine.

If not active: If your program invokes the termination routine and the COBOL environment is not already active, the invocation has no effect on execution, and control is returned to the invoker with an error code of 0.

"Example: preinitializing the COBOL environment"

Example: preinitializing the COBOL environment

The following figure illustrates how the preinitialized COBOL environment works. The example shows a C program initializing the COBOL environment, calling COBOL programs, then terminating the COBOL environment.



The following example shows the use of COBOL preinitialization. A C main program calls the COBOL program XIO several times. The first call to XIO opens the file, the second call writes one record, and so on. The final call closes the file. The C program then uses C-stream I/O to open and read the file.

To test and run the program, enter the following commands from a command shell:

```
xlc -c testinit.c
cob2 testinit.o xio.cbl
a.out
```

The result is:

```
_iwzCOBOLinit got 0
xio entered with x=0000000000
xio entered with x=0000000001
xio entered with x=0000000002
xio entered with x=0000000003
xio entered with x=0000000004
xio entered with x=0000000099
StopArg=0
_iwzCOBOLTerm expects rc=0 and got rc=0
FILE1 contains ----
11111
22222
33333
---- end of FILE1
```

Note that in this example, the run unit was not terminated by a COBOL `STOP RUN`; it was terminated when the main program called `_iwzCOBOLTerm`.

The following C program is in the file `testinit.c`:

```
#ifdef _AIX
typedef int (*PFN)();
#define LINKAGE
#else
#include <windows.h>
#define LINKAGE _System
#endif

#include   <stdio.h>
#include   <setjmp.h>

extern void _iwzCOBOLInit(int fcode, PFN StopFun, int *err_code, void *StopArg);
extern void _iwzCOBOLTerm(int fcode, int *err_code);
extern void LINKAGE XIO(long *k);

jmp_buf Jmpbuf;
long StopArg = 0;

int LINKAGE
StopFun(long *stoparg)
{
        printf("inside StopFun\n");
        *stoparg = 123;
        longjmp(Jmpbuf,1);
}

main()
{
        int rc;
        long k;
        FILE *s;
        int c;

        if (setjmp(Jmpbuf) ==0) {
                _iwzCOBOLInit(1, StopFun, &rc, &StopArg);
                printf( "_iwzCOBOLinit got %d\n",rc);
                for (k=0; k <= 4; k++) XIO(&k);
                k = 99; XIO(&k);
        }
        else printf("return after STOP RUN\n");
        printf("StopArg=%d\n", StopArg);
        _iwzCOBOLTerm(1, &rc);
        printf("_iwzCOBOLTerm expects rc=0 and got rc=%d\n",rc);
        printf("FILE1 contains ---- \n");
        s = fopen("FILE1", "r");
        if (s) {
                while (  (c = fgetc(s) ) != EOF ) putchar(c);
        }
        printf("---- end of FILE1\n");
}
```

The following COBOL program is in the file `xio.cbl`:

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.     xio.
******************************************************************
 ENVIRONMENT    DIVISION.
 CONFIGURATION   SECTION.
 INPUT-OUTPUT    SECTION.
 FILE-CONTROL.
     SELECT file1 ASSIGN TO FILE1
       ORGANIZATION IS LINE SEQUENTIAL
       FILE STATUS IS file1-status.
 . . .
 DATA           DIVISION.
 FILE SECTION.
 FD FILE1.
 01 file1-id pic x(5).
 . . .
 WORKING-STORAGE SECTION.
 01 file1-status  pic xx    value is zero.
 . . .
 LINKAGE SECTION.
*
 01 x               PIC S9(8) COMP-5.
 . . .
 PROCEDURE DIVISION using x.
 . . .
     display "xio entered with x=" x
     if x = 0 then
       OPEN output FILE1
     end-if
     if x = 1  then
        MOVE ALL "1" to file1-id
        WRITE file1-id
     end-if
     if x = 2 then
        MOVE ALL "2" to file1-id
        WRITE file1-id
     end-if
     if x = 3 then
        MOVE ALL "3" to file1-id
        WRITE file1-id
     end-if
     if x = 99 then
       CLOSE file1
     end-if
     GOBACK.
```

Chapter 31. Processing two-digit-year dates

With the millennium language extensions (MLE), you can make simple changes in your COBOL programs to define date fields. The compiler recognizes and acts on these dates by using a century window to ensure consistency.

Use the following steps to implement automatic date recognition in a COBOL program:

1. Add the `DATE FORMAT` clause to the data description entries of the data items in the program that contain dates. You must identify all dates with `DATE FORMAT` clauses, even those that are not used in comparisons.
2. To expand dates, use `MOVE` or `COMPUTE` statements to copy the contents of windowed date fields to expanded date fields.
3. If necessary, use the `DATEVAL` and `UNDATE` intrinsic functions to convert between date fields and nondates.
4. Use the `YEARWINDOW` compiler option to set the century window as either a fixed window or a sliding window.
5. Compile the program with the `DATEPROC(FLAG)` compiler option, and review the diagnostic messages to see if date processing has produced any unexpected side effects.
6. When the compilation has only information-level diagnostic messages, you can recompile the program with the `DATEPROC(NOFLAG)` compiler option to produce a clean listing.

You can use certain programming techniques to take advantage of date processing and control the effects of using date fields such as when comparing dates, sorting and merging by date, and performing arithmetic operations involving dates. The millennium language extensions support year-first, year-only, and year-last date fields for the most common operations on date fields: comparisons, moving and storing, and incrementing and decrementing.

RELATED CONCEPTS
"Millennium language extensions (MLE)" on page 520

RELATED TASKS
"Resolving date-related logic problems" on page 521
"Using year-first, year-only, and year-last date fields" on page 527
"Manipulating literals as dates" on page 529
"Performing arithmetic on date fields" on page 533
"Controlling date processing explicitly" on page 535
"Analyzing and avoiding date-related diagnostic messages" on page 537
"Avoiding problems in processing dates" on page 538

RELATED REFERENCES
"DATEPROC" on page 256
"YEARWINDOW" on page 292
DATE FORMAT clause (*COBOL for AIX Language Reference*)

Millennium language extensions (MLE)

The term *millennium language extensions* (MLE) refers to the features of COBOL for AIX that the DATEPROC compiler option activates to help with logic problems that involve dates in the year 2000 and beyond.

When enabled, the extensions include:

- The DATE FORMAT clause. Add this clause to items in the DATA DIVISION to identify date fields and to specify the location of the year component within the date.

  There are several restrictions on use of the DATE FORMAT clause; for example, you cannot specify it for items that have USAGE NATIONAL. See the related references below for details.
- The reinterpretation as a date field of the function return value for the following intrinsic functions:
  - DATE-OF-INTEGER
  - DATE-TO-YYYYMMDD
  - DAY-OF-INTEGER
  - DAY-TO-YYYYDDD
  - YEAR-TO-YYYY
- The reinterpretation as a date field of the conceptual data items DATE, DATE YYYYMMDD, DAY, and DAY YYYYDDD in the following forms of the ACCEPT statement:
  - ACCEPT *identifier* FROM DATE
  - ACCEPT *identifier* FROM DATE YYYYMMDD
  - ACCEPT *identifier* FROM DAY
  - ACCEPT *identifier* FROM DAY YYYYDDD
- The intrinsic functions UNDATE and DATEVAL, used for selective reinterpretation of date fields and nondates.
- The intrinsic function YEARWINDOW, which retrieves the starting year of the century window set by the YEARWINDOW compiler option.

The DATEPROC compiler option enables special date-oriented processing of identified date fields. The YEARWINDOW compiler option specifies the 100-year window (the century window) to use for interpreting two-digit windowed years.

RELATED CONCEPTS
"Principles and objectives of these extensions"

RELATED REFERENCES
"DATEPROC" on page 256
"YEARWINDOW" on page 292
Restrictions on using date fields (*COBOL for AIX Language Reference*)

Principles and objectives of these extensions

To gain the most benefit from the millennium language extensions, you need to understand the reasons for their introduction into the COBOL language.

The millennium language extensions focus on a few key principles:

- Programs to be recompiled with date semantics are fully tested and valuable assets of the enterprise. Their only relevant limitation is that two-digit years in the programs are restricted to the range 1900-1999.

- No special processing is done for the nonyear part of dates. That is why the nonyear part of the supported date formats is denoted by `X`s. To do otherwise might change the meaning of existing programs. The only date-sensitive semantics that are provided involve automatically expanding (and contracting) the two-digit year part of dates with respect to the century window for the program.
- Dates with four-digit year parts are generally of interest only when used in combination with windowed dates. Otherwise there is little difference between four-digit year dates and nondates.

Based on these principles, the millennium language extensions are designed to meet several objectives. You should evaluate the objectives that you need to meet in order to resolve your date-processing problems, and compare them with the objectives of the millennium language extensions, to determine how your application can benefit from them. You should not consider using the extensions in new applications or in enhancements to existing applications, unless the applications are using old data that cannot be expanded until later.

The objectives of the millennium language extensions are as follows:
- Extend the useful life of your application programs as they are currently specified.
- Keep source changes to a minimum, preferably limited to augmenting the declarations of date fields in the `DATA DIVISION`. To implement the century window solution, you should not need to change the program logic in the `PROCEDURE DIVISION`.
- Preserve the existing semantics of the programs when adding date fields. For example, when a date is expressed as a literal, as in the following statement, the literal is considered to be compatible (windowed or expanded) with the date field to which it is compared:

  ```
  If Expiry-Date Greater Than 980101 . . .
  ```

  Because the existing program assumes that two-digit-year dates expressed as literals are in the range 1900-1999, the extensions do not change this assumption.
- The windowing feature is not intended for long-term use. It can extend the useful life of applications as a start toward a long-term solution that can be implemented later.
- The expanded date field feature is intended for long-term use, as an aid for expanding date fields in files and databases.

The extensions do not provide fully specified or complete date-oriented data types, with semantics that recognize, for example, the month and day parts of Gregorian dates. They do, however, provide special semantics for the year part of dates.

Resolving date-related logic problems

You can adopt any of three approaches to assist with date-processing problems: use a century window, internal bridging, or full field expansion.

Century window
: You define a century window and specify the fields that contain windowed dates. The compiler then interprets the two-digit years in these data fields according to the century window.

Internal bridging
: If your files and databases have not yet been converted to four-digit-year

dates, but you prefer to use four-digit expanded-year logic in your programs, you can use an internal bridging technique to process the dates as four-digit-year dates.

Full field expansion
This solution involves explicitly expanding two-digit-year date fields to contain full four-digit years in your files and databases and then using these fields in expanded form in your programs. This is the only method that assures reliable date processing for all applications.

You can use the millennium language extensions with each approach to achieve a solution, but each has advantages and disadvantages, as shown below.

Table 68. **Advantages and disadvantages of Year 2000 solutions**

Aspect	Century window	Internal bridging	Full field expansion
Implementation	Fast and easy but might not suit all applications	Some risk of corrupting data	Must ensure that changes to databases, copybooks, and programs are synchronized
Testing	Less testing is required because no changes to program logic	Testing is easy because changes to program logic are straightforward	
Duration of fix	Programs can function beyond 2000, but not a long-term solution	Programs can function beyond 2000, but not a permanent solution	Permanent solution
Performance	Might degrade performance	Good performance	Best performance
Maintenance			Maintenance is easier.

"Example: century window" on page 523
"Example: internal bridging" on page 524
"Example: converting files to expanded date form" on page 525

RELATED TASKS
"Using a century window"
"Using internal bridging" on page 523
"Moving to full field expansion" on page 525

Using a century window

A *century window* is a 100-year interval, such as 1950-2049, within which any two-digit year is unique. For windowed date fields, you specify the century window start date by using the `YEARWINDOW` compiler option.

When the `DATEPROC` option is in effect, the compiler applies this window to two-digit date fields in the program. For example, with a century window of 1930-2029, COBOL interprets two-digit years as follows:

- Year values from 00 through 29 are interpreted as years 2000-2029.
- Year values from 30 through 99 are interpreted as years 1930-1999.

To implement this century window, you use the DATE FORMAT clause to identify the date fields in your program and use the YEARWINDOW compiler option to define the century window as either a fixed window or a sliding window:

- For a fixed window, specify a four-digit year between 1900 and 1999 as the YEARWINDOW option value. For example, YEARWINDOW(1950) defines a fixed window of 1950-2049.
- For a sliding window, specify a negative integer from -1 through -99 as the YEARWINDOW option value. For example, YEARWINDOW(-50) defines a sliding window that starts 50 years before the year in which the program is running. So if the program is running in 2008, then the century window is 1958-2057; in 2009 it automatically becomes 1959-2058, and so on.

The compiler automatically applies the century window to operations on the date fields that you have identified. You do not need any extra program logic to implement the windowing.

"Example: century window"

RELATED REFERENCES
"DATEPROC" on page 256
"YEARWINDOW" on page 292
DATE FORMAT clause (*COBOL for AIX Language Reference*)
Restrictions on using date fields (*COBOL for AIX Language Reference*)

Example: century window

The following example shows (in bold) how to modify a program with the DATE FORMAT clause to use the automatic date windowing capability.

```
CBL LIB,QUOTE,NOOPT,DATEPROC(FLAG),YEARWINDOW(-60)
. . .
01  Loan-Record.
    05  Member-Number   Pic X(8).
    05  DVD-ID          Pic X(8).
    05  Date-Due-Back   Pic X(6) Date Format yyxxxx.
    05  Date-Returned   Pic X(6) Date Format yyxxxx.
. . .
    If Date-Returned > Date-Due-Back Then
       Perform Fine-Member.
```

There are no changes to the PROCEDURE DIVISION. The addition of the DATE FORMAT clause on the two date fields means that the compiler recognizes them as windowed date fields, and therefore applies the century window when processing the IF statement. For example, if Date-Due-Back contains 080102 (January 2, 2008) and Date-Returned contains 071231 (December 31, 2007), Date-Returned is less than (earlier than) Date-Due-Back, so the program does not perform the Fine-Member paragraph. (The program checks whether a DVD was returned on time.)

Using internal bridging

For internal bridging, you need to structure your program appropriately.

Do the following steps:

1. Read the input files with two-digit-year dates.
2. Declare these two-digit dates as windowed date fields and move them to expanded date fields, so that the compiler automatically expands them to four-digit-year dates.

3. In the main body of the program, use the four-digit-year dates for all date processing.
4. Window the dates back to two-digit years.
5. Write the two-digit-year dates to the output files.

This process provides a convenient migration path to a full expanded-date solution, and can have performance advantages over using windowed dates.

When you use this technique, your changes to the program logic are minimal. You simply add statements to expand and contract the dates, and change the statements that refer to dates to use the four-digit-year date fields in `WORKING-STORAGE` instead of the two-digit-year fields in the records.

Because you are converting the dates back to two-digit years for output, you should allow for the possibility that the year is outside the century window. For example, if a date field contains the year 2020, but the century window is 1920-2019, then the date is outside the window. Simply moving the year to a two-digit-year field will be incorrect. To protect against this problem, you can use a `COMPUTE` statement to store the date, with the `ON SIZE ERROR` phrase to detect whether the date is outside the century window.

"Example: internal bridging"

RELATED TASKS
"Using a century window" on page 522
"Performing arithmetic on date fields" on page 533
"Moving to full field expansion" on page 525

Example: internal bridging

The following example shows (in bold) how a program can be changed to implement internal bridging.

```
CBL  DATEPROC(FLAG),YEARWINDOW(-60)
    . . .
    File Section.
    FD  Customer-File.
    01  Cust-Record.
        05  Cust-Number     Pic 9(9) Binary.
        . . .
        05  Cust-Date       Pic 9(6) Date Format yyxxxx.
    Working-Storage Section.
    77  Exp-Cust-Date       Pic 9(8) Date Format yyyyxxxx.
    . . .
    Procedure Division.
        Open I-O Customer-File.
        Read Customer-File.
        Move Cust-Date to Exp-Cust-Date.
        . . .
   *=====================================================*
   * Use expanded date in the rest of the program logic  *
   *=====================================================*
        . . .
        Compute Cust-Date = Exp-Cust-Date
             On Size Error
                Display "Exp-Cust-Date outside century window"
        End-Compute
        Rewrite Cust-Record.
```

Moving to full field expansion

Using the millennium language extensions, you can move gradually toward a solution that fully expands the date field.

Do the following steps:

1. Apply the century window solution, and use this solution until you have the resources to implement a more permanent solution.
2. Apply the internal bridging solution. This way you can use expanded dates in your programs while your files continue to hold dates in two-digit-year form. You can progress more easily to a full-field-expansion solution because there will be no further changes to the logic in the main body of the programs.
3. Change the file layouts and database definitions to use four-digit-year dates.
4. Change your COBOL copybooks to reflect these four-digit-year date fields.
5. Run a utility program (or special-purpose COBOL program) to copy files from the old format to the new format.
6. Recompile your programs and do regression testing and date testing.

After you have completed the first two steps, you can repeat the remaining steps any number of times. You do not need to change every date field in every file at the same time. Using this method, you can select files for progressive conversion based on criteria such as business needs or interfaces with other applications.

When you use this method, you need to write special-purpose programs to convert your files to expanded-date form.

"Example: converting files to expanded date form"

Example: converting files to expanded date form

The following example shows a simple program that copies from one file to another while expanding the date fields. The record length of the output file is larger than that of the input file because the dates are expanded.

```
CBL  LIB,QUOTE,NOOPT,DATEPROC(FLAG),YEARWINDOW(-80)
      ************************************************
      ** CONVERT - Read a file, convert the date    **
      **           fields to expanded form, write   **
      **           the expanded records to a new    **
      **           file.                            **
      ************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  CONVERT.

       ENVIRONMENT DIVISION.

       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT INPUT-FILE
                  ASSIGN TO INFILE
                  FILE STATUS IS INPUT-FILE-STATUS.

           SELECT OUTPUT-FILE
                  ASSIGN TO OUTFILE
                  FILE STATUS IS OUTPUT-FILE-STATUS.

       DATA DIVISION.
       FILE SECTION.
       FD  INPUT-FILE
```

```
    RECORDING MODE IS F.
01  INPUT-RECORD.
    03  CUST-NAME.
        05  FIRST-NAME  PIC X(10).
        05  LAST-NAME   PIC X(15).
    03  ACCOUNT-NUM     PIC 9(8).
    03  DUE-DATE        PIC X(6) DATE FORMAT YYXXXX.    (1)
    03  REMINDER-DATE   PIC X(6) DATE FORMAT YYXXXX.
    03  DUE-AMOUNT      PIC S9(5)V99 COMP-3.

FD  OUTPUT-FILE
    RECORDING MODE IS F.
01  OUTPUT-RECORD.
    03  CUST-NAME.
        05  FIRST-NAME  PIC X(10).
        05  LAST-NAME   PIC X(15).
    03  ACCOUNT-NUM     PIC 9(8).
    03  DUE-DATE        PIC X(8) DATE FORMAT YYYYXXXX.  (2)
    03  REMINDER-DATE   PIC X(8) DATE FORMAT YYYYXXXX.
    03  DUE-AMOUNT      PIC S9(5)V99 COMP-3.

WORKING-STORAGE SECTION.

01  INPUT-FILE-STATUS   PIC 99.
01  OUTPUT-FILE-STATUS  PIC 99.

PROCEDURE DIVISION.

    OPEN INPUT INPUT-FILE.
    OPEN OUTPUT OUTPUT-FILE.

READ-RECORD.
    READ INPUT-FILE
         AT END GO TO CLOSE-FILES.
    MOVE CORRESPONDING INPUT-RECORD TO OUTPUT-RECORD.   (3)
    WRITE OUTPUT-RECORD.

    GO TO READ-RECORD.

CLOSE-FILES.
    CLOSE INPUT-FILE.
    CLOSE OUTPUT-FILE.

    EXIT PROGRAM.

END PROGRAM CONVERT.
```

Notes:

(1) The fields DUE-DATE and REMINDER-DATE in the input record are Gregorian dates with two-digit year components. They are defined with a DATE FORMAT clause so that the compiler recognizes them as windowed date fields.

(2) The output record contains the same two fields in expanded date format. They are defined with a DATE FORMAT clause so that the compiler treats them as four-digit-year date fields.

(3) The MOVE CORRESPONDING statement moves each item in INPUT-RECORD to its matching item in OUTPUT-RECORD. When the two windowed date fields are moved to the corresponding expanded date fields, the compiler expands the year values using the current century window.

Using year-first, year-only, and year-last date fields

When you compare two date fields of either year-first or year-only types, the two dates must be compatible; that is, they must have the same number of nonyear characters. The number of digits for the year component need not be the same.

A *year-first* date field is a date field whose DATE FORMAT specification consists of YY or YYYY, followed by one or more Xs. The date format of a *year-only* date field has just the YY or YYYY. A *year-last* date field is a date field whose DATE FORMAT clause specifies one or more Xs preceding YY or YYYY.

Year-last date formats are commonly used to display dates, but are less useful computationally because the year, which is the most significant part of the date, is in the least significant position of the date representation.

Functional support for year-last date fields is limited to equal or unequal comparisons and certain kinds of assignment. The operands must be either dates with identical (year-last) date formats, or a date and a nondate. The compiler does not provide automatic windowing for operations on year-last dates. When an unsupported usage (such as arithmetic on year-last dates) occurs, the compiler provides an error-level message.

If you need more general date-processing capability for year-last dates, you should isolate and operate on the year part of the date.

"Example: comparing year-first date fields" on page 528

RELATED CONCEPTS
"Compatible dates"

RELATED TASKS
"Using other date formats" on page 528

Compatible dates

The meaning of the term *compatible dates* depends on whether the usage occurs in the DATA DIVISION or the PROCEDURE DIVISION.

The DATA DIVISION usage deals with the declaration of date fields, and the rules that govern COBOL language elements such as subordinate data items and the REDEFINES clause. In the following example, Review-Date and Review-Year are compatible because Review-Year can be declared as a subordinate data item to Review-Date:

```
01  Review-Record.
    03  Review-Date              Date Format yyxxxx.
        05  Review-Year Pic XX    Date Format yy.
        05  Review-M-D  Pic XXXX.
```

The PROCEDURE DIVISION usage deals with how date fields can be used together in operations such as comparisons, moves, and arithmetic expressions. For year-first and year-only date fields to be considered compatible, date fields must have the same number of nonyear characters. For example, a field with DATE FORMAT YYXXXX is compatible with another field that has the same date format and with a YYYYXXXX field, but not with a YYXXX field.

Year-last date fields must have identical DATE FORMAT clauses. In particular, operations between windowed date fields and expanded year-last date fields are not allowed. For example, you can move a date field that has a date format of XXXXYY to another XXXXYY date field, but not to a date field that has a format of XXXXYYYY.

You can perform operations on date fields, or on a combination of date fields and nondates, provided that the date fields in the operation are compatible. For example, assume the following definitions:

```
01  Date-Gregorian-Win  Pic 9(6) Packed-Decimal Date Format yyxxxx.
01  Date-Julian-Win     Pic 9(5) Packed-Decimal Date Format yyxxx.
01  Date-Gregorian-Exp  Pic 9(8) Packed-Decimal Date Format yyyyxxxx.
```

The following statement is inconsistent because the number of nonyear digits is different between the two fields:

```
If Date-Gregorian-Win Less than Date-Julian-Win . . .
```

The following statement is accepted because the number of nonyear digits is the same for both fields:

```
If Date-Gregorian-Win Less than Date-Gregorian-Exp . . .
```

In this case the century window is applied to the windowed date field (Date-Gregorian-Win) to ensure that the comparison is meaningful.

When a nondate is used in conjunction with a date field, the nondate is either assumed to be compatible with the date field or is treated as a simple numeric value.

Example: comparing year-first date fields

The following example shows a windowed date field that is compared with an expanded date field.

```
77  Todays-Date         Pic X(8) Date Format yyyyxxxx.
01  Loan-Record.
    05  Date-Due-Back   Pic X(6) Date Format yyxxxx.
. . .
    If Date-Due-Back > Todays-Date Then . . .
```

The century window is applied to Date-Due-Back. Todays-Date must have a DATE FORMAT clause to define it as an expanded date field. If it did not, it would be treated as a nondate field and would therefore be considered to have the same number of year digits as Date-Due-Back. The compiler would apply the assumed century window of 1900-1999, which would create an inconsistent comparison.

Using other date formats

To be eligible for automatic windowing, a date field should contain a two-digit year as the first or only part of the field. The remainder of the field, if present, must contain between one and four characters, but its content is not important.

If there are date fields in your application that do not fit these criteria, you might have to make some code changes to define just the year part of the date as a date field with the DATE FORMAT clause. Some examples of these types of date formats are:

- A seven-character field that consists of a two-digit year, three characters that contain an abbreviation of the month, and two digits for the day of the month. This format is not supported because date fields can have only one through four nonyear characters.
- A Gregorian date of the form DDMMYY. Automatic windowing is not provided because the year component is not the first part of the date. Year-last dates such as these are fully supported as windowed keys in SORT or MERGE statements, and are also supported in a limited number of other COBOL operations.

If you need to use date windowing in cases like these, you will need to add some code to isolate the year portion of the date.

Example: isolating the year

The following example shows how you can isolate the year portion of a data field that is in the form DDMMYY.

```
03  Last-Review-Date Pic 9(6).
03  Next-Review-Date Pic 9(6).
. . .
Add 1 to Last-Review-Date Giving Next-Review-Date.
```

In the code above, if Last-Review-Date contains 230108 (January 23, 2008), then Next-Review-Date will contain 230109 (January 23, 2009) after the ADD statement is executed. This is a simple method for setting the next date for an annual review. However, if Last-Review-Date contains 230199, then adding 1 yields 230200, which is not the desired result.

Because the year is not the first part of these date fields, the DATE FORMAT clause cannot be applied without some code to isolate the year component. In the next example, the year component of both date fields has been isolated so that COBOL can apply the century window and maintain consistent results:

```
03  Last-Review-Date Date Format xxxxyy.
    05  Last-R-DDMM  Pic 9(4).
    05  Last-R-YY    Pic 99 Date Format yy.
03  Next-Review-Date Date Format xxxxyy.
    05  Next-R-DDMM  Pic 9(4).
    05  Next-R-YY    Pic 99 Date Format yy.
. . .
Move Last-R-DDMM to Next-R-DDMM.
Add 1 to Last-R-YY Giving Next-R-YY.
```

Manipulating literals as dates

If a windowed date field has a level-88 condition-name associated with it, the literal in the VALUE clause is windowed against the century window of the compile unit rather than against the assumed century window of 1900-1999.

For example, suppose you have these data definitions:

```
05  Date-Due        Pic 9(6)  Date Format yyxxxx.
    88  Date-Target           Value 081220.
```

If the century window is 1950-2049, and the contents of Date-Due are 081220 (representing December 20, 2008), then the first condition below evaluates to true, but the second condition evaluates to false:

```
If Date-Target. . .
If Date-Due = 081220
```

The literal 081220 is treated as a nondate; therefore it is windowed against the assumed century window of 1900-1999, and represents December 20, 1908. But where the literal is specified in the VALUE clause of a level-88 condition-name, the literal becomes part of the data item to which it is attached. Because this data item is a windowed date field, the century window is applied whenever it is referenced.

You can also use the DATEVAL intrinsic function in a comparison expression to convert a literal to a date field. The resulting date field will be treated as either a windowed date field or an expanded date field to ensure a consistent comparison. For example, using the above definitions, both of the following conditions evaluate to true:

```
If Date-Due = Function DATEVAL (081220 "YYXXXX")
If Date-Due = Function DATEVAL (20081220 "YYYYXXXX")
```

With a level-88 condition-name, you can specify the THRU option on the VALUE clause, but you must specify a fixed century window on the YEARWINDOW compiler option rather than a sliding window. For example:

```
05  Year-Field  Pic 99  Date Format yy.
    88 In-Range         Value 98 Thru 06.
```

With this form, the windowed value of the second item in the range must be greater than the windowed value of the first item. However, the compiler can verify this difference only if the YEARWINDOW compiler option specifies a fixed century window (for example, YEARWINDOW(1940) rather than YEARWINDOW(-68)).

The windowed order requirement does not apply to year-last date fields. If you specify a condition-name VALUE clause with the THROUGH phrase for a year-last date field, the two literals must follow normal COBOL rules. That is, the first literal must be less than the second literal.

RELATED CONCEPTS
"Assumed century window"
"Treatment of nondates" on page 531

RELATED TASKS
"Controlling date processing explicitly" on page 535

Assumed century window

When a program uses windowed date fields, the compiler applies the century window that is defined by the YEARWINDOW compiler option to the compilation unit. When a windowed date field is used in conjunction with a nondate, and the context demands that the nondate be treated as a windowed date, the compiler uses an assumed century window to resolve the nondate field.

The assumed century window is 1900-1999, which typically is not the same as the century window for the compilation unit.

In many cases, particularly for literal nondates, this assumed century window is the correct choice. In the following construct, the literal should retain its original meaning of January 1, 1972, and not change to 2072 if the century window is, for example, 1975-2074:

```
01  Manufacturing-Record.
    03  Makers-Date Pic X(6) Date Format yyxxxx.
. . .
    If Makers-Date Greater than "720101" . . .
```

Even if the assumption is correct, it is better to make the year explicit and eliminate the warning-level diagnostic message (which results from applying the assumed century window) by using the DATEVAL intrinsic function:

```
If Makers-Date Greater than
    Function Dateval("19720101" "YYYYXXXX") . . .
```

In some cases, the assumption might not be correct. For the following example, assume that Project-Controls is in a copy member that is used by other applications that have not yet been upgraded for year 2000 processing, and therefore Date-Target cannot have a DATE FORMAT clause:

```
01  Project-Controls.
    03  Date-Target     Pic 9(6).
. . .
01  Progress-Record.
    03  Date-Complete   Pic 9(6) Date Format yyxxxx.
. . .
    If Date-Complete Less than Date-Target . . .
```

In the example above, the following three conditions need to be true to make Date-Complete earlier than (less than) Date-Target:

- The century window is 1910-2009.
- Date-Complete is 991202 (Gregorian date: December 2, 1999).
- Date-Target is 000115 (Gregorian date: January 15, 2000).

However, because Date-Target does not have a DATE FORMAT clause, it is a nondate. Therefore, the century window applied to it is the assumed century window of 1900-1999, and it is processed as January 15, 1900. So Date-Complete will be greater than Date-Target, which is not the desired result.

In this case, you should use the DATEVAL intrinsic function to convert Date-Target to a date field for this comparison. For example:

```
If Date-Complete Less than
    Function Dateval (Date-Target "YYXXXX") . . .
```

RELATED TASKS
"Controlling date processing explicitly" on page 535

Treatment of nondates

How the compiler treats a nondate depends upon its context.

The following items are nondates:

- A literal value.
- A data item whose data description does not include a DATE FORMAT clause.
- The results (intermediate or final) of some arithmetic expressions. For example, the difference of two date fields is a nondate, whereas the sum of a date field and a nondate is a date field.
- The output from the UNDATE intrinsic function.

When you use a nondate in conjunction with a date field, the compiler interprets the nondate either as a date whose format is compatible with the date field or as a simple numeric value. This interpretation depends on the context in which the date field and nondate are used, as follows:

- Comparison

When a date field is compared with a nondate, the nondate is considered to be compatible with the date field in the number of year and nonyear characters. In the following example, the nondate literal 971231 is compared with a windowed date field:

```
01  Date-1    Pic 9(6) Date Format yyxxxx.
. . .
    If Date-1 Greater than 971231 . . .
```

The nondate literal 971231 is treated as if it had the same DATE FORMAT as Date-1, but with a base year of 1900.

- Arithmetic operations

  In all supported arithmetic operations, nondate fields are treated as simple numeric values. In the following example, the numeric value 10000 is added to the Gregorian date in Date-2, effectively adding one year to the date:

  ```
  01  Date-2    Pic 9(6) Date Format yyxxxx.
  . . .
      Add 10000 to Date-2.
  ```

- MOVE statement

  Moving a date field to a nondate is not supported. However, you can use the UNDATE intrinsic function to do this.

  When you move a nondate to a date field, the sending field is assumed to be compatible with the receiving field in the number of year and nonyear characters. For example, when you move a nondate to a windowed date field, the nondate field is assumed to contain a compatible date with a two-digit year.

Using sign conditions

Some applications use special values such as zeros in date fields to act as a trigger, that is, to signify that some special processing is required.

For example, in an Orders file, a value of zero in Order-Date might signify that the record is a customer totals record rather than an order record. The program compares the date to zero, as follows:

```
01  Order-Record.
    05  Order-Date      Pic S9(5) Comp-3 Date Format yyxxx.
. . .
    If Order-Date Equal Zero Then . . .
```

However, this comparison is not valid because the literal value Zero is a nondate, and is therefore windowed against the assumed century window to give a value of 1900000.

Alternatively, you can use a sign condition instead of a literal comparison as follows. With a sign condition, Order-Date is treated as a nondate, and the century window is not considered.

```
If Order-Date Is Zero Then . . .
```

This approach applies only if the operand in the sign condition is a simple identifier rather than an arithmetic expression. If an expression is specified, the expression is evaluated first, with the century window being applied where appropriate. The sign condition is then compared with the results of the expression.

You could use the UNDATE intrinsic function instead to achieve the same result.

RELATED CONCEPTS
"Treatment of nondates" on page 531

RELATED TASKS
"Controlling date processing explicitly" on page 535

RELATED REFERENCES
"DATEPROC" on page 256

Performing arithmetic on date fields

You can perform arithmetic operations on numeric date fields in the same manner as on any numeric data item. Where appropriate, the century window will be used in the calculation.

However, there are some restrictions on where date fields can be used in arithmetic expressions and statements. Arithmetic operations that include date fields are restricted to:

- Adding a nondate to a date field
- Subtracting a nondate from a date field
- Subtracting a date field from a compatible date field to give a nondate result

The following arithmetic operations are not allowed:

- Any operation between incompatible date fields
- Adding two date fields
- Subtracting a date field from a nondate
- Unary minus applied to a date field
- Multiplication, division, or exponentiation of or by a date field
- Arithmetic expressions that specify a year-last date field
- Arithmetic expressions that specify a year-last date field, except as a receiving data item when the sending field is a nondate

Date semantics are provided for the year parts of date fields but not for the nonyear parts. For example, adding 1 to a windowed Gregorian date field that contains the value 980831 gives a result of 980832, not 980901.

RELATED TASKS
"Allowing for overflow from windowed date fields"
"Specifying the order of evaluation" on page 534

Allowing for overflow from windowed date fields

A (nonyear-last) windowed date field that participates in an arithmetic operation is processed as if the value of the year component of the field were first incremented by 1900 or 2000, depending on the century window.

```
01 Review-Record.
    03 Last-Review-Year Pic 99 Date Format yy.
    03 Next-Review-Year Pic 99 Date Format yy.
. . .
    Add 10 to Last-Review-Year Giving Next-Review-Year.
```

In the example above, if the century window is 1910-2009, and the value of `Last-Review-Year` is 98, then the computation proceeds as if `Last-Review-Year` is first incremented by 1900 to give 1998. Then the ADD operation is performed, giving a result of 2008. This result is stored in Next-Review-Year as 08.

However, the following statement would give a result of 2018:

```
Add 20 to Last-Review-Year Giving Next-Review-Year.
```

This result falls outside the range of the century window. If the result is stored in Next-Review-Year, it will be incorrect because later references to Next-Review-Year will interpret it as 1918. In this case, the result of the operation depends on whether the ON SIZE ERROR phrase is specified on the ADD statement:

- If SIZE ERROR is specified, the receiving field is not changed, and the SIZE ERROR imperative statement is executed.
- If SIZE ERROR is not specified, the result is stored in the receiving field with the left-hand digits truncated.

This consideration is important when you use internal bridging. When you contract a four-digit-year date field back to two digits to write it to the output file, you need to ensure that the date falls within the century window. Then the two-digit-year date will be represented correctly in the field.

To ensure appropriate calculations, use a COMPUTE statement to do the contraction, with a SIZE ERROR phrase to handle the out-of-window condition. For example:

```
Compute Output-Date-YY = Work-Date-YYYY
On Size Error Perform CenturyWindowOverflow.
```

SIZE ERROR processing for windowed date receivers recognizes any year value that falls outside the century window. That is, a year value less than the starting year of the century window raises the SIZE ERROR condition, as does a year value greater than the ending year of the century window.

RELATED TASKS
"Using internal bridging" on page 523

Specifying the order of evaluation

Because of the restrictions on date fields in arithmetic expressions, you might find that programs that previously compiled successfully now produce diagnostic messages when some of the data items are changed to date fields.

```
01 Dates-Record.
    03 Start-Year-1 Pic 99 Date Format yy.
    03 End-Year-1   Pic 99 Date Format yy.
    03 Start-Year-2 Pic 99 Date Format yy.
    03 End-Year-2   Pic 99 Date Format yy.
. . .
    Compute End-Year-2 = Start-Year-2 + End-Year-1 - Start-Year-1.
```

In the example above, the first arithmetic expression evaluated is:

```
Start-Year-2 + End-Year-1
```

However, the addition of two date fields is not permitted. To resolve these date fields, you should use parentheses to isolate the parts of the arithmetic expression that are allowed. For example:

```
Compute End-Year-2 = Start-Year-2 + (End-Year-1 - Start-Year-1).
```

In this case, the first arithmetic expression evaluated is:

```
End-Year-1 - Start-Year-1
```

The subtraction of one date field from another is permitted and gives a nondate result. This nondate result is then added to the date field `End-Year-1`, giving a date field result that is stored in `End-Year-2`.

Controlling date processing explicitly

There might be times when you want COBOL data items to be treated as date fields only under certain conditions or only in specific parts of the program. Or your application might contain two-digit-year date fields that cannot be declared as windowed date fields because of some interaction with another software product.

For example, if a date field is used in a context where it is recognized only by its true binary contents without further interpretation, the date in that field cannot be windowed. Such date fields include:

- A key in a VSAM file
- A search field in a database system such as DB2
- A key field in a CICS command

Conversely, there might be times when you want a date field to be treated as a nondate in specific parts of the program.

COBOL provides two intrinsic functions to deal with these conditions:

DATEVAL	Converts a nondate to a date field
UNDATE	Converts a date field to a nondate

RELATED TASKS
"Using DATEVAL"
"Using UNDATE" on page 536

Using DATEVAL

You can use the `DATEVAL` intrinsic function to convert a nondate to a date field, so that COBOL will apply the relevant date processing to the field.

The first argument in the function is the nondate to be converted, and the second argument specifies the date format. The second argument is a literal string with a specification similar to that of the date pattern in the `DATE FORMAT` clause.

In most cases, the compiler makes the correct assumption about the interpretation of a nondate but accompanies this assumption with a warning-level diagnostic message. This message typically happens when a windowed date is compared with a literal:

```
03  When-Made        Pic x(6) Date Format yyxxxx.
. . .
If When-Made = "850701" Perform Warranty-Check.
```

The literal is assumed to be a compatible windowed date but with a century window of 1900-1999, thus representing July 15, 1985. You can use the DATEVAL intrinsic function to make the year of the literal date explicit and eliminate the warning message:

```
If When-Made = Function Dateval("19850701" "YYYYXXXX")
    Perform Warranty-Check.
```

"Example: DATEVAL"

Using UNDATE

You can use the UNDATE intrinsic function to convert a date field to a nondate so that it can be referenced without any date processing.

Attention: Avoid using UNDATE except as a last resort, because the compiler will lose the flow of date fields in your program. This problem could result in date comparisons not being windowed properly.

Use more DATE FORMAT clauses instead of function UNDATE for MOVE and COMPUTE.

"Example: UNDATE"

Example: DATEVAL

This example shows a case where it is better to leave a field as a nondate, and use the DATEVAL intrinsic function in a comparison statement.

Assume that a field Date-Copied is referenced many times in a program, but that most of the references just move the value between records or reformat it for printing. Only one reference relies on it to contain a date (for comparison with another date). In this case, it is better to leave the field as a nondate, and use the DATEVAL intrinsic function in the comparison statement. For example:

```
03  Date-Distributed Pic 9(6) Date Format yyxxxx.
03  Date-Copied      Pic 9(6).
. . .
If Function DATEVAL(Date-Copied "YYXXXX") Less than Date-Distributed . . .
```

In this example, DATEVAL converts Date-Copied to a date field so that the comparison will be meaningful.

RELATED REFERENCES
DATEVAL (*COBOL for AIX Language Reference*)

Example: UNDATE

The following example shows a case where you might want to convert a date field to a nondate.

The field Invoice-Date is a windowed Julian date. In some records, it contains the value 00999 to indicate that the record is not a true invoice record, but instead contains file-control information.

Invoice-Date has a DATE FORMAT clause because most of its references in the program are date-specific. However, when it is checked for the existence of a control record, the value 00 in the year component will lead to some confusion. A year value of 00 in Invoice-Date could represent either 1900 or 2000, depending on the century window. This is compared with a nondate (the literal 00999 in the example), which will always be windowed against the assumed century window and therefore always represents the year 1900.

To ensure a consistent comparison, you should use the UNDATE intrinsic function to convert Invoice-Date to a nondate. Therefore, if the IF statement is not comparing date fields, it does not need to apply windowing. For example:

```
01  Invoice-Record.
    03  Invoice-Date    Pic x(5) Date Format yyxxx.
. . .
    If FUNCTION UNDATE(Invoice-Date) Equal "00999" . . .
```

RELATED REFERENCES
UNDATE (*COBOL for AIX Language Reference*)

Analyzing and avoiding date-related diagnostic messages

When the DATEPROC(FLAG) compiler option is in effect, the compiler produces diagnostic messages for every statement that defines or references a date field.

As with all compiler-generated messages, each date-related message has one of the following severity levels:

- Information-level, to draw your attention to the definition or use of a date field.
- Warning-level, to indicate that the compiler has had to make an assumption about a date field or nondate because of inadequate information coded in the program, or to indicate the location of date logic that should be manually checked for correctness. Compilation proceeds, with any assumptions continuing to be applied.
- Error-level, to indicate that the usage of the date field is incorrect. Compilation continues, but runtime results are unpredictable.
- Severe-level, to indicate that the usage of the date field is incorrect. The statement that caused this error is discarded from the compilation.

The easiest way to use the MLE messages is to compile with a FLAG option setting that embeds the messages in the source listing after the line to which the messages refer. You can choose to see all MLE messages or just certain severities.

To see all MLE messages, specify the FLAG(I,I) and DATEPROC(FLAG) compiler options. Initially, you might want to see all of the messages to understand how MLE is processing the date fields in your program. For example, if you want to do a static analysis of the date usage in a program by using the compile listing, use FLAG (I,I).

However, it is recommended that you specify FLAG(W,W) for MLE-specific compiles. You must resolve all severe-level (S-level) error messages, and you should resolve all error-level (E-level) messages as well. For the warning-level (W-level) messages, you need to examine each message and use the following guidelines to either eliminate the message or, for unavoidable messages, ensure that the compiler makes correct assumptions:

- The diagnostic messages might indicate some date data items that should have had a DATE FORMAT clause. Either add DATE FORMAT clauses to these items or use the DATEVAL intrinsic function in references to them.

- Pay particular attention to literals in relation conditions that involve date fields or in arithmetic expressions that include date fields. You can use the DATEVAL function on literals (as well as nondate data items) to specify a DATE FORMAT pattern to be used. As a last resort, you can use the UNDATE function to enable a date field to be used in a context where you do not want date-oriented behavior.
- With the REDEFINES and RENAMES clauses, the compiler might produce a warning-level diagnostic message if a date field and a nondate occupy the same storage location. You should check these cases carefully to confirm that all uses of the aliased data items are correct, and that none of the perceived nondate redefinitions actually is a date or can adversely affect the date logic in the program.

In some cases, a the W-level message might be acceptable, but you might want to change the code to get a compile with a return code of zero.

To avoid warning-level diagnostic messages, follow these guidelines:

- Add DATE FORMAT clauses to any data items that will contain dates, even if the items are not used in comparisons. But see the related references below about restrictions on using date fields. For example, you cannot use the DATE FORMAT clause on a data item that is described implicitly or explicitly as USAGE NATIONAL.
- Do not specify a date field in a context where a date field does not make sense, such as a FILE STATUS, PASSWORD, ASSIGN USING, LABEL RECORD, or LINAGE item. If you do, you will get a warning-level message and the date field will be treated as a nondate.
- Ensure that implicit or explicit aliases for date fields are compatible, such as in a group item that consists solely of a date field.
- Ensure that if a date field is defined with a VALUE clause, the value is compatible with the date field definition.
- Use the DATEVAL intrinsic function if you want a nondate treated as a date field, such as when moving a nondate to a date field or when comparing a windowed date with a nondate and you want a windowed date comparison. If you do not use DATEVAL, the compiler will make an assumption about the use of the nondate and produce a warning-level diagnostic message. Even if the assumption is correct, you might want to use DATEVAL to eliminate the message.
- Use the UNDATE intrinsic function if you want a date field treated as a nondate, such as moving a date field to a nondate, or comparing a nondate and a windowed date field when you do not want a windowed comparison.

RELATED TASKS

"Controlling date processing explicitly" on page 535
COBOL Millennium Language Extensions Guide (Analyzing date-related diagnostic messages)

RELATED REFERENCES

Restrictions on using date fields (*COBOL for AIX Language Reference*)

Avoiding problems in processing dates

When you change a COBOL program to use the millennium language extensions, you might find that some parts of the program need special attention to resolve unforeseen changes in behavior. For example, you might need to avoid problems with packed-decimal fields and problems that occur if you move from expanded to windowed date fields.

RELATED TASKS
"Avoiding problems with packed-decimal fields"
"Moving from expanded to windowed date fields"

Avoiding problems with packed-decimal fields

COMPUTATIONAL-3 fields (packed-decimal format) are often defined as having an odd number of digits even if the field will not be used to hold a number of that magnitude. The internal representation of packed-decimal numbers always allows for an odd number of digits.

A field that holds a six-digit Gregorian date, for example, can be declared as PIC S9(6) COMP-3. This declaration will reserve 4 bytes of storage. But a programmer might have declared the field as PIC S9(7), knowing that this would reserve 4 bytes with the high-order digit always containing a zero.

If you add a DATE FORMAT YYXXXX clause to this field, the compiler will issue a diagnostic message because the number of digits in the PICTURE clause does not match the size of the date format specification. In this case, you need to carefully check each use of the field. If the high-order digit is never used, you can simply change the field definition to PIC S9(6). If it is used (for example, if the same field can hold a value other than a date), you need to take some other action, such as:

- Using a REDEFINES clause to define the field as both a date and a nondate (this usage will also produce a warning-level diagnostic message)
- Defining another WORKING-STORAGE field to hold the date, and moving the numeric field to the new field
- Not adding a DATE FORMAT clause to the data item, and using the DATEVAL intrinsic function when referring to it as a date field

Moving from expanded to windowed date fields

When you move an expanded alphanumeric date field to a windowed date field, the move does not follow the normal COBOL conventions for alphanumeric moves. When both the sending and receiving fields are date fields, the move is right justified, not left justified as normal. For an expanded-to-windowed (contracting) move, the leading two digits of the year are truncated.

Depending on the contents of the sending field, the results of such a move might be incorrect. For example:

```
77  Year-Of-Birth-Exp  Pic x(4) Date Format yyyy.
77  Year-Of-Birth-Win  Pic xx   Date Format yy.
. . .
    Move Year-Of-Birth-Exp to Year-Of-Birth-Win.
```

If Year-Of-Birth-Exp contains '1925', Year-Of-Birth-Win will contain '25'. However, if the century window is 1930-2029, subsequent references to Year-Of-Birth-Win will treat it as 2025, which is incorrect.

Part 8. Improving performance and productivity

Chapter 32. Tuning your program 543
Using an optimal programming style 544
  Using structured programming 544
  Factoring expressions. 544
  Using symbolic constants 545
  Grouping constant computations 545
  Grouping duplicate computations 545
Choosing efficient data types 546
  Choosing efficient computational data items . . 546
  Using consistent data types. 546
  Making arithmetic expressions efficient. . . . 547
  Making exponentiations efficient 547
Handling tables efficiently 547
  Optimization of table references 549
    Optimization of constant and variable items 549
    Optimization of duplicate items 549
    Optimization of variable-length items . . . 550
    Comparison of direct and relative indexing 550
Optimizing your code 550
  Optimization 551
    Contained program procedure integration 551
Choosing compiler features to enhance performance. 552
  Performance-related compiler options 552
  Evaluating performance 554

Chapter 33. Simplifying coding. 557
Eliminating repetitive coding 557
  Example: using the COPY statement. 558
Manipulating dates and times 559
  Getting feedback from date and time callable services 559
  Handling conditions from date and time callable services 560
  Example: manipulating dates 560
  Example: formatting dates for output 560
  Feedback token. 561
  Picture character terms and strings 562
  Example: date-and-time picture strings 564
  Century window 565
    Example: querying and changing the century window 565

Chapter 32. Tuning your program

When a program is comprehensible, you can assess its performance. A program that has a tangled control flow is difficult to understand and maintain. The tangled control flow also inhibits the optimization of the code.

Therefore, before you try to improve the performance directly, you need to assess certain aspects of your program:

1. Examine the underlying algorithms for your program. For top performance, a sound algorithm is essential. For example, a sophisticated algorithm for sorting a million items can be hundreds of thousands times faster than a simple algorithm.
2. Look at the data structures. They should be appropriate for the algorithm. When your program frequently accesses data, reduce the number of steps needed to access the data wherever possible.
3. After you have improved the algorithm and data structures, look at other details of the COBOL source code that affect performance.

You can write programs that result in better generated code sequences and use system services better. These areas affect program performance:

- Coding techniques. These include using a programming style that helps the optimizer, choosing efficient data types, and handling tables efficiently.
- Optimization. You can optimize your code by using the `OPTIMIZE` compiler option.
- Compiler options and `USE FOR DEBUGGING ON ALL PROCEDURES`. Certain compiler options and language affect the efficiency of your program.
- Runtime environment. Carefully consider your choice of runtime options and other runtime considerations that control how your compiled program runs.
- Running under CICS. Convert instances of `EXEC CICS LINK` to `CALL` to improve transaction response time.

RELATED CONCEPTS
"Optimization" on page 551

RELATED TASKS
"Using an optimal programming style" on page 544
"Choosing efficient data types" on page 546
"Handling tables efficiently" on page 547
"Optimizing your code" on page 550
"Choosing compiler features to enhance performance" on page 552
"Improving SFS performance" on page 139

RELATED REFERENCES
"Performance-related compiler options" on page 552
Chapter 16, "Runtime options," on page 301

Using an optimal programming style

The coding style you use can affect how the optimizer handles your code. You can improve optimization by using structured programming techniques, factoring expressions, using symbolic constants, and grouping constant and duplicate computations.

RELATED TASKS
"Using structured programming"
"Factoring expressions"
"Using symbolic constants" on page 545
"Grouping constant computations" on page 545
"Grouping duplicate computations" on page 545

Using structured programming

Using structured programming statements, such as `EVALUATE` and inline `PERFORM`, makes your program more comprehensible and generates a more linear control flow. As a result, the optimizer can operate over larger regions of the program, which gives you more efficient code.

Use top-down programming constructs. Out-of-line `PERFORM` statements are a natural means of doing top-down programming. Out-of-line `PERFORM` statements can often be as efficient as inline `PERFORM` statements, because the optimizer can simplify or remove the linkage code.

Avoid using the following constructs:
- `ALTER` statement
- Backward branches (except as needed for loops for which `PERFORM` is unsuitable)
- `PERFORM` procedures that involve irregular control flow (such as preventing control from passing to the end of the procedure and returning to the `PERFORM` statement)

Factoring expressions

By factoring expressions in your programs, you can potentially eliminate a lot of unnecessary computation.

For example, the first block of code below is more efficient than the second block of code:

```
MOVE ZERO TO TOTAL
PERFORM VARYING I FROM 1 BY 1 UNTIL I = 10
  COMPUTE TOTAL = TOTAL + ITEM(I)
END-PERFORM
COMPUTE TOTAL = TOTAL * DISCOUNT
```

```
MOVE ZERO TO TOTAL
PERFORM VARYING I FROM 1 BY 1 UNTIL I = 10
  COMPUTE TOTAL = TOTAL + ITEM(I) * DISCOUNT
END-PERFORM
```

The optimizer does not factor expressions.

Using symbolic constants

To have the optimizer recognize a data item as a constant throughout the program, initialize it with a VALUE clause and do not change it anywhere in the program.

If you pass a data item to a subprogram BY REFERENCE, the optimizer treats it as an external data item and assumes that it is changed at every subprogram call.

If you move a literal to a data item, the optimizer recognizes the data item as a constant only in a limited area of the program after the MOVE statement.

Grouping constant computations

When several items in an expression are constant, ensure that the optimizer is able to optimize them. The compiler is bound by the left-to-right evaluation rules of COBOL. Therefore, either move all the constants to the left side of the expression or group them inside parentheses.

For example, if V1, V2, and V3 are variables and C1, C2, and C3 are constants, the expressions on the left below are preferable to the corresponding expressions on the right:

More efficient	Less efficient
`V1 * V2 * V3 * (C1 * C2 * C3)`	`V1 * V2 * V3 * C1 * C2 * C3`
`C1 + C2 + C3 + V1 + V2 + V3`	`V1 + C1 + V2 + C2 + V3 + C3`

In production programming, there is often a tendency to place constant factors on the right-hand side of expressions. However, such placement can result in less efficient code because optimization is lost.

Grouping duplicate computations

When components of different expressions are duplicates, ensure that the compiler is able to optimize them. For arithmetic expressions, the compiler is bound by the left-to-right evaluation rules of COBOL. Therefore, either move all the duplicates to the left side of the expressions or group them inside parentheses.

If V1 through V5 are variables, the computation V2 * V3 * V4 is a duplicate (known as a common subexpression) in the following two statements:

```
COMPUTE A = V1 * (V2 * V3 * V4)
COMPUTE B = V2 * V3 * V4 * V5
```

In the following example, V2 + V3 is a common subexpression:

```
COMPUTE C = V1 + (V2 + V3)
COMPUTE D = V2 + V3 + V4
```

In the following example, there is no common subexpression:

```
COMPUTE A = V1 * V2 * V3 * V4
COMPUTE B = V2 * V3 * V4 * V5
COMPUTE C = V1 + (V2 + V3)
COMPUTE D = V4 + V2 + V3
```

The optimizer can eliminate duplicate computations. You do not need to introduce artificial temporary computations; a program is often more comprehensible without them.

Choosing efficient data types

Choosing the appropriate data type and PICTURE clause can produce more efficient code, as can avoiding USAGE DISPLAY and USAGE NATIONAL data items in areas that are heavily used for computations.

Consistent data types can reduce the need for conversions during operations on data items. You can also improve program performance by carefully determining when to use fixed-point and floating-point data types.

RELATED CONCEPTS
"Formats for numeric data" on page 43

RELATED TASKS
"Choosing efficient computational data items"
"Using consistent data types"
"Making arithmetic expressions efficient" on page 547
"Making exponentiations efficient" on page 547

Choosing efficient computational data items

When you use a data item mainly for arithmetic or as a subscript, code USAGE BINARY on the data description entry for the item. The operations for manipulating binary data are faster than those for manipulating decimal data.

However, if a fixed-point arithmetic statement has intermediate results with a large precision (number of significant digits), the compiler uses decimal arithmetic anyway, after converting the operands to packed-decimal form. For fixed-point arithmetic statements, the compiler normally uses binary arithmetic for simple computations with binary operands if the precision is eight or fewer digits. Above 18 digits, the compiler always uses decimal arithmetic. With a precision of nine to 18 digits, the compiler uses either form.

To produce the most efficient code for a BINARY data item, ensure that it has:

- A sign (an S in its PICTURE clause)
- Eight or fewer digits

For a data item that is larger than eight digits or is used with DISPLAY or NATIONAL data items, use PACKED-DECIMAL.

To produce the most efficient code for a PACKED-DECIMAL data item, ensure that it has:

- A sign (an S in its PICTURE clause)
- An odd number of digits (9s in the PICTURE clause), so that it occupies an exact number of bytes without a half byte left over

Using consistent data types

In operations on operands of different types, one of the operands must be converted to the same type as the other. Each conversion requires several instructions. For example, one of the operands might need to be scaled to give it the appropriate number of decimal places.

You can largely avoid conversions by using consistent data types and by giving both operands the same usage and also appropriate PICTURE specifications. That is, you should ensure that two numbers to be compared, added, or subtracted not only have the same usage but also the same number of decimal places (9s after the V in the PICTURE clause).

Making arithmetic expressions efficient

Computation of arithmetic expressions that are evaluated in floating point is most efficient when the operands need little or no conversion. Use operands that are COMP-1 or COMP-2 to produce the most efficient code.

Declare integer items as BINARY or PACKED-DECIMAL with nine or fewer digits to afford quick conversion to floating-point data. Also, conversion from a COMP-1 or COMP-2 item to a fixed-point integer with nine or fewer digits, without SIZE ERROR in effect, is efficient when the value of the COMP-1 or COMP-2 item is less than 1,000,000,000.

Making exponentiations efficient

Use floating point for exponentiations for large exponents to achieve faster evaluation and more accurate results.

For example, the first statement below is computed more quickly and accurately than the second statement:

```
COMPUTE fixed-point1 = fixed-point2 ** 100000.E+00

COMPUTE fixed-point1 = fixed-point2 ** 100000
```

A floating-point exponent causes floating-point arithmetic to be used to compute the exponentiation.

Handling tables efficiently

You can use several techniques to improve the efficiency of table-handling operations, and to influence the optimizer. The return for your efforts can be significant, particularly when table-handling operations are a major part of an application.

The following two guidelines affect your choice of how to refer to table elements:

- Use indexing rather than subscripting.

  Although the compiler can eliminate duplicate indexes and subscripts, the original reference to a table element is more efficient with indexes (even if the subscripts were BINARY). The value of an index has the element size factored into it, whereas the value of a subscript must be multiplied by the element size when the subscript is used. The index already contains the displacement from the start of the table, and this value does not have to be calculated at run time. However, subscripting might be easier to understand and maintain.
- Use relative indexing.

  Relative index references (that is, references in which an unsigned numeric literal is added to or subtracted from the index-name) are executed at least as fast as direct index references, and sometimes faster. There is no merit in keeping alternative indexes with the offset factored in.

Whether you use indexes or subscripts, the following coding guidelines can help you get better performance:

- Put constant and duplicate indexes or subscripts on the left.

  You can reduce or eliminate runtime computations this way. Even when all the indexes or subscripts are variable, try to use your tables so that the rightmost subscript varies most often for references that occur close to each other in the program. This practice also improves the pattern of storage references and also paging. If all the indexes or subscripts are duplicates, then the entire index or subscript computation is a common subexpression.
- Specify the element length so that it matches that of related tables.

  When you index or subscript tables, it is most efficient if all the tables have the same element length. That way, the stride for the last dimension of the tables is the same, and the optimizer can reuse the rightmost index or subscript computed for one table. If both the element lengths and the number of occurrences in each dimension are equal, then the strides for dimensions other than the last are also equal, resulting in greater commonality between their subscript computations. The optimizer can then reuse indexes or subscripts other than the rightmost.
- Avoid errors in references by coding index and subscript checks into your program.

  If you need to validate indexes and subscripts, it might be faster to code your own checks than to use the `SSRANGE` compiler option.

You can also improve the efficiency of tables by using these guidelines:

- Use binary data items for all subscripts.

  When you use subscripts to address a table, use a `BINARY` signed data item with eight or fewer digits. In some cases, using four or fewer digits for the data item might also improve processing time.
- Use binary data items for variable-length table items.

  For tables with variable-length items, you can improve the code for `OCCURS DEPENDING ON` (ODO). To avoid unnecessary conversions each time the variable-length items are referenced, specify `BINARY` for `OCCURS . . . DEPENDING ON` objects.
- Use fixed-length data items whenever possible.

  Copying variable-length data items into a fixed-length data item before a period of high-frequency use can reduce some of the overhead associated with using variable-length data items.
- Organize tables according to the type of search method used.

  If the table is searched sequentially, put the data values most likely to satisfy the search criteria at the beginning of the table. If the table is searched using a binary search algorithm, put the data values in the table sorted alphabetically on the search key field.

RELATED CONCEPTS
"Optimization of table references" on page 549

RELATED TASKS
"Referring to an item in a table" on page 64
"Choosing efficient data types" on page 546

RELATED REFERENCES
"SSRANGE" on page 285

Optimization of table references

The COBOL compiler optimizes table references in several ways.

For the table element reference `ELEMENT(S1 S2 S3)`, where `S1`, `S2`, and `S3` are subscripts, the compiler evaluates the following expression:

```
comp_s1 * d1 + comp_s2 * d2 + comp_s3 * d3 + base_address
```

Here `comp_s1` is the value of `S1` after conversion to binary, `comp-s2` is the value of `S2` after conversion to binary, and so on. The strides for each dimension are `d1`, `d2`, and `d3`. The *stride* of a given dimension is the distance in bytes between table elements whose occurrence numbers in that dimension differ by 1 and whose other occurrence numbers are equal. For example, the stride `d2` of the second dimension in the above example is the distance in bytes between `ELEMENT(S1 1 S3)` and `ELEMENT(S1 2 S3)`.

Index computations are similar to subscript computations, except that no multiplication needs to be done. Index values have the stride factored into them. They involve loading the indexes into registers, and these data transfers can be optimized, much as the individual subscript computation terms are optimized.

Because the compiler evaluates expressions from left to right, the optimizer finds the most opportunities to eliminate computations when the constant or duplicate subscripts are the leftmost.

Optimization of constant and variable items

Assume that `C1`, `C2`, . . . are constant data items and that `V1`, `V2`, . . . are variable data items. Then, for the table element reference `ELEMENT(V1 C1 C2)` the compiler can eliminate only the individual terms `comp_c1 * d2` and `comp_c2 * d3` as constant from the expression:

```
comp_v1 * d1 + comp_c1 * d2 + comp_c2 * d3 + base_address
```

However, for the table element reference `ELEMENT(C1 C2 V1)` the compiler can eliminate the entire subexpression `comp_c1 * d1 + comp_c2 * d2` as constant from the expression:

```
comp_c1 * d1 + comp_c2 * d2 + comp_v1 * d3 + base_address
```

In the table element reference `ELEMENT(C1 C2 C3)`, all the subscripts are constant, and so no subscript computation is done at run time. The expression is:

```
comp_c1 * d1 + comp_c2 * d2 + comp_c3 * d3 + base_address
```

With the optimizer, this reference can be as efficient as a reference to a scalar (nontable) item.

Optimization of duplicate items

In the table element references `ELEMENT(V1 V3 V4)` and `ELEMENT(V2 V3 V4)` only the individual terms `comp_v3 * d2` and `comp_v4 * d3` are common subexpressions in the expressions needed to reference the table elements:

```
comp_v1 * d1 + comp_v3 * d2 + comp_v4 * d3 + base_address
comp_v2 * d1 + comp_v3 * d2 + comp_v4 * d3 + base_address
```

However, for the two table element references `ELEMENT(V1 V2 V3)` and `ELEMENT(V1 V2 V4)` the entire subexpression `comp_v1 * d1 + comp_v2 * d2` is common between the two expressions needed to reference the table elements:

```
comp_v1 * d1 + comp_v2 * d2 + comp_v3 * d3 + base_address
comp_v1 * d1 + comp_v2 * d2 + comp_v4 * d3 + base_address
```

In the two references `ELEMENT(V1 V2 V3)` and `ELEMENT(V1 V2 V3)`, the expressions are the same:

```
comp_v1 * d1 + comp_v2 * d2 + comp_v3 * d3 + base_address
comp_v1 * d1 + comp_v2 * d2 + comp_v3 * d3 + base_address
```

With the optimizer, the second (and any subsequent) reference to the same element can be as efficient as a reference to a scalar (nontable) item.

Optimization of variable-length items

A group item that contains a subordinate `OCCURS DEPENDING ON` data item has a variable length. The program must perform special code every time a variable-length data item is referenced.

Because this code is out-of-line, it might interrupt optimization. Furthermore, the code to manipulate variable-length data items is much less efficient than that for fixed-size data items and can significantly increase processing time. For instance, the code to compare or move a variable-length data item might involve calling a library routine and is much slower than the same code for fixed-length data items.

Comparison of direct and relative indexing

Relative index references are as fast as or faster than direct index references.

The direct indexing in `ELEMENT (I5, J3, K2)` requires this preprocessing:

```
SET I5 TO I
SET I5 UP BY 5
SET J3 TO J
SET J3 DOWN BY 3
SET K2 TO K
SET K2 UP BY 2
```

This processing makes the direct indexing less efficient than the relative indexing in `ELEMENT (I + 5, J - 3, K + 2)`.

RELATED CONCEPTS
"Optimization" on page 551

RELATED TASKS
"Handling tables efficiently" on page 547

Optimizing your code

When your program is ready for final testing, specify the `OPTIMIZE` compiler option so that the tested code and the production code are identical.

You might also want to use this compiler option during development if a program is used frequently without recompilation. However, the overhead for `OPTIMIZE` might outweigh its benefits if you recompile frequently, unless you are using the assembler language expansion (`LIST` compiler option) to fine-tune the program.

For unit-testing a program, you will probably find it easier to debug code that has not been optimized.

To see how the optimizer works on a program, compile it with and without the `OPTIMIZE` option and compare the generated code. (Use the `LIST` compiler option to request the assembler listing of the generated code.)

RELATED CONCEPTS
"Optimization"

RELATED REFERENCES
"LIST" on page 269
"OPTIMIZE" on page 275

Optimization

To improve the efficiency of the generated code, you can use the `OPTIMIZE` compiler option.

`OPTIMIZE` causes the COBOL optimizer to do the following optimizations:

- Eliminate unnecessary transfers of control and inefficient branches, including those generated by the compiler that are not evident from looking at the source program.
- Simplify the compiled code for a `CALL` statement to a contained (nested) program. Where possible, the optimizer places the statements inline, eliminating the need for linkage code. This optimization is known as *procedure integration*. If procedure integration cannot be done, the optimizer uses the simplest linkage possible (perhaps as few as two instructions) to get to and from the called program.
- Eliminate duplicate computations (such as subscript computations and repeated statements) that have no effect on the results of the program.
- Eliminate constant computations by performing them when the program is compiled.
- Eliminate constant conditional expressions.
- Aggregate moves of contiguous items (such as those that often occur with the use of `MOVE CORRESPONDING`) into a single move. Both the source and target must be contiguous for the moves to be aggregated.
- Discard unreferenced data items from the `DATA DIVISION`, and suppress generation of code to initialize these data items to their `VALUE` clauses. (The optimizer takes this action only when you use the `FULL` suboption.)

Contained program procedure integration

In contained program procedure integration, the contained program code replaces a `CALL` to a contained program. The resulting program runs faster without the overhead of `CALL` linkage and with more linear control flow.

Program size: If several `CALL` statements call contained programs and these programs replace each such statement, the containing program can become large. The optimizer limits this increase to no more than 50 percent, after which it no longer integrates the programs. The optimizer then chooses the next best optimization for the `CALL` statement. The linkage overhead can be as few as two instructions.

Unreachable code: As a result of this integration, one contained program might be repeated several times. As further optimization proceeds on each copy of the program, portions might be found to be unreachable, depending on the context into which the code was copied.

RELATED CONCEPTS
"Optimization of table references" on page 549

RELATED REFERENCES
"OPTIMIZE" on page 275

Choosing compiler features to enhance performance

Your choice of performance-related compiler options and your use of the `USE FOR DEBUGGING ON ALL PROCEDURES` statement can affect how well your program is optimized.

You might have a customized system that requires certain options for optimum performance. Do these steps:

1. To see what your system defaults are, get a short listing for any program and review the listed option settings.
2. Select performance-related options for compiling your programs.

   Important: Confer with your system programmer about how to tune COBOL programs. Doing so will ensure that the options you choose are appropriate for programs at your site.

Another compiler feature to consider is the `USE FOR DEBUGGING ON ALL PROCEDURES` statement. It can greatly affect the compiler optimizer. The `ON ALL PROCEDURES` option generates extra code at each transfer to a procedure name. Although very useful for debugging, it can make the program significantly larger and inhibit optimization substantially.

RELATED CONCEPTS
"Optimization" on page 551

RELATED TASKS
"Optimizing your code" on page 550
"Getting listings" on page 316

RELATED REFERENCES
"Performance-related compiler options"

Performance-related compiler options

In the table below you can see a description of the purpose of each option, its performance advantages and disadvantages, and usage notes where applicable.

Table 69. **Performance-related compiler options**

Compiler option	Purpose	Performance advantages	Performance disadvantages	Usage notes
`ARITH(EXTEND)` (see "ARITH" on page 248)	To increase the maximum number of digits allowed for decimal numbers	In general, none	`ARITH(EXTEND)` causes some degradation in performance for all decimal data types due to larger intermediate results.	The amount of degradation that you experience depends directly on the amount of decimal data that you use.

Table 69. **Performance-related compiler options** *(continued)*

Compiler option	Purpose	Performance advantages	Performance disadvantages	Usage notes
"DYNAM" on page 258	To have subprograms (called through the CALL statement) dynamically loaded at run time	Subprograms are easier to maintain, because the application does not have to be link-edited again if a subprogram is changed.	There is a slight performance penalty, because the call must go through a library routine.	To free virtual storage that is no longer needed, issue the CANCEL statement.
OPTIMIZE(STD) (see "OPTIMIZE" on page 275)	To optimize generated code for better performance	Generally results in more efficient runtime code	Longer compile time: OPTIMIZE requires more processing time for compiles than NOOPTIMIZE.	NOOPTIMIZE is generally used during program development when frequent compiles are needed; it also allows for symbolic debugging. For production runs, OPTIMIZE is recommended.
OPTIMIZE(FULL) (see "OPTIMIZE" on page 275)	To optimize generated code for better performance and also optimize the DATA DIVISION	Generally results in more efficient runtime code and less storage usage	Longer compile time: OPTIMIZE requires more processing time for compiles than NOOPTIMIZE.	OPT(FULL) deletes unused data items, which might be undesirable in the case of time stamps or data items that are used only as markers for dump reading.
NOSSRANGE (see "SSRANGE" on page 285)	To verify that all table references and reference modification expressions are in proper bounds	SSRANGE generates additional code for verifying table references. Using NOSSRANGE causes that code not to be generated.	None	In general, if you need to verify the table references only a few times instead of at every reference, coding your own checks might be faster than using SSRANGE. You can turn off SSRANGE at run time by using the CHECK(OFF) runtime option. For performance-sensitive applications, NOSSRANGE is recommended.

Table 69. **Performance-related compiler options** *(continued)*

Compiler option	Purpose	Performance advantages	Performance disadvantages	Usage notes
NOTEST (see "TEST" on page 286)	To avoid the additional object code that is needed to take full advantage of the Debugger for AIX.	TEST significantly enlarges the object file because it adds debugging information. When linking the program, you can direct the linker to exclude the debugging information, resulting in approximately the same size executable as would be created if the modules were compiled with NOTEST. If the debugging information is included, a slight performance degradation might occur because a larger executable takes longer to load and could increase paging.	None	TEST forces compiler option NOOPT into effect. For production runs, using NOTEST is recommended.
TRUNC(OPT) (see "TRUNC" on page 288)	To avoid having code generated to truncate the receiving fields of arithmetic operations	Does not generate extra code and generally improves performance	Both TRUNC(BIN) and TRUNC(STD) generate extra code whenever a BINARY data item is changed. TRUNC(BIN) is usually the slowest of these options, though its performance was improved in COBOL for OS/390 & VM V2R2.	TRUNC(STD) conforms to Standard COBOL 85, but TRUNC(BIN) and TRUNC(OPT) do not. With TRUNC(OPT), the compiler assumes that the data conforms to the PICTURE and USAGE specifications. TRUNC(OPT) is recommended where possible.

RELATED CONCEPTS
"Optimization" on page 551

RELATED TASKS
"Generating a list of compiler error messages" on page 225
"Choosing compiler features to enhance performance" on page 552
"Handling tables efficiently" on page 547

RELATED REFERENCES
"Sign representation of zoned and packed-decimal data" on page 50
Chapter 14, "Compiler options," on page 243

Evaluating performance

Fill in the following worksheet to help you evaluate the performance of your program. If you answer yes to each question, you are probably improving the performance.

In thinking about the performance tradeoff, be sure you understand the function of each option as well as the performance advantages and disadvantages. You might prefer function over increased performance in many instances.

Table 70. **Performance-tuning worksheet**

Compiler option	Consideration	Yes?
`DYNAM`	Can you use `NODYNAM`? Consider the performance tradeoffs.	
`OPTIMIZE`	Do you use `OPTIMIZE` for production runs? Can you use `OPTIMIZE(FULL)`?	
`SSRANGE`	Do you use `NOSSRANGE` for production runs?	
`TEST`	Do you use `NOTEST` for production runs?	
`TRUNC`	Do you use `TRUNC(OPT)` when possible?	

RELATED TASKS
"Choosing compiler features to enhance performance" on page 552

RELATED REFERENCES
"Performance-related compiler options" on page 552

Chapter 33. Simplifying coding

You can use coding techniques to improve your productivity. By using the COPY statement, COBOL intrinsic functions, and callable services, you can avoid repetitive coding and having to code many arithmetic calculations or other complex tasks.

If your program contains frequently used code sequences (such as blocks of common data items, input-output routines, error routines, or even entire COBOL programs), write the code sequences once and put them in a COBOL copy library. You can use the COPY statement to retrieve these code sequences and have them included in your program at compile time. Using copybooks in this manner eliminates repetitive coding.

COBOL provides various capabilities for manipulating strings and numbers. These capabilities can help you simplify your coding.

The date and time callable services store dates as fullword binary integers and store timestamps as long (64-bit) floating-point values. These formats let you do arithmetic calculations on date and time values simply and efficiently. You do not need to write special subroutines that use services outside the language library to perform such calculations.

RELATED TASKS
"Using numeric intrinsic functions" on page 53
"Eliminating repetitive coding"
"Converting data items (intrinsic functions)" on page 104
"Evaluating data items (intrinsic functions)" on page 107
"Manipulating dates and times" on page 559

Eliminating repetitive coding

Use the COPY statement in any program division and at any code sequence level to include stored source statements in a program. You can nest COPY statements to any depth.

To specify more than one copy library, either set the environment variable SYSLIB to multiple path names separated by a semicolon (;) or define your own environment variables and include the following phrase in the COPY statement:

```
IN/OF library-name
```

For example:

```
COPY MEMBER1 OF COPYLIB
```

If you omit this qualifying phrase, the default is SYSLIB.

Use a command such as the following example to set an environment variable that defines COPYLIB at compile time:

```
SET COPYLIB=D:\CPYFILES\COBCOPY
```

COPY and debugging line: In order for the text copied to be treated as debug lines, for example, as if there were a D inserted in column 7, put the D on the first line of the COPY statement. A COPY statement itself cannot be a debugging line; if it contains a D and WITH DEBUGGING mode is not specified, the COPY statement is nevertheless processed.

"Example: using the COPY statement"

RELATED REFERENCES
Chapter 15, "Compiler-directing statements," on page 295

Example: using the COPY statement

These examples show how you can use the COPY statement to include library text in a program.

Suppose the library entry CFILEA consists of the following FD entries:

```
    BLOCK CONTAINS 20 RECORDS
    RECORD CONTAINS 120 CHARACTERS
    LABEL RECORDS ARE STANDARD
    DATA RECORD IS FILE-OUT.
01  FILE-OUT       PIC X(120).
```

You can retrieve the text-name CFILEA by using the COPY statement in a source program as follows:

```
FD FILEA
         COPY CFILEA.
```

The library entry is copied into your program, and the resulting program listing looks like this:

```
FD FILEA
         COPY CFILEA.
C    BLOCK CONTAINS 20 RECORDS
C    RECORD CONTAINS 120 CHARACTERS
C    LABEL RECORDS ARE STANDARD
C    DATA RECORD IS FILE-OUT.
C    01  FILE-OUT   PIC X(120).
```

In the compiler source listing, the COPY statement prints on a separate line. C precedes copied lines.

Assume that a copybook with the text-name DOWORK is stored by using the following statements:

```
COMPUTE QTY-ON-HAND = TOTAL-USED-NUMBER-ON-HAND
MOVE QTY-ON-HAND to PRINT-AREA
```

To retrieve the copybook identified as DOWORK, code:

```
paragraph-name.
    COPY DOWORK.
```

The statements that are in the DOWORK procedure will follow *paragraph-name*.

If you use the EXIT compiler option to provide a LIBEXIT module, your results might differ from those shown here.

RELATED TASKS
"Eliminating repetitive coding" on page 557

RELATED REFERENCES
Chapter 15, "Compiler-directing statements," on page 295

Manipulating dates and times

To invoke a date or time callable service, use a CALL statement with the correct parameters for that service. You define the data items for the CALL statement in the DATA DIVISION with the data definitions required by that service.

```
77  argument         pic  s9(9)  comp.
01  format.
    05 format-length  pic  s9(4)  comp.
    05 format-string  pic  x(80).
77  result           pic  x(80).
77  feedback-code    pic  x(12)  display.
. . .
    CALL "CEEDATE" using argument, format, result, feedback-code.
```

In the example above, the callable service CEEDATE converts a number that represents a Lilian date in the data item argument to a date in character format, which is written to the data item result. The picture string contained in the data item format controls the format of the conversion. Information about the success or failure of the call is returned in the data item feedback-code.

In the CALL statements that you use to invoke the date and time callable services, you must use a literal for the program-name rather than an identifier.

A program calls the date and time callable services by using the standard system linkage convention.

"Example: manipulating dates" on page 560

RELATED CONCEPTS
Appendix F, "Date and time callable services," on page 599

RELATED TASKS
"Getting feedback from date and time callable services"
"Handling conditions from date and time callable services" on page 560

RELATED REFERENCES
"Feedback token" on page 561
"Picture character terms and strings" on page 562
CALL statement (*COBOL for AIX Language Reference*)

Getting feedback from date and time callable services

You can specify a feedback code parameter (which is optional) in any date and time callable service. Specify OMITTED for this parameter if you do not want the service to return information about the success or failure of the call.

However, if you do not specify this parameter and the callable service does not complete successfully, the program will abend.

When you call a date and time callable service and specify OMITTED for the feedback code, the RETURN-CODE special register is set to 0 if the service is successful, but it is not altered if the service is unsuccessful. If the feedback code is not OMITTED, the RETURN-CODE special register is always set to 0 regardless of whether the service completed successfully.

"Example: formatting dates for output" on page 560

RELATED REFERENCES
"Feedback token" on page 561

Handling conditions from date and time callable services

Condition handling by COBOL for AIX is significantly different from that provided by IBM Language Environment® on the host. COBOL for AIX adheres to the native COBOL condition handling scheme and does not provide the level of support that is in Language Environment.

If you pass a feedback token an argument, it will simply be returned after the appropriate information has been filled in. You can code logic in the calling routine to examine the contents and perform any actions if necessary. The condition will not be signaled.

RELATED REFERENCES
"Feedback token" on page 561

Example: manipulating dates

The following example shows how to use date and time callable services to convert a date to a different format and do a simple calculation with the formatted date.

```
CALL CEEDAYS USING dateof_hire, 'YYMMDD', doh_lilian, fc.
CALL CEELOCT USING todayLilian, today_seconds, today_Gregorian, fc.
COMPUTE servicedays  = today_Lilian - doh_Lilian.
COMPUTE serviceyears = service_days / 365.25.
```

The example above uses the original date of hire in the format YYMMDD to calculate the number of years of service for an employee. The calculation is as follows:

1. Call CEEDAYS (Convert Date to Lilian Format) to convert the date to Lilian format.
2. Call CEELOCT (Get Current Local Time) to get the current local time.
3. Subtract `doh_Lilian` from `today_Lilian` (the number of days from the beginning of the Gregorian calendar to the current local time) to calculate the employee's number of days of employment.
4. Divide the number of days by 365.25 to get the number of service years.

Example: formatting dates for output

The following example uses date and time callable services to format and display a date obtained from an `ACCEPT` statement.

Many callable services offer capabilities that would require extensive coding using previous versions of COBOL. Two such services are CEEDAYS and CEEDATE, which you can use effectively when you want to format dates.

```
CBL QUOTE
       ID DIVISION.
       PROGRAM-ID. HOHOHO.
      ************************************************************
      * FUNCTION:  DISPLAY TODAY'S DATE IN THE FOLLOWING FORMAT: *
      *            WWWWWWWWW, MMMMMMMM DD, YYYY                  *
      *                                                          *
      *            For example:  MONDAY, OCTOBER 20, 2008        *
*                                                          *
************************************************************
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.

 01   CHRDATE.
     05 CHRDATE-LENGTH     PIC S9(4) COMP VALUE 10.
     05 CHRDATE-STRING     PIC X(10).
 01   PICSTR.
     05 PICSTR-LENGTH      PIC  S9(4) COMP.
     05 PICSTR-STRING      PIC  X(80).

 77   LILIAN PIC           S9(9) COMP.
 77   FORMATTED-DATE       PIC X(80).

 PROCEDURE DIVISION.
***************************************************************
*    USE  DATE/TIME CALLABLE SERVICES TO PRINT OUT            *
*    TODAY'S DATE FROM COBOL ACCEPT STATEMENT.                *
***************************************************************
     ACCEPT CHRDATE-STRING FROM DATE.

     MOVE "YYMMDD" TO PICSTR-STRING.
     MOVE 6 TO PICSTR-LENGTH.
     CALL "CEEDAYS" USING CHRDATE , PICSTR , LILIAN , OMITTED.

     MOVE " WWWWWWWWWZ, MMMMMMMMMZ DD, YYYY " TO PICSTR-STRING.
     MOVE 50 TO PICSTR-LENGTH.
     CALL "CEEDATE" USING LILIAN , PICSTR , FORMATTED-DATE ,
           OMITTED.

     DISPLAY "******************************".
     DISPLAY FORMATTED-DATE.
     DISPLAY "******************************".

     STOP RUN.
```

Feedback token

A feedback token contains feedback information in the form of a condition token. The condition token set by the callable service is returned to the calling routine, indicating whether the service completed successfully.

COBOL for AIX uses the same feedback token as Language Environment, which is defined as follows:

```
01  FC.
    02  Condition-Token-Value.
    COPY  CEEIGZCT.
        03  Case-1-Condition-ID.
            04  Severity    PIC S9(4) COMP.
            04  Msg-No      PIC S9(4) COMP.
        03  Case-2-Condition-ID
                  REDEFINES Case-1-Condition-ID.
            04  Class-Code  PIC S9(4) COMP.
            04  Cause-Code  PIC S9(4) COMP.
        03  Case-Sev-Ctl    PIC X.
        03  Facility-ID     PIC XXX.
    02  I-S-Info            PIC S9(9) COMP.
```

The contents of each field and the differences from IBM Language Environment on the host are as follows:

`Severity`
: This is the severity number with the following possible values:

0	Information only (or, if the entire token is zero, no information)
1	Warning: service completed, probably correctly
2	Error detected: correction was attempted; service completed, perhaps incorrectly
3	Severe error: service did not complete
4	Critical error: service did not complete

`Msg-No` This is the associated message number.

`Case-Sev-Ctl`
: This field always contains the value 1.

`Facility-ID`
: This field always contains the characters CEE.

`I-S-Info`
: This field always contains the value 0.

The sample copybook CEEIGZCT.CPY defines the condition tokens. The condition tokens in the file are equivalent to those provided by Language Environment, except that character representations are in ASCII instead of EBCDIC. You must take these differences into account if you compare the condition tokens with those provided by Language Environment.

The descriptions of the individual callable services include a listing of the symbolic feedback codes that might be returned in the feedback code output field specified on invocation of the service. In addition to these, the symbolic feedback code CEE0PD might be returned for any callable service. See message IWZ0813S for details.

All date and time callable services are based on the Gregorian calendar. Date variables associated with this calendar have architectural limits. These limits are:

Starting Lilian date
: The beginning of the Lilian date range is Friday 15 October 1582, the date of adoption of the Gregorian calendar. Lilian dates before this date are undefined. Therefore:
- Day zero is 00:00:00 14 October 1582.
- Day one is 00:00:00 15 October 1582.

All valid input dates must be after 00:00:00 15 October 1582.

End Lilian date
: The end Lilian date is set to 31 December 9999. Lilian dates after this date are undefined because 9999 is the highest possible four-digit year.

RELATED REFERENCES
Appendix J, "Runtime messages," on page 725

Picture character terms and strings

You use picture strings (templates that indicate the format of the input data or the desired format of the output data) for several of the date and time callable services.

Table 71. **Picture character terms and strings**

Picture terms	Explanations	Valid values	Notes
Y YY YYY ZYY YYYY	One-digit year Two-digit year Three-digit year Three-digit year within era Four-digit year	0-9 00-99 000-999 1-999 1582-9999	Y valid for output only. YY assumes range set by CEESCEN. YYY/ZYY used with <JJJJ>, <CCCC>, and <CCCCCCCC>.
<JJJJ>	Japanese Era name in Kanji characters with UTF-16 hexadecimal encoding	Heisei (NX'5E736210') Showa (NX'662D548C') Taisho (NX'59276B63') Meiji (NX'660E6CBB')	Affects YY field: if <JJJJ> is specified, YY means the year within Japanese Era. For example, 1988 equals Showa 63.
MM ZM	Two-digit month One- or two-digit month	01-12 1-12	For output, leading zero suppressed. For input, ZM treated as MM.
RRRR RRRZ	Roman numeral month	I*bbb*-XII*b* (left justified)	For input, source string is folded to uppercase. For output, uppercase only. I=Jan, II=Feb, ..., XII=Dec.
MMM Mmm MMMM...M Mmmm...m MMMMMMMMMZ Mmmmmmmmmz	Three-char month, uppercase Three-char month, mixed case Three-20 char mo., uppercase Three-20 char mo., mixed case Trailing blanks suppressed Trailing blanks suppressed	JAN-DEC Jan-Dec JANUARY*bb*-DECEMBER*b* January*bb*-December*b* JANUARY-DECEMBER January-December	For input, source string always folded to uppercase. For output, M generates uppercase and m generates lowercase. Output is padded with blanks (*b*) (unless Z specified) or truncated to match the number of Ms, up to 20.
DD ZD DDD	Two-digit day of month One- or two-digit day of mo. Day of year (Julian day)	01-31 1-31 001-366	For output, leading zero is always suppressed. For input, ZD treated as DD.
HH ZH	Two-digit hour One- or two-digit hour	00-23 0-23	For output, leading zero suppressed. For input, ZH treated as HH. If AP specified, valid values are 01-12.
MI	Minute	00-59	
SS	Second	00-59	
9 99 999	Tenths of a second Hundredths of a second Thousandths of a second	0-9 00-99 000-999	No rounding
AP ap A.P. a.p.	AM/PM indicator	AM or PM am or pm A.M. or P.M. a.m. or p.m.	AP affects HH/ZH field. For input, source string always folded to uppercase. For output, AP generates uppercase and ap generates lowercase.
W WWW Www WWW...W Www...w WWWWWWWWWZ Wwwwwwwwwz	One-char day-of-week Three-char day, uppercase Three-char day, mixed case Three-20 char day, uppercase Three-20 char day, mixed case Trailing blanks suppressed Trailing blanks suppressed	S, M, T, W, T, F, S SUN-SAT Sun-Sat SUNDAY*bbb*-SATURDAY*b* Sunday*bbb*-Saturday*b* SUNDAY-SATURDAY Sunday-Saturday	For input, Ws are ignored. For output, W generates uppercase and w generates lowercase. Output padded with blanks (unless Z specified) or truncated to match the number of Ws, up to 20.
All others	Delimiters	X'01'-X'FF' (X'00' is reserved for "internal" use by the date and time callable services)	For input, treated as delimiters between the month, day, year, hour, minute, second, and fraction of a second. For output, copied exactly as is to the target string.
Note: Blank characters are indicated by the symbol *b*.			

The following table defines Japanese Eras used by date and time services when <JJJJ> is specified.

Table 72. **Japanese Eras**

First date of Japanese Era	Era name	Era name in Kanji with UTF-16 hexadecimal encoding	Valid year values
1868-09-08	Meiji	`NX'660E6CBB'`	01-45
1912-07-30	Taisho	`NX'59276B63'`	01-15
1926-12-25	Showa	`NX'662D548C'`	01-64
1989-01-08	Heisei	`NX'5E736210'`	01-999 (01 = 1989)

"Example: date-and-time picture strings"

Example: date-and-time picture strings

These are examples of picture strings that are recognized by the date and time services.

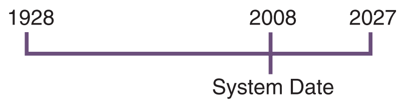
Table 73. **Examples of date-and-time picture strings**

Picture strings	Examples	Comments
`YYMMDD`	880516	
`YYYYMMDD`	19880516	
`YYYY-MM-DD`	1988-05-16	1988-5-16 would also be valid input.
`<JJJJ> YY.MM.DD`	Showa 63.05.16	*Showa* is a Japanese Era name. Showa 63 equals 1988.
`MMDDYY`	050688	
`MM/DD/YY`	05/06/88	
`ZM/ZD/YY`	5/6/88	
`MM/DD/YYYY`	05/06/1988	
`MM/DD/Y`	05/06/8	One-digit year format (`Y`) is valid for output only.
`DD.MM.YY`	09.06.88	
`DD-RRRR-YY`	09-VI -88	
`DD MMM YY`	09 JUN 88	
`DD Mmmmmmmmmm YY`	09 June 88	
`ZD Mmmmmmmmmz YY`	9 June 88	
`Mmmmmmmmmz ZD, YYYY`	June 9, 1988	
`ZDMMMMMMMMzYY`	9JUNE88	`Z` suppresses zeros and blanks.
`YY.DDD`	88.137	Julian date
`YYDDD`	88137	
`YYYY/DDD`	1988/137	
`YYMMDDHHMISS`	880516204229	Timestamp: valid only for CEESECS and CEEDATM. If used with CEEDATE, time positions are filled with zeros. If used with CEEDAYS, `HH`, `MI`, `SS`, and `999` fields are ignored.
`YYYYMMDDHHMISS`	19880516204229	
`YYYY-MM-DD HH:MI:SS.999`	1988-05-16 20:42:29.046	
`WWW, ZM/ZD/YY HH:MI AP`	MON, 5/16/88 08:42 PM	
`Wwwwwwwwwz, DD Mmm`	Monday, 16 May	
`YYYY, ZH:MI AP`	1988, 8:42 PM	
Note: Lowercase characters can be used only for alphabetic picture terms.		

Century window

To process two-digit years in the year 2000 and beyond, the date and time callable services use a sliding scheme in which all two-digit years are assumed to lie within a 100-year interval (the *century window*) that starts 80 years before the current system date.

In the year 2008 for example, the 100 years that span from 1928 to 2027 are the default century window for the date and time callable services. In 2008, years 28 through 99 are recognized as 1928-1999, and years 00 through 27 are recognized as 2000-2027.



By year 2080, all two-digit years will be recognized as 20*nn*. In 2081, 00 will be recognized as year 2100.

Some applications might need to set up a different 100-year interval. For example, banks often deal with 30-year bonds, which could be due 01/31/28. The two-digit year 28 would be interpreted as 1928 if the century window described above were in effect. The CEESCEN callable service lets you change the century window. A companion service, CEEQCEN, queries the current century window.

You can use CEEQCEN and CEESCEN, for example, to cause a subroutine to use a different interval for date processing than that used by its parent routine. Before returning, the subroutine should reset the interval to its previous value.

"Example: querying and changing the century window"

Example: querying and changing the century window

The following example shows how to query, set, and restore the starting point of the century window using the CEEQCEN and CEESCEN services.

The example calls CEEQCEN to obtain an integer (`OLDCEN`) that indicates how many years earlier the current century window began. It then temporarily changes the starting point of the current century window to a new value (`TEMPCEN`) by calling CEESCEN with that value. Because the century window is set to 30, any two-digit years that follow the CEESCEN call are assumed to lie within the 100-year interval starting 30 years before the current system date.

Finally, after it processes dates (not shown) using the temporary century window, the example again calls CEESCEN to reset the starting point of the century window to its original value.

```
 WORKING-STORAGE SECTION.
 77 OLDCEN  PIC S9(9) COMP.
 77 TEMPCEN PIC S9(9) COMP.
 77 QCENFC  PIC X(12).
 . . .
 77 SCENFC1 PIC X(12).
 77 SCENFC2 PIC X(12).
 . . .
 PROCEDURE DIVISION.
     . . .
** Call CEEQCEN to retrieve and save current century window
     CALL "CEEQCEN" USING OLDCEN, QCENFC.
```

```
** Call CEESCEN to temporarily change century window to 30
     MOVE 30 TO TEMPCEN.
     CALL "CEESCEN" USING TEMPCEN, SCENFC1.
** Perform date processing with two-digit years
     . . .
** Call CEESCEN again to reset century window
     CALL "CEESCEN" USING OLDCEN, SCENFC2.
     . . .
     GOBACK.
```

RELATED REFERENCES

Appendix F, “Date and time callable services,” on page 599

Part 9. Appendixes

Appendix A. Migration and compatibility

This information describes some areas you might need to consider if migrating COBOL for AIX Version 2 programs to COBOL for AIX Version 3.

RELATED REFERENCES
"Object code"
"Object-oriented syntax, and Java 5 or Java 6 SDKs"
"Data size limits"
"XML generation" on page 570
"SFS files" on page 570
"Migration of 32-bit applications to 64-bit mode" on page 570
Reserved words (*COBOL for AIX Language Reference*)

Object code

COBOL for AIX programs can call and be called by programs compiled with COBOL for AIX Version 2.

You can run programs compiled with COBOL for AIX Version 2 using the current COBOL for AIX runtime library.

Object-oriented syntax, and Java 5 or Java 6 SDKs

COBOL for AIX Version 2 applications that use object-oriented syntax for Java interoperability are supported with Java SDK 1.4.2.

To run these existing applications with Java 5 or Java 6, do these steps:

1. Recompile and relink the applications using the current version of COBOL for AIX.
2. Recompile the generated Java class that is associated with each object-oriented COBOL class using the `javac` command from Java 5 or Java 6.

RELATED CONCEPTS
"Java SDK support" on page 241

RELATED TASKS
"Preparing OO applications" on page 238

Data size limits

If COBOL for AIX programs are compiled with option `ADDR(32)` in effect, the maximum size limit of a data item could in some cases be smaller than the size of the data item in programs compiled with COBOL for AIX Version 2.

For full details about compiler limits, see the related reference about compiler limits.

RELATED REFERENCES
"ADDR" on page 246
Compiler limits (*COBOL for AIX Language Reference*)

XML generation

XML GENERATE returns exception code 417 if element character content contains characters that are illegal in XML content. XML generation continues with the element tag name prefixed with 'hex.', and the original data value represented in the document in hexadecimal.

The generated XML document is compatible with documents generated by COBOL for AIX Version 2, but the new exception code is raised.

RELATED TASKS
"Handling XML GENERATE exceptions" on page 384

RELATED REFERENCES
"XML GENERATE exceptions" on page 649

SFS files

If you use SFS files in a COBOL for AIX application, compile the application using the cob2_r command. The cob2_r4 command, which was recommended for compiling COBOL for AIX Version 2 applications that use the SFS file system, is not supported.

If you use shell scripts to compile COBOL for AIX applications, update them accordingly.

RELATED TASKS
"Compiling using shell scripts" on page 220
"Compiling for SFS files" on page 220

RELATED REFERENCES
"SFS file system" on page 117

Migration of 32-bit applications to 64-bit mode

Execution results can change in 64-bit mode. If you migrate 32-bit applications to 64-bit mode, you might need to do some recoding to accommodate the differences.

Although the main consideration in migrating to 64-bit mode is the change in size of data items, other areas of an application could potentially be impacted, as described below.

LENGTH OF special register:

In 32-bit mode, the implicit definition of the LENGTH OF special register is PICTURE 9(9) USAGE IS BINARY. In 64-bit mode, the implicit definition is PICTURE 9(18) USAGE IS BINARY.

LENGTH intrinsic function:

In 32-bit mode, FUNCTION LENGTH returns a 9-digit integer. In 64-bit mode, it returns an 18-digit integer.

Address data items and index data items:

The storage allocation for data items that contain addresses or indexes is 4 bytes in 32-bit mode, and 8 bytes in 64-bit mode. The data items that are affected are those that are defined with any of the following usages:

- POINTER
- FUNCTION-POINTER
- PROCEDURE-POINTER
- OBJECT REFERENCE
- INDEX

The special registers ADDRESS OF and JNIEnvPtr are implicitly defined as USAGE POINTER. Therefore storage is allocated for them as described above for USAGE POINTER data items.

Layout of group items:

The change in allocation size of address data items and index data items between 32-bit mode and 64-bit mode programs, described above, affects the layout of group items. In particular, data items that are located after an address or index data item could be located at different offsets within a group. The number of *slack bytes* (bytes inserted by the compiler to ensure correct alignment) could differ, and thus the total size of a group could change.

SYNCHRONIZED data items:

If the SYNCHRONIZED clause is specified for address data items or index data items, the items are aligned on a fullword boundary in 32-bit mode and on a doubleword boundary in 64-bit mode. Offsets of data items within a group, number of slack bytes, and total group size could change, as described above for group items.

Data files:

Data files created by 32-bit mode programs can be processed by 64-bit mode programs, and data files created by 64-bit mode programs can be processed by 32-bit mode programs. No addressing mode is attached to a data file. However, if a record definition contains an address data item or index data item, migration to 64-bit mode could cause the length of the record, and thus the length of the file, to change.

Middleware subsystems:

If a program depends on a middleware subsystem to do data-processing tasks, make sure that that subsystem works with 64-bit programs.

Restriction: DB2 Version 8 and CICS TXSeries support only 32-bit COBOL applications.

DB2 Version 9 supports either 32-bit or 64-bit COBOL applications.

Other considerations:

AIX does not support mixing 32-bit mode and 64-bit mode programs within an application, and the linker does not support linking object files of different addressing modes. If a 64-bit program makes interlanguage calls, or calls routines in a user library, make sure that 64-bit versions of the called routines are available.

RELATED REFERENCES
"ADDR" on page 246
"Data size limits" on page 569
Compiler limits (*COBOL for AIX Language Reference*)

Appendix B. Summary of differences with host COBOL

IBM COBOL for AIX implements certain items differently from the way that Enterprise COBOL for z/OS implements them. See the related references below for details.

RELATED TASKS
Chapter 25, "Porting applications between platforms," on page 457

RELATED REFERENCES
"Compiler options"
"Data representation"
"Environment variables" on page 575
"File specification" on page 576
"Interlanguage communication (ILC)" on page 576
"Input and output" on page 576
"Runtime options" on page 576
"Source code line size" on page 577
"Language elements" on page 577

Compiler options

COBOL for AIX treats the following compiler options as comments: `ADV`, `AWO`, `BINARY`, `BUFSIZE`, `CODEPAGE`, `DATA`, `DECK`, `DBCS`, `FASTSRT`, `FLAGMIG`, `INTDATE`, `LANGUAGE`, `NAME`, `OUTDD`, and `RENT` (the compiler always produces reentrant code). These options are flagged with I-level messages.

COBOL for AIX also treats the `NOADV` compiler option as a comment. It is flagged with a W-level message. If you specify this option, the application might yield unexpected results when you use a `WRITE` statement with the `ADVANCING` phrase.

For OO COBOL applications, you can specify the `-host` option of the `cob2_j` command, any setting of the `CHAR` or `FLOAT` compiler options, or both. However, you must specify any floating-point items that are used as method arguments or parameters, and any arguments to JNI services, in native format (for example, by using the `NATIVE` phrase on the data description entry).

RELATED TASKS
"Getting mainframe applications to compile" on page 457

RELATED REFERENCES
"cob2, cob2_j, and cob2_r options" on page 227
"CHAR" on page 250

Data representation

The representation of data can differ between IBM host COBOL and IBM COBOL for AIX.

Zoned decimal data

Sign representation for zoned decimal data is based on ASCII or EBCDIC depending on the setting of the `CHAR` compiler option (`NATIVE` or `EBCDIC`) and whether the `USAGE` clause is specified with the `NATIVE` phrase. COBOL for AIX processes the sign representation of zoned decimal data consistently with the processing that occurs on the host when the compiler option `NUMPROC(NOPFD)` is in effect.

Packed-decimal data

Sign representation for unsigned packed-decimal numbers is different between COBOL for AIX and host COBOL. COBOL for AIX always uses a sign nibble of `x'C'` for unsigned packed-decimal numbers. Host COBOL uses a sign nibble of `x'F'` for unsigned packed-decimal numbers. If you are going to share data files that contain packed-decimal numbers between AIX and z/OS, it is recommended that you use signed packed-decimal numbers instead of unsigned packed-decimal numbers.

Display floating-point data

You can use the `FLOAT(S390)` compiler option to indicate that display floating-point data items are in the zSeries data representation (hexadecimal) as opposed to the native (IEEE) format.

Do not specify the zSeries formats of display floating-point items as arguments on `INVOKE` statements or as method parameters. You must specify these arguments and parameters in the native formats in order for them to be interoperable with the Java data types.

National data

COBOL for AIX and host COBOL use UTF-16 big-endian format for national data.

EBCDIC and ASCII data

You can specify the EBCDIC collating sequence for alphanumeric data items using the following language elements:

- `ALPHABET` clause
- `PROGRAM COLLATING SEQUENCE` clause
- `COLLATING SEQUENCE` phrase of the `SORT` or `MERGE` statement

You can specify the `CHAR(EBCDIC)` compiler option to indicate that `DISPLAY` data items are in the zSeries data representation (EBCDIC).

Code-page determination for data conversion

For alphabetic, alphanumeric, DBCS, and national data items, the source code page used for implicit conversion of native characters is determined from the locale in effect at run time.

For alphanumeric, DBCS, and national literals, the source code page used for implicit conversion of characters is determined from the locale in effect at compile time.

DBCS character strings

Under COBOL for AIX, ASCII DBCS character strings are not delimited with the shift-in and shift-out characters except possibly with the dummy shift-in and shift-out characters as discussed below.

Use the SOSI compiler option to indicate that AIX workstation shift-out (X'1E') and shift-in (X'1F') control characters delimit DBCS character strings in the source program, including user-defined words, DBCS literals, alphanumeric literals, national literals, and comments. Host shift-out and shift-in control characters (X'0E' and X'0F', respectively) are usually converted to workstation shift-out and shift-in control characters when COBOL for AIX source code is downloaded, depending on the download method that you use.

Using control characters X'00' through X'1F' within an alphanumeric literal can yield unpredictable results.

RELATED TASKS
Chapter 11, "Setting the locale," on page 193
"Fixing differences caused by data representations" on page 458

RELATED REFERENCES
"CHAR" on page 250
"SOSI" on page 281

Environment variables

COBOL for AIX recognizes several environment variables, as listed below.

- *assignment-name*
- COBMSGS
- COBPATH
- COBRTOPT
- DB2DBDFT
- EBCDIC_CODEPAGE
- ENCINA_CDS_ROOT
- ENCINA_SFS_DATA_VOLUME
- ENCINA_SFS_INDEX_VOLUME
- ENCINA_VSAM_AUTO_FLUSH
- ENCINA_VSAM_CACHE
- LANG
- LC_ALL
- LC_COLLATE
- LC_CTYPE
- LC_MESSAGES
- LC_TIME
- LOCPATH

- NLSPATH
- TMP
- TZ

File specification

COBOL for AIX treats all files as single-volume files. All other file specifications are treated as comments. This change affects the following items: REEL, UNIT, MULTIPLE FILE TAPE clause, and CLOSE. . .UNIT/REEL.

Interlanguage communication (ILC)

ILC is available with C/C++ and PL/I programs.

These are the differences in ILC behavior on the AIX workstation compared to using ILC on the host with Language Environment:

- There are differences in termination behavior when a COBOL STOP RUN, a C exit(), or a PL/I STOP is used.
- There is no coordinated condition handling on the workstation. Avoid using a C longjmp() that crosses COBOL programs.
- On the host, the first program that is invoked within the process and that is enabled for Language Environment is considered to be the ″main″ program. On AIX, the first COBOL program invoked within the process is considered to be the main program by COBOL. This difference affects language semantics that are sensitive to the definition of the run unit (the execution unit that starts with a main program). For example, a STOP RUN results in the return of control to the invoker of the main program, which in a mixed-language environment might be different as stated above.

RELATED CONCEPTS
Chapter 30, “Preinitializing the COBOL runtime environment,” on page 513

Input and output

COBOL for AIX supports input and output for sequential, relative, and indexed files by using the STL file system and VSAM file system. Line-sequential input and output is supported by using the native byte stream file support of the platform.

Sizes and values are different for the file status *data-name* returned from the file system.

COBOL for AIX does not provide direct support for tape drives or diskette drives.

RELATED CONCEPTS
“File systems” on page 115

Runtime options

COBOL for AIX does not recognize the following host runtime options, and treats them as not valid: AIXBLD, ALL31, CBLPSHPOP, CBLQDA, COUNTRY, HEAP, MSGFILE, NATLANG, SIMVRD, and STACK.

On the host, you can use the STORAGE runtime option to initialize COBOL WORKING-STORAGE. With COBOL for AIX, use the WSCLEAR compiler option.

RELATED REFERENCES
"WSCLEAR" on page 291

Source code line size

A COBOL source line can contain fewer than 72 characters. A line ends on column 72 or where a newline control character is found.

Language elements

The following table lists language elements that are different between the host COBOL and COBOL for AIX compilers, and where possible offers advice about how to handle such differences in COBOL for AIX programs.

Many COBOL clauses and phrases that are valid in host COBOL are syntax checked but have no effect on the execution of COBOL for AIX programs. These clauses and phrases should have minimal effect on existing applications that you download. COBOL for AIX recognizes most host COBOL language syntax even if that syntax has no functional effect.

Table 74. **Language differences between Enterprise COBOL for z/OS and COBOL for AIX**

Language element	COBOL for AIX implementation or restriction
ACCEPT statement	If your host COBOL program expects ddnames as the targets of ACCEPT statements, define these targets by using equivalent environment variables with values set to appropriate file-names. In COBOL for AIX, *environment-name* and the associated environment-variable value, if set, determine file identification.
APPLY WRITE-ONLY clause	Syntax checked, but has no effect on the execution of the program in COBOL for AIX
ASSIGN clause	COBOL for AIX uses a different syntax and mapping to the system file-name based on *assignment-name*. ASSIGN. . .USING *data-name* is not supported in host COBOL.
BLOCK CONTAINS clause	Syntax checked, but has no effect on the execution of the program in COBOL for AIX
CALL statement	A file-name as a CALL argument is not supported in COBOL for AIX.
CLOSE statement	The following phrases are syntax checked, but have no effect on the execution of the program in COBOL for AIX: FOR REMOVAL, WITH NO REWIND, and UNIT/REEL. Avoid use of these phrases in programs that are intended to be portable.
CODE-SET clause	Syntax checked, but has no effect on the execution of the program in COBOL for AIX
DATA RECORDS clause	Syntax checked, but has no effect on the execution of the program in COBOL for AIX
DISPLAY statement	If your host COBOL program expects ddnames as the targets of DISPLAY statements, define these targets by using equivalent environment variables with values set to appropriate file-names. In COBOL for AIX, *environment-name* and the associated environment-variable value, if set, determine file identification.
File status *data-name-1*	Some values and meanings for file status 9*x* are different in host COBOL than in COBOL for AIX.
File status *data-name-8*	The format and values are different depending on the platform and the file system.
INDEX data items	In host COBOL, INDEX data items are implicitly defined as 4 bytes. In COBOL for AIX programs compiled with ADDR(32), their size is 4 bytes; with ADDR(64), their size is 8 bytes.

Table 74. **Language differences between Enterprise COBOL for z/OS and COBOL for AIX** *(continued)*

Language element	**COBOL for AIX implementation or restriction**
LABEL RECORDS clause	The phrases LABEL RECORD IS *data-name*, USE. . .AFTER. . .LABEL PROCEDURE, and GO TO MORE-LABELS are syntax checked, but have no effect on the execution of the program in COBOL for AIX. A warning is issued if you use any of these phrases. The user-label declaratives are not called at run time. You cannot port programs that depend on the user-label processing that z/OS QSAM supports.
MULTIPLE FILE TAPE	Syntax checked, but has no effect on the execution of the program in COBOL for AIX. On the AIX workstation, all files are treated as single-volume files.
OBJECT REFERENCE data items	In host COBOL, OBJECT REFERENCE data items are implicitly defined as 4 bytes. In COBOL for AIX programs compiled with ADDR(32), their size is 4 bytes; with ADDR(64), their size is 8 bytes.
OPEN statement	The following phrases are syntax checked, but have no effect on the execution of the program in COBOL for AIX: REVERSED and WITH NO REWIND.
PASSWORD clause	Syntax checked, but has no effect on the execution of the program in COBOL for AIX
POINTER, PROCEDURE-POINTER, and FUNCTION-POINTER data items	In host COBOL, POINTER and FUNCTION-POINTER data items are implicitly defined as 4 bytes as are the special registers ADDRESS OF and JNIEnvPtr; PROCEDURE-POINTER data items are implicitly defined as 8 bytes. In COBOL for AIX programs compiled with ADDR(32), the size of each of these items is 4 bytes; with ADDR(64), their size is 8 bytes.
READ. . .PREVIOUS	In COBOL for AIX only, allows you to read the previous record for relative or indexed files with DYNAMIC access mode
RECORD CONTAINS clause	The RECORD CONTAINS *n* CHARACTERS clause is accepted with one exception: RECORD CONTAINS 0 CHARACTERS is syntax checked, but has no effect on the execution of the program in COBOL for AIX.
RECORDING MODE clause	Syntax checked, but has no effect on the execution of the program in COBOL for AIX for relative, indexed, and line-sequential files. RECORDING MODE U is syntax checked, but has no effect on the execution of the program for sequential files.
RERUN clause	Syntax checked, but has no effect on the execution of the program in COBOL for AIX
RESERVE clause	Syntax checked, but has no effect on the execution of the program in COBOL for AIX
SAME AREA clause	Syntax checked, but has no effect on the execution of the program in COBOL for AIX
SAME SORT clause	Syntax checked, but has no effect on the execution of the program in COBOL for AIX
SHIFT-IN, SHIFT-OUT special registers	The COBOL for AIX compiler puts out an E-level message if it encounters these registers unless the CHAR(EBCDIC) compiler option is in effect.
SORT-CONTROL special register	The implicit definition and contents of this special register differ between host and workstation COBOL.
SORT-CORE-SIZE special register	The contents of this special register differ between host and workstation COBOL.
SORT-FILE-SIZE special register	Syntax checked, but has no effect on the execution of the program in COBOL for AIX. Values in this special register are not used.
SORT-MESSAGE special register	Syntax checked, but has no effect on the execution of the program in COBOL for AIX
SORT-MODE-SIZE special register	Syntax checked, but has no effect on the execution of the program in COBOL for AIX. Values in this special register are not used.
SORT MERGE AREA clause	Syntax checked, but has no effect on the execution of the program in COBOL for AIX
START. . .	In COBOL for AIX, the following relational operators are allowed: IS LESS THAN, IS <, IS NOT GREATER THAN, IS NOT >, IS LESS THAN OR EQUAL TO, IS <=.
STOP RUN	Not supported in COBOL for AIX multithreaded programs; can be replaced in a multithreaded program with a call to the C exit() function
WRITE statement	In COBOL for AIX, if you specify WRITE. . .ADVANCING with environment names C01 through C12 or S01 through S05, one line is advanced.

Table 74. **Language differences between Enterprise COBOL for z/OS and COBOL for AIX** *(continued)*

Language element	**COBOL for AIX implementation or restriction**
XML PARSE statement	In host COBOL programs compiled with the host-only option XMLPARSE(XMLSS), additional syntax (the ENCODING phrase and RETURNING NATIONAL phrase) and special registers for namespace processing are available that are not available with COBOL for AIX.
Names known to the platform environment	The following names are identified differently: *program-name*, *text-name*, *library-name*, *assignment-name*, file-name in the SORT-CONTROL special register, *basis-name*, DISPLAY or ACCEPT target identification, and system-dependent names.

Appendix C. zSeries host data format considerations

The following are considerations, restrictions, and limitations that apply to the use of zSeries host data internal representation.

The `CHAR` and `FLOAT` compiler options determine whether zSeries host data format or native data format is used (other than for `COMP-5` items or items defined with the `NATIVE` phrase in the `USAGE` clause). (The terms *host data format* and *native data format* in this information refer to the internal representation of data items.)

CICS access

There is no zSeries host data format support for COBOL programs that are translated by the separate or integrated CICS translator and run on TXSeries.

Date and time callable services

You can use the date and time callable services with the zSeries host data format internal representations. All of the parameters passed to the callable services must be in zSeries host data format. You cannot mix native and host data internal representations in the same call to a date and time service.

Floating-point overflow exceptions

Due to differences in the limits of floating-point data representations on the AIX workstation and the zSeries host, it is possible if `FLOAT(HEX)` is in effect that a floating-point overflow exception could occur during conversion between the two formats. For example, you might receive the following message on the workstation when you run a program that runs successfully on the host:

```
IWZ053S An overflow occurred on conversion to floating point
```

To avoid this problem, you must be aware of the maximum floating-point values supported on each platform for the respective data types. The limits are shown in the following table.

Table 75. **Maximum floating-point values**

Data type	Maximum workstation value	Maximum zSeries host value
`COMP-1`	*(2**128 - 2**4) (approx.*3.4028E+38)	*(16**63 - 16**57) (approx.*7.2370E+75)
`COMP-2`	*(2**1024 - 2**971) (approx.*1.7977E+308)	*(16**63 - 16**49) (approx.*7.2370E+75)
* Indicates that the value can be positive or negative.		

As shown above, the host can carry a larger `COMP-1` value than the workstation and the workstation can carry a larger `COMP-2` value than the host.

DB2

The zSeries host data format compiler options can be used with DB2 programs.

Object-oriented syntax for Java interoperability

The zSeries format of floating-point data items must not be specified for arguments or returning items in `INVOKE` statements.

Distributed Computing Environment applications

The zSeries host data format compiler options should not be used with Distributed Computing Environment programs.

File data

- EBCDIC data and hexadecimal binary data can be read from and written to any sequential, relative, or indexed files. No automatic conversion takes place.
- If you are accessing files that contain host data, use the compiler options `COLLSEQ(EBCDIC)`, `CHAR(EBCDIC)`, and `FLOAT(S390)` to process EBCDIC character data and hexadecimal floating-point data that is acquired from these files.

SORT

| All of the zSeries host data formats except DBCS (`USAGE DISPLAY-1`) can be used as sort keys.

RELATED CONCEPTS
"Formats for numeric data" on page 43

RELATED TASKS
"Coding interoperable data types in COBOL and Java" on page 448

RELATED REFERENCES
Chapter 14, "Compiler options," on page 243

Appendix D. Intermediate results and arithmetic precision

The compiler handles arithmetic statements as a succession of operations performed according to operator precedence, and sets up intermediate fields to contain the results of those operations. The compiler uses algorithms to determine the number of integer and decimal places to reserve.

Intermediate results are possible in the following cases:

- In an ADD or SUBTRACT statement that contains more than one operand immediately after the verb
- In a COMPUTE statement that specifies a series of arithmetic operations or multiple result fields
- In an arithmetic expression contained in a conditional statement or in a reference-modification specification
- In an ADD, SUBTRACT, MULTIPLY, or DIVIDE statement that uses the GIVING option and multiple result fields
- In a statement that uses an intrinsic function as an operand

"Example: calculation of intermediate results" on page 585

The precision of intermediate results depends on whether you compile using the default option ARITH(COMPAT) (referred to as *compatibility mode*) or using ARITH(EXTEND) (referred to as *extended mode*).

In compatibility mode, evaluation of arithmetic operations is unchanged from that in IBM COBOL Set for AIX:

- 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 43
"Fixed-point contrasted with floating-point arithmetic" on page 56

RELATED REFERENCES
"Fixed-point data and intermediate results" on page 585
"Floating-point data and intermediate results" on page 590
"Arithmetic expressions in nonarithmetic statements" on page 591
"ARITH" on page 248

Terminology used for intermediate results

To understand this information about intermediate results, you need to understand the following terminology.

i	The number of integer places carried for an intermediate result. (If you use the `ROUNDED` phrase, one more integer place might be carried for accuracy if necessary.)
d	The number of decimal places carried for an intermediate result. (If you use the `ROUNDED` phrase, one more decimal place might be carried for accuracy if necessary.)
dmax	In a particular statement, the largest of the following items: • The number of decimal places needed for the final result field or fields • The maximum number of decimal places defined for any operand, except divisors or exponents • The *outer-dmax* for any function operand
inner-dmax	In reference to a function, the largest of the following items: • The number of decimal places defined for any of its elementary arguments • The *dmax* for any of its arithmetic expression arguments • The *outer-dmax* for any of its embedded functions
outer-dmax	The number of decimal places that a function result contributes to operations outside of its own evaluation (for example, if the function is an operand in an arithmetic expression, or an argument to another function).
op1	The first operand in a generated arithmetic statement (in division, the divisor).
op2	The second operand in a generated arithmetic statement (in division, the dividend).
i1, i2	The number of integer places in *op1* and *op2*, respectively.
d1, d2	The number of decimal places in *op1* and *op2*, respectively.
ir	The intermediate result when a generated arithmetic statement or operation is performed. (Intermediate results are generated either in registers or storage locations.)
ir1, ir2	Successive intermediate results. (Successive intermediate results might have the same storage location.)

RELATED REFERENCES
ROUNDED phrase (*COBOL for AIX 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 52

RELATED REFERENCES
"Terminology used for intermediate results" on page 584

Fixed-point data and intermediate results

The compiler determines the number of integer and decimal places in an intermediate result.

Addition, subtraction, multiplication, and division

The following table shows the precision theoretically possible as the result of addition, subtraction, multiplication, or division.

Operation	Integer places	Decimal places
+ or -	(*i1* or *i2*) + 1, whichever is greater	*d1* or *d2*, whichever is greater
*	*i1* + *i2*	*d1* + *d2*
/	*i2* + *d1*	(*d2* - *d1*) or *dmax*, whichever is greater

You must define the operands of any arithmetic statements with enough decimal places to obtain the accuracy you want in the final result.

The following table shows the number of places the compiler carries for fixed-point intermediate results of arithmetic operations that involve addition, subtraction, multiplication, or division in *compatibility mode* (that is, when the default compiler option `ARITH(COMPAT)` is in effect):

Value of *i* + *d*	Value of *d*	Value of *i* + *dmax*	Number of places carried for *ir*
<30 or =30	Any value	Any value	*i* integer and *d* decimal places

Value of *i* + *d*	Value of *d*	Value of *i* + *dmax*	Number of places carried for *ir*
>30	<*dmax* or =*dmax*	Any value	30-*d* integer and *d* decimal places
	>*dmax*	<30 or =30	*i* integer and 30-*i* decimal places
		>30	30-*dmax* integer and *dmax* decimal places

The following table shows the number of places the compiler carries for fixed-point intermediate results of arithmetic operations that involve addition, subtraction, multiplication, or division in *extended mode* (that is, when the compiler option `ARITH(EXTEND)` is in effect):

Value of *i* + *d*	Value of *d*	Value of *i* + *dmax*	Number of places carried for *ir*
<31 or =31	Any value	Any value	*i* integer and *d* decimal places
>31	<*dmax* or =*dmax*	Any value	31-*d* integer and *d* decimal places
	>*dmax*	<31 or =31	*i* integer and 31-*i* decimal places
		>31	31-*dmax* integer and *dmax* decimal places

Exponentiation

Exponentiation is represented by the expression *op1* ** *op2*. Based on the characteristics of *op2*, the compiler handles exponentiation of fixed-point numbers in one of three ways:

- When *op2* is expressed with decimals, floating-point instructions are used.
- When *op2* is an integral literal or constant, the value *d* is computed as

  `d = d1 * |op2|`

  and the value *i* is computed based on the characteristics of *op1*:
  - When *op1* is a data-name or variable,

    `i = i1 * |op2|`
  - When *op1* is a literal or constant, *i* is set equal to the number of integers in the value of *op1* `** |op2|`.

  In compatibility mode (compilation using `ARITH(COMPAT)`), the compiler having calculated *i* and *d* takes the action indicated in the table below to handle the intermediate results *ir* of the exponentiation.

Value of *i* + *d*	Other conditions	Action taken
<30	Any	*i* integer and *d* decimal places are carried for *ir*.
=30	*op1* has an odd number of digits.	*i* integer and *d* decimal places are carried for *ir*.
	op1 has an even number of digits.	Same action as when *op2* is an integral data-name or variable (shown below). Exception: for a 30-digit integer raised to the power of literal 1, *i* integer and *d* decimal places are carried for *ir*.
>30	Any	Same action as when *op2* is an integral data-name or variable (shown below)

In extended mode (compilation using `ARITH(EXTEND)`), the compiler having calculated *i* and *d* takes the action indicated in the table below to handle the intermediate results *ir* of the exponentiation.

Value of *i* + *d*	**Other conditions**	**Action taken**
<31	Any	*i* integer and *d* decimal places are carried for *ir*.
=31 or >31	Any	Same action as when *op2* is an integral data-name or variable (shown below). Exception: for a 31-digit integer raised to the power of literal 1, *i* integer and *d* decimal places are carried for *ir*.

If *op2* is negative, the value of 1 is then divided by the result produced by the preliminary computation. The values of *i* and *d* that are used are calculated following the division rules for fixed-point data already shown above.

- When *op2* is an integral data-name or variable, *dmax* decimal places and 30-*dmax* (compatibility mode) or 31-*dmax* (extended mode) integer places are used. *op1* is multiplied by itself (|*op2*| - 1) times for nonzero *op2*.

  If *op2* is equal to 0, the result is 1. Division-by-0 and exponentiation `SIZE ERROR` conditions apply.

Fixed-point exponents with more than nine significant digits are always truncated to nine digits. If the exponent is a literal or constant, an E-level compiler diagnostic message is issued; otherwise, an informational message is issued at run time.

"Example: exponentiation in fixed-point arithmetic"

RELATED REFERENCES
"Terminology used for intermediate results" on page 584
"Truncated intermediate results" on page 588
"Binary data and intermediate results" on page 588
"Floating-point data and intermediate results" on page 590
"Intrinsic functions evaluated in fixed-point arithmetic" on page 588
"ARITH" on page 248
SIZE ERROR phrases (*COBOL for AIX Language Reference*)

Example: exponentiation in fixed-point arithmetic

The following example shows how the compiler performs an exponentiation to a nonzero integer power as a succession of multiplications, storing intermediate results as needed.

```
COMPUTE Y = A ** B
```

If `B` is equal to 4, the result is computed as shown below. The values of *i* and *d* that are used are calculated according to the multiplication rules for fixed-point data and intermediate results (referred to below).

1. Multiply `A` by `A` yielding *ir1*.
2. Multiply *ir1* by `A` yielding *ir2*.
3. Multiply *ir2* by `A` yielding *ir3*.
4. Move *ir3* to *ir4*.

   ir4 has *dmax* decimal places. Because `B` is positive, *ir4* is moved to `Y`. If `B` were equal to -4, however, an additional fifth step would be performed:
5. Divide *ir4* into 1 yielding *ir5*.

ir5 has *dmax* decimal places, and would then be moved to `Y`.

RELATED REFERENCES
"Terminology used for intermediate results" on page 584
"Fixed-point data and intermediate results" on page 585

Truncated intermediate results

Whenever the number of digits in an intermediate result exceeds 30 in compatibility mode or 31 in extended mode, the compiler truncates to 30 (compatibility mode) or 31 (extended mode) digits and issues a warning. If truncation occurs at run time, a message is issued and the program continues running.

If you want to avoid the truncation of intermediate results that can occur in fixed-point calculations, use floating-point operands (`COMP-1` or `COMP-2`) instead.

RELATED CONCEPTS
"Formats for numeric data" on page 43

RELATED REFERENCES
"Fixed-point data and intermediate results" on page 585
"ARITH" on page 248

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 585
"ARITH" on page 248

Intrinsic functions evaluated in fixed-point arithmetic

The compiler determines the *inner-dmax* and *outer-dmax* values for an intrinsic function from the characteristics of the function.

Integer functions

Integer intrinsic functions return an integer; thus their *outer-dmax* is always zero. For those integer functions whose arguments must all be integers, the *inner-dmax* is thus also always zero.

The following table summarizes the *inner-dmax* and the precision of the function result.

Function	*Inner-dmax*	**Digit precision of function result**
`DATE-OF-INTEGER`	0	8

Function	*Inner-dmax*	**Digit precision of function result**
DATE-TO-YYYYMMDD	0	8
DAY-OF-INTEGER	0	7
DAY-TO-YYYYDDD	0	7
FACTORIAL	0	30 in compatibility mode, 31 in extended mode
INTEGER-OF-DATE	0	7
INTEGER-OF-DAY	0	7
LENGTH	n/a	9
MOD	0	min(*i1 i2*)
ORD	n/a	3
ORD-MAX		9
ORD-MIN		9
YEAR-TO-YYYY	0	4
INTEGER		For a fixed-point argument: one more digit than in the argument. For a floating-point argument: 30 in compatibility mode, 31 in extended mode.
INTEGER-PART		For a fixed-point argument: same number of digits as in the argument. For a floating-point argument: 30 in compatibility mode, 31 in extended mode.

Mixed functions

A *mixed* intrinsic function is a function whose result type depends on the type of its arguments. A mixed function is fixed point if all of its arguments are numeric and none of its arguments is floating point. (If any argument of a mixed function is floating point, the function is evaluated with floating-point instructions and returns a floating-point result.) When a mixed function is evaluated with fixed-point arithmetic, the result is integer if all of the arguments are integer; otherwise, the result is fixed point.

For the mixed functions MAX, MIN, RANGE, REM, and SUM, the *outer-dmax* is always equal to the *inner-dmax* (and both are thus zero if all the arguments are integer). To determine the precision of the result returned for these functions, apply the rules for fixed-point arithmetic and intermediate results (as referred to below) to each step in the algorithm.

MAX

1. Assign the first argument to the function result.
2. For each remaining argument, do the following 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 584
"Fixed-point data and intermediate results" on page 585
"Floating-point data and intermediate results"
"ARITH" on page 248

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 584
"ARITH" on page 248

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 56

RELATED REFERENCES
"Terminology used for intermediate results" on page 584
"Fixed-point data and intermediate results" on page 585
"Floating-point data and intermediate results" on page 590
IF statement (*COBOL for AIX Language Reference*)
EVALUATE statement (*COBOL for AIX Language Reference*)
Conditional expressions (*COBOL for AIX Language Reference*)

Appendix E. Complex OCCURS DEPENDING ON

Several types of complex OCCURS DEPENDING ON (*complex ODO*) are possible. Complex ODO is supported as an extension to Standard COBOL 85.

The basic forms of complex ODO permitted by the compiler are as follows:

- Variably located item or group: A data item described by an OCCURS clause with the DEPENDING ON phrase is followed by a nonsubordinate elementary or group data item.
- Variably located table: A data item described by an OCCURS clause with the DEPENDING ON phrase is followed by a nonsubordinate data item described by an OCCURS clause.
- Table that has variable-length elements: A data item described by an OCCURS clause contains a subordinate data item described by an OCCURS clause with the DEPENDING ON phrase.
- Index name for a table that has variable-length elements.
- Element of a table that has variable-length elements.

"Example: complex ODO"

RELATED TASKS
"Preventing index errors when changing ODO object value" on page 595
"Preventing overlay when adding elements to a variable table" on page 595

RELATED REFERENCES
"Effects of change in ODO object value" on page 594
OCCURS DEPENDING ON clause (*COBOL for AIX Language Reference*)

Example: complex ODO

The following example illustrates the possible types of occurrence of complex ODO.

```
01  FIELD-A.
    02 COUNTER-1                           PIC S99.
    02 COUNTER-2                           PIC S99.
    02 TABLE-1.
       03 RECORD-1 OCCURS 1 TO 5 TIMES
                   DEPENDING ON COUNTER-1   PIC X(3).
    02 EMPLOYEE-NUMBER                     PIC X(5). (1)
    02 TABLE-2 OCCURS 5 TIMES                        (2)(3)
               INDEXED BY INDX.                      (4)
       03 TABLE-ITEM                       PIC 99.   (5)
       03 RECORD-2 OCCURS 1 TO 3 TIMES
                   DEPENDING ON COUNTER-2.
          04 DATA-NUM                      PIC S99.
```

Definition: In the example, COUNTER-1 is an *ODO object*, that is, it is the object of the DEPENDING ON clause of RECORD-1. RECORD-1 is said to be an *ODO subject*. Similarly, COUNTER-2 is the ODO object of the corresponding ODO subject, RECORD-2.

The types of complex ODO occurrences shown in the example above are as follows:

(1) A variably located item: EMPLOYEE-NUMBER is a data item that follows, but is not subordinate to, a variable-length table in the same level-01 record.

(2) A variably located table: TABLE-2 is a table that follows, but is not subordinate to, a variable-length table in the same level-01 record.

(3) A table with variable-length elements: TABLE-2 is a table that contains a subordinate data item, RECORD-2, whose number of occurrences varies depending on the content of its ODO object.

(4) An index-name, INDX, for a table that has variable-length elements.

(5) An element, TABLE-ITEM, of a table that has variable-length elements.

How length is calculated

The length of the variable portion of each record is the product of its ODO object and the length of its ODO subject. For example, whenever a reference is made to one of the complex ODO items shown above, the actual length, if used, is computed as follows:

- The length of TABLE-1 is calculated by multiplying the contents of COUNTER-1 (the number of occurrences of RECORD-1) by 3 (the length of RECORD-1).
- The length of TABLE-2 is calculated by multiplying the contents of COUNTER-2 (the number of occurrences of RECORD-2) by 2 (the length of RECORD-2), and adding the length of TABLE-ITEM.
- The length of FIELD-A is calculated by adding the lengths of COUNTER-1, COUNTER-2, TABLE-1, EMPLOYEE-NUMBER, and TABLE-2 times 5.

Setting values of ODO objects

You must set *every* ODO object in a group item before you reference any complex ODO item in the group. For example, before you refer to EMPLOYEE-NUMBER in the code above, you must set COUNTER-1 and COUNTER-2 even though EMPLOYEE-NUMBER does not directly depend on either ODO object for its value.

Restriction: An ODO object cannot be variably located.

Effects of change in ODO object value

If a data item that is described by an OCCURS clause with the DEPENDING ON phrase is followed in the same group by one or more nonsubordinate data items (a form of complex ODO), any change in value of the ODO object affects subsequent references to complex ODO items in the record.

For example:

- The size of any group that contains the relevant ODO clause reflects the new value of the ODO object.
- A MOVE to a group that contains the ODO subject is made based on the new value of the ODO object.
- The location of any nonsubordinate items that follow the item described with the ODO clause is affected by the new value of the ODO object. (To preserve the contents of the nonsubordinate items, move them to a work area before the value of the ODO object changes, then move them back.)

The value of an ODO object can change when you move data to the ODO object or to the group in which it is contained. The value can also change if the ODO object is contained in a record that is the target of a READ statement.

RELATED TASKS
"Preventing index errors when changing ODO object value"
"Preventing overlay when adding elements to a variable table"

Preventing index errors when changing ODO object value

Be careful if you reference a complex-ODO index-name, that is, an index-name for a table that has variable-length elements, after having changed the value of the ODO object for a subordinate data item in the table.

When you change the value of an ODO object, the byte offset in an associated complex-ODO index is no longer valid because the table length has changed. Unless you take precautions, you will have unexpected results if you then code a reference to the index-name such as:

- A reference to an element of the table
- A SET statement of the form SET *integer-data-item* TO *index-name* (format 1)
- A SET statement of the form SET *index-name* UP|DOWN BY *integer* (format 2)

To avoid this type of error, do these steps:

1. Save the index in an integer data item. (Doing so causes an implicit conversion: the integer item receives the table element occurrence number that corresponds to the offset in the index.)
2. Change the value of the ODO object.
3. Immediately restore the index from the integer data item. (Doing so causes an implicit conversion: the index-name receives the offset that corresponds to the table element occurrence number in the integer item. The offset is computed according to the table length then in effect.)

The following code shows how to save and restore the index-name (shown in "Example: complex ODO" on page 593) when the ODO object COUNTER-2 changes.

```
 77  INTEGER-DATA-ITEM-1      PIC 99.
 . . .
     SET INDX TO 5.
*        INDX is valid at this point.
     SET INTEGER-DATA-ITEM-1 TO INDX.
*        INTEGER-DATA-ITEM-1 now has the
*        occurrence number that corresponds to INDX.
     MOVE NEW-VALUE TO COUNTER-2.
*        INDX is not valid at this point.
     SET INDX TO INTEGER-DATA-ITEM-1.
*        INDX is now valid, containing the offset
*        that corresponds to INTEGER-DATA-ITEM-1, and
*        can be used with the expected results.
```

RELATED REFERENCES
SET statement (*COBOL for AIX 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.

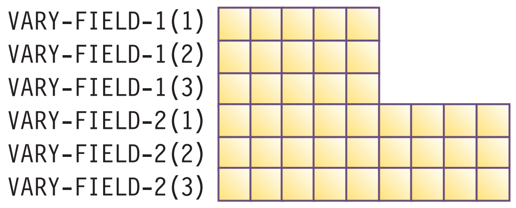
To avoid this type of error, do these steps:

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).
```

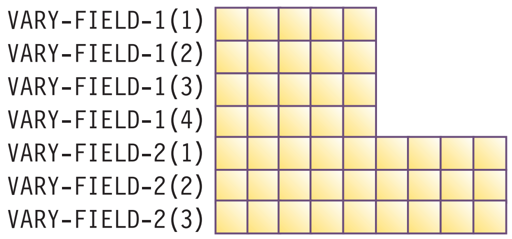
Each element of `VARY-FIELD-1` has 5 bytes, and each element of `VARY-FIELD-2` has 9 bytes. If `CONTROL-1` and `CONTROL-2` both contain the value 3, you can picture storage for `VARY-FIELD-1` and `VARY-FIELD-2` as follows:



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`.

Appendix F. Date and time callable services

With the date and time callable services, you can get the current local time and date in several formats and convert dates and times.

The available date and time callable services are shown below. Two of the services, CEEQCEN and CEESCEN, provide a predictable way to handle two-digit years, such as 91 for 1991 or 08 for 2008.

Table 76. **Date and time callable services**

Callable service	Description
"CEECBLDY: convert date to COBOL integer format" on page 600	Converts character date value to COBOL integer date format. Day one is 01 January 1601 and the value is incremented by one for each subsequent day.
"CEEDATE: convert Lilian date to character format" on page 604	Converts dates in the Lilian format back to character values.
"CEEDATM: convert seconds to character timestamp" on page 608	Converts number of seconds to character timestamp.
"CEEDAYS: convert date to Lilian format" on page 611	Converts character date values to the Lilian format. Day one is 15 October 1582 and the value is incremented by one for each subsequent day.
"CEEDYWK: calculate day of week from Lilian date" on page 615	Provides day of week calculation.
"CEEGMT: get current Greenwich Mean Time" on page 617	Gets current Greenwich Mean Time (date and time).
"CEEGMTO: get offset from Greenwich Mean Time to local time" on page 619	Gets difference between Greenwich Mean Time and local time.
"CEEISEC: convert integers to seconds" on page 621	Converts binary year, month, day, hour, second, and millisecond to a number representing the number of seconds since 00:00:00 15 October 1582.
"CEELOCT: get current local date or time" on page 623	Gets current date and time.
"CEEQCEN: query the century window" on page 625	Queries the callable services century window.
"CEESCEN: set the century window" on page 627	Sets the callable services century window.
"CEESECI: convert seconds to integers" on page 629	Converts a number representing the number of seconds since 00:00:00 15 October 1582 to seven separate binary integers representing year, month, day, hour, minute, second, and millisecond.
"CEESECS: convert timestamp to seconds" on page 632	Converts character timestamps (a date and time) to the number of seconds since 00:00:00 15 October 1582.
"CEEUTC: get coordinated universal time" on page 636	Same as CEEGMT.
"IGZEDT4: get current date" on page 636	Returns the current date with a four-digit year in the form YYYMMDD.

All of these date and time callable services allow source code compatibility with Enterprise COBOL for z/OS. There are, however, significant differences in the way conditions are handled.

The date and time callable services are in addition to the date/time intrinsic functions shown below.

Table 77. **Date and time intrinsic functions**

Intrinsic function	Description
CURRENT-DATE	Current date and time and difference from Greenwich mean time
DATE-OF-INTEGER[1]	Standard date equivalent (YYYYMMDD) of integer date
DATE-TO-YYYYMMDD[1]	Standard date equivalent (YYYYMMDD) of integer date with a windowed year, according to the specified 100-year interval
DATEVAL[1]	Date field equivalent of integer or alphanumeric date
DAY-OF-INTEGER[1]	Julian date equivalent (YYYYDDD) of integer date
DAY-TO-YYYYDDD[1]	Julian date equivalent (YYYYMMDD) of integer date with a windowed year, according to the specified 100-year interval
INTEGER-OF-DATE	Integer date equivalent of standard date (YYYYMMDD)
INTEGER-OF-DAY	Integer date equivalent of Julian date (YYYYDDD)
UNDATE[1]	Nondate equivalent of integer or alphanumeric date field
YEAR-TO-YYYY[1]	Expanded year equivalent (YYYY) of windowed year, according to the specified 100-year interval
YEARWINDOW[1]	Starting year of the century window specified by the YEARWINDOW compiler option
1. Behavior depends on the setting of the DATEPROC compiler option.	

"Example: formatting dates for output" on page 560

RELATED REFERENCES
"Feedback token" on page 561
CALL statement (*COBOL for AIX Language Reference*)
Function definitions (*COBOL for AIX Language Reference*)

CEECBLDY: convert date to COBOL integer format

CEECBLDY converts a string that represents a date into the number of days since 31 December 1600. Use CEECBLDY to access the century window of the date and time callable services and to perform date calculations with COBOL intrinsic functions.

This service is similar to CEEDAYS except that it provides a string in COBOL integer format, which is compatible with COBOL intrinsic functions.

CALL CEECBLDY syntax

```
►►—CALL—"CEECBLDY"—USING—input_char_date,—picture_string,——————————►

►—output_Integer_date,—fc.————————————————————————————————————————►◄
```

input_char_date **(input)**

A halfword length-prefixed character string that represents a date or timestamp, in a format conforming to that specified by *picture_string*.

The character string must contain between 5 and 255 characters, inclusive. *input_char_date* can contain leading or trailing blanks. Parsing for a date begins with the first nonblank character (unless the picture string itself contains leading blanks, in which case CEECBLDY skips exactly that many positions before parsing begins).

After parsing a valid date, as determined by the format of the date specified in *picture_string*, CEECBLDY ignores all remaining characters. Valid dates range between and include 01 January 1601 to 31 December 9999.

picture_string **(input)**

A halfword length-prefixed character string indicating the format of the date specified in *input_char_date*.

Each character in the *picture_string* corresponds to a character in *input_char_date*. For example, if you specify MMDDYY as the *picture_string*, CEECBLDY reads an *input_char_date* of 060288 as 02 June 1988.

If delimiters such as the slash (/) appear in the picture string, you can omit leading zeros. For example, the following calls to CEECBLDY would each assign the same value, 141502 (02 June 1988), to COBINTDTE:

```
MOVE '6/2/88' TO DATEVAL-STRING.
MOVE 6 TO DATEVAL-LENGTH.
MOVE 'MM/DD/YY' TO PICSTR-STRING.
MOVE 8 TO PICSTR-LENGTH.
CALL CEECBLDY USING DATEVAL, PICSTR, COBINTDTE, FC.

MOVE '06/02/88' TO DATEVAL-STRING.
MOVE 8 TO DATEVAL-LENGTH.
MOVE 'MM/DD/YY' TO PICSTR-STRING.
MOVE 8 TO PICSTR-LENGTH.
CALL CEECBLDY USING DATEVAL, PICSTR, COBINTDTE, FC.

MOVE '060288' TO DATEVAL-STRING.
MOVE 6 TO DATEVAL-LENGTH.
MOVE 'MMDDYY' TO PICSTR-STRING.
MOVE 6 TO PICSTR-LENGTH.
CALL CEECBLDY USING DATEVAL, PICSTR, COBINTDTE, FC.

MOVE '88154' TO DATEVAL-STRING.
MOVE 5 TO DATEVAL-LENGTH.
MOVE 'YYDDD' TO PICSTR-STRING.
MOVE 5 TO PICSTR-LENGTH.
CALL CEECBLDY USING DATEVAL, PICSTR, COBINTDTE, FC.
```

Whenever characters such as colons or slashes are included in the *picture_string* (such as HH:MI:SS YY/MM/DD), they count as placeholders but are otherwise ignored.

If *picture_string* includes a Japanese Era symbol <JJJJ>, the YY position in *input_char_date* is replaced by the year number within the Japanese Era. For example, the year 1988 equals the Japanese year 63 in the Showa era.

output_Integer_date **(output)**

A 32-bit binary integer that represents the COBOL integer date, the number of days since 31 December 1600. For example, 16 May 1988 is day number 141485.

If *input_char_date* does not contain a valid date, *output_Integer_date* is set to 0 and CEECBLDY terminates with a non-CEE000 symbolic feedback code.

Date calculations are performed easily on the *output_Integer_date*, because *output_Integer_date* is an integer. Leap year and end-of-year anomalies do not affect the calculations.

fc **(output)**

A 12-byte feedback code (optional) that indicates the result of this service.

Table 78. **CEECBLDY symbolic conditions**

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	--	The service completed successfully.
CEE2EB	3	2507	Insufficient data was passed to CEEDAYS or CEESECS. The Lilian value was not calculated.
CEE2EC	3	2508	The date value passed to CEEDAYS or CEESECS was invalid.
CEE2ED	3	2509	The era passed to CEEDAYS or CEESECS was not recognized.
CEE2EH	3	2513	The input date passed in a CEEISEC, CEEDAYS, or CEESECS call was not within the supported range.
CEE2EL	3	2517	The month value in a CEEISEC call was not recognized.
CEE2EM	3	2518	An invalid picture string was specified in a call to a date/time service.
CEE2EO	3	2520	CEEDAYS detected nonnumeric data in a numeric field, or the date string did not match the picture string.
CEE2EP	3	2521	The <JJJJ>, <CCCC>, or <CCCCCCCC> year-within-era value passed to CEEDAYS or CEESECS was zero.

Usage notes

- Call CEECBLDY only from COBOL programs that use the returned value as input to COBOL intrinsic functions. Unlike CEEDAYS, there is no inverse function of CEECBLDY, because it is only for COBOL users who want to use the date and time century window service together with COBOL intrinsic functions for date calculations. The inverse of CEECBLDY is provided by the `DATE-OF-INTEGER` and `DAY-OF-INTEGER` intrinsic functions.
- To perform calculations on dates earlier than 1 January 1601, add 4000 to the year in each date, convert the dates to COBOL integer format, then do the calculation. If the result of the calculation is a date, as opposed to a number of days, convert the result to a date string and subtract 4000 from the year.

- By default, two-digit years lie within the 100-year range that starts 80 years before the system date. Thus in 2008, all two-digit years represent dates between 1928 and 2027, inclusive. You can change this default range by using the CEESCEN callable service.

Example

```
CBL LIB
      *************************************************
      **                                             **
      ** Function: Invoke CEECBLDY callable service  **
      ** to convert date to COBOL integer format.    **
      ** This service is used when using the         **
      ** Century Window feature of the date and time **
      ** callable services mixed with COBOL          **
      ** intrinsic functions.                        **
      **                                             **
      *************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLDY.
      *
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  CHRDATE.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char    PIC X
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                               of CHRDATE.
       01  PICSTR.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char    PIC X
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                               of  PICSTR.
       01  INTEGER                 PIC S9(9) BINARY.
       01  NEWDATE                 PIC 9(8).
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) COMP.
                   04  Msg-No      PIC S9(4) COMP.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) COMP.
                   04  Cause-Code  PIC S9(4) COMP.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info            PIC S9(9) COMP.
      *
       PROCEDURE DIVISION.
       PARA-CBLDAYS.
      *************************************************
      ** Specify input date and length               **
      *************************************************
           MOVE 25 TO Vstring-length of CHRDATE.
           MOVE '1 January 00'
               to Vstring-text of CHRDATE.
      *************************************************
      ** Specify a picture string that describes     **
      ** input date, and set the string's length.    **
      *************************************************
           MOVE 23 TO Vstring-length of PICSTR.
           MOVE 'ZD Mmmmmmmmmmmmmmz YY'
```

```
               TO Vstring-text of PICSTR.
*************************************************
** Call CEECBLDY to convert input date to a    **
** COBOL integer date                          **
*************************************************
     CALL 'CEECBLDY' USING CHRDATE, PICSTR,
                           INTEGER, FC.
*************************************************
** If CEECBLDY runs successfully, then compute **
**     the date of the 90th day after the      **
**     input date using Intrinsic Functions    **
*************************************************
     IF CEE000 of FC  THEN
         COMPUTE INTEGER = INTEGER + 90
         COMPUTE NEWDATE = FUNCTION
             DATE-OF-INTEGER (INTEGER)
         DISPLAY NEWDATE
             ' is Lilian day: ' INTEGER
     ELSE
         DISPLAY 'CEEBLDY failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.
*
     GOBACK.
```

RELATED REFERENCES
"Picture character terms and strings" on page 562

CEEDATE: convert Lilian date to character format

CEEDATE converts a number that represents a Lilian date to a date written in character format. The output is a character string, such as 2008/04/23.

CALL CEEDATE syntax

```
►►──CALL──"CEEDATE"──USING──input_Lilian_date,──picture_string,──────────────►

►─output_char_date,──fc.──────────────────────────────────────────────────────►◄
```

input_Lilian_date **(input)**
A 32-bit integer that represents the Lilian date. The Lilian date is the number of days since 14 October 1582. For example, 16 May 1988 is Lilian day number 148138. The valid range of Lilian dates is 1 to 3,074,324 (15 October 1582 to 31 December 9999).

picture_string **(input)**
A halfword length-prefixed character string that represents the desired format of *output_char_date*, for example MM/DD/YY. Each character in *picture_string* represents a character in *output_char_date*. If delimiters such as the slash (/) appear in the picture string, they are copied as is to *output_char_date*.

If *picture_string* includes a Japanese Era symbol <JJJJ>, the YY position in *output_char_date* is replaced by the year number within the Japanese Era. For example, the year 1988 equals the Japanese year 63 in the Showa era.

output_char_date **(output)**

A fixed-length 80-character string that is the result of converting *input_Lilian_date* to the format specified by *picture_string*. If *input_Lilian_date* is invalid, *output_char_date* is set to all blanks and CEEDATE terminates with a non-CEE000 symbolic feedback code.

fc **(output)**

A 12-byte feedback code (optional) that indicates the result of this service.

Table 79. **CEEDATE symbolic conditions**

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	--	The service completed successfully.
CEE2EG	3	2512	The Lilian date value passed in a call to CEEDATE or CEEDYWK was not within the supported range.
CEE2EM	3	2518	An invalid picture string was specified in a call to a date/time service.
CEE2EQ	3	2522	An era (<JJJJ>, <CCCC>, or <CCCCCCCC>) was used in a picture string passed to CEEDATE, but the Lilian date value was not within the supported range. The era could not be determined.
CEE2EU	2	2526	The date string returned by CEEDATE was truncated.
CEE2F6	1	2534	Insufficient field width was specified for a month or weekday name in a call to CEEDATE or CEEDATM. Output set to blanks.

Usage note: The inverse of CEEDATE is CEEDAYS, which converts character dates to the Lilian format.

Example

```
CBL LIB
      ************************************************
      **                                            **
      ** Function: CEEDATE - convert Lilian date to **
      **                     character format       **
      **                                            **
      ** In this example, a call is made to CEEDATE **
      ** to convert a Lilian date (the number of    **
      ** days since 14 October 1582) to a character **
      ** format (such as 6/22/98). The result is    **
      ** displayed.  The Lilian date is obtained    **
      ** via a call to CEEDAYS.                     **
      **                                            **
      ************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLDATE.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  LILIAN                  PIC S9(9) BINARY.
       01  CHRDATE                 PIC X(80).
       01  IN-DATE.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char    PIC X
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                               of IN-DATE.
```

```
 01  PICSTR.
     02  Vstring-length      PIC S9(4) BINARY.
     02  Vstring-text.
         03  Vstring-char    PIC X
                     OCCURS 0 TO 256 TIMES
                     DEPENDING ON Vstring-length
                        of PICSTR.
 01  FC.
     02  Condition-Token-Value.
     COPY  CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity    PIC S9(4) COMP.
             04  Msg-No      PIC S9(4) COMP.
         03  Case-2-Condition-ID
                   REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) COMP.
             04  Cause-Code  PIC S9(4) COMP.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info            PIC S9(9) COMP.
*
 PROCEDURE DIVISION.
 PARA-CBLDAYS.
*************************************************
** Call CEEDAYS to convert date of 6/2/98 to   **
**     Lilian representation                   **
*************************************************
     MOVE 6 TO Vstring-length of IN-DATE.
     MOVE '6/2/98' TO Vstring-text of IN-DATE(1:6).
     MOVE 8 TO Vstring-length of PICSTR.
     MOVE 'MM/DD/YY' TO Vstring-text of PICSTR(1:8).
     CALL 'CEEDAYS' USING IN-DATE, PICSTR,
                          LILIAN, FC.

*************************************************
** If CEEDAYS runs successfully, display result**
*************************************************
     IF  CEE000 of FC  THEN
         DISPLAY Vstring-text of IN-DATE
             ' is Lilian day: ' LILIAN
     ELSE
         DISPLAY 'CEEDAYS failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

*************************************************
** Specify picture string that describes the   **
**  desired format of the output from CEEDATE, **
**  and the picture string's length.           **
*************************************************
     MOVE 23 TO Vstring-length OF PICSTR.
     MOVE 'ZD Mmmmmmmmmmmmmmz YYYY' TO
                 Vstring-text OF PICSTR(1:23).

*************************************************
** Call CEEDATE to convert the Lilian date     **
**     to  a picture string.                   **
*************************************************
     CALL 'CEEDATE' USING LILIAN, PICSTR,
                          CHRDATE, FC.

*************************************************
** If CEEDATE runs successfully, display result**
*************************************************
     IF CEE000 of FC  THEN
         DISPLAY 'Input Lilian date of ' LILIAN
```

```
        ' corresponds to:  ' CHRDATE
ELSE
    DISPLAY 'CEEDATE failed with msg '
        Msg-No of FC UPON CONSOLE
    STOP RUN
END-IF.

GOBACK.
```

The following table shows the sample output from CEEDATE.

input_Lilian_date	picture_string	output_char_date

148138	`YY`
	`YYMM`
	`YY-MM`
	`YYMMDD`
	`YYYYMMDD`
	`YYYY-MM-DD`
	`YYYY-ZM-ZD`
	`<JJJJ> YY.MM.DD`
		98
		9805
		98-05
		980516
		19980516
		1998-05-16
		1998-5-16
		Showa 63.05.16 (in a DBCS string)

148139	`MM`
	`MMDD`
	`MM/DD`
	`MMDDYY`
	`MM/DD/YYYY`
	`ZM/DD/YYYY`
		05
		0517
		05/17
		051798
		05/17/1998
		5/17/1998

148140	`DD`
	`DDMM`
	`DDMMYY`
	`DD.MM.YY`
	`DD.MM.YYYY`
	`DD Mmm YYYY`
		18
		1805
		180598
		18.05.98
		18.05.1998
		18 May 1998

148141	`DDD`
	`YYDDD`
	`YY.DDD`
	`YYYY.DDD`
		140
		98140
		98.140
		1998.140

148142	`YY/MM/DD HH:MI:SS.99`
	`YYYY/ZM/ZD ZH:MI AP`
		98/05/20 00:00:00.00
		1998/5/20 0:00 AM

148143	`WWW., MMM DD, YYYY`
	`Www., Mmm DD, YYYY`
	`Wwwwwwwwww,`
	`Mmmmmmmmmm DD, YYYY`
	`Wwwwwwwwwz,`
	`Mmmmmmmmmz DD, YYYY`
		SAT., MAY 21, 1998
		Sat., May 21, 1998
		Saturday,
		May 21, 1998
		Saturday,
		May 21, 1998

"Example: date-and-time picture strings" on page 564

RELATED REFERENCES
"Picture character terms and strings" on page 562

CEEDATM: convert seconds to character timestamp

CEEDATM converts a number that represents the number of seconds since 00:00:00 14 October 1582 to a character string. The output is a character string timestamp such as `1988/07/26 20:37:00`.

CALL CEEDATM syntax

```
►►──CALL──"CEEDATM"──USING──input_seconds,──picture_string,──────────────►

►──output_timestamp,──fc.──────────────────────────────────────────────►◄
```

input_seconds **(input)**
A 64-bit long floating-point number that represents the number of seconds since 00:00:00 on 14 October 1582, not counting leap seconds.

For example, 00:00:01 on 15 October 1582 is second number 86,401 (24*60*60 + 01). The valid range of *input_seconds* is 86,400 to 265,621,679,999.999 (23:59:59.999 31 December 9999).

picture_string **(input)**
A halfword length-prefixed character string that represents the desired format of *output_timestamp*, for example, `MM/DD/YY HH:MI AP`.

Each character in the *picture_string* represents a character in *output_timestamp*. If delimiters such as a slash (/) appear in the picture string, they are copied as is to *output_timestamp*.

If *picture_string* includes the Japanese Era symbol `<JJJJ>`, the `YY` position in *output_timestamp* represents the year within Japanese Era.

output_timestamp **(output)**
A fixed-length 80-character string that is the result of converting *input_seconds* to the format specified by *picture_string*.

If necessary, the output is truncated to the length of *output_timestamp*.

If *input_seconds* is invalid, *output_timestamp* is set to all blanks and CEEDATM terminates with a non-CEE000 symbolic feedback code.

fc **(output)**
A 12-byte feedback code (optional) that indicates the result of this service.

Table 80. **CEEDATM symbolic conditions**

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	--	The service completed successfully.
CEE2E9	3	2505	The input_seconds value in a call to CEEDATM or CEESECI was not within the supported range.
CEE2EA	3	2506	An era (<JJJJ>, <CCCC>, or <CCCCCCCC>) was used in a picture string passed to CEEDATM, but the input number-of-seconds value was not within the supported range. The era could not be determined.

Table 80. **CEEDATM symbolic conditions** *(continued)*

Symbolic feedback code	Severity	Message number	Message text
CEE2EM	3	2518	An invalid picture string was specified in a call to a date or time service.
CEE2EV	2	2527	The timestamp string returned by CEEDATM was truncated.
CEE2F6	1	2534	Insufficient field width was specified for a month or weekday name in a call to CEEDATE or CEEDATM. Output set to blanks.

Usage note: The inverse of CEEDATM is CEESECS, which converts a timestamp to number of seconds.

Example

```
CBL LIB
      *************************************************
      **                                             **
      ** Function: CEEDATM - convert seconds to      **
      **                     character timestamp     **
      **                                             **
      ** In this example, a call is made to CEEDATM  **
      ** to convert a date represented in Lilian     **
      ** seconds (the number of seconds since        **
      ** 00:00:00 14 October 1582) to a character    **
      ** format (such as 06/02/88 10:23:45). The     **
      ** result is displayed.                        **
      **                                             **
      *************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLDATM.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  DEST          PIC S9(9) BINARY VALUE 2.
       01  SECONDS                 COMP-2.
       01  IN-DATE.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char    PIC X
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                               of IN-DATE.
       01  PICSTR.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char    PIC X
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                              of PICSTR.
       01  TIMESTP                 PIC X(80).
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) COMP.
                   04  Msg-No      PIC S9(4) COMP.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) COMP.
                   04  Cause-Code  PIC S9(4) COMP.
```

```
        03  Case-Sev-Ctl    PIC X.
        03  Facility-ID     PIC XXX.
    02  I-S-Info            PIC S9(9) COMP.
*
 PROCEDURE DIVISION.
 PARA-CBLDATM.
**************************************************
** Call CEESECS to convert timestamp of 6/2/88 **
**     at 10:23:45 AM to Lilian representation **
**************************************************
     MOVE 20 TO Vstring-length of IN-DATE.
     MOVE '06/02/88 10:23:45 AM'
         TO Vstring-text of IN-DATE.
     MOVE 20 TO Vstring-length of PICSTR.
     MOVE 'MM/DD/YY HH:MI:SS AP'
         TO Vstring-text of PICSTR.
     CALL 'CEESECS' USING IN-DATE, PICSTR,
                          SECONDS, FC.

**************************************************
** If CEESECS runs successfully, display result**
**************************************************
     IF  CEE000 of FC  THEN
         DISPLAY Vstring-text of IN-DATE
             ' is Lilian second:  ' SECONDS
     ELSE
         DISPLAY 'CEESECS failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

**************************************************
** Specify desired format of the output.       **
**************************************************
     MOVE 35 TO Vstring-length OF PICSTR.
     MOVE 'ZD Mmmmmmmmmmmmmmz YYYY at HH:MI:SS'
             TO Vstring-text OF PICSTR.

**************************************************
** Call CEEDATM to convert Lilian seconds to   **
**     a character timestamp                   **
**************************************************
     CALL 'CEEDATM' USING SECONDS, PICSTR,
                          TIMESTP, FC.

**************************************************
** If CEEDATM runs successfully, display result**
**************************************************
     IF CEE000 of FC  THEN
         DISPLAY 'Input seconds of ' SECONDS
             ' corresponds to: ' TIMESTP
     ELSE
         DISPLAY 'CEEDATM failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

     GOBACK.
```

The following table shows the sample output of CEEDATM.

input_seconds	picture_string	output_timestamp
12,799,191,601.000	YYMMDD HH:MI:SS YY-MM-DD YYMMDDHHMISS YY-MM-DD HH:MI:SS YYYY-MM-DD HH:MI:SS AP	880516 19:00:01 88-05-16 880516190001 88-05-16 19:00:01 1988-05-16 07:00:01 PM
12,799,191,661.986	DD Mmm YY DD MMM YY HH:MM WWW, MMM DD, YYYY ZH:MI AP Wwwwwwwwwz, ZM/ZD/YY HH:MI:SS.99	16 May 88 16 MAY 88 19:01 MON, MAY 16, 1988 7:01 PM Monday, 5/16/88 19:01:01.98
12,799,191,662.009	YYYY YY Y MM ZM RRRR MMM Mmm Mmmmmmmmmm Mmmmmmmmmz DD ZD DDD HH ZH MI SS 99 999 AP WWW Www Wwwwwwwwww Wwwwwwwwwz	1988 88 8 05 5 V MAY May May May 16 16 137 19 19 01 02 00 009 PM MON Mon Monday Monday

"Example: date-and-time picture strings" on page 564

RELATED REFERENCES
"Picture character terms and strings" on page 562

CEEDAYS: convert date to Lilian format

CEEDAYS converts a string that represents a date into a Lilian format, which represents a date as the number of days from the beginning of the Gregorian calendar (Friday, 14 October, 1582).

Do not use CEEDAYS in combination with COBOL intrinsic functions. Use CEECBLDY for programs that use intrinsic functions.

CALL CEEDAYS syntax

```
►►──CALL──"CEEDAYS"──USING──input_char_date,──picture_string,──────────────►

►─output_Lilian_date,──fc.──────────────────────────────────────────────────►◄
```

input_char_date **(input)**
A halfword length-prefixed character string that represents a date or timestamp, in a format conforming to that specified by *picture_string*.

The character string must contain between 5 and 255 characters, inclusive. *input_char_date* can contain leading or trailing blanks. Parsing for a date begins with the first nonblank character (unless the picture string itself contains leading blanks, in which case CEEDAYS skips exactly that many positions before parsing begins).

After parsing a valid date, as determined by the format of the date specified in *picture_string*, CEEDAYS ignores all remaining characters. Valid dates range between and include 15 October 1582 to 31 December 9999.

picture_string **(input)**
A halfword length-prefixed character string, indicating the format of the date specified in *input_char_date*.

Each character in the *picture_string* corresponds to a character in *input_char_date*. For example, if you specify MMDDYY as the *picture_string*, CEEDAYS reads an *input_char_date* of 060288 as 02 June 1988.

If delimiters such as a slash (/) appear in the picture string, leading zeros can be omitted. For example, the following calls to CEEDAYS each assign the same value, 148155 (02 June 1988), to *lildate*:

```
CALL CEEDAYS USING '6/2/88'  , 'MM/DD/YY', lildate, fc.
CALL CEEDAYS USING '06/02/88', 'MM/DD/YY', lildate, fc.
CALL CEEDAYS USING '060288'  , 'MMDDYY'  , lildate, fc.
CALL CEEDAYS USING '88154'   , 'YYDDD'   , lildate, fc.
```

Whenever characters such as colons or slashes are included in the *picture_string* (such as `HH:MI:SS YY/MM/DD`), they count as placeholders but are otherwise ignored.

If *picture_string* includes a Japanese Era symbol <JJJJ>, the `YY` position in *input_char_date* is replaced by the year number within the Japanese Era. For example, the year 1988 equals the Japanese year 63 in the Showa era.

output_Lilian_date **(output)**
A 32-bit binary integer that represents the Lilian date, the number of days since 14 October 1582. For example, 16 May 1988 is day number 148138.

If *input_char_date* does not contain a valid date, *output_Lilian_date* is set to 0 and CEEDAYS terminates with a non-CEE000 symbolic feedback code.

Date calculations are performed easily on the *output_Lilian_date*, because it is an integer. Leap year and end-of-year anomalies do not affect the calculations.

fc **(output)**
A 12-byte feedback code (optional) that indicates the result of this service.

Table 81. **CEEDAYS symbolic conditions**

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	--	The service completed successfully.
CEE2EB	3	2507	Insufficient data was passed to CEEDAYS or CEESECS. The Lilian value was not calculated.
CEE2EC	3	2508	The date value passed to CEEDAYS or CEESECS was invalid.
CEE2ED	3	2509	The era passed to CEEDAYS or CEESECS was not recognized.
CEE2EH	3	2513	The input date passed in a CEEISEC, CEEDAYS, or CEESECS call was not within the supported range.
CEE2EL	3	2517	The month value in a CEEISEC call was not recognized.
CEE2EM	3	2518	An invalid picture string was specified in a call to a date/time service.
CEE2EO	3	2520	CEEDAYS detected nonnumeric data in a numeric field, or the date string did not match the picture string.
CEE2EP	3	2521	The <JJJJ>, <CCCC>, or <CCCCCCCC> year-within-era value passed to CEEDAYS or CEESECS was zero.

Usage notes

- The inverse of CEEDAYS is CEEDATE, which converts *output_Lilian_date* from Lilian format to character format.
- To perform calculations on dates earlier than 15 October 1582, add 4000 to the year in each date, convert the dates to Lilian, then do the calculation. If the result of the calculation is a date, as opposed to a number of days, convert the result to a date string and subtract 4000 from the year.
- By default, two-digit years lie within the 100-year range that starts 80 years before the system date. Thus in 2008, all two-digit years represent dates between 1928 and 2027, inclusive. You can change the default range by using the callable service CEESCEN.
- You can easily perform date calculations on the *output_Lilian_date*, because it is an integer. Leap-year and end-of-year anomalies are avoided.

Example

```
CBL LIB
      *******************************************
      **                                       **
      ** Function: CEEDAYS - convert date to   **
      **                     Lilian format     **
      **                                       **
      *******************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLDAYS.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  CHRDATE.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char    PIC X
```

```
                    OCCURS 0 TO 256 TIMES
                    DEPENDING ON Vstring-length
                       of CHRDATE.
 01  PICSTR.
     02  Vstring-length      PIC S9(4) BINARY.
     02  Vstring-text.
         03  Vstring-char    PIC X
                    OCCURS 0 TO 256 TIMES
                    DEPENDING ON Vstring-length
                       of PICSTR.
 01  LILIAN                  PIC S9(9) BINARY.
 01  FC.
     02  Condition-Token-Value.
     COPY  CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity    PIC S9(4) COMP.
             04  Msg-No      PIC S9(4) COMP.
         03  Case-2-Condition-ID
                  REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) COMP.
             04  Cause-Code  PIC S9(4) COMP.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info            PIC S9(9) COMP.
*
 PROCEDURE DIVISION.
 PARA-CBLDAYS.
*************************************************
** Specify input date and length               **
*************************************************
     MOVE 16 TO Vstring-length of CHRDATE.
     MOVE '1 January 2005'
         TO Vstring-text of CHRDATE.

*************************************************
** Specify a picture string that describes     **
** input date, and the picture string's length.**
*************************************************
     MOVE 25 TO Vstring-length of PICSTR.
     MOVE 'ZD Mmmmmmmmmmmmmmz YYYY'
             TO Vstring-text of PICSTR.

*************************************************
** Call CEEDAYS to convert input date to a     **
** Lilian date                                 **
*************************************************
     CALL 'CEEDAYS' USING CHRDATE, PICSTR,
                          LILIAN, FC.

*************************************************
** If CEEDAYS runs successfully, display result**
*************************************************
     IF  CEE000 of FC  THEN
         DISPLAY Vstring-text of CHRDATE
             ' is Lilian day: ' LILIAN
     ELSE
         DISPLAY 'CEEDAYS failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

     GOBACK.
```

"Example: date-and-time picture strings" on page 564

RELATED REFERENCES
"Picture character terms and strings" on page 562

CEEDYWK: calculate day of week from Lilian date

CEEDYWK calculates the day of the week on which a Lilian date falls as a number between 1 and 7.

The number returned by CEEDYWK is useful for end-of-week calculations.

CALL CEEDYWK syntax

```
►►──CALL──"CEEDYWK"──USING──input_Lilian_date,──output_day_no,──fc.──────────►◄
```

***input_Lilian_date* (input)**
: A 32-bit binary integer that represents the Lilian date, the number of days since 14 October 1582.

  For example, 16 May 1988 is day number 148138. The valid range of *input_Lilian_date* is between 1 and 3,074,324 (15 October 1582 and 31 December 9999).

***output_day_no* (output)**
: A 32-bit binary integer that represents *input_Lilian_date*'s day-of-week: 1 equals Sunday, 2 equals Monday, . . ., 7 equals Saturday.

  If *input_Lilian_date* is invalid, *output_day_no* is set to 0 and CEEDYWK terminates with a non-CEE000 symbolic feedback code.

***fc* (output)**
: A 12-byte feedback code (optional) that indicates the result of this service.

Table 82. **CEEDYWK symbolic conditions**

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	--	The service completed successfully.
CEE2EG	3	2512	The Lilian date value passed in a call to CEEDATE or CEEDYWK was not within the supported range.

Example

```
CBL LIB
      ************************************************
      **                                            **
      ** Function: Call CEEDYWK to calculate the    **
      **           day of the week from Lilian date **
      **                                            **
      ** In this example, a call is made to CEEDYWK **
      ** to return the day of the week on which a   **
      ** Lilian date falls. (A Lilian date is the   **
      ** number of days since 14 October 1582)      **
      **                                            **
      ************************************************
       IDENTIFICATION DIVISION.
```

```
 PROGRAM-ID. CBLDYWK.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  LILIAN                 PIC S9(9) BINARY.
 01  DAYNUM                 PIC S9(9) BINARY.
 01  IN-DATE.
     02  Vstring-length      PIC S9(4) BINARY.
     02  Vstring-text.
         03  Vstring-char        PIC X,
                     OCCURS 0 TO 256 TIMES
                     DEPENDING ON Vstring-length
                         of IN-DATE.
 01  PICSTR.
     02  Vstring-length      PIC S9(4) BINARY.
     02  Vstring-text.
         03  Vstring-char        PIC X,
                     OCCURS 0 TO 256 TIMES
                     DEPENDING ON Vstring-length
                         of PICSTR.
 01  FC.
     02  Condition-Token-Value.
     COPY  CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity    PIC S9(4) COMP.
             04  Msg-No      PIC S9(4) COMP.
         03  Case-2-Condition-ID
                   REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) COMP.
             04  Cause-Code  PIC S9(4) COMP.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info            PIC S9(9) COMP.

 PROCEDURE DIVISION.
 PARA-CBLDAYS.
** Call CEEDAYS to convert date of 6/2/88 to
**     Lilian representation
     MOVE 6 TO Vstring-length of IN-DATE.
     MOVE '6/2/88' TO Vstring-text of IN-DATE(1:6).
     MOVE 8 TO Vstring-length of PICSTR.
     MOVE 'MM/DD/YY' TO Vstring-text of PICSTR(1:8).
     CALL 'CEEDAYS' USING IN-DATE, PICSTR,
         LILIAN, FC.

** If CEEDAYS runs successfully, display result.
     IF  CEE000 of FC  THEN
         DISPLAY Vstring-text of IN-DATE
             ' is Lilian day: ' LILIAN
     ELSE
         DISPLAY 'CEEDAYS failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

 PARA-CBLDYWK.

** Call CEEDYWK to return the day of the week on
** which the Lilian date falls
     CALL 'CEEDYWK' USING LILIAN , DAYNUM , FC.

** If CEEDYWK runs successfully, print results
     IF CEE000 of FC  THEN
         DISPLAY 'Lilian day ' LILIAN
             ' falls on day ' DAYNUM
             ' of the week, which is a:'
** Select DAYNUM to display the name of the day
**     of the week.
```

```
        EVALUATE DAYNUM
          WHEN 1
            DISPLAY 'Sunday.'
          WHEN 2
            DISPLAY 'Monday.'
          WHEN 3
            DISPLAY 'Tuesday'
          WHEN 4
            DISPLAY 'Wednesday.'
          WHEN 5
            DISPLAY 'Thursday.'
          WHEN 6
            DISPLAY 'Friday.'
          WHEN 7
            DISPLAY 'Saturday.'
        END-EVALUATE
    ELSE
        DISPLAY 'CEEDYWK failed with msg '
            Msg-No of FC UPON CONSOLE
        STOP RUN
    END-IF.

    GOBACK.
```

CEEGMT: get current Greenwich Mean Time

CEEGMT returns the current Greenwich Mean Time (GMT) as both a Lilian date and as the number of seconds since 00:00:00 14 October 1582. The returned values are compatible with those generated and used by the other date and time callable services.

CALL CEEGMT syntax

```
►►──CALL──"CEEGMT"──USING──output_GMT_Lilian,──output_GMT_seconds,──fc.──────────►◄
```

***output_GMT_Lilian* (output)**
A 32-bit binary integer that represents the current date in Greenwich, England, in the Lilian format (the number of days since 14 October 1582).

For example, 16 May 1988 is day number 148138. If GMT is not available from the system, *output_GMT_Lilian* is set to 0 and CEEGMT terminates with a non-CEE000 symbolic feedback code.

***output_GMT_seconds* (output)**
A 64-bit long floating-point number that represents the current date and time in Greenwich, England, as the number of seconds since 00:00:00 on 14 October 1582, not counting leap seconds.

For example, 00:00:01 on 15 October 1582 is second number 86,401 (24*60*60 + 01). 19:00:01.078 on 16 May 1988 is second number 12,799,191,601.078. If GMT is not available from the system, *output_GMT_seconds* is set to 0 and CEEGMT terminates with a non-CEE000 symbolic feedback code.

***fc* (output)**
A 12-byte feedback code (optional) that indicates the result of this service.

Table 83. **CEEGMT symbolic conditions**

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	--	The service completed successfully.
CEE2E6	3	2502	The UTC/GMT was not available from the system.

Usage notes

- CEEDATE converts *output_GMT_Lilian* to a character date, and CEEDATM converts *output_GMT_seconds* to a character timestamp.
- In order for the results of this service to be meaningful, your system's clock must be set to the local time and the environment variable TZ must be set correctly.
- The values returned by CEEGMT are handy for elapsed time calculations. For example, you can calculate the time elapsed between two calls to CEEGMT by calculating the differences between the returned values.
- CEEUTC is identical to this service.

Example

```
CBL LIB
      ************************************************
      **                                            **
      ** Function: Call CEEGMT to get current       **
      **           Greenwich Mean Time              **
      **                                            **
      ** In this example, a call is made to CEEGMT   **
      ** to return the current GMT as a Lilian date  **
      ** and as Lilian seconds. The results are      **
      ** displayed.                                  **
      **                                            **
      ************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. IGZTGMT.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  LILIAN                  PIC S9(9) BINARY.
       01  SECS                    COMP-2.
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) COMP.
                   04  Msg-No      PIC S9(4) COMP.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) COMP.
                   04  Cause-Code  PIC S9(4) COMP.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info            PIC S9(9) COMP.
       PROCEDURE DIVISION.
       PARA-CBLGMT.
           CALL 'CEEGMT' USING LILIAN , SECS , FC.

           IF CEE000 of FC  THEN
               DISPLAY 'The current GMT is also '
                   'known as Lilian day: ' LILIAN
               DISPLAY 'The current GMT in Lilian '
                   'seconds is: ' SECS
           ELSE
```

```
    DISPLAY 'CEEGMT failed with msg '
        Msg-No of FC UPON CONSOLE
    STOP RUN
END-IF.

GOBACK.
```

RELATED TASKS
"Setting environment variables" on page 209

CEEGMTO: get offset from Greenwich Mean Time to local time

CEEGMTO returns values to the calling routine that represent the difference between the local system time and Greenwich Mean Time (GMT).

CALL CEEGMTO syntax

```
►►──CALL──"CEEGMTO"──USING──offset_hours,──offset_minutes,──offset_seconds,────►

►──fc.──────────────────────────────────────────────────────────────────────►◄
```

offset_hours **(output)**
: A 32-bit binary integer that represents the offset from GMT to local time, in hours.

  For example, for Pacific Standard Time, *offset_hours* equals -8.

  The range of *offset_hours* is -12 to +13 (+13 = Daylight Savings Time in the +12 time zone).

  If local time offset is not available, *offset_hours* equals 0 and CEEGMTO terminates with a non-CEE000 symbolic feedback code.

offset_minutes **(output)**
: A 32-bit binary integer that represents the number of additional minutes that local time is ahead of or behind GMT.

  The range of *offset_minutes* is 0 to 59.

  If the local time offset is not available, *offset_minutes* equals 0 and CEEGMTO terminates with a non-CEE000 symbolic feedback code.

offset_seconds **(output)**
: A 64-bit long floating-point number that represents the offset from GMT to local time, in seconds.

  For example, Pacific Standard Time is eight hours behind GMT. If local time is in the Pacific time zone during standard time, CEEGMTO would return -28,800 (-8 * 60 * 60). The range of *offset_seconds* is -43,200 to +46,800. *offset_seconds* can be used with CEEGMT to calculate local date and time.

  If the local time offset is not available from the system, *offset_seconds* is set to 0 and CEEGMTO terminates with a non-CEE000 symbolic feedback code.

fc **(output)**
: A 12-byte feedback code (optional) that indicates the result of this service.

Table 84. **CEEGMTO symbolic conditions**

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	--	The service completed successfully.
CEE2E7	3	2503	The offset from UTC/GMT to local time was not available from the system.

Usage notes

- CEEDATM converts *offset_seconds* to a character timestamp.
- In order for the results of this service to be meaningful, your system clock must be set to the local time, and the environment variable TZ must be set correctly.

Example

```
CBL LIB
      *************************************************
      **                                             **
      ** Function:  Call CEEGMTO to get offset from  **
      **            Greenwich Mean Time to local     **
      **            time                             **
      **                                             **
      ** In this example, a call is made to CEEGMTO  **
      ** to return the offset from GMT to local time **
      ** as separate binary integers representing    **
      ** offset hours, minutes, and seconds. The     **
      ** results are displayed.                      **
      **                                             **
      *************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. IGZTGMTO.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  HOURS                  PIC S9(9) BINARY.
       01  MINUTES                PIC S9(9) BINARY.
       01  SECONDS COMP-2.
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) COMP.
                   04  Msg-No      PIC S9(4) COMP.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) COMP.
                   04  Cause-Code  PIC S9(4) COMP.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info           PIC S9(9) COMP.
       PROCEDURE DIVISION.
       PARA-CBLGMTO.
           CALL 'CEEGMTO' USING HOURS , MINUTES ,
               SECONDS , FC.

           IF CEE000 of FC  THEN
               DISPLAY 'Local time differs from GMT '
                   'by: ' HOURS ' hours, '
                   MINUTES ' minutes, OR  '
                   SECONDS ' seconds. '
           ELSE
               DISPLAY 'CEEGMTO failed with msg '
```

```
            Msg-No of FC UPON CONSOLE
        STOP RUN
    END-IF.

    GOBACK.
```

RELATED TASKS
"Setting environment variables" on page 209

RELATED REFERENCES
"CEEGMT: get current Greenwich Mean Time" on page 617
"Runtime environment variables" on page 211

CEEISEC: convert integers to seconds

CEEISEC converts binary integers that represent year, month, day, hour, minute, second, and millisecond to a number that represents the number of seconds since 00:00:00 14 October 1582.

CALL CEEISEC syntax

```
►►──CALL──"CEEISEC"──USING──input_year,──input_months,──input_day,──────────►

►──input_hours,──input_minutes,──input_seconds,──input_milliseconds,────────►

►──output_seconds,──fc.───────────────────────────────────────────────────►◄
```

input_year **(input)**
: A 32-bit binary integer that represents the year.

  The range of valid values for *input_year* is 1582 to 9999, inclusive.

input_month **(input)**
: A 32-bit binary integer that represents the month.

  The range of valid values for *input_month* is 1 to 12.

input_day **(input)**
: A 32-bit binary integer that represents the day.

  The range of valid values for *input_day* is 1 to 31.

input_hours **(input)**
: A 32-bit binary integer that represents the hours.

  The range of valid values for *input_hours* is 0 to 23.

input_minutes **(input)**
: A 32-bit binary integer that represents the minutes.

  The range of valid values for *input_minutes* is 0 to 59.

input_seconds **(input)**
: A 32-bit binary integer that represents the seconds.

  The range of valid values for *input_seconds* is 0 to 59.

input_milliseconds **(input)**
A 32-bit binary integer that represents milliseconds.

The range of valid values for *input_milliseconds* is 0 to 999.

output_seconds **(output)**
A 64-bit long floating-point number that represents the number of seconds since 00:00:00 on 14 October 1582, not counting leap seconds.

For example, 00:00:01 on 15 October 1582 is second number 86,401 (24*60*60 + 01). The valid range of *output_seconds* is 86,400 to 265,621,679,999.999 (23:59:59.999 31 December 9999).

If any input values are invalid, *output_seconds* is set to zero.

To convert *output_seconds* to a Lilian day number, divide *output_seconds* by 86,400 (the number of seconds in a day).

fc **(output)**
A 12-byte feedback code (optional) that indicates the result of this service.

Table 85. **CEEISEC symbolic conditions**

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	--	The service completed successfully.
CEE2EE	3	2510	The hours value in a call to CEEISEC or CEESECS was not recognized.
CEE2EF	3	2511	The day parameter passed in a CEEISEC call was invalid for year and month specified.
CEE2EH	3	2513	The input date passed in a CEEISEC, CEEDAYS, or CEESECS call was not within the supported range.
CEE2EI	3	2514	The year value passed in a CEEISEC call was not within the supported range.
CEE2EJ	3	2515	The milliseconds value in a CEEISEC call was not recognized.
CEE2EK	3	2516	The minutes value in a CEEISEC call was not recognized.
CEE2EL	3	2517	The month value in a CEEISEC call was not recognized.
CEE2EN	3	2519	The seconds value in a CEEISEC call was not recognized.

Usage note: The inverse of CEEISEC is CEESECI, which converts number of seconds to integer year, month, day, hour, minute, second, and millisecond.

Example

```
CBL LIB
      *************************************************
      **                                             **
      ** Function: Call CEEISEC to convert integers  **
      **           to seconds                        **
      **                                             **
      *************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLISEC.
       DATA DIVISION.
```

```
 WORKING-STORAGE SECTION.
 01  YEAR                    PIC S9(9) BINARY.
 01  MONTH                   PIC S9(9) BINARY.
 01  DAYS                    PIC S9(9) BINARY.
 01  HOURS                   PIC S9(9) BINARY.
 01  MINUTES                 PIC S9(9) BINARY.
 01  SECONDS                 PIC S9(9) BINARY.
 01  MILLSEC                 PIC S9(9) BINARY.
 01  OUTSECS                 COMP-2.
 01  FC.
     02  Condition-Token-Value.
     COPY  CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity    PIC S9(4) COMP.
             04  Msg-No      PIC S9(4) COMP.
         03  Case-2-Condition-ID
                   REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) COMP.
             04  Cause-Code  PIC S9(4) COMP.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info            PIC S9(9) COMP.
 PROCEDURE DIVISION.
 PARA-CBLISEC.
*************************************************
** Specify seven binary integers representing  **
** the date and time as input to be converted  **
** to Lilian seconds                           **
*************************************************
     MOVE 2000 TO YEAR.
     MOVE 1 TO MONTH.
     MOVE 1 TO DAYS.
     MOVE 0 TO HOURS.
     MOVE 0 TO MINUTES.
     MOVE 0 TO SECONDS.
     MOVE 0 TO MILLSEC.
*************************************************
** Call CEEISEC to convert the integers        **
** to seconds                                  **
*************************************************
     CALL 'CEEISEC' USING YEAR, MONTH, DAYS,
                          HOURS, MINUTES, SECONDS,
                          MILLSEC, OUTSECS , FC.
*************************************************
** If CEEISEC runs successfully, display result**
*************************************************
     IF CEE000 of FC  THEN
         DISPLAY MONTH '/' DAYS '/' YEAR
           ' AT ' HOURS ':' MINUTES ':' SECONDS
           ' is equivalent to ' OUTSECS ' seconds'
     ELSE
         DISPLAY 'CEEISEC failed with msg '
           Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

     GOBACK.
```

CEELOCT: get current local date or time

CEELOCT returns the current local date or time as a Lilian date (the number of days since 14 October 1582), as Lilian seconds (the number of seconds since 00:00:00 14 October 1582), and as a Gregorian character string (YYYYMMDDHHMISS999).

These values are compatible with other date and time callable services and with existing intrinsic functions.

CEELOCT performs the same function as calling the CEEGMT, CEEGMTO, and CEEDATM services separately. Calling CEELOCT, however, is much faster.

CALL CEELOCT syntax

```
►►──CALL──"CEELOCT"──USING──output_Lilian,──output_seconds,──────────────────►

►─output_Gregorian,──fc.─────────────────────────────────────────────────────►◄
```

output_Lilian **(output)**

A 32-bit binary integer that represents the current local date in the Lilian format, that is, day 1 equals 15 October 1582, day 148,887 equals 4 June 1990.

If the local time is not available from the system, *output_Lilian* is set to 0 and CEELOCT terminates with a non-CEE000 symbolic feedback code.

output_seconds **(output)**

A 64-bit long floating-point number that represents the current local date and time as the number of seconds since 00:00:00 on 14 October 1582, not counting leap seconds. For example, 00:00:01 on 15 October 1582 is second number 86,401 (24*60*60 + 01). 19:00:01.078 on 4 June 1990 is second number 12,863,905,201.078.

If the local time is not available from the system, *output_seconds* is set to 0 and CEELOCT terminates with a non-CEE000 symbolic feedback code.

output_Gregorian **(output)**

A 17-byte fixed-length character string in the form YYYYMMDDHHMISS999 that represents local year, month, day, hour, minute, second, and millisecond.

If the format of *output_Gregorian* does not meet your needs, you can use the CEEDATM callable service to convert *output_seconds* to another format.

fc **(output)**

A 12-byte feedback code (optional) that indicates the result of this service.

Table 86. **CEELOCT symbolic conditions**

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	--	The service completed successfully.
CEE2F3	3	2531	The local time was not available from the system.

Usage notes

- You can use the CEEGMT callable service to determine Greenwich Mean Time (GMT).
- You can use the CEEGMTO callable service to obtain the offset from GMT to local time.

- The character value returned by CEELOCT is designed to match that produced by existing intrinsic functions. The numeric values returned can be used to simplify date calculations.

Example

```
CBL LIB
      ************************************************
      **                                            **
      ** Function: Call CEELOCT to get current      **
      **           local time                       **
      **                                            **
      ** In this example, a call is made to CEELOCT **
      ** to return the current local time in Lilian **
      ** days (the number of days since 14 October  **
      ** 1582), Lilian seconds (the number of       **
      ** seconds since 00:00:00 14 October 1582),   **
      ** and a Gregorian string (in the form        **
      ** YYYMMDDMISS999). The Gregorian character   **
      ** string is then displayed.                  **
      **                                            **
      ************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLLOCT.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  LILIAN                 PIC S9(9) BINARY.
       01  SECONDS                COMP-2.
       01  GREGORN                PIC X(17).
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) COMP.
                   04  Msg-No      PIC S9(4) COMP.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) COMP.
                   04  Cause-Code  PIC S9(4) COMP.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info           PIC S9(9) COMP.
       PROCEDURE DIVISION.
       PARA-CBLLOCT.
           CALL 'CEELOCT' USING LILIAN, SECONDS,
                                GREGORN, FC.
      ************************************************
      ** If CEELOCT runs successfully, display      **
      **     Gregorian character string             **
      ************************************************
           IF CEE000 of FC  THEN
               DISPLAY 'Local Time is ' GREGORN
           ELSE
               DISPLAY 'CEELOCT failed with msg '
                   Msg-No of FC UPON CONSOLE
               STOP RUN
           END-IF.

           GOBACK.
```

CEEQCEN: query the century window

CEEQCEN queries the century window, which is a two-digit year value.

When you want to change the century window, use CEEQCEN to get the setting and then use CEESCEN to save and restore the current setting.

CALL CEEQCEN syntax

```
►►──CALL──"CEEQCEN"──USING──century_start,──fc.──────────────────────►◄
```

century_start **(output)**
An integer between 0 and 100 that indicates the year on which the century window is based.

For example, if the date and time callable services default is in effect, all two-digit years lie within the 100-year window that starts 80 years before the system date. CEEQCEN then returns the value 80. For example, in the year 2008, 80 indicates that all two-digit years lie within the 100-year window between 1928 and 2027, inclusive.

fc **(output)**
A 12-byte feedback code (optional) that indicates the result of this service.

Table 87. **CEEQCEN symbolic conditions**

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	--	The service completed successfully.

Example

```
CBL LIB
      *************************************************
      **                                             **
      ** Function: Call CEEQCEN to query the         **
      **           date and time callable services   **
      **           century window                    **
      **                                             **
      ** In this example, CEEQCEN is called to query **
      ** the date at which the century window starts **
      ** The century window is the 100-year window   **
      ** within which the date and time callable     **
      ** services assume all two-digit years lie.    **
      **                                             **
      *************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLQCEN.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  STARTCW                 PIC S9(9) BINARY.
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) COMP.
                   04  Msg-No      PIC S9(4) COMP.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) COMP.
                   04  Cause-Code  PIC S9(4) COMP.
```

```
          03  Case-Sev-Ctl    PIC X.
          03  Facility-ID     PIC XXX.
      02  I-S-Info            PIC S9(9) COMP.
  PROCEDURE DIVISION.

  PARA-CBLQCEN.
*************************************************
** Call CEEQCEN to return the start of the     **
**     century window                          **
*************************************************

     CALL 'CEEQCEN' USING STARTCW, FC.
*************************************************
** CEEQCEN has no nonzero feedback codes to    **
**     check, so just display result.          **
*************************************************
     IF CEE000 of FC  THEN
         DISPLAY 'The start of the century '
             'window is: ' STARTCW
     ELSE
         DISPLAY 'CEEQCEN failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

     GOBACK.
```

CEESCEN: set the century window

CEESCEN sets the century window to a two-digit year value for use by other date and time callable services.

Use CEESCEN in conjunction with CEEDAYS or CEESECS when:

- You process date values that contain two-digit years (for example, in the YYMMDD format).
- The default century interval does not meet the requirements of a particular application.

To query the century window, use CEEQCEN.

CALL CEESCEN syntax

```
►►──CALL──"CEESCEN"──USING──century_start,──fc.──────────────────────►◄
```

century_start
: An integer between 0 and 100, which sets the century window.

  A value of 80, for example, places all two-digit years within the 100-year window that starts 80 years before the system date. In 2008, therefore, all two-digit years are assumed to represent dates between 1928 and 2027, inclusive.

fc **(output)**
: A 12-byte feedback code (optional) that indicates the result of this service.

Table 88. **CEESCEN symbolic conditions**

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	--	The service completed successfully.
CEE2E6	3	2502	The UTC/GMT was not available from the system.
CEE2F5	3	2533	The value passed to CEESCEN was not between 0 and 100.

Example

```
CBL LIB
      **************************************************
      **                                              **
      ** Function: Call CEESCEN to set the            **
      **           date and time callable services    **
      **           century window                     **
      **                                              **
      ** In this example, CEESCEN is called to change **
      ** the start of the century window to 30 years  **
      ** before the system date. CEEQCEN is then      **
      ** called to query that the change made.  A     **
      ** message that this has been done is then      **
      ** displayed.                                   **
      **                                              **
      **************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLSCEN.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  STARTCW                 PIC S9(9) BINARY.
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) COMP.
                   04  Msg-No      PIC S9(4) COMP.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) COMP.
                   04  Cause-Code  PIC S9(4) COMP.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info            PIC S9(9) COMP.
       PROCEDURE DIVISION.
       PARA-CBLSCEN.
      **************************************************
      ** Specify 30 as century start, and two-digit
      **     years will be assumed to lie in the
      **     100-year window starting 30 years before
      **     the system date.
      **************************************************
           MOVE 30 TO STARTCW.

      **************************************************
      ** Call CEESCEN to change the start of the century
      **     window.
      **************************************************
           CALL 'CEESCEN' USING STARTCW, FC.
           IF NOT CEE000 of FC  THEN
               DISPLAY 'CEESCEN failed with msg '
                   Msg-No of FC UPON CONSOLE
               STOP RUN
           END-IF.

       PARA-CBLQCEN.
      **************************************************
      ** Call CEEQCEN to return the start of the century
      **     window
      **************************************************
           CALL 'CEEQCEN' USING STARTCW, FC.

      **************************************************
      ** CEEQCEN has no nonzero feedback codes to
      **     check, so just display result.
      **************************************************
               DISPLAY 'The start of the century '
                   'window is: ' STARTCW
           GOBACK.
```

CEESECI: convert seconds to integers

CEESECI converts a number that represents the number of seconds since 00:00:00 14 October 1582 to binary integers that represent year, month, day, hour, minute, second, and millisecond.

Use CEESECI instead of CEEDATM when the output is needed in numeric format rather than in character format.

CALL CEESECI syntax

```
►►──CALL──"CEESECI"──USING──input_seconds,──output_year,──output_month,──────────►

►──output_day,──output_hours,──output_minutes,──output_seconds,──────────────────►

►──output_milliseconds,──fc.────────────────────────────────────────────────────►◄
```

input_seconds
: A 64-bit long floating-point number that represents the number of seconds since 00:00:00 on 14 October 1582, not counting leap seconds.

  For example, 00:00:01 on 15 October 1582 is second number 86,401 (24*60*60 + 01). The range of valid values for *input_seconds* is 86,400 to 265,621,679,999.999 (23:59:59.999 31 December 9999).

  If *input_seconds* is invalid, all output parameters except the feedback code are set to 0.

***output_year* (output)**
: A 32-bit binary integer that represents the year.

  The range of valid values for *output_year* is 1582 to 9999, inclusive.

***output_month* (output)**
: A 32-bit binary integer that represents the month.

  The range of valid values for *output_month* is 1 to 12.

***output_day* (output)**
: A 32-bit binary integer that represents the day.

The range of valid values for *output_day* is 1 to 31.

***output_hours* (output)**
A 32-bit binary integer that represents the hour.

The range of valid values for *output_hours* is 0 to 23.

***output_minutes* (output)**
A 32-bit binary integer that represents the minutes.

The range of valid values for *output_minutes* is 0 to 59.

***output_seconds* (output)**
A 32-bit binary integer that represents the seconds.

The range of valid values for *output_seconds* is 0 to 59.

***output_milliseconds* (output)**
A 32-bit binary integer that represents milliseconds.

The range of valid values for *output_milliseconds* is 0 to 999.

***fc* (output)**
A 12-byte feedback code (optional) that indicates the result of this service.

Table 89. **CEESECI symbolic conditions**

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	--	The service completed successfully.
CEE2E9	3	2505	The input_seconds value in a call to CEEDATM or CEESECI was not within the supported range.

Usage notes

- The inverse of CEESECI is CEEISEC, which converts separate binary integers that represent year, month, day, hour, second, and millisecond to a number of seconds.
- If the input value is a Lilian date instead of seconds, multiply the Lilian date by 86,400 (number of seconds in a day), and pass the new value to CEESECI.

Example

```
CBL LIB
      ************************************************
      **                                            **
      ** Function: Call CEESECI to convert seconds   **
      **           to integers                      **
      **                                            **
      ** In this example a call is made to CEESECI   **
      ** to convert a number representing the number **
      ** of seconds since 00:00:00 14 October 1582   **
      ** to seven binary integers representing year, **
      ** month, day, hour, minute, second, and       **
      ** millisecond.  The results are displayed in  **
      ** this example.                              **
      **                                            **
      ************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLSECI.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  INSECS                 COMP-2.
```

```
 01  YEAR                    PIC S9(9) BINARY.
 01  MONTH                   PIC S9(9) BINARY.
 01  DAYS                    PIC S9(9) BINARY.
 01  HOURS                   PIC S9(9) BINARY.
 01  MINUTES                 PIC S9(9) BINARY.
 01  SECONDS                 PIC S9(9) BINARY.
 01  MILLSEC                 PIC S9(9) BINARY.
 01  IN-DATE.
     02  Vstring-length      PIC S9(4) BINARY.
     02  Vstring-text.
         03  Vstring-char        PIC X,
                     OCCURS 0 TO 256 TIMES
                     DEPENDING ON Vstring-length
                         of IN-DATE.
 01  PICSTR.
     02  Vstring-length      PIC S9(4) BINARY.
     02  Vstring-text.
         03  Vstring-char        PIC X,
                     OCCURS 0 TO 256 TIMES
                     DEPENDING ON Vstring-length
                        of PICSTR.
 01  FC.
     02  Condition-Token-Value.
     COPY  CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity    PIC S9(4) COMP.
             04  Msg-No      PIC S9(4) COMP.
         03  Case-2-Condition-ID
                   REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) COMP.
             04  Cause-Code  PIC S9(4) COMP.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info            PIC S9(9) COMP.
 PROCEDURE DIVISION.
 PARA-CBLSECS.
*************************************************
** Call CEESECS to convert timestamp of 6/2/88
**     at 10:23:45 AM to Lilian representation
*************************************************
     MOVE 20 TO Vstring-length of IN-DATE.
     MOVE '06/02/88 10:23:45 AM'
             TO Vstring-text of IN-DATE.
     MOVE 20 TO Vstring-length of PICSTR.
     MOVE 'MM/DD/YY HH:MI:SS AP'
             TO Vstring-text of PICSTR.
     CALL 'CEESECS' USING IN-DATE, PICSTR,
                          INSECS, FC.
     IF NOT CEE000 of FC  THEN
         DISPLAY 'CEESECS failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

 PARA-CBLSECI.
*************************************************
** Call CEESECI to convert seconds to integers
*************************************************
     CALL 'CEESECI' USING INSECS, YEAR, MONTH,
                          DAYS, HOURS,  MINUTES,
                          SECONDS, MILLSEC, FC.
*************************************************
** If CEESECI runs successfully, display results
*************************************************
     IF CEE000 of FC  THEN
         DISPLAY 'Input seconds of ' INSECS
             ' represents:'
```

```
    DISPLAY '   Year......... ' YEAR
    DISPLAY '   Month........ ' MONTH
    DISPLAY '   Day.......... ' DAYS
    DISPLAY '   Hour......... ' HOURS
    DISPLAY '   Minute....... ' MINUTES
    DISPLAY '   Second....... ' SECONDS
    DISPLAY '   Millisecond.. ' MILLSEC
ELSE
    DISPLAY 'CEESECI failed with msg '
        Msg-No of FC UPON CONSOLE
    STOP RUN
END-IF.

GOBACK.
```

CEESECS: convert timestamp to seconds

CEESECS converts a string that represents a timestamp into Lilian seconds (the number of seconds since 00:00:00 14 October 1582). This service makes it easier to perform time arithmetic, such as calculating the elapsed time between two timestamps.

CALL CEESECS syntax

```
►►──CALL──"CEESECS"──USING──input_timestamp,──picture_string,──────────────►

►─output_seconds,──fc.───────────────────────────────────────────────────►◄
```

***input_timestamp* (input)**
A halfword length-prefixed character string that represents a date or timestamp in a format matching that specified by *picture_string*.

The character string must contain between 5 and 80 picture characters, inclusive. *input_timestamp* can contain leading or trailing blanks. Parsing begins with the first nonblank character (unless the picture string itself contains leading blanks; in this case, CEESECS skips exactly that many positions before parsing begins).

After a valid date is parsed, as determined by the format of the date you specify in *picture_string*, all remaining characters are ignored by CEESECS. Valid dates range between and including the dates 15 October 1582 to 31 December 9999. A full date must be specified. Valid times range from 00:00:00.000 to 23:59:59.999.

If any part or all of the time value is omitted, zeros are substituted for the remaining values. For example:

```
1992-05-17-19:02 is equivalent to 1992-05-17-19:02:00
1992-05-17       is equivalent to 1992-05-17-00:00:00
```

***picture_string* (input)**
A halfword length-prefixed character string, indicating the format of the date or timestamp value specified in *input_timestamp*.

Each character in the *picture_string* represents a character in *input_timestamp*. For example, if you specify MMDDYY HH.MI.SS as the *picture_string*, CEESECS reads an *input_char_date* of 060288 15.35.02 as

3:35:02 PM on 02 June 1988. If delimiters such as the slash (/) appear in the picture string, leading zeros can be omitted. For example, the following calls to CEESECS all assign the same value to data item *secs*:

```
CALL CEESECS USING '92/06/03 15.35.03',
                   'YY/MM/DD HH.MI.SS', secs, fc.
CALL CEESECS USING '92/6/3 15.35.03',
                   'YY/MM/DD HH.MI.SS', secs, fc.
CALL CEESECS USING '92/6/3 3.35.03 PM',
                   'YY/MM/DD HH.MI.SS AP', secs, fc.
CALL CEESECS USING '92.155 3.35.03 pm',
                   'YY.DDD   HH.MI.SS AP', secs, fc.
```

If *picture_string* includes a Japanese Era symbol <JJJJ>, the YY position in *input_timestamp* represents the year number within the Japanese Era. For example, the year 1988 equals the Japanese year 63 in the Showa era.

***output_seconds* (output)**

A 64-bit long floating-point number that represents the number of seconds since 00:00:00 on 14 October 1582, not counting leap seconds. For example, 00:00:01 on 15 October 1582 is second 86,401 (24*60*60 + 01) in the Lilian format. 19:00:01.12 on 16 May 1988 is second 12,799,191,601.12.

The largest value represented is 23:59:59.999 on 31 December 9999, which is second 265,621,679,999.999 in the Lilian format.

A 64-bit long floating-point value can accurately represent approximately 16 significant decimal digits without loss of precision. Therefore, accuracy is available to the nearest millisecond (15 decimal digits).

If *input_timestamp* does not contain a valid date or timestamp, *output_seconds* is set to 0 and CEESECS terminates with a non-CEE000 symbolic feedback code.

Elapsed time calculations are performed easily on the *output_seconds*, because it represents elapsed time. Leap year and end-of-year anomalies do not affect the calculations.

***fc* (output)**

A 12-byte feedback code (optional) that indicates the result of this service.

Table 90. **CEESECS symbolic conditions**

Symbolic feedback code	Severity	Message number	Message text
CEE000	0	--	The service completed successfully.
CEE2EB	3	2507	Insufficient data was passed to CEEDAYS or CEESECS. The Lilian value was not calculated.
CEE2EC	3	2508	The date value passed to CEEDAYS or CEESECS was invalid.
CEE2ED	3	2509	The era passed to CEEDAYS or CEESECS was not recognized.
CEE2EE	3	2510	The hours value in a call to CEEISEC or CEESECS was not recognized.
CEE2EH	3	2513	The input date passed in a CEEISEC, CEEDAYS, or CEESECS call was not within the supported range.
CEE2EK	3	2516	The minutes value in a CEEISEC call was not recognized.

Table 90. **CEESECS symbolic conditions** *(continued)*

Symbolic feedback code	Severity	Message number	Message text
CEE2EL	3	2517	The month value in a CEEISEC call was not recognized.
CEE2EM	3	2518	An invalid picture string was specified in a call to a date/time service.
CEE2EN	3	2519	The seconds value in a CEEISEC call was not recognized.
CEE2EP	3	2521	The <JJJJ>, <CCCC>, or <CCCCCCCC> year-within-era value passed to CEEDAYS or CEESECS was zero.
CEE2ET	3	2525	CEESECS detected nonnumeric data in a numeric field, or the timestamp string did not match the picture string.

Usage notes

- The inverse of CEESECS is CEEDATM, which converts *output_seconds* to character format.
- By default, two-digit years lie within the 100-year range that starts 80 years before the system date. Thus in 2008, all two-digit years represent dates between 1928 and 2027, inclusive. You can change this range by using the callable service CEESCEN.

Example

```
CBL LIB
      ************************************************
      **                                            **
      ** Function: Call CEESECS to convert          **
      **           timestamp to number of seconds   **
      **                                            **
      ** In this example, calls are made to CEESECS **
      ** to convert two timestamps to the number of **
      ** seconds since 00:00:00 14 October 1582.    **
      ** The Lilian seconds for the earlier         **
      ** timestamp are then subtracted from the     **
      ** Lilian seconds for the later timestamp     **
      ** to determine the number of between the     **
      ** two.  This result is displayed.            **
      **                                            **
      ************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLSECS.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  SECOND1                COMP-2.
       01  SECOND2                COMP-2.
       01  TIMESTP.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char        PIC X,
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                              of TIMESTP.
       01  TIMESTP2.
           02  Vstring-length      PIC S9(4) BINARY.
```

```
    02  Vstring-text.
        03  Vstring-char        PIC X,
                    OCCURS 0 TO 256 TIMES
                    DEPENDING ON Vstring-length
                       of TIMESTP2.
01  PICSTR.
    02  Vstring-length      PIC S9(4) BINARY.
    02  Vstring-text.
        03  Vstring-char        PIC X,
                    OCCURS 0 TO 256 TIMES
                    DEPENDING ON Vstring-length
                       of PICSTR.
01  FC.
    02  Condition-Token-Value.
    COPY  CEEIGZCT.
        03  Case-1-Condition-ID.
            04  Severity    PIC S9(4) COMP.
            04  Msg-No      PIC S9(4) COMP.
        03  Case-2-Condition-ID
                  REDEFINES Case-1-Condition-ID.
            04  Class-Code  PIC S9(4) COMP.
            04  Cause-Code  PIC S9(4) COMP.
        03  Case-Sev-Ctl    PIC X.
        03  Facility-ID     PIC XXX.
    02  I-S-Info            PIC S9(9) COMP.
PROCEDURE DIVISION.

PARA-SECS1.
************************************************
** Specify first timestamp and a picture string
**     describing the format of the timestamp
**     as input to CEESECS
************************************************
    MOVE 25 TO Vstring-length of TIMESTP.
    MOVE '1969-05-07 12:01:00.000'
            TO Vstring-text of TIMESTP.
    MOVE 25 TO Vstring-length of PICSTR.
    MOVE 'YYYY-MM-DD HH:MI:SS.999'
            TO Vstring-text of PICSTR.

************************************************
** Call CEESECS to convert the first timestamp
** to Lilian seconds
************************************************
    CALL 'CEESECS' USING TIMESTP, PICSTR,
                         SECOND1, FC.
    IF NOT CEE000 of FC  THEN
        DISPLAY 'CEESECS failed with msg '
            Msg-No of FC UPON CONSOLE
        STOP RUN
    END-IF.

PARA-SECS2.
************************************************
** Specify second timestamp and a picture string
**     describing the format of the timestamp as
**     input to CEESECS.
************************************************
    MOVE 25 TO Vstring-length of TIMESTP2.
    MOVE '2004-01-01 00:00:01.000'
            TO Vstring-text of TIMESTP2.
    MOVE 25 TO Vstring-length of PICSTR.
    MOVE 'YYYY-MM-DD HH:MI:SS.999'
            TO Vstring-text of PICSTR.

************************************************
** Call CEESECS to convert the second timestamp
```

```
      **     to Lilian seconds
      *************************************************
           CALL 'CEESECS' USING TIMESTP2, PICSTR,
                                SECOND2, FC.
           IF NOT CEE000 of FC  THEN
               DISPLAY 'CEESECS failed with msg '
                   Msg-No of FC UPON CONSOLE
               STOP RUN
           END-IF.

       PARA-SECS2.
      *************************************************
      ** Subtract SECOND2 from SECOND1 to determine the
      **     number of seconds between the two timestamps
      *************************************************
           SUBTRACT SECOND1 FROM SECOND2.
           DISPLAY 'The number of seconds between '
               Vstring-text OF TIMESTP ' and '
               Vstring-text OF TIMESTP2 ' is: ' SECOND2.

           GOBACK.
```

"Example: date-and-time picture strings" on page 564

RELATED REFERENCES
"Picture character terms and strings" on page 562

CEEUTC: get coordinated universal time

CEEUTC is identical to CEEGMT.

RELATED REFERENCES
"CEEGMT: get current Greenwich Mean Time" on page 617

IGZEDT4: get current date

IGZEDT4 returns the current date with a four-digit year in the form YYYYMMDD.

CALL IGZEDT4 syntax

```
►►──CALL──"IGZEDT4"──USING──output_char_date.──────────────────────────►◄
```

output_char_date **(output)**
: An 8-byte fixed-length character string in the form YYYYMMDD, which represents current year, month, and day.

Usage note: IGZEDT4 is not supported under CICS.

Example

```
CBL LIB
      ***************************************************
      ** Function: IGZEDT4 - get current date in the  **
      **                       format YYYYMMDD.        **
      ***************************************************
       IDENTIFICATION DIVISION.
```

```
 PROGRAM-ID. CBLEDT4.
 . . .
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  CHRDATE                 PIC S9(8) USAGE DISPLAY.
 . . .
 PROCEDURE DIVISION.
 PARA-CBLEDT4.
**************************************************
** Call IGZEDT4.
**************************************************
     CALL 'IGZEDT4' USING BY REFERENCE CHRDATE.
**************************************************
** IGZEDT4 has no nonzero return code to
**     check, so just display result.
**************************************************
     DISPLAY 'The current date is: '
         CHRDATE
     GOBACK.
```

Appendix G. XML reference material

The following information describes the XML exception codes that the XML parser and the `XML GENERATE` statement return in special register `XML-CODE`. It also documents the well-formedness constraints from the *XML specification* that the parser checks.

RELATED REFERENCES
"XML PARSE exceptions that allow continuation"
"XML PARSE exceptions that do not allow continuation" on page 643
"XML conformance" on page 646
"XML GENERATE exceptions" on page 649
XML specification

XML PARSE exceptions that allow continuation

| When an exception event occurs, the parser sets special register `XML-CODE` to a
| value that identifies the exception. Depending on the value of `XML-CODE`, the parser
| might be able to continue processing.

| The parser can continue processing after exception events if the associated
| exception code is within one of the following ranges:

- 1-99
- 100,001-165,535
- 200,001-265,535

The following table describes each exception and the actions that the parser takes when you request that it continue after the exception. Some of the descriptions use the following terms:

- *Actual document encoding*
- *Document encoding declaration*
- *External ASCII code page*
- *External EBCDIC code page*

For definitions of the terms, see the related concept below about XML input document encoding.

Table 91. **XML PARSE exceptions that allow continuation**

Code	Description	Parser action on continuation
1	The parser found an invalid character while scanning white space outside element content.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
2	The parser found an invalid start of a processing instruction, element, comment, or document type declaration outside element content.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.

Table 91. **XML PARSE exceptions that allow continuation** *(continued)*

Code	Description	Parser action on continuation
3	The parser found a duplicate attribute name.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
4	The parser found the markup character ’<’ in an attribute value.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
5	The start and end tag names of an element did not match.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
6	The parser found an invalid character in element content.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
7	The parser found an invalid start of an element, comment, processing instruction, or CDATA section in element content.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
8	The parser found in element content the CDATA closing character sequence ’]]>’ without the matching opening character sequence ’<![CDATA[’.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
9	The parser found an invalid character in a comment.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
10	The parser found in a comment the character sequence ’--’ (two hyphens) not followed by ’>’.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
11	The parser found an invalid character in a processing instruction data segment.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.

Table 91. **XML PARSE exceptions that allow continuation** *(continued)*

Code	Description	Parser action on continuation
12	The XML declaration was not at the beginning of the document.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
13	The parser found an invalid digit in a hexadecimal character reference (of the form &#x*dddd*;).	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
14	The parser found an invalid digit in a decimal character reference (of the form &#*dddd*;).	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
15	The encoding declaration value in the XML declaration did not begin with lowercase or uppercase A through Z.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
16	A character reference did not refer to a legal XML character.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
17	The parser found an invalid character in an entity reference name.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
18	The parser found an invalid character in an attribute value.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the `END-OF-DOCUMENT` event.
70	The actual document encoding was EBCDIC, and the external EBCDIC code page is supported, but the document encoding declaration did not specify a supported EBCDIC code page.	The parser uses the encoding specified by the external EBCDIC code page.
71	The actual document encoding was EBCDIC, and the document encoding declaration specified a supported EBCDIC encoding, but the external EBCDIC code page is not supported.	The parser uses the encoding specified by the document encoding declaration.

Table 91. **XML PARSE exceptions that allow continuation** *(continued)*

Code	Description	Parser action on continuation
72	The actual document encoding was EBCDIC, the external EBCDIC code page is not supported, and the document did not contain an encoding declaration.	The parser uses EBCDIC code page 1140 (USA, Canada, . . . Euro Country Extended Code Page).
73	The actual document encoding was EBCDIC, but neither the external EBCDIC code page nor the document encoding declaration specified a supported EBCDIC code page.	The parser uses EBCDIC code page 1140 (USA, Canada, . . . Euro Country Extended Code Page).
80	The actual document encoding was ASCII, and the external ASCII code page is supported, but the document encoding declaration did not specify a supported ASCII code page.	The parser uses the encoding specified by the external ASCII code page.
81	The actual document encoding was ASCII, and the document encoding declaration specified a supported ASCII encoding, but the external ASCII code page is not supported.	The parser uses the encoding specified by the document encoding declaration.
82	The actual document encoding was ASCII, but the external ASCII code page is not supported, and the document did not contain an encoding declaration.	The parser uses ASCII code page 819 (ISO-8859-1 Latin 1/Open Systems).
83	The actual document encoding was ASCII, but neither the external ASCII code page nor the document encoding declaration specified a supported ASCII code page.	The parser uses ASCII code page 819 (ISO-8859-1 Latin 1/Open Systems).
84	The actual document encoding was ASCII but not valid UTF-8, the external code page specified UTF-8, and the document did not contain an encoding declaration.	The parser uses UTF-8.
85	The actual document encoding was ASCII but not valid UTF-8, the external code page specified UTF-8, and the document encoding declaration specified neither a supported ASCII code page nor UTF-8.	The parser uses UTF-8.
86	The actual document encoding was ASCII but not valid UTF-8, the external code page specified a supported ASCII code page, and the document encoding declaration specified UTF-8.	The parser uses UTF-8.
87	The actual document encoding was ASCII but not valid UTF-8, and the external code page and the document encoding declaration both specified UTF-8.	The parser uses UTF-8.

Table 91. **XML PARSE exceptions that allow continuation** *(continued)*

Code	Description	Parser action on continuation
88	The actual document encoding was ASCII but not valid UTF-8, the external code page specified neither a supported ASCII code page nor UTF-8, and the document encoding declaration specified UTF-8.	The parser uses UTF-8.
89	The actual document encoding was ASCII but not valid UTF-8, the external code page specified UTF-8, and the document encoding declaration specified a supported ASCII code page.	The parser uses UTF-8.
92	The document data item was alphanumeric, but the actual document encoding was Unicode UTF-16.	The parser uses code page 1200 (Unicode UTF-16).
100,001 - 165,535	The external EBCDIC code page and the document encoding declaration specified different supported EBCDIC code pages. XML-CODE contains the code page CCSID for the encoding declaration plus 100,000.	If you set XML-CODE to zero before returning from the EXCEPTION event, the parser uses the encoding specified by the external EBCDIC code page. If you set XML-CODE to the CCSID for the document encoding declaration (by subtracting 100,000), the parser uses this encoding.
200,001 - 265,535	The external ASCII code page and the document encoding declaration specified different supported ASCII code pages. XML-CODE contains the CCSID for the encoding declaration plus 200,000.	If you set XML-CODE to zero before returning from the EXCEPTION event, the parser uses the encoding specified by the external ASCII code page. If you set XML-CODE to the CCSID for the document encoding declaration (by subtracting 200,000), the parser uses this encoding.

RELATED CONCEPTS
"XML-CODE" on page 366
"XML input document encoding" on page 371

RELATED TASKS
"Handling XML PARSE exceptions" on page 374

XML PARSE exceptions that do not allow continuation

When an exception event occurs, the parser sets special register XML-CODE to a value that identifies the exception. The information below describes exceptions that do not let the parser continue processing.

No further events are returned from the parser for the exceptions shown in the table below, even if you set XML-CODE to zero and return control to the parser after processing the exception. The parser transfers control to the statement that you specify in the ON EXCEPTION phrase, or to the end of the XML PARSE statement if you did not code an ON EXCEPTION phrase.

Table 92. **XML PARSE exceptions that do not allow continuation**

Code	Description
100	The parser reached the end of the document while scanning the start of the XML declaration.
101	The parser reached the end of the document while looking for the end of the XML declaration.
102	The parser reached the end of the document while looking for the root element.
103	The parser reached the end of the document while looking for the version information in the XML declaration.
104	The parser reached the end of the document while looking for the version information value in the XML declaration.
106	The parser reached the end of the document while looking for the encoding declaration value in the XML declaration.
108	The parser reached the end of the document while looking for the `standalone` declaration value in the XML declaration.
109	The parser reached the end of the document while scanning an attribute name.
110	The parser reached the end of the document while scanning an attribute value.
111	The parser reached the end of the document while scanning a character reference or entity reference in an attribute value.
112	The parser reached the end of the document while scanning an empty element tag.
113	The parser reached the end of the document while scanning the root element name.
114	The parser reached the end of the document while scanning an element name.
115	The parser reached the end of the document while scanning character data in element content.
116	The parser reached the end of the document while scanning a processing instruction in element content.
117	The parser reached the end of the document while scanning a comment or CDATA section in element content.
118	The parser reached the end of the document while scanning a comment in element content.
119	The parser reached the end of the document while scanning a CDATA section in element content.
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.

Table 92. **XML PARSE exceptions that do not allow continuation** *(continued)*

Code	Description
127	The parser found a character other than ’=’ following an attribute name.
128	The parser found an invalid attribute value delimiter.
130	The first character of an attribute name was not a letter, ’_’, or ’:’.
131	The parser found an invalid character either in or following an attribute name.
132	An empty element tag was not terminated by a ’>’ following the ’/’.
133	The first character of an element end tag name was not a letter, ’_’, or ’:’.
134	An element end tag name was not terminated by a ’>’.
135	The first character of an element name was not a letter, ’_’, or ’:’.
136	The parser found an invalid start of a comment or CDATA section in element content.
137	The parser found an invalid start of a comment.
138	The first character of a processing instruction target name was not a letter, ’_’, or ’:’.
139	The parser found an invalid character in or following a processing instruction target name.
140	A processing instruction was not terminated by the closing character sequence ’?>’.
141	The parser found an invalid character following ’&’ in a character reference or entity reference.
142	The version information was not present in the XML declaration.
143	’version’ in the XML declaration was not followed by ’=’.
144	The version declaration value in the XML declaration is either missing or improperly delimited.
145	The version information value in the XML declaration specified a bad character, or the start and end delimiters did not match.
146	The parser found an invalid character following the version information value closing delimiter in the XML declaration.
147	The parser found an invalid attribute instead of the optional encoding declaration in the XML declaration.
148	’encoding’ in the XML declaration was not followed by ’=’.
149	The encoding declaration value in the XML declaration is either missing or improperly delimited.
150	The encoding declaration value in the XML declaration specified a bad character, or the start and end delimiters did not match.
151	The parser found an invalid character following the encoding declaration value closing delimiter in the XML declaration.
152	The parser found an invalid attribute instead of the optional `standalone` declaration in the XML declaration.
153	`standalone` in the XML declaration was not followed by `=`.
154	The `standalone` declaration value in the XML declaration is either missing or improperly delimited.
155	The `standalone` declaration value was neither ’yes’ nor ’no’ only.
156	The `standalone` declaration value in the XML declaration specified a bad character, or the start and end delimiters did not match.

Table 92. **XML PARSE exceptions that do not allow continuation** (continued)

Code	Description
157	The parser found an invalid character following the `standalone` declaration value closing delimiter in the XML declaration.
158	The XML declaration was not terminated by the proper character sequence '?>', or contained an invalid attribute.
159	The parser found the start of a document type declaration after the end of the root element.
160	The parser found the start of an element after the end of the root element.
161	The parser found an invalid UTF-8 byte sequence.
162	The parser found a UTF-8 character that has a Unicode scalar value greater than x'FFFF'.
315	The *actual document encoding* was UTF-16 little-endian, which the parser does not support on this platform.
316	The actual document encoding was UCS4, which the parser does not support.
317	The parser cannot determine the document encoding. The document might be damaged.
318	The actual document encoding was UTF-8, which the parser does not support.
320	The document data item was national, but the actual document encoding was EBCDIC.
321	The document data item was national, but the actual document encoding was ASCII.
322	The document data item was a native alphanumeric data item, but the actual document encoding was EBCDIC.
323	The document data item was a host alphanumeric data item, but the actual document encoding was ASCII.
324	The document data item was national, but the actual document encoding was UTF-8.
325	The document data item was a host alphanumeric data item, but the actual document encoding was UTF-8.
500-599	Internal error. Please report the error to your service representative.

RELATED CONCEPTS
"XML-CODE" on page 366

RELATED TASKS
"Handling XML PARSE exceptions" on page 374

XML conformance

The built-in COBOL XML parser that is included in COBOL for AIX is not a conforming XML processor according to the definition in the *XML specification*. The parser does not validate the XML documents that you parse. Although it does check for many well-formedness errors, it does not perform all of the actions required of a nonvalidating XML processor.

In particular, it does not process the internal document type definition (DTD internal subset). Thus it does not supply default attribute values, does not normalize attribute values, and does not include the replacement text of internal entities except for the predefined entities. Instead, it passes the entire document type declaration as the contents of `XML-TEXT` or `XML-NTEXT` for the `DOCUMENT-TYPE-DECLARATION` XML event, which allows the application to perform these actions if required.

The parser optionally lets programs continue processing an XML document after some errors. The purpose of allowing processing to continue is to facilitate the debugging of XML documents and processing procedures.

Recapitulating the definition in the *XML specification*, a textual object is a *well-formed* XML document if:

- Taken as a whole, it conforms to the grammar for XML documents.
- It meets all the explicit well-formedness constraints listed in the *XML specification*.
- Each parsed entity (piece of text) that is referenced directly or indirectly within the document is well formed.

The COBOL XML parser does check that documents conform to the XML grammar, except for any document type declaration. The declaration is supplied in its entirety, unchecked, to your application.

The *XML specification* also contains twelve explicit well-formedness constraints. The constraints that the COBOL XML parser checks partly or completely are shown in **bold** type:

1. Parameter Entities (PEs) in Internal Subset: "In the internal DTD subset, parameter-entity references can occur only where markup declarations can occur, not within markup declarations. (This does not apply to references that occur in external parameter entities or to the external subset.)"

   The parser does not process the internal DTD subset, so it does not enforce this constraint.
2. External Subset: "The external subset, if any, must match the production for extSubset."

   The parser does not process the external subset, so it does not enforce this constraint.
3. Parameter Entity Between Declarations: "The replacement text of a parameter entity reference in a DeclSep must match the production extSubsetDecl."

   The parser does not process the internal DTD subset, so it does not enforce this constraint.
4. **Element Type Match:** "The Name in an element's end-tag must match the element type in the start-tag."

   The parser enforces this constraint.

5. **Unique Attribute Specification:** “No attribute name may appear more than once in the same start-tag or empty-element tag.”

   The parser partly supports this constraint by checking up to 10 attribute names in a given element for uniqueness. The application can check any attribute names beyond this limit.
6. No External Entity References: “Attribute values cannot contain direct or indirect entity references to external entities.”

   The parser does not enforce this constraint.
7. No ′<′ in Attribute Values: “The replacement text of any entity referred to directly or indirectly in an attribute value must not contain a ′<′.”

   The parser does not enforce this constraint.
8. **Legal Character:** “Characters referred to using character references must match the production for Char.”

   The parser enforces this constraint.
9. Entity Declared: “In a document without any DTD, a document with only an internal DTD subset which contains no parameter entity references, or a document with standalone=′yes′, for an entity reference that does not occur within the external subset or a parameter entity, the Name given in the entity reference must match that in an entity declaration that does not occur within the external subset or a parameter entity, except that well-formed documents need not declare any of the following entities: amp, lt, gt, apos, quot. The declaration of a general entity must precede any reference to it which appears in a default value in an attribute-list declaration.”

   “Note that if entities are declared in the external subset or in external parameter entities, a non-validating processor is not obligated to read and process their declarations; for such documents, the rule that an entity must be declared is a well-formedness constraint only if standalone=′yes′.”

   The parser does not enforce this constraint.
10. Parsed Entity: “An entity reference must not contain the name of an unparsed entity. Unparsed entities may be referred to only in attribute values declared to be of type ENTITY or ENTITIES.”

    The parser does not enforce this constraint.
11. No Recursion: “A parsed entity must not contain a recursive reference to itself, either directly or indirectly.”

    The parser does not enforce this constraint.
12. In DTD: “Parameter-entity references may only appear in the DTD.”

    The parser does not enforce this constraint, because the error cannot occur.

RELATED CONCEPTS
“XML parser in COBOL” on page 361

RELATED REFERENCES
Extensible Markup Language (XML)
XML specification (Prolog and document type declaration)

XML GENERATE exceptions

One of several exception codes might be returned in the `XML-CODE` special register during XML generation. If one of these exceptions occurs, control is passed to the statement in the `ON EXCEPTION` phrase, or to the end of the `XML GENERATE` statement if you did not code an `ON EXCEPTION` phrase.

Table 93. **XML GENERATE exceptions**

Code	Description
400	The receiver was too small to contain the generated XML document. The `COUNT IN` data item, if specified, contains the count of character positions that were actually generated.
401	A multibyte data-name contained a character that, when converted to Unicode, was not valid in an XML element or attribute name.
402	The first character of a multibyte data-name, when converted to Unicode, was not valid as the first character of an XML element or attribute name.
403	The value of an `OCCURS DEPENDING ON` variable exceeded 16,777,215.
410	The encoding specified for the XML document is not supported for conversion to Unicode.
411	The receiver was host alphanumeric, but the encoding specified for the document is not a supported single-byte EBCDIC code page.
412	The receiver was native alphanumeric, but the encoding specified for the document was not UTF-8 or a supported single-byte ASCII code page.
413	The receiver was alphanumeric, but the runtime locale was not consistent with the compile-time locale.
414	The encoding specified for the XML document was invalid or was not a supported code page.
415	The receiver was national, but the encoding specified for the document was not UTF-16.
416	The XML namespace identifier contained invalid XML characters.
417	Element character content or an attribute value contained characters that are illegal in XML content. XML generation has continued, with the element tag name or the attribute name prefixed with 'hex.' and the original data value represented in the document in hexadecimal.
418	Substitution characters were generated by encoding conversion.
419	The XML namespace prefix was invalid.
420	The source data item included a multibyte name or multibyte content, and the receiver was native alphanumeric, but the encoding specified for the document was not UTF-8.
600-699	Internal error. Report the error to your service representative.

RELATED TASKS
"Handling XML GENERATE exceptions" on page 384

RELATED REFERENCES
"XML generation" on page 570

Appendix H. JNI.cpy

This listing shows the copybook JNI.cpy, which you can use to access the Java Native Interface (JNI) services from your COBOL programs.

JNI.cpy contains sample COBOL data definitions that correspond to the Java JNI types, and contains JNINativeInterface, the JNI environment structure that contains function pointers for accessing the JNI callable services.

JNI.cpy is in the `include` subdirectory of the COBOL install directory (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.


* Possible return values for JNI functions.
```

```
 01 JNI-RC pic S9(9) comp-5.
* success
   88 JNI-OK        value  0.
* unknown error
   88 JNI-ERR       value -1.
* thread detached from the VM
   88 JNI-EDETACHED value -2.
* JNI version error
   88 JNI-EVERSION  value -3.
* not enough memory
   88 JNI-ENOMEM    value -4.
* VM already created
   88 JNI-EEXIST    value -5.
* invalid arguments
   88 JNI-EINVAL    value -6.

* Used in ReleaseScalarArrayElements
 01 releaseMode pic s9(9) comp-5.
   88 JNI-COMMIT value 1.
   88 JNI-ABORT  value 2.

 01 JNIenv pointer.

* JNI Native Method Interface - environment structure.
 01 JNINativeInterface.
   02 pointer.
   02 pointer.
   02 pointer.
   02 pointer.
   02 GetVersion                      function-pointer.
   02 DefineClass                     function-pointer.
   02 FindClass                       function-pointer.
   02 FromReflectedMethod             function-pointer.
   02 FromReflectedField              function-pointer.
   02 ToReflectedMethod               function-pointer.
   02 GetSuperclass                   function-pointer.
   02 IsAssignableFrom                function-pointer.
   02 ToReflectedField                function-pointer.
   02 Throw                           function-pointer.
   02 ThrowNew                        function-pointer.
   02 ExceptionOccurred               function-pointer.
   02 ExceptionDescribe               function-pointer.
   02 ExceptionClear                  function-pointer.
   02 FatalError                      function-pointer.
   02 PushLocalFrame                  function-pointer.
   02 PopLocalFrame                   function-pointer.
   02 NewGlobalRef                    function-pointer.
   02 DeleteGlobalRef                 function-pointer.
   02 DeleteLocalRef                  function-pointer.
   02 IsSameObject                    function-pointer.
   02 NewLocalRef                     function-pointer.
   02 EnsureLocalCapacity             function-pointer.
   02 AllocObject                     function-pointer.
   02 NewObject                       function-pointer.
   02 NewObjectV                      function-pointer.
   02 NewObjectA                      function-pointer.
   02 GetObjectClass                  function-pointer.
   02 IsInstanceOf                    function-pointer.
   02 GetMethodID                     function-pointer.
   02 CallObjectMethod                function-pointer.
   02 CallObjectMethodV               function-pointer.
   02 CallObjectMethodA               function-pointer.
   02 CallBooleanMethod               function-pointer.
   02 CallBooleanMethodV              function-pointer.
   02 CallBooleanMethodA              function-pointer.
   02 CallByteMethod                  function-pointer.
   02 CallByteMethodV                 function-pointer.
```

```
02 CallByteMethodA                  function-pointer.
02 CallCharMethod                   function-pointer.
02 CallCharMethodV                  function-pointer.
02 CallCharMethodA                  function-pointer.
02 CallShortMethod                  function-pointer.
02 CallShortMethodV                 function-pointer.
02 CallShortMethodA                 function-pointer.
02 CallIntMethod                    function-pointer.
02 CallIntMethodV                   function-pointer.
02 CallIntMethodA                   function-pointer.
02 CallLongMethod                   function-pointer.
02 CallLongMethodV                  function-pointer.
02 CallLongMethodA                  function-pointer.
02 CallFloatMethod                  function-pointer.
02 CallFloatMethodV                 function-pointer.
02 CallFloatMethodA                 function-pointer.
02 CallDoubleMethod                 function-pointer.
02 CallDoubleMethodV                function-pointer.
02 CallDoubleMethodA                function-pointer.
02 CallVoidMethod                   function-pointer.
02 CallVoidMethodV                  function-pointer.
02 CallVoidMethodA                  function-pointer.
02 CallNonvirtualObjectMethod       function-pointer.
02 CallNonvirtualObjectMethodV      function-pointer.
02 CallNonvirtualObjectMethodA      function-pointer.
02 CallNonvirtualBooleanMethod      function-pointer.
02 CallNonvirtualBooleanMethodV     function-pointer.
02 CallNonvirtualBooleanMethodA     function-pointer.
02 CallNonvirtualByteMethod         function-pointer.
02 CallNonvirtualByteMethodV        function-pointer.
02 CallNonvirtualByteMethodA        function-pointer.
02 CallNonvirtualCharMethod         function-pointer.
02 CallNonvirtualCharMethodV        function-pointer.
02 CallNonvirtualCharMethodA        function-pointer.
02 CallNonvirtualShortMethod        function-pointer.
02 CallNonvirtualShortMethodV       function-pointer.
02 CallNonvirtualShortMethodA       function-pointer.
02 CallNonvirtualIntMethod          function-pointer.
02 CallNonvirtualIntMethodV         function-pointer.
02 CallNonvirtualIntMethodA         function-pointer.
02 CallNonvirtualLongMethod         function-pointer.
02 CallNonvirtualLongMethodV        function-pointer.
02 CallNonvirtualLongMethodA        function-pointer.
02 CallNonvirtualFloatMethod        function-pointer.
02 CallNonvirtualFloatMethodV       function-pointer.
02 CallNonvirtualFloatMethodA       function-pointer.
02 CallNonvirtualDoubleMethod       function-pointer.
02 CallNonvirtualDoubleMethodV      function-pointer.
02 CallNonvirtualDoubleMethodA      function-pointer.
02 CallNonvirtualVoidMethod         function-pointer.
02 CallNonvirtualVoidMethodV        function-pointer.
02 CallNonvirtualVoidMethodA        function-pointer.
02 GetFieldID                       function-pointer.
02 GetObjectField                   function-pointer.
02 GetBooleanField                  function-pointer.
02 GetByteField                     function-pointer.
02 GetCharField                     function-pointer.
02 GetShortField                    function-pointer.
02 GetIntField                      function-pointer.
02 GetLongField                     function-pointer.
02 GetFloatField                    function-pointer.
02 GetDoubleField                   function-pointer.
02 SetObjectField                   function-pointer.
02 SetBooleanField                  function-pointer.
02 SetByteField                     function-pointer.
02 SetCharField                     function-pointer.
02 SetShortField                    function-pointer.
```

```
02 SetIntField                      function-pointer.
02 SetLongField                     function-pointer.
02 SetFloatField                    function-pointer.
02 SetDoubleField                   function-pointer.
02 GetStaticMethodID                function-pointer.
02 CallStaticObjectMethod           function-pointer.
02 CallStaticObjectMethodV          function-pointer.
02 CallStaticObjectMethodA          function-pointer.
02 CallStaticBooleanMethod          function-pointer.
02 CallStaticBooleanMethodV         function-pointer.
02 CallStaticBooleanMethodA         function-pointer.
02 CallStaticByteMethod             function-pointer.
02 CallStaticByteMethodV            function-pointer.
02 CallStaticByteMethodA            function-pointer.
02 CallStaticCharMethod             function-pointer.
02 CallStaticCharMethodV            function-pointer.
02 CallStaticCharMethodA            function-pointer.
02 CallStaticShortMethod            function-pointer.
02 CallStaticShortMethodV           function-pointer.
02 CallStaticShortMethodA           function-pointer.
02 CallStaticIntMethod              function-pointer.
02 CallStaticIntMethodV             function-pointer.
02 CallStaticIntMethodA             function-pointer.
02 CallStaticLongMethod             function-pointer.
02 CallStaticLongMethodV            function-pointer.
02 CallStaticLongMethodA            function-pointer.
02 CallStaticFloatMethod            function-pointer.
02 CallStaticFloatMethodV           function-pointer.
02 CallStaticFloatMethodA           function-pointer.
02 CallStaticDoubleMethod           function-pointer.
02 CallStaticDoubleMethodV          function-pointer.
02 CallStaticDoubleMethodA          function-pointer.
02 CallStaticVoidMethod             function-pointer.
02 CallStaticVoidMethodV            function-pointer.
02 CallStaticVoidMethodA            function-pointer.
02 GetStaticFieldID                 function-pointer.
02 GetStaticObjectField             function-pointer.
02 GetStaticBooleanField            function-pointer.
02 GetStaticByteField               function-pointer.
02 GetStaticCharField               function-pointer.
02 GetStaticShortField              function-pointer.
02 GetStaticIntField                function-pointer.
02 GetStaticLongField               function-pointer.
02 GetStaticFloatField              function-pointer.
02 GetStaticDoubleField             function-pointer.
02 SetStaticObjectField             function-pointer.
02 SetStaticBooleanField            function-pointer.
02 SetStaticByteField               function-pointer.
02 SetStaticCharField               function-pointer.
02 SetStaticShortField              function-pointer.
02 SetStaticIntField                function-pointer.
02 SetStaticLongField               function-pointer.
02 SetStaticFloatField              function-pointer.
02 SetStaticDoubleField             function-pointer.
02 NewString                        function-pointer.
02 GetStringLength                  function-pointer.
02 GetStringChars                   function-pointer.
02 ReleaseStringChars               function-pointer.
02 NewStringUTF                     function-pointer.
02 GetStringUTFLength               function-pointer.
02 GetStringUTFChars                function-pointer.
02 ReleaseStringUTFChars            function-pointer.
02 GetArrayLength                   function-pointer.
02 NewObjectArray                   function-pointer.
02 GetObjectArrayElement            function-pointer.
02 SetObjectArrayElement            function-pointer.
02 NewBooleanArray                  function-pointer.
```

```
   02 NewByteArray                     function-pointer.
   02 NewCharArray                     function-pointer.
   02 NewShortArray                    function-pointer.
   02 NewIntArray                      function-pointer.
   02 NewLongArray                     function-pointer.
   02 NewFloatArray                    function-pointer.
   02 NewDoubleArray                   function-pointer.
   02 GetBooleanArrayElements          function-pointer.
   02 GetByteArrayElements             function-pointer.
   02 GetCharArrayElements             function-pointer.
   02 GetShortArrayElements            function-pointer.
   02 GetIntArrayElements              function-pointer.
   02 GetLongArrayElements             function-pointer.
   02 GetFloatArrayElements            function-pointer.
   02 GetDoubleArrayElements           function-pointer.
   02 ReleaseBooleanArrayElements      function-pointer.
   02 ReleaseByteArrayElements         function-pointer.
   02 ReleaseCharArrayElements         function-pointer.
   02 ReleaseShortArrayElements        function-pointer.
   02 ReleaseIntArrayElements          function-pointer.
   02 ReleaseLongArrayElements         function-pointer.
   02 ReleaseFloatArrayElements        function-pointer.
   02 ReleaseDoubleArrayElements       function-pointer.
   02 GetBooleanArrayRegion            function-pointer.
   02 GetByteArrayRegion               function-pointer.
   02 GetCharArrayRegion               function-pointer.
   02 GetShortArrayRegion              function-pointer.
   02 GetIntArrayRegion                function-pointer.
   02 GetLongArrayRegion               function-pointer.
   02 GetFloatArrayRegion              function-pointer.
   02 GetDoubleArrayRegion             function-pointer.
   02 SetBooleanArrayRegion            function-pointer.
   02 SetByteArrayRegion               function-pointer.
   02 SetCharArrayRegion               function-pointer.
   02 SetShortArrayRegion              function-pointer.
   02 SetIntArrayRegion                function-pointer.
   02 SetLongArrayRegion               function-pointer.
   02 SetFloatArrayRegion              function-pointer.
   02 SetDoubleArrayRegion             function-pointer.
   02 RegisterNatives                  function-pointer.
   02 UnregisterNatives                function-pointer.
   02 MonitorEnter                     function-pointer.
   02 MonitorExit                      function-pointer.
   02 GetJavaVM                        function-pointer.
   02 GetStringRegion                  function-pointer.
   02 GetStringUTFRegion               function-pointer.
   02 GetPrimitiveArrayCritical        function-pointer.
   02 ReleasePrimitiveArrayCritical    function-pointer.
   02 GetStringCritical                function-pointer.
   02 ReleaseStringCritical            function-pointer.
   02 NewWeakGlobalRef                 function-pointer.
   02 DeleteWeakGlobalRef              function-pointer.
   02 ExceptionCheck                   function-pointer.
```

RELATED TASKS
"Compiling OO applications" on page 237
"Accessing JNI services" on page 443

Appendix I. COBOL SYSADATA file contents

When you use the `ADATA` compiler option, the compiler produces a file that contains program data. You can use this file instead of the compiler listing to extract information about the program. For example, you can extract information about the program for symbolic debugging tools or cross-reference tools.

"Example: SYSADATA" on page 659

RELATED REFERENCES
"ADATA" on page 246
"Existing compiler options that affect the SYSADATA file"
"SYSADATA record types" on page 658
"SYSADATA record descriptions" on page 660

Existing 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.

TEST `TEST` causes additional object text records to be created that also affect the contents of the SYSADATA file.

NUM `NUM` causes the compiler to use the contents of columns 1-6 in the source records for line numbering, rather than using generated sequence numbers. Any invalid (nonnumeric) or out-of-sequence numbers are replaced with a number one higher than that of the previous record.

The following SYSADATA fields contain line numbers whose contents differ depending on the `NUM|NONUM` setting:

Type	Field	Record
0020	AE_LINE	External Symbol record
0030	ATOK_LINE	Token record
0032	AF_STMT	Source Error record
0038	AS_STMT	Source record
0039	AS_REP_EXP_SLIN	COPY REPLACING record
0039	AS_REP_EXP_ELIN	COPY REPLACING record
0042	ASY_STMT	Symbol record
0044	AX_DEFN	Symbol Cross Reference record
0044	AX_STMT	Symbol Cross Reference record
0046	AN_STMT	Nested Program record

The Type 0038 Source record contains two fields that relate to line numbers and record numbers:

- AS_STMT contains the compiler line number in both the NUM and NONUM cases.
- AS_CUR_REC# contains the physical source record number.

These two fields can always be used to correlate the compiler line numbers, used in all the above fields, with physical source record numbers.

The remaining compiler options have no direct effect on the SYSADATA file, but might trigger generation of additional error messages associated with the specific option, such as FLAGSAA, FLAGSTD, or SSRANGE.

"Example: SYSADATA" on page 659

RELATED REFERENCES
"SYSADATA record types"
"COMPILE" on page 255
"EXIT" on page 259
"NUMBER" on page 274
"TEST" on page 286

SYSADATA record types

The SYSADATA file contains records classified into different record types. Each type of record provides information about the COBOL program being compiled.

Each record consists of two parts:
- A 12-byte header section, which has the same structure for all record types, and contains the record code that identifies the type of record
- A variable-length data section, which varies by record type

Table 94. **SYSADATA record types**

Record type	What it does
"Job identification record: X'0000'" on page 662	Provides information about the environment used to process the source data
"ADATA identification record: X'0001'" on page 663	Provides common information about the records in the SYSADATA file
"Compilation unit start|end record: X'0002'" on page 663	Marks the beginning and ending of compilation units in a source file
"Options record: X'0010'" on page 664	Describes the compiler options used for the compilation
"External symbol record: X'0020'" on page 673	Describes all external names in the program, definitions, and references
"Parse tree record: X'0024'" on page 674	Defines a node in the parse tree of the program
"Token record: X'0030'" on page 689	Defines a source token
"Source error record: X'0032'" on page 702	Describes errors in source program statements
"Source record: X'0038'" on page 703	Describes a single source line
"COPY REPLACING record: X'0039'" on page 704	Describes an instance of text replacement as a result of a match of COPY. . .REPLACING *operand-1* with text in the copybook

Table 94. **SYSADATA record types** *(continued)*

Record type	**What it does**
"Symbol record: X'0042'" on page 704	Describes a single symbol defined in the program. There is one symbol record for each symbol defined in the program.
"Symbol cross-reference record: X'0044'" on page 717	Describes references to a single symbol
"Nested program record: X'0046'" on page 718	Describes the name and nesting level of a program
"Library record: X'0060'" on page 719	Describes the library files and members used from each library
"Statistics record: X'0090'" on page 719	Describes the statistics about the compilation
"EVENTS record: X'0120'" on page 720	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"

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 (L*n*). 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 661
"Job identification record: X'0000'" on page 662
"ADATA identification record: X'0001'" on page 663
"Compilation unit start|end record: X'0002'" on page 663
"Options record: X'0010'" on page 664
"External symbol record: X'0020'" on page 673
"Parse tree record: X'0024'" on page 674
"Token record: X'0030'" on page 689
"Source error record: X'0032'" on page 702
"Source record: X'0038'" on page 703
"COPY REPLACING record: X'0039'" on page 704
"Symbol record: X'0042'" on page 704
"Symbol cross-reference record: X'0044'" on page 717
"Nested program record: X'0046'" on page 718

"Library record: X'0060'" on page 719
"Statistics record: X'0090'" on page 719
"EVENTS record: X'0120'" on page 720

Common header section

The table below shows the format of the header section that is common for all record types. For MVS™ and VSE, each record is preceded by a 4-byte RDW (record-descriptor word) that is normally used only by access methods and stripped off by download utilities.

Table 95. **SYSADATA common header section**

Field	Size	Description
Language code	XL1	**16** High Level Assembler **17** COBOL on all platforms **40** PL/I on supported platforms
Record type	HL2	The record type, which can be one of the following: **X'0000'** Job Identification record[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	XL1	**.... ..1.** ADATA record integers are in little-endian (Intel) format **.... ...1** This record is continued in the next record **1111 11..** Reserved for future use
Associated data record edition level	XL1	Used to indicate a new format for a specific record type, usually 0

Table 95. **SYSADATA common header section** *(continued)*

Field	Size	Description
Reserved	CL4	Reserved for future use
Associated data field length	HL2	The length in bytes of the data following the header
1. When a batch compilation (sequence of programs) is run with the ADATA option, there will be multiple Job Identification, Options, and Statistics records for each compilation.		

The mapping of the 12-byte header does not include the area used for the variable-length record-descriptor word required by the access method on MVS and VSE.

Job identification record: X'0000'

The following table shows the contents of the job identification record.

Table 96. **SYSADATA job identification record**

Field	Size	Description
Date	CL8	The date of the compilation in the format YYYYMMDD
Time	CL4	The time of the compilation in the format HHMM
Product number	CL8	The product number of the compiler that produced the associated data file
Product version	CL8	The version number of the product that produced the associated data file, in the form V.R.M
PTF level	CL8	The PTF level number of the product that produced the associated data file. (This field is blank if the PTF number is not available.)
System ID	CL24	The system identification of the system on which the compilation was run
Job name	CL8	The MVS job name of the compilation job
Step name	CL8	The MVS step name of the compilation step
Proc step	CL8	The MVS procedure step name of the compilation procedure
Number of input files[1]	HL2	The number of input files recorded in this record. The following group of seven fields will occur *n* 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(*n*)	The name of the input file for the compilation
...Volume serial number	CL(*n*)	The volume serial number of the (first) volume on which the input file resides

Table 96. **SYSADATA job identification record** *(continued)*

Field	Size	Description
...Member name	CL(*n*)	Where applicable, the name of the member in the input file
1. Where the number of input files would exceed the record size for the associated data file, the record is continued on the next record. The current number of input files (for that record) is stored in the record, and the record is written to the associated data file. The next record contains the rest of the input files. The count of the number of input files is a count for the current record.		

ADATA identification record: X'0001'

The following table shows the contents of the ADATA identification record.

Table 97. **ADATA identification record**

Field	Size	Description
Time (binary)	XL8	Universal Time (UT) as a binary number of microseconds since midnight Greenwich Mean Time, with the low-order bit representing 1 microsecond. This time can be used as a time-zone-independent time stamp. On Windows and AIX systems, only bytes 5-8 of the field are used as a fullword binary field that contains the time.
CCSID[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
1. The appropriate CCS flag will always be set. If the CCSID is set to nonzero, the code-page name length will be zero. If the CCSID is set to zero, the code-page name length will be nonzero and the code-page name will be present.		

Compilation unit start|end record: X'0002'

The following table shows the contents of the compilation unit start|end record.

Table 98. **SYSADATA compilation unit start|end record**

Field	Size	Description
Type	HL2	Compilation unit type, which can be one of the following: **X'0000'** Start compilation unit **X'0001'** End compilation unit
Reserved	CL2	Reserved for future use
Reserved	FL4	Reserved for future use

Options record: X'0010'

The following table shows the contents of the options record.

Table 99. **SYSADATA options record**

Field	Size	Description
Option byte 0	XL1	**1111 1111** Reserved for future use
Option byte 1	XL1	**1... ....** Bit 1 = DECK, Bit 0 = NODECK **.1.. ....** Bit 1 = ADATA, Bit 0 = NOADATA **..1. ....** Bit 1 = COLLSEQ(EBCDIC), Bit 0 = COLLSEQ(LOCALE|BINARY) (Windows and AIX only) **...1 ....** Bit 1 = SEPOBJ, Bit 0 = NOSEPOBJ (Windows and AIX only) **.... 1...** Bit 1 = NAME, Bit 0 = NONAME **.... .1..** Bit 1 = OBJECT, Bit 0 = NOOBJECT **.... ..1.** Bit 1 = SQL, Bit 0 = NOSQL **.... ...1** Bit 1 = CICS, Bit 0 = NOCICS
Option byte 2	XL1	**1... ....** Bit 1 = OFFSET, Bit 0 = NOOFFSET **.1.. ....** Bit 1 = MAP, Bit 0 = NOMAP **..1. ....** Bit 1 = LIST, Bit 0 = NOLIST **...1 ....** Bit 1 = DBCSXREF, Bit 0 = NODBCSXREF **.... 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. **.... .1..** Bit 1 = SOURCE, Bit 0 = NOSOURCE **.... ..1.** Bit 1 = VBREF, Bit 0 = NOVBREF **.... ...1** Bit 1 = XREF, Bit 0 = not XREF. See also flag at bit 4 above.

Table 99. **SYSADATA options record** *(continued)*

Field	Size	Description
Option byte 3	XL1	`1... ....` Bit 1 = FLAG imbedded diagnostics level specified (a value *y* is specified as in FLAG(*x*,*y*)) `.1.. ....` Bit 1 = FLAGSTD, Bit 0 = NOFLAGSTD `..1. ....` Bit 1 = NUM, Bit 0 = NONUM `...1 ....` Bit 1 = SEQUENCE, Bit 0 = NOSEQUENCE `.... 1...` Bit 1 = SOSI, Bit 0 = NOSOSI (Windows and AIX only) `.... .1..` Bit 1 = NSYMBOL(NATIONAL), Bit 0 = NSYMBOL(DBCS) `.... ..1.` Bit 1 = PROFILE, Bit 0 = NOPROFILE (AIX only) `.... ...1` Bit 1 = WORD, Bit 0 = NOWORD
Option byte 4	XL1	`1... ....` Bit 1 = ADV, Bit 0 = NOADV `.1.. ....` Bit 1 = APOST, Bit 0 = QUOTE `..1. ....` Bit 1 = DYNAM, Bit 0 = NODYNAM `...1 ....` Bit 1 = AWO, Bit 0 = NOAWO `.... 1...` Bit 1 = RMODE specified, Bit 0 = RMODE(AUTO) `.... .1..` Bit 1 = RENT, Bit 0 = NORENT `.... ..1.` Bit 1 = RES: this flag will always be set on for COBOL. `.... ...1` Bit 1 = RMODE(24), Bit 0 = RMODE(ANY)

Table 99. **SYSADATA options record** *(continued)*

Field	Size	Description
Option byte 5	XL1	`1... ....` Bit 1 = SQLCCSID, Bit 0 = NOSQLCCSID `.1.. ....` Bit 1 = OPT, Bit 0 = NOOPT `..1. ....` Bit 1 = LIB, Bit 0 = NOLIB `...1 ....` Bit 1 = DBCS, Bit 0 = NODBCS `.... 1...` Bit 1 = OPT(FULL), Bit 0 = not OPT(FULL) `.... .1..` Bit 1 = SSRANGE, Bit 0 = NOSSRANGE `.... ..1.` Bit 1 = TEST, Bit 0 = NOTEST `.... ...1` Bit 1 = PROBE, Bit 0 = NOPROBE (Windows only)
Option byte 6	XL1	`..1. ....` Bit 1 = NUMPROC(PFD), Bit 0 = NUMPROC(NOPFD) `...1 ....` Bit 1 = NUMCLS(ALT), Bit 0 = NUMCLS(PRIM) `.... .1..` Bit 1 = BINARY(S390), Bit 0 = BINARY(NATIVE) (Windows and AIX only) `.... ..1.` Bit 1 = TRUNC(STD), Bit 0 = TRUNC(OPT) `.... ...1` Bit 1 = ZWB, Bit 0 = NOZWB `11.. 1...` Reserved for future use
Option byte 7	XL1	`1... ....` Bit 1 = ALOWCBL, Bit 0 = NOALOWCBL `.1.. ....` Bit 1 = TERM, Bit 0 = NOTERM `..1. ....` Bit 1 = DUMP, Bit 0 = NODUMP `.... ..1.` Bit 1 = CURRENCY, Bit 0 = NOCURRENCY `...1 11.1` Reserved for future use

Table 99. **SYSADATA options record** *(continued)*

Field	Size	Description
Option byte 8	XL1	**1... ....** Bit 1 = XMLPARSE(XMLSS), Bit 0 = XMLPARSE(COMPAT) **.1.. ....** Bit 1 = OPTFILE, Bit 0 = not OPTFILE **..1. ....** Bit 1 = ADDR(64), Bit 0 = ADDR(32) **...1 1111** Reserved for future use
Option byte 9	XL1	**1... ....** Bit 1 = DATA(24), Bit 0 = DATA(31) **.1.. ....** Bit 1 = FASTSRT, Bit 0 = NOFASTSRT **..1. ....** Bit 1 = SIZE(MAX), Bit 0 = SIZE(*nnnn*) or SIZE(*nnnn*K) **.... .1..** Bit 1 = THREAD, Bit 0 = NOTHREAD **...1 1.11** Reserved for future use
Option byte A	XL1	**1111 1111** Reserved for future use
Option byte B	XL1	**1111 1111** Reserved for future use

Table 99. **SYSADATA options record** *(continued)*

Field	Size	Description
Option byte C	XL1	`1... ....` Bit 1 = NCOLLSEQ(LOCALE) (Windows and AIX only) `.1.. ....` Reserved for future use `..1. ....` Bit 1 = INTDATE(LILIAN), Bit 0 = INTDATE(ANSI) `...1 ....` Bit 1 = NCOLLSEQ(BINARY) (Windows and AIX only) `.... 1...` Bit 1 = CHAR(EBCDIC), Bit 0 = CHAR(NATIVE) (Windows and AIX only) `.... .1..` Bit 1 = FLOAT(HEX), Bit 0 = FLOAT(NATIVE) (Windows and AIX only) `.... ..1.` Bit 1 = COLLSEQ(BINARY) (Windows and AIX only) `.... ...1` Bit 1 = COLLSEQ(LOCALE) (Windows and AIX only)
Option byte D	XL1	`1... ....` Bit 1 = DLL Bit 0 = NODLL `.1.. ....` Bit 1 = EXPORTALL, Bit 0 = NOEXPORTALL `..1. ....` Bit 1 = CODEPAGE `...1 ....` Bit 1 = DATEPROC, Bit 0 = NODATEPROC `.... 1...` Bit 1 = DATEPROC(FLAG), Bit 0 = DATEPROC(NOFLAG) `.... .1..` Bit 1 = YEARWINDOW `.... ..1.` Bit 1 = WSCLEAR, Bit 0 = NOWSCLEAR (Windows and AIX only) `.... ...1` Bit 1 = BEOPT, Bit 0 = NOBEOPT (Windows and AIX only)

Table 99. **SYSADATA options record** *(continued)*

Field	Size	Description
Option byte E	XL1	**1... ....** Bit 1 = DATEPROC(TRIG), Bit 0 = DATEPROC(NOTRIG) **.1.. ....** Bit 1 = DIAGTRUNC, Bit 0 = NODIAGTRUNC **.... .1..** Bit 1 = LSTFILE(UTF-8), Bit 0 = LSTFILE(LOCALE) (Windows and AIX only) **.... ..1.** Bit 1 = MDECK, Bit 0 = NOMDECK **.... ...1** Bit 1 = MDECK(NOCOMPILE) **..11 1...** Reserved for future use
Option byte F	XL1	**1111 1111** Reserved for future use
Flag 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
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

Table 99. **SYSADATA options record** *(continued)*

Field	Size	Description
FLAGSTD (FIPS) specification	XL1	**1... ....** Minimum **.1.. ....** Intermediate **..1. ....** High **...1 ....** IBM extensions **.... 1...** Level-1 segmentation **.... .1..** Level-2 segmentation **.... ..1.** Debugging **.... ...1** 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	
Data for 3-valued options	XL1	**1... ....** NAME(ALIAS) specified **.1.. ....** NUMPROC(MIG) specified **..1. ....** TRUNC(BIN) specified **...1 1111** Reserved for future use
TEST suboptions	XL1	**1... ....** TEST(HOOK) **.1.. ....** TEST(SEP) **..1. ....** TEST(EJPD) **...1 1111** Reserved for TEST suboptions
OUTDD name length	HL2	Length of OUTDD name
RWT ID Length	HL2	Length of Reserved Word Table identifier

Table 99. **SYSADATA options record** *(continued)*

Field	Size	Description
LVLINFO	CL4	User-specified LVLINFO data
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
CallInterface suboptions	XL1	**1... ....** Bit 1 = CallInterface(System) (Windows and AIX only) **.1.. ....** Bit 1 = CallInterface(OptLink) (Windows only) **...1 ....** Bit 1 = CallInterface(Cdecl) (Windows only) **.... 1...** Bit 1 = CallInterface(System(Desc)) (Windows and AIX only) **..1. .111** 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	CL6	Reserved for future use
Converted SO	CL1	Converted SO hexadecimal value
Converted SI	CL1	Converted SI hexadecimal value
Language id	CL2	This field holds the two-character abbreviation (one of EN, UE, JA, or JP) from the LANGUAGE option.

Table 99. **SYSADATA options record** *(continued)*

Field	Size	Description
Reserved	CL8	Reserved for future use
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 ’Library’ user-exit name
ADEXIT name length	HL2	Length of ADATA user-exit name
CURROPT	CL5	CURRENCY option value
Reserved	CL1	Reserved for future use
YEARWINDOW	HL2	YEARWINDOW option value
CODEPAGE	HL2	CODEPAGE CCSID option value
Reserved	CL50	Reserved for future use
LINECNT	HL2	LINECOUNT value
Reserved	CL2	Reserved for future use
BUFSIZE	FL4	BUFSIZE option value
Size value	FL4	SIZE option value
Reserved	FL4	Reserved for future use
Phase residence bits byte 1	XL1	**`1... ....`** Bit 1 = IGYCLIBR in user region **`.1.. ....`** Bit 1 = IGYCSCAN in user region **`..1. ....`** Bit 1 = IGYCDSCN in user region **`...1 ....`** Bit 1 = IGYCGROU in user region **`.... 1...`** Bit 1 = IGYCPSCN in user region **`.... .1..`** Bit 1 = IGYCPANA in user region **`.... ..1.`** Bit 1 = IGYCFGEN in user region **`.... ...1`** Bit 1 = IGYCPGEN in user region

Table 99. **SYSADATA options record** *(continued)*

Field	Size	Description
Phase residence bits byte 2	XL1	`1... ....` Bit 1 = IGYCOPTM in user region `.1.. ....` Bit 1 = IGYCLSTR in user region `..1. ....` Bit 1 = IGYCXREF in user region `...1 ....` Bit 1 = IGYCDMAP in user region `.... 1...` Bit 1 = IGYCASM1 in user region `.... .1..` Bit 1 = IGYCASM2 in user region `.... ..1.` Bit 1 = IGYCDIAG in user region `.... ...1` Reserved for future use
Phase residence bits bytes 3 and 4	XL2	Reserved
Reserved	CL8	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.

Table 100. **SYSADATA 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.

Table 101. **SYSADATA parse tree record**

Field	Size	Description
Node number	FL4	The node number generated by the compiler, starting at 1

Table 101. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
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
		501 Sentence **502** File definition **503** Sort file definition **504** Program-name **505** Program attribute **508** ENVIRONMENT DIVISION clause **509** CLASS attribute **510** METHOD attribute **511** USE statement
		601 Statement **602** Data description clause **603** Data entry **604** File description clause **605** Data entry name **606** Data entry level **607** EXEC entry

Table 101. **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 **720** DATA DIVISION phrase
		801 Phrase **802** ON phrase **803** NOT phrase **804** THEN phrase **805** ELSE phrase **806** Condition **807** Expression **808** Relative indexing **809** EXEC CICS Option **810** Reserved word **811** INITIALIZE REPLACING category

Table 101. **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
		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 101. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
		For Paragraph type: **0001** PROGRAM-ID paragraph **0002** AUTHOR paragraph **0003** INSTALLATION paragraph **0004** DATE-WRITTEN paragraph **0005** SECURITY paragraph **0006** SOURCE-COMPUTER paragraph **0007** OBJECT-COMPUTER paragraph **0008** SPECIAL-NAMES paragraph **0009** FILE-CONTROL paragraph **0010** I-O-CONTROL paragraph **0011** DATE-COMPILED paragraph **0012** CLASS-ID paragraph **0013** METHOD-ID paragraph **0014** REPOSITORY paragraph
		For Environment Division clause type: **0001** WITH DEBUGGING MODE **0002** MEMORY-SIZE **0003** SEGMENT-LIMIT **0004** CURRENCY-SIGN **0005** DECIMAL POINT **0006** PROGRAM COLLATING SEQUENCE **0007** ALPHABET **0008** SYMBOLIC-CHARACTER **0009** CLASS **0010** ENVIRONMENT NAME **0011** SELECT

Table 101. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
		For Data description clause type: **0001** BLANK WHEN ZERO **0002** DATA-NAME OR FILLER **0003** JUSTIFIED **0004** OCCURS **0005** PICTURE **0006** REDEFINES **0007** RENAMES **0008** SIGN **0009** SYNCHRONIZED **0010** USAGE **0011** VALUE **0023** GLOBAL **0024** EXTERNAL

Table 101. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
		For File Description clause type:
		0001 FILE STATUS
		0002 ORGANIZATION
		0003 ACCESS MODE
		0004 RECORD KEY
		0005 ASSIGN
		0006 RELATIVE KEY
		0007 PASSWORD
		0008 PROCESSING MODE
		0009 RECORD DELIMITER
		0010 PADDING CHARACTER
		0011 BLOCK CONTAINS
		0012 RECORD CONTAINS
		0013 LABEL RECORDS
		0014 VALUE OF
		0015 DATA RECORDS
		0016 LINAGE
		0017 ALTERNATE KEY
		0018 LINES AT TOP
		0019 LINES AT BOTTOM
		0020 CODE-SET
		0021 RECORDING MODE
		0022 RESERVE
		0023 GLOBAL
		0024 EXTERNAL
		0025 LOCK

Table 101. **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 101. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
		0023 INVOKE **0024** MERGE **0025** MOVE **0026** MULTIPLY **0027** OPEN **0028** PERFORM **0029** READ **0030** READY **0031** RELEASE **0032** RESET **0033** RETURN **0034** REWRITE **0035** SEARCH **0036** SERVICE **0037** SET **0038** SORT **0039** START **0040** STOP **0041** STRING **0042** SUBTRACT **0043** UNSTRING **0044** EXEC SQL **0144** EXEC CICS **0045** WRITE **0046** XML

Table 101. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
		For Phrase type: **0001** INTO **0002** DELIMITED **0003** INITIALIZE. . .REPLACING **0004** INSPECT. . .ALL **0005** INSPECT. . .LEADING **0006** SET. . .TO **0007** SET. . .UP **0008** SET. . .DOWN **0009** PERFORM. . .TIMES **0010** DIVIDE. . .REMAINDER **0011** INSPECT. . .FIRST **0012** SEARCH. . .VARYING **0013** MORE-LABELS **0014** SEARCH ALL **0015** SEARCH. . .AT END **0016** SEARCH. . .TEST INDEX **0017** GLOBAL **0018** LABEL **0019** DEBUGGING **0020** SEQUENCE **0021** Reserved for future use **0022** Reserved for future use **0023** Reserved for future use **0024** TALLYING **0025** Reserved for future use **0026** ON SIZE ERROR **0027** ON OVERFLOW **0028** ON ERROR **0029** AT END **0030** INVALID KEY

Table 101. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
		0031 END-OF-PAGE **0032** USING **0033** BEFORE **0034** AFTER **0035** EXCEPTION **0036** CORRESPONDING **0037** Reserved for future use **0038** RETURNING **0039** GIVING **0040** THROUGH **0041** KEY **0042** DELIMITER **0043** POINTER **0044** COUNT **0045** METHOD **0046** PROGRAM **0047** INPUT **0048** OUTPUT **0049** I-O **0050** EXTEND **0051** RELOAD **0052** ASCENDING **0053** DESCENDING **0054** DUPLICATES **0055** NATIVE (USAGE) **0056** INDEXED **0057** FROM **0058** FOOTING **0059** LINES AT BOTTOM **0060** LINES AT TOP **0061** XML ENCODING **0062** XML GENERATE XML-DECLARATION **0063** XML GENERATE ATTRIBUTES **0064** XML GENERATE NAMESPACE **0065** XML PARSE PROCESSING

Table 101. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
		For Function identifier type: **0001** COS **0002** LOG **0003** MAX **0004** MIN **0005** MOD **0006** ORD **0007** REM **0008** SIN **0009** SUM **0010** TAN **0011** ACOS **0012** ASIN **0013** ATAN **0014** CHAR **0015** MEAN **0016** SQRT **0017** LOG10 **0018** RANGE **0019** LENGTH **0020** MEDIAN **0021** NUMVAL **0022** RANDOM **0023** ANNUITY **0024** INTEGER **0025** ORD-MAX **0026** ORD-MIN **0027** REVERSE **0028** MIDRANGE **0029** NUMVAL-C **0030** VARIANCE **0031** FACTORIAL **0032** LOWER-CASE

Table 101. **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 **0046** UNDATE **0047** DATEVAL **0048** YEARWINDOW **0049** DISPLAY-OF **0050** NATIONAL-OF
		For Special Register type: **0001** ADDRESS OF **0002** LENGTH OF
		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

Table 101. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
		For Identifier, Data-name, Index-name, Condition-name or Mnemonic-name type: **0001** REFERENCED **0002** CHANGED **0003** REFERENCED & CHANGED
		For Initialize literal type: **0001** ALPHABETIC **0002** ALPHANUMERIC **0003** NUMERIC **0004** ALPHANUMERIC-EDITED **0005** NUMERIC-EDITED **0006** DBCS/EGCS **0007** NATIONAL **0008** NATIONAL-EDITED
		For Procedure-name type: **0001** SECTION **0002** PARAGRAPH
		For Reserved word at identifier level type: **0001** ROUNDED **0002** TRUE **0003** ON **0004** OFF **0005** SIZE **0006** DATE **0007** DAY **0008** DAY-OF-WEEK **0009** TIME **0010** WHEN-COMPILED **0011** PAGE **0012** DATE YYYYMMDD **0013** DAY YYYYDDD

Table 101. **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.
Symbol ID	FL4	The Symbol ID of the node, if it is a user-name of one of the following types: • 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

Table 101. **SYSADATA parse tree record** *(continued)*

Field	Size	Description
Last token number	FL4	The number of the last token associated with the node
Reserved	FL4	Reserved for future use
Flags	CL1	Information about the node: **X'80'** Reserved **X'40'** Generated node, no tokens
Reserved	CL3	Reserved for future use

Token record: X'0030'

The compiler does not generate token records for any lines that are treated as comment lines, which include, but are not limited to, items in the following list.

- Comment lines, which are lines that have an asterisk (*) or a slash (/) in column 7
- The following compiler-directing statements:
  - `*CBL` (`*CONTROL`)
  - `BASIS`
  - `>>CALLINT`
  - `COPY`
  - `DELETE`
  - `EJECT`
  - `INSERT`
  - `REPLACE`
  - `SKIP1`
  - `SKIP2`
  - `SKIP3`
  - `TITLE`
- Debugging lines, which are lines that have a `D` in column 7, if `WITH DEBUGGING MODE` is not specified

Table 102. **SYSADATA token record**

Field	Size	Description
Token number	FL4	The token number within the source file generated by the compiler, starting at 1. Any copybooks have already been included in the source.

Table 102. **SYSADATA token record** *(continued)*

Field	Size	Description
Token code	HL2	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: **0001** ACCEPT **0002** ADD **0003** ALTER **0004** CALL **0005** CANCEL **0007** CLOSE **0009** COMPUTE **0011** DELETE **0013** DISPLAY **0014** DIVIDE **0017** READY **0018** END-PERFORM **0019** ENTER **0020** ENTRY **0021** EXIT **0022** EXEC EXECUTE **0023** GO **0024** IF **0025** INITIALIZE **0026** INVOKE **0027** INSPECT **0028** MERGE **0029** MOVE

Table 102. **SYSADATA token record** *(continued)*

Field	Size	Description
		0030 MULTIPLY
		0031 OPEN
		0032 PERFORM
		0033 READ
		0035 RELEASE
		0036 RETURN
		0037 REWRITE
		0038 SEARCH
		0040 SET
		0041 SORT
		0042 START
		0043 STOP
		0044 STRING
		0045 SUBTRACT
		0048 UNSTRING
		0049 USE
		0050 WRITE
		0051 CONTINUE
		0052 END-ADD
		0053 END-CALL
		0054 END-COMPUTE
		0055 END-DELETE
		0056 END-DIVIDE
		0057 END-EVALUATE
		0058 END-IF
		0059 END-MULTIPLY
		0060 END-READ
		0061 END-RETURN
		0062 END-REWRITE
		0063 END-SEARCH
		0064 END-START
		0065 END-STRING
		0066 END-SUBTRACT
		0067 END-UNSTRING
		0068 END-WRITE
		0069 GOBACK

Table 102. **SYSADATA token record** *(continued)*

Field	Size	Description	
		0070	EVALUATE
		0071	RESET
		0072	SERVICE
		0073	END-INVOKE
		0074	END-EXEC
		0075	XML
		0076	END-XML
		0099	FOREIGN-VERB
		0101	DATA-NAME
		0105	DASHED-NUM
		0106	DECIMAL
		0107	DIV-SIGN
		0108	EQ
		0109	EXPONENTIATION
		0110	GT
		0111	INTEGER
		0112	LT
		0113	LPAREN
		0114	MINUS-SIGN
		0115	MULT-SIGN
		0116	NONUMLIT
		0117	PERIOD
		0118	PLUS-SIGN
		0121	RPAREN
		0122	SIGNED-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 102. **SYSADATA token record** *(continued)*

Field	Size	Description
		0132 GE
		0133 IDREF
		0134 EXPREF
		0136 CICS
		0137 NEW
		0138 NATIONAL-LIT
		0200 ADDRESS
		0201 ADVANCING
		0202 AFTER
		0203 ALL
		0204 ALPHABETIC
		0205 ALPHANUMERIC
		0206 ANY
		0207 AND
		0208 ALPHANUMERIC-EDITED
		0209 BEFORE
		0210 BEGINNING
		0211 FUNCTION
		0212 CONTENT
		0213 CORR
		CORRESPONDING
		0214 DAY
		0215 DATE
		0216 DEBUG-CONTENTS
		0217 DEBUG-ITEM
		0218 DEBUG-LINE
		0219 DEBUG-NAME
		0220 DEBUG-SUB-1
		0221 DEBUG-SUB-2
		0222 DEBUG-SUB-3
		0223 DELIMITED
		0224 DELIMITER
		0225 DOWN

Table 102. **SYSADATA token record** *(continued)*

Field	Size	Description	
		0226	NUMERIC-EDITED
		0227	XML-EVENT
		0228	END-OF-PAGE
			EOP
		0229	EQUAL
		0230	ERROR
		0231	XML-NTEXT
		0232	EXCEPTION
		0233	EXTEND
		0234	FIRST
		0235	FROM
		0236	GIVING
		0237	GREATER
		0238	I-O
		0239	IN
		0240	INITIAL
		0241	INTO
		0242	INVALID
		0243	SQL
		0244	LESS
		0245	LINAGE-COUNTER
		0246	XML-TEXT
		0247	LOCK
		0248	GENERATE
		0249	NEGATIVE
		0250	NEXT
		0251	NO
		0252	NOT
		0253	NUMERIC
		0254	KANJI
		0255	OR
		0256	OTHER
		0257	OVERFLOW
		0258	PAGE
		0259	CONVERTING

Table 102. **SYSADATA token record** *(continued)*

Field	Size	Description
		0260 POINTER
		0261 POSITIVE
		0262 DBCS
		0263 PROCEDURES
		0264 PROCEED
		0265 REFERENCES
		0266 DAY-OF-WEEK
		0267 REMAINDER
		0268 REMOVAL
		0269 REPLACING
		0270 REVERSED
		0271 REWIND
		0272 ROUNDED
		0273 RUN
		0274 SENTENCE
		0275 STANDARD
		0276 RETURN-CODE
		SORT-CORE-SIZE
		SORT-FILE-SIZE
		SORT-MESSAGE
		SORT-MODE-SIZE
		SORT-RETURN
		TALLY
		XML-CODE
		0277 TALLYING
		0278 SUM
		0279 TEST
		0280 THAN
		0281 UNTIL
		0282 UP
		0283 UPON
		0284 VARYING
		0285 RELOAD
		0286 TRUE

Table 102. **SYSADATA token record** *(continued)*

Field	Size	Description
		0287 THEN
		0288 RETURNING
		0289 ELSE
		0290 SELF
		0291 SUPER
		0292 WHEN-COMPILED
		0293 ENDING
		0294 FALSE
		0295 REFERENCE
		0296 NATIONAL-EDITED
		0297 COM-REG
		0298 ALPHABETIC-LOWER
		0299 ALPHABETIC-UPPER
		0301 REDEFINES
		0302 OCCURS
		0303 SYNC
		SYNCHRONIZED
		0304 MORE-LABELS
		0305 JUST
		JUSTIFIED
		0306 SHIFT-IN
		0307 BLANK
		0308 VALUE
		0309 COMP
		COMPUTATIONAL
		0310 COMP-1
		COMPUTATIONAL-1
		0311 COMP-3
		COMPUTATIONAL-3
		0312 COMP-2
		COMPUTATIONAL-2
		0313 COMP-4
		COMPUTATIONAL-4
		0314 DISPLAY-1
		0315 SHIFT-OUT

Table 102. **SYSADATA token record** *(continued)*

Field	Size	Description
		0316 INDEX
		0317 USAGE
		0318 SIGN
		0319 LEADING
		0320 SEPARATE
		0321 INDEXED
		0322 LEFT
		0323 RIGHT
		0324 PIC
		PICTURE
		0325 VALUES
		0326 GLOBAL
		0327 EXTERNAL
		0328 BINARY
		0329 PACKED-DECIMAL
		0330 EGCS
		0331 PROCEDURE-POINTER
		0332 COMP-5
		COMPUTATIONAL-5
		0333 FUNCTION-POINTER
		0334 TYPE
		0335 JNIENVPTR
		0336 NATIONAL
		0337 GROUP-USAGE
		0401 HIGH-VALUE
		HIGH-VALUES
		0402 LOW-VALUE
		LOW-VALUES
		0403 QUOTE
		QUOTES
		0404 SPACE
		SPACES
		0405 ZERO

Table 102. **SYSADATA token record** *(continued)*

Field	Size	Description
		0406 ZEROES
		ZEROS
		0407 NULL
		NULLS
		0501 BLOCK
		0502 BOTTOM
		0505 CHARACTER
		0506 CODE
		0507 CODE-SET
		0514 FILLER
		0516 FOOTING
		0520 LABEL
		0521 LENGTH
		0524 LINAGE
		0526 OMITTED
		0531 RENAMES
		0543 TOP
		0545 TRAILING
		0549 RECORDING
		0601 INHERITS
		0603 RECURSIVE
		0701 ACCESS
		0702 ALSO
		0703 ALTERNATE
		0704 AREA
		AREAS
		0705 ASSIGN
		0707 COLLATING
		0708 COMMA
		0709 CURRENCY
		0710 CLASS
		0711 DECIMAL-POINT
		0712 DUPLICATES
		0713 DYNAMIC
		0714 EVERY

Table 102. **SYSADATA token record** *(continued)*

Field	Size	Description
		0716 MEMORY
		0717 MODE
		0718 MODULES
		0719 MULTIPLE
		0720 NATIVE
		0721 OFF
		0722 OPTIONAL
		0723 ORGANIZATION
		0724 POSITION
		0725 PROGRAM
		0726 RANDOM
		0727 RELATIVE
		0728 RERUN
		0729 RESERVE
		0730 SAME
		0731 SEGMENT-LIMIT
		0732 SELECT
		0733 SEQUENCE
		0734 SEQUENTIAL
		0736 SORT-MERGE
		0737 STANDARD-1
		0738 TAPE
		0739 WORDS
		0740 PROCESSING
		0741 APPLY
		0742 WRITE-ONLY
		0743 COMMON
		0744 ALPHABET
		0745 PADDING
		0746 SYMBOLIC
		0747 STANDARD-2
		0748 OVERRIDE
		0750 PASSWORD

Table 102. **SYSADATA token record** *(continued)*

Field	Size	Description
		0801 ARE
		IS
		0802 ASCENDING
		0803 AT
		0804 BY
		0805 CHARACTERS
		0806 CONTAINS
		0808 COUNT
		0809 DEBUGGING
		0810 DEPENDING
		0811 DESCENDING
		0812 DIVISION
		0814 FOR
		0815 ORDER
		0816 INPUT
		0817 REPLACE
		0818 KEY
		0819 LINE
		LINES
		0821 OF
		0822 ON
		0823 OUTPUT
		0825 RECORD
		0826 RECORDS
		0827 REEL
		0828 SECTION
		0829 SIZE
		0830 STATUS
		0831 THROUGH
		THRU
		0832 TIME
		0833 TIMES
		0834 TO
		0836 UNIT

Table 102. **SYSADATA token record** *(continued)*

Field	Size	Description
		0837 USING
		0838 WHEN
		0839 WITH
		0901 PROCEDURE
		0902 DECLARATIVES
		0903 END
		1001 DATA
		1002 FILE
		1003 FD
		1004 SD
		1005 WORKING-STORAGE
		1006 LOCAL-STORAGE
		1007 LINKAGE
		1101 ENVIRONMENT
		1102 CONFIGURATION
		1103 SOURCE-COMPUTER
		1104 OBJECT-COMPUTER
		1105 SPECIAL-NAMES
		1106 REPOSITORY
		1107 INPUT-OUTPUT
		1108 FILE-CONTROL
		1109 I-O-CONTROL
		1201 ID
		IDENTIFICATION
		1202 PROGRAM-ID
		1203 AUTHOR
		1204 INSTALLATION
		1205 DATE-WRITTEN
		1206 DATE-COMPILED
		1207 SECURITY
		1208 CLASS-ID
		1209 METHOD-ID
		1210 METHOD
		1211 FACTORY

Table 102. **SYSADATA token record** *(continued)*

Field	Size	Description
		1212 OBJECT **2020** TRACE **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 **5001** PARSE **5002** AUTOMATIC **5003** PREVIOUS **9999** COBOL
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
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(*n*)	The actual token string

Source error record: X'0032'

The following table shows the contents of the source error record.

Table 103. **SYSADATA source error record**

Field	Size	Description
Statement number	FL4	The statement number of the statement in error
Error identifier	CL16	The error message identifier (left-justified and padded with blanks)
Error severity	HL2	The severity of the error
Error message length	HL2	The length of the error message text
Line position	XL1	The line position indicator provided in FIPS messages
Reserved	CL7	Reserved for future use
Error message	CL(*n*)	The error message text

Source record: X'0038'

The following table shows the contents of the source record.

Table 104. **SYSADATA source record**

Field	Size	Description
Line number	FL4	The listing line number of the source record
Input record number	FL4	The input source record number in the current input file
Primary file number	HL2	The input file's assigned sequence number if this record is from the primary input file. (Refer to the Input file *n* 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 *n* field in the Library record.)
Reserved	CL8	Reserved for future use
Parent record number	FL4	The parent source record number. This will be the record number of the `COPY|BASIS` statement.
Parent primary file number	HL2	The parent file's assigned sequence number if the parent of this record is from the primary input file. (Refer to the Input file *n* 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 *n* 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(*n*)	

COPY REPLACING record: X’0039’

One COPY REPLACING type record will be emitted each time a REPLACING action takes place. That is, whenever *operand-1* of the REPLACING phrase is matched with text in the copybook, a COPY REPLACING TEXT record will be written.

The following table shows the contents of the COPY REPLACING record.

Table 105. **SYSADATA COPY REPLACING record**

Field	Size	Description
Starting line number of replaced string	FL4	The listing line number of the start of the text that resulted from REPLACING
Starting column number of replaced string	FL4	The listing column number of the start of the text that resulted from REPLACING
Ending line number of replaced string	FL4	The listing line number of the end of the text that resulted from REPLACING
Ending column number of replaced string	FL4	The listing column number of the end of the text that resulted from REPLACING
Starting line number of original string	FL4	The source file line number of the start of the text that was changed by REPLACING
Starting column number of original string	FL4	The source file column number of the start of the text that was changed by REPLACING
Ending line number of original string	FL4	The source file line number of the end of the text that was changed by REPLACING
Ending column number of original string	FL4	The source file column number of the end of the text that was changed by REPLACING

Symbol record: X’0042’

The following table shows the contents of the symbol record.

Table 106. **SYSADATA symbol record**

Field	Size	Description
Symbol ID	FL4	Unique ID of symbol
Line number	FL4	The listing line number of the source record in which the symbol is defined or declared
Level	XL1	True level-number of symbol (or relative level-number of a data item within a structure). For COBOL, this can be in the range 01-49, 66 (for RENAMES items), 77, or 88 (for condition items).
Qualification indicator	XL1	**X'00'** Unique name; no qualification needed. **X'01'** This data item needs qualification. The name is not unique within the program. This field applies only when this data item is *not* the level-01 name.

Table 106. **SYSADATA symbol record** *(continued)*

Field	Size	Description
Symbol type	XL1	**X'68'** Class-name (Class-ID) **X'58'** Method-name **X'40'** Data-name **X'20'** Procedure-name **X'10'** Mnemonic-name **X'08'** Program-name **X'81'** Reserved The following are ORed into the above types, when applicable: **X'04'** External **X'02'** Global
Symbol attribute	XL1	**X'01'** Numeric **X'02'** Elementary character of one of these classes: • Alphabetic • Alphanumeric • DBCS • National **X'03'** Group **X'04'** Pointer **X'05'** Index data item **X'06'** Index-name **X'07'** Condition **X'0F'** File **X'10'** Sort file **X'17'** Class-name (repository) **X'18'** Object reference

Table 106. **SYSADATA symbol record** *(continued)*

Field	Size	Description
Clauses	XL1	Clauses specified in symbol definition. 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'`): **`1... ....`** Value **`.1.. ....`** Indexed **`..1. ....`** Redefines **`...1 ....`** Renames **`.... 1...`** Occurs **`.... .1..`** Has Occurs keys **`.... ..1.`** Occurs Depending On **`.... ...1`** Occurs in parent For both file types: **`1... ....`** Select **`.1.. ....`** Assign **`..1. ....`** Rerun **`...1 ....`** Same area **`.... 1...`** Same record area **`.... .1..`** Recording mode **`.... ..1.`** Reserved **`.... ...1`** Record

Table 106. **SYSADATA symbol record** *(continued)*

Field	Size	Description
		For mnemonic-name symbols:
		01 CSP
		02 C01
		03 C02
		04 C03
		05 C04
		06 C05
		07 C06
		08 C07
		09 C08
		10 C09
		11 C10
		12 C11
		13 C12
		14 S01
		15 S02
		16 S03
		17 S04
		18 S05
		19 CONSOLE
		20 SYSIN|SYSIPT
		22 SYSOUT|SYSLST|SYSLIST
		24 SYSPUNCH|SYSPCH
		26 UPSI-0
		27 UPSI-1
		28 UPSI-2
		29 UPSI-3
		30 UPSI-4
		31 UPSI-5
		32 UPSI-6
		33 UPSI-7
		34 AFP-5A

Table 106. **SYSADATA symbol record** *(continued)*

Field	**Size**	**Description**

Data flags 1	XL1	For both file types, and for symbols that have a symbol attribute of Numeric (`X'01'`), Elementary character (`X'02'`), Group (`X'03'`), Pointer (`X'04'`), Index data item (`X'05'`), or Object reference (`X'18'`):
		`1... ....`
		Redefined
		`.1.. ....`
		Renamed
		`..1. ....`
		Synchronized
		`...1 ....`
		Implicitly redefined
		`.... 1...`
		Date field
		`.... .1..`
		Implicit redefines
		`.... ..1.`
		`FILLER`
		`.... ...1`
		Level 77

Table 106. **SYSADATA symbol record** *(continued)*

Field	**Size**	**Description**
Data flags 2	XL1	For symbols that have a symbol attribute of Numeric (`X'01'`): **`1... ....`** Binary **`.1.. ....`** External floating point (of `USAGE DISPLAY` or `USAGE NATIONAL`) **`..1. ....`** Internal floating point **`...1 ....`** Packed **`.... 1...`** External decimal (of `USAGE DISPLAY` or `USAGE NATIONAL`) **`.... .1..`** Scaled negative **`.... ..1.`** Numeric edited (of `USAGE DISPLAY` or `USAGE NATIONAL`) **`.... ...1`** Reserved for future use For symbols that have a symbol attribute of Elementary character (`X'02'`) or Group (`X'03'`): **`1... ....`** Alphabetic **`.1.. ....`** Alphanumeric **`..1. ....`** Alphanumeric edited **`...1 ....`** Group contains its own ODO object **`.... 1...`** DBCS item **`.... .1..`** Group variable length **`.... ..1.`** EGCS item **`.... ...1`** EGCS edited

Table 106. **SYSADATA symbol record** *(continued)*

Field	Size	Description
		For both file types: `1... ....` Object of ODO in record `.1.. ....` Subject of ODO in record `..1. ....` Sequential access `...1 ....` Random access `.... 1...` Dynamic access `.... .1..` Locate mode `.... ..1.` Record area `.... ...1` Reserved for future use Field will be zero for all other data types.
Data flags 3	XL1	For both file types: `1... ....` All records are the same length `.1.. ....` Fixed length `..1. ....` Variable length `...1 ....` Undefined `.... 1...` Spanned `.... .1..` Blocked `.... ..1.` Apply write only `.... ...1` Same sort merge area Field will be zero for all other data types.

Table 106. **SYSADATA symbol record** *(continued)*

Field	Size	Description
File organization	XL1	For both file types: **1... ....** QSAM **.1.. ....** ASCII **..1. ....** Standard label **...1 ....** User label **.... 1...** VSAM sequential **.... .1..** VSAM indexed **.... ..1.** VSAM relative **.... ...1** Line sequential Field will be zero for all other data types.
USAGE clause	FL1	**X'00'** USAGE IS DISPLAY **X'01'** USAGE IS COMP-1 **X'02'** USAGE IS COMP-2 **X'03'** USAGE IS PACKED-DECIMAL or USAGE IS COMP-3 **X'04'** USAGE IS BINARY, USAGE IS COMP, or USAGE IS COMP-4 **X'05'** USAGE IS DISPLAY-1 **X'06'** USAGE IS POINTER **X'07'** USAGE IS INDEX **X'08'** USAGE IS PROCEDURE-POINTER **X'09'** USAGE IS OBJECT-REFERENCE **X'0B'** NATIONAL **X'0A'** FUNCTION-POINTER
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.

Table 106. **SYSADATA symbol record** *(continued)*

Field	Size	Description
Size	FL4	The size of this data item. The actual number of bytes this item occupies in storage. If a DBCS item, the number is in bytes, not characters. For variable-length items, this field will reflect the maximum size of storage reserved for this item by the compiler. Also known as the "Length attribute."
Precision	FL1	The precision of a fixed or float data item
Scale	FL1	The scale factor of a fixed data item. This is the number of digits to the right of the decimal point.

Table 106. **SYSADATA symbol record** *(continued)*

Field	Size	Description
Base locator type	FL1	For host: **01** Base Locator File **02** Base Locator Working-Storage **03** Base Locator Linkage Section **05** Base Locator special regs **07** Indexed by variable **09** COMREG special reg **10** UPSI switch **13** Base Locator for Varloc items **14** Base Locator for Extern data **15** Base Locator alphanumeric FUNC **16** Base Locator alphanumeric EVAL **17** Base Locator for Object data **19** Base Locator for Local-Storage **20** Factory data **21** `XML-TEXT` and `XML-NTEXT` For Windows and AIX: **01** Base Locator File **02** Base Locator Linkage Section **03** Base Locator for Varloc items **04** Base Locator for Extern data **05** Base Locator for Object data **06** `XML-TEXT` and `XML-NTEXT` **10** Base Locator Working-Storage **11** Base Locator special regs **12** Base Locator alphanumeric FUNC **13** Base Locator alphanumeric EVAL **14** Indexed by variable **16** COMREG special reg **17** UPSI switch **18** Factory data **22** Base Locator for Local-Storage

Table 106. **SYSADATA symbol record** *(continued)*

Field	Size	Description
Date format	FL1	Date format: **01** YY **02** YYXX **03** YYXXXX **04** YYXXX **05** YYYY **06** YYYYXX **07** YYYYXXXX **08** YYYYXXX **09** YYX **10** YYYYX **22** XXYY **23** XXXXYY **24** XXXYY **26** XXYYYY **27** XXXXYYYY **28** XXXYYYY **29** XYY **30** XYYYY
Data flags 4	XL1	For symbols that have a symbol attribute of Numeric (X'01'): **1... ....** Numeric national For symbols that have a symbol attribute of Elementary character (X'02'): **1... ....** National **.1.. ....** National edited For symbols that have a symbol attribute of Group (X'03'): **1... ....** Group-Usage National
Reserved	FL3	Reserved for future use

Table 106. **SYSADATA symbol record** *(continued)*

Field	Size	Description
Addressing information	FL4	For host, the Base Locator number and displacement: **Bits 0-4** Unused **Bits 5-19** Base Locator (BL) number **Bits 20-31** Displacement off Base Locator For Windows and AIX, the W-code SymId.
Structure displacement	AL4	Offset of symbol within structure. This offset is set to 0 for variably located items.
Parent displacement	AL4	Byte offset from immediate parent of the item being defined.
Parent ID	FL4	The symbol ID of the immediate parent of the item being defined.
Redefined ID	FL4	The symbol ID of the data item that this item redefines, if applicable.
Start-renamed ID	FL4	If this item is a level-66 item, the symbol ID of the starting COBOL data item that this item renames. If not a level-66 item, this field is set to 0.
End-renamed ID	FL4	If this item is a level-66 item, the symbol ID of the ending COBOL data item that this item renames. If not a level-66 item, this field is set to 0.
Program-name symbol ID	FL4	ID of the program-name of the program or the class-name of the class where this symbol is defined.
`OCCURS` minimum 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
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.
Reserved	CL8	Reserved for future use
Value pairs count	HL2	Count of value pairs
Symbol name length	HL2	Number of characters in the symbol name

Table 106. **SYSADATA symbol record** *(continued)*

Field	Size	Description
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 *exactly* 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
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

Table 106. **SYSADATA symbol record** *(continued)*

Field	**Size**	**Description**
...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 107. **SYSADATA symbol cross-reference record**

Field	**Size**	**Description**
Symbol length	HL2	The length of the symbol
Statement definition	FL4	The statement number where the symbol is defined or declared **For VERB XREF only:** Verb count - total number of references to this verb.
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’** Verb **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.

Table 107. **SYSADATA symbol cross-reference record** *(continued)*

Field	Size	Description
...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)
...Statement number	XL4	The statement number on which the symbol or verb is referenced
1. The reference flag field and the statement number field occur as many times as the number of references field dictates. For example, if there is a value of 10 in the number of references field, there will be 10 occurrences of the reference flag and statement number pair for data-name, procedure, or program symbols, or 10 occurrences of the statement number for verbs. 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.		

Nested program record: X'0046'

The following table shows the contents of the nested program record.

Table 108. **SYSADATA nested program record**

Field	Size	Description
Statement definition	FL4	The statement number where the symbol is defined or declared
Nesting level	XL1	Program nesting level
Program attributes	XL1	**1... ....** Initial **.1.. ....** Common **..1. ....** `PROCEDURE DIVISION using` **...1 1111** Reserved for future use
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 109. **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
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

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.
2. If COPY/BASIS members are retrieved from different libraries, a library record is written to the SYSADATA file for each unique library.

Statistics record: X'0090'

The following table shows the contents of the statistics record.

Table 110. **SYSADATA statistics record**

Field	Size	Description
Source records	FL4	The number of source records processed
DATA DIVISION statements	FL4	The number of DATA DIVISION statements processed
PROCEDURE DIVISION statements	FL4	The number of PROCEDURE DIVISION statements processed
Compilation number	HL2	Batch compilation number
Error severity	XL1	The highest error message severity

Table 110. **SYSADATA statistics record** *(continued)*

Field	Size	Description
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

Table 111. **SYSADATA EVENTS TIMESTAMP record layout**

Field	Size	Description
Header	CL12	Standard ADATA record header
Record length	HL2	Length of following EVENTS record data (excluding this halfword)
EVENTS record type TIMESTAMP record	CL12	C’TIMESTAMP’
Blank separator	CL1	
Revision level	XL1	
Blank separator	CL1	
Date	XL8	YYYYMMDD
Hour	XL2	HH
Minutes	XL2	MI
Seconds	XL2	SS

Table 112. **SYSADATA EVENTS PROCESSOR record layout**

Field	Size	Description
Header	CL12	Standard ADATA record header
Record length	HL2	Length of following EVENTS record data (excluding this halfword)

Table 112. **SYSADATA EVENTS PROCESSOR record layout** *(continued)*

Field	Size	Description
EVENTS record type PROCESSOR record	CL9	C'PROCESSOR'
Blank separator	CL1	
Revision level	XL1	
Blank separator	CL1	
Output file ID	XL1	
Blank separator	CL1	
Line-class indicator	XL1	

Table 113. **SYSADATA EVENTS FILE END record layout**

Field	Size	Description
Header	CL12	Standard ADATA record header
Record length	HL2	Length of following EVENTS record data (excluding this halfword)
EVENTS record type FILE END record	CL7	C'FILEEND'
Blank separator	CL1	
Revision level	XL1	
Blank separator	CL1	
Input file ID	XL1	
Blank separator	CL1	
Expansion indicator	XL1	

Table 114. **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 115. **SYSADATA EVENTS FILE ID record layout**

Field	Size	Description
Header	CL12	Standard ADATA record header

Table 115. **SYSADATA EVENTS FILE ID record layout** *(continued)*

Field	Size	Description
Record length	HL2	Length of following EVENTS record data (excluding this halfword)
EVENTS record type FILE ID record	CL7	C'FILEID'
Blank separator	CL1	
Revision level	XL1	
Blank separator	CL1	
Input source file ID	XL1	File ID of source file
Blank separator	CL1	
Reference indicator	XL1	
Blank separator	CL1	
Source file name length	H2	
Blank separator	CL1	
Source file name	CL(n)	

Table 116. **SYSADATA EVENTS ERROR record layout**

Field	Size	Description
Header	CL12	Standard ADATA record header
Record length	HL2	Length of following EVENTS record data (excluding this halfword)
EVENTS record type ERROR record	CL5	C'ERROR'
Blank separator	CL1	
Revision level	XL1	
Blank separator	CL1	
Input source file ID	XL1	File ID of source file
Blank separator	CL1	
Annot class	XL1	Annot-class message placement
Blank separator	CL1	
Error input record number	XL10	
Blank separator	CL1	
Error start line number	XL10	
Blank separator	CL1	
Error token start number	XL1	Column number of error token start
Blank separator	CL1	
Error end line number	XL10	
Blank separator	CL1	

Table 116. **SYSADATA EVENTS ERROR record layout** *(continued)*

Field	Size	Description
Error token end number	XL1	Column number of error token end
Blank separator	CL1	
Error message ID number	XL9	
Blank separator	CL1	
Error message severity code	XL1	
Blank separator	CL1	
Error message severity level number	XL2	
Blank separator	CL1	
Error message length	HL3	
Blank separator	CL1	
Error message text	CL(n)	

Appendix J. Runtime messages

Messages for COBOL for AIX contain a message prefix, message number, severity code, and descriptive text.

The message prefix is always IWZ. The severity code is either I (information), W (warning), S (severe), or C (critical). The message text provides a brief explanation of the condition.

```
IWZ2519S  The seconds value in a CEEISEC call was not recognized.
```

In the example message above:
- The message prefix is IWZ.
- The message number is 2519.
- The severity code is S.
- The message text is ″The seconds value in a CEEISEC call was not recognized.″

The date and time callable services messages also contain a symbolic feedback code, which represents the first 8 bytes of a 12-byte condition token. You can think of the symbolic feedback code as the nickname for a condition. The callable services messages contain a four-digit message number.

To see a complete list at your terminal of the messages in the message catalog, issue the command:

```
dspcat cobol2run.cat
```

To see the content of only the message whose message number is *nnnn*, issue the command:

```
dspcat cobol2run.cat nnnn
```

When running your application from the command line, you can capture any runtime messages by redirecting stdout and stderr to a file. For example:

```
program-name program-arguments >combined-output-file 2>&1
```

The following example shows how to write the output to separate files:

```
program-name program-arguments >output-file 2>error-file
```

Table 117. **Runtime messages**

Message number	Message text
"IWZ006S" on page 732	The reference to table *table-name* by verb number *verb-number* on line *line-number* addressed an area outside the region of the table.
"IWZ007S" on page 733	The reference to variable-length group *group-name* by verb number *verb-number* on line *line-number* addressed an area outside the maximum defined length of the group.
"IWZ012I" on page 733	Invalid run unit termination occurred while sort or merge is running.
"IWZ013S" on page 733	Sort or merge requested while sort or merge is running in a different thread.

Table 117. **Runtime messages** *(continued)*

"IWZ026W" on page 734	The SORT-RETURN special register was never referenced, but the current content indicated the sort or merge operation in program *program-name* on line number *line-number* was unsuccessful. The sort or merge return code was *return code*.
"IWZ029S" on page 734	Argument-1 for function *function-name* in program *program-name* at line *line-number* was less than zero.
"IWZ030S" on page 734	Argument-2 for function *function-name* in program *program-name* at line *line-number* was not a positive integer.
"IWZ036W" on page 734	Truncation of high order digit positions occurred in program *program-name* on line number *line-number*.
"IWZ037I" on page 735	The flow of control in program *program-name* proceeded beyond the last line of the program. Control returned to the caller of the program *program-name*.
"IWZ038S" on page 735	A reference modification length value of *reference-modification-value* on line *line-number* which was not equal to 1 was found in a reference to data item *data-item*.
"IWZ039S" on page 735	An invalid overpunched sign was detected.
"IWZ040S" on page 736	An invalid separate sign was detected.
"IWZ045S" on page 736	Unable to invoke method *method-name* on line number *line number* in program *program-name*.
"IWZ047S" on page 736	Unable to invoke method *method-name* on line number *line number* in class *class-name*.
"IWZ048W" on page 737	A negative base was raised to a fractional power in an exponentiation expression. The absolute value of the base was used.
"IWZ049W" on page 737	A zero base was raised to a zero power in an exponentiation expression. The result was set to one.
"IWZ050S" on page 737	A zero base was raised to a negative power in an exponentiation expression.
"IWZ051W" on page 738	No significant digits remain in a fixed-point exponentiation operation in program *program-name* due to excessive decimal positions specified in the operands or receivers.
"IWZ053S" on page 738	An overflow occurred on conversion to floating point.
"IWZ054S" on page 738	A floating-point exception occurred.
"IWZ055W" on page 738	An underflow occurred on conversion to floating point. The result was set to zero.
"IWZ058S" on page 739	Exponent overflow occurred.
"IWZ059W" on page 739	An exponent with more than nine digits was truncated.
"IWZ060W" on page 739	Truncation of high order digit positions occurred.
"IWZ061S" on page 739	Division by zero occurred.

Table 117. **Runtime messages** *(continued)*

"IWZ063S" on page 740	An invalid sign was detected in a numeric edited sending field in *program-name* on line number *line-number*.
"IWZ064S" on page 740	A recursive call to active program *program-name* in compilation unit *compilation-unit* was attempted.
"IWZ065I" on page 740	A CANCEL of active program *program-name* in compilation unit *compilation-unit* was attempted.
"IWZ066S" on page 740	The length of external data record *data-record* in program *program-name* did not match the existing length of the record.
"IWZ071S" on page 741	ALL subscripted table reference to table *table-name* by verb number *verb-number* on line *line-number* had an ALL subscript specified for an OCCURS DEPENDING ON dimension, and the object was less than or equal to 0.
"IWZ072S" on page 741	A reference modification start position value of *reference-modification-value* on line *line-number* referenced an area outside the region of data item *data-item*.
"IWZ073S" on page 741	A nonpositive reference modification length value of *reference-modification-value* on line *line-number* was found in a reference to data item *data-item*.
"IWZ074S" on page 742	A reference modification start position value of *reference-modification-value* and length value of *length* on line *line-number* caused reference to be made beyond the rightmost character of data item *data-item*.
"IWZ075S" on page 742	Inconsistencies were found in EXTERNAL file *file-name* in program *program-name*. The following file attributes did not match those of the established external file: *attribute-1 attribute-2 attribute-3 attribute-4 attribute-5 attribute-6 attribute-7*.
"IWZ076W" on page 742	The number of characters in the INSPECT REPLACING CHARACTERS BY data-name was not equal to one. The first character was used.
"IWZ077W" on page 742	The lengths of the INSPECT data items were not equal. The shorter length was used.
"IWZ078S" on page 743	ALL subscripted table reference to table *table-name* by verb number *verb-number* on line *line-number* will exceed the upper bound of the table.
"IWZ096C" on page 743	Dynamic call of program *program-name* failed. Message variants include: • A load of module *module-name* failed with an error code of *error-code*. • A load of module *module-name* failed with a return code of *return-code*. • Dynamic call of program *program-name* failed. Insufficient resources. • Dynamic call of program *program-name* failed. COBPATH not found in environment. • Dynamic call of program *program-name* failed. Entry *entry-name* not found. • Dynamic call failed. The name of the target program does not contain any valid characters. • Dynamic call of program *program-name* failed. The load module *load-module* could not be found in the directories identified in the COBPATH environment variable.

Table 117. **Runtime messages** *(continued)*

"IWZ097S" on page 744	Argument-1 for function *function-name* contained no digits.
"IWZ100S" on page 744	Argument-1 for function *function-name* was less than or equal to -1.
"IWZ103S" on page 744	Argument-1 for function *function-name* was less than zero or greater than 99.
"IWZ104S" on page 744	Argument-1 for function *function-name* was less than zero or greater than 99999.
"IWZ105S" on page 744	Argument-1 for function *function-name* was less than zero or greater than 999999.
"IWZ151S" on page 745	Argument-1 for function *function-name* contained more than 18 digits.
"IWZ152S" on page 745	Invalid character *character* was found in column *column-number* in argument-1 for function *function-name*.
"IWZ155S" on page 745	Invalid character *character* was found in column *column-number* in argument-2 for function *function-name*.
"IWZ156S" on page 745	Argument-1 for function *function-name* was less than zero or greater than 28.
"IWZ157S" on page 745	The length of Argument-1 for function *function-name* was not equal to 1.
"IWZ158S" on page 746	Argument-1 for function *function-name* was less than zero or greater than 29.
"IWZ159S" on page 746	Argument-1 for function *function-name* was less than 1 or greater than 3067671.
"IWZ160S" on page 746	Argument-1 for function *function-name* was less than 16010101 or greater than 99991231.
"IWZ161S" on page 746	Argument-1 for function *function-name* was less than 1601001 or greater than 9999365.
"IWZ162S" on page 746	Argument-1 for function *function-name* was less than 1 or greater than the number of positions in the program collating sequence.
"IWZ163S" on page 747	Argument-1 for function *function-name* was less than zero.
"IWZ165S" on page 747	A reference modification start position value of *start-position-value* on line *line number* referenced an area outside the region of the function result of *function-result*.
"IWZ166S" on page 747	A nonpositive reference modification length value of *length* on line *line-number* was found in a reference to the function result of *function-result*.
"IWZ167S" on page 747	A reference modification start position value of *start-position* and length value of *length* on line *line-number* caused reference to be made beyond the rightmost character of the function result of *function-result*.
"IWZ168W" on page 748	SYSPUNCH/SYSPCH will default to the system logical output device. The corresponding environment variable has not been set.
"IWZ170S" on page 748	Illegal data type for DISPLAY operand.
"IWZ171I" on page 748	*string-name* is not a valid runtime option.

Table 117. **Runtime messages** *(continued)*

"IWZ172I" on page 749	The string *string-name* is not a valid suboption of the runtime option *option-name*.
"IWZ173I" on page 749	The suboption string *string-name* of the runtime option *option-name* must be *number* characters long. The default will be used.
"IWZ174I" on page 749	The suboption string *string-name* of the runtime option *option-name* contains one or more invalid characters. The default will be used.
"IWZ175S" on page 749	There is no support for routine *routine-name* on this system.
"IWZ176S" on page 749	Argument-1 for function *function-name* was greater than *decimal-value*.
"IWZ177S" on page 750	Argument-2 for function *function-name* was equal to *decimal-value*.
"IWZ178S" on page 750	Argument-1 for function *function-name* was less than or equal to *decimal-value*.
"IWZ179S" on page 750	Argument-1 for function *function-name* was less than *decimal-value*.
"IWZ180S" on page 750	Argument-1 for function *function-name* was not an integer.
"IWZ181I" on page 750	An invalid character was found in the numeric string *string* of the runtime option *option-name*. The default will be used.
"IWZ182I" on page 751	The number *number* of the runtime option *option-name* exceeded the range of *min-range* to *max-range*. The default will be used.
"IWZ183S" on page 751	The function name in _IWZCOBOLInit did a return.
"IWZ200S" on page 751	Error detected during *I/O operation* for file *file-name*. File status is: *file-status*.
"IWZ200S" on page 751	STOP or ACCEPT failed with an I/O error, *error-code*. The run unit is terminated.
"IWZ201C" on page 752	
"IWZ203W" on page 752	The code page in effect is not a DBCS code page.
"IWZ204W" on page 753	An error occurred during conversion from ASCII DBCS to EBCDIC DBCS.
"IWZ221S" on page 753	ICU converter for code page, *codepage value,* can not be opened.
"IWZ222S" on page 753	Data conversion via ICU failed with error code *error code value*.
"IWZ223S" on page 753	Close of ICU converter failed with error code *error code value*.
"IWZ224S" on page 754	ICU collator for the locale value, *locale value*, can not be opened. The error code is *error code value*.
"IWZ225S" on page 754	Unicode case mapping function via ICU failed with error code *error code value*.
"IWZ230W" on page 754	The conversion table for the current codeset, *ASCII codeset-id*, to the EBCDIC codeset, *EBCDIC codeset-id*, is not available. The default ASCII to EBCDIC conversion table will be used.
"IWZ230W" on page 754	The EBCDIC code page specified, *EBCDIC codepage*, is not consistent with the locale *locale*, but will be used as requested.

Table 117. **Runtime messages** *(continued)*

"IWZ230W" on page 755	The EBCDIC code page specified, *EBCDIC codepage*, is not supported. The default EBCDIC code page, *EBCDIC codepage*, will be used.
"IWZ230S" on page 755	The EBCDIC conversion table cannot be opened.
"IWZ230S" on page 755	The EBCDIC conversion table cannot be built.
"IWZ230S" on page 755	The main program was compiled with both the `-host` flag and the `CHAR(NATIVE)` option, which are not compatible.
"IWZ231S" on page 756	Query of current locale setting failed.
"IWZ232W" on page 756	Message variants include: • An error occurred during the conversion of data item *data-name* to EBCDIC in program *program-name* on line number *decimal-value*. • An error occurred during the conversion of data item *data-name* to ASCII in program *program-name* on line number *decimal-value*. • An error occurred during the conversion to EBCDIC for data item *data-name* in program *program-name* on line number *decimal-value*. • An error occurred during the conversion to ASCII for data item *data-name* in program *program-name* on line number *decimal-value*. • An error occurred during the conversion from ASCII to EBCDIC in program *program-name* on line number *decimal-value*. • An error occurred during the conversion from EBCDIC to ASCII in program *program-name* on line number *decimal-value*.
"IWZ233S" on page 756	COBOL program *program-name* cannot be run because it was compiled with the `NOTHREAD` compiler option and linked with the multithreaded library.
"IWZ240S" on page 757	The base year for program *program-name* was outside the valid range of 1900 through 1999. The sliding window value *window-value* resulted in a base year of *base-year*.
"IWZ241S" on page 757	The current year was outside the 100-year window, *year-start* through *year-end*, for program *program-name*.
"IWZ242S" on page 757	There was an invalid attempt to start an XML PARSE statement.
"IWZ243S" on page 758	There was an invalid attempt to end an XML PARSE statement.
"IWZ250S" on page 758	Internal error: the call to JNI_GetCreatedJavaVMs returned an error, return code *nn*.
"IWZ251S" on page 758	Internal error: *n* active Java VMs were detected, when only 1 was expected.
"IWZ253S" on page 758	More than *nn* JVM initialization options was specified.
"IWZ254S" on page 759	Internal error: the call to JNI_CreateJavaVM returned an error.
"IWZ255S" on page 759	Internal error: the call to GetEnv returned code *nn* because the current thread is not attached to the JVM.
"IWZ256S" on page 759	Internal error: the call to GetEnv returned code *nn* because the JVM version is not supported.

Table 117. **Runtime messages** *(continued)*

"IWZ257S" on page 759	Internal error: the call to GetEnv returned the unrecognized return code *nn*.
"IWZ258S" on page 759	Internal error: GetByteArrayElements was unable to acquire a pointer to the instance data.
"IWZ259S" on page 760	Unable to acquire a direct pointer to the instance data. The installed JVM does not support pinned byte arrays.
"IWZ260S" on page 760	The Java class *name* could not be found.
"IWZ813S" on page 760	Insufficient storage was available to satisfy a get storage request.
"IWZ901S" on page 760	Message variants include: • Program exits due to severe or critical error. • Program exits: more than ERRCOUNT errors occurred.
"IWZ902S" on page 761	The system detected a decimal-divide exception.
"IWZ903S" on page 761	The system detected a data exception.
"IWZ907S" on page 761	Message variants include: • Insufficient storage. • Insufficient storage. Cannot get *number-bytes* bytes of space for *storage*.
"IWZ993W" on page 761	Insufficient storage. Cannot find space for message *message-number*.
"IWZ994W" on page 762	Cannot find message *message-number* in message-catalog.
"IWZ995C" on page 762	Message variants include: • *System exception* signal received while executing routine *routine-name* at offset 0x*offset-value*. • *System exception* signal received while executing code at location 0x*offset-value*. • *System exception* signal received. The location could not be determined.
"IWZ2502S" on page 762	The UTC/GMT was not available from the system.
"IWZ2503S" on page 763	The offset from UTC/GMT to local time was not available from the system.
"IWZ2505S" on page 763	The input_seconds value in a call to CEEDATM or CEESECI was not within the supported range.
"IWZ2506S" on page 763	An era (<JJJJ>, <CCCC>, or <CCCCCCCC>) was used in a picture string passed to CEEDATM, but the input number-of-seconds value was not within the supported range. The era could not be determined.
"IWZ2507S" on page 763	Insufficient data was passed to CEEDAYS or CEESECS. The Lilian value was not calculated.
"IWZ2508S" on page 764	The date value passed to CEEDAYS or CEESECS was invalid.
"IWZ2509S" on page 764	The era passed to CEEDAYS or CEESECS was not recognized.

Table 117. **Runtime messages** *(continued)*

"IWZ2510S" on page 764	The hours value in a call to CEEISEC or CEESECS was not recognized.
"IWZ2511S" on page 765	The day parameter passed in a CEEISEC call was invalid for year and month specified.
"IWZ2512S" on page 765	The Lilian date value passed in a call to CEEDATE or CEEDYWK was not within the supported range.
"IWZ2513S" on page 765	The input date passed in a CEEISEC, CEEDAYS, or CEESECS call was not within the supported range.
"IWZ2514S" on page 766	The year value passed in a CEEISEC call was not within the supported range.
"IWZ2515S" on page 766	The milliseconds value in a CEEISEC call was not recognized.
"IWZ2516S" on page 766	The minutes value in a CEEISEC call was not recognized.
"IWZ2517S" on page 766	The month value in a CEEISEC call was not recognized.
"IWZ2518S" on page 767	An invalid picture string was specified in a call to a date/time service.
"IWZ2519S" on page 767	The seconds value in a CEEISEC call was not recognized.
"IWZ2520S" on page 767	CEEDAYS detected nonnumeric data in a numeric field, or the date string did not match the picture string.
"IWZ2521S" on page 768	The <JJJJ>, <CCCC>, or <CCCCCCCC> year-within-era value passed to CEEDAYS or CEESECS was zero.
"IWZ2522S" on page 768	An era (<JJJJ>, <CCCC>, or <CCCCCCCC>) was used in a picture string passed to CEEDATE, but the Lilian date value was not within the supported range. The era could not be determined.
"IWZ2525S" on page 768	CEESECS detected nonnumeric data in a numeric field, or the timestamp string did not match the picture string.
"IWZ2526S" on page 769	The date string returned by CEEDATE was truncated.
"IWZ2527S" on page 769	The timestamp string returned by CEEDATM was truncated.
"IWZ2531S" on page 769	The local time was not available from the system.
"IWZ2533S" on page 769	The value passed to CEESCEN was not between 0 and 100.
"IWZ2534W" on page 770	Insufficient field width was specified for a month or weekday name in a call to CEEDATE or CEEDATM. Output set to blanks.

IWZ006S

The reference to table *table-name* by verb number *verb-number* on line *line-number* addressed an area outside the region of the table.

Explanation: When the SSRANGE option is in effect, this message is issued to indicate that a fixed-length table has been subscripted in a way that exceeds the defined size of the table, or, for variable-length tables, the maximum size of the table.

The range check was performed on the composite of the subscripts and resulted in an address outside the region of the table. For variable-length tables, the address is outside the region of the table defined when all OCCURS DEPENDING ON objects are at their maximum values; the ODO object's current value is not considered. The check was not performed on individual subscripts.

Programmer response: Ensure that the value of literal subscripts and the value of variable subscripts as evaluated at run time do not exceed the subscripted dimensions for subscripted data in the failing statement.

System action: The application was terminated.

IWZ007S

The reference to variable-length group *group-name* by verb number *verb-number* on line *line-number* addressed an area outside the maximum defined length of the group.

Explanation: When the `SSRANGE` option is in effect, this message is issued to indicate that a variable-length group generated by OCCURS DEPENDING ON has a length that is less than zero, or is greater than the limits defined in the OCCURS DEPENDING ON clauses.

The range check was performed on the composite length of the group, and not on the individual OCCURS DEPENDING ON objects.

Programmer response: Ensure that OCCURS DEPENDING ON objects as evaluated at run time do not exceed the maximum number of occurrences of the dimension for tables within the referenced group item.

System action: The application was terminated.

IWZ012I

Invalid run unit termination occurred while sort or merge is running.

Explanation: A sort or merge initiated by a COBOL program was in progress and one of the following was attempted:
1. A STOP RUN was issued.
2. A GOBACK or an EXIT PROGRAM was issued within the input procedure or the output procedure of the COBOL program that initiated the sort or merge. Note that the GOBACK and EXIT PROGRAM statements are allowed in a program called by an input procedure or an output procedure.

Programmer response: Change the application so that it does not use one of the above methods to end the sort or merge.

System action: The application was terminated.

IWZ013S

Sort or merge requested while sort or merge is running in a different thread.

Explanation: Running sort or merge in two or more threads at the same time is not supported.

Programmer response: Always run sort or merge in the same thread. Alternatively, include code before each call to the sort or merge that determines if sort or merge is running in another thread. If sort or merge is running in another thread, then wait for that thread to finish. If it isn't, then set a flag to indicate sort or merge is running and call sort or merge.

System action: The thread is terminated.

IWZ026W

The SORT-RETURN special register was never referenced, but the current content indicated the sort or merge operation in program *program-name* on line number *line-number* was unsuccessful. The sort or merge return code was *return code*.

Explanation: The COBOL source does not contain any references to the `SORT-RETURN` register. The compiler generates a test after each sort or merge verb. A nonzero return code has been passed back to the program by Sort or Merge.

Programmer response: Determine why the Sort or Merge was unsuccessful and fix the problem. See "Sort and merge error numbers" on page 152 for the list of possible return codes.

System action: No system action was taken.

IWZ029S

Argument-1 for function *function-name* in program *program-name* at line *line-number* was less than zero.

Explanation: An illegal value for argument-1 was used.

Programmer response: Ensure that argument-1 is greater than or equal to zero.

System action: The application was terminated.

IWZ030S

Argument-2 for function *function-name* in program *program-name* at line *line-number* was not a positive integer.

Explanation: An illegal value for argument-1 was used.

Programmer response: Ensure that argument-2 is a positive integer.

System action: The application was terminated.

IWZ036W

Truncation of high order digit positions occurred in program *program-name* on line number *line-number*.

Explanation: The generated code has truncated an intermediate result (that is, temporary storage used during an arithmetic calculation) to 30 digits; some of the truncated digits were not 0.

Programmer response: See the related concepts at the end of this section for a description of intermediate results.

System action: No system action was taken.

IWZ037I

The flow of control in program *program-name* proceeded beyond the last line of the program. Control returned to the caller of the program *program-name*.

Explanation: The program did not have a terminator (STOP, GOBACK, or EXIT), and control fell through the last instruction.

Programmer response: Check the logic of the program. Sometimes this error occurs because of one of the following logic errors:

- The last paragraph in the program was only supposed to receive control as the result of a PERFORM statement, but due to a logic error it was branched to by a GO TO statement.
- The last paragraph in the program was executed as the result of a ″fall-through″ path, and there was no statement at the end of the paragraph to end the program.

System action: The application was terminated.

IWZ038S

A reference modification length value of *reference-modification-value* on line *line-number* which was not equal to 1 was found in a reference to data item *data-item*.

Explanation: The length value in a reference modification specification was not equal to 1. The length value must be equal to 1.

Programmer response: Check the indicated line number in the program to ensure that any reference modified length values are (or will resolve to) 1.

System action: The application was terminated.

IWZ039S

An invalid overpunched sign was detected.

Explanation: The value in the sign position was not valid.

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

Positive:	0, 1, 2, 3, 8, 9, A, and B
Negative:	4, 5, 6, 7, C, D, E, and F

Signs generated internally are 3 for positive and unsigned, and 7 for negative.

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

Positive:	A, C, E, and F
Negative:	B and D

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

Programmer response: This error might have occurred because of a REDEFINES clause involving the sign position or a group move involving the sign position, or the position was never initialized. Check for the above cases.

System action: The application was terminated.

IWZ040S

An invalid separate sign was detected.

Explanation: An operation was attempted on data defined with a separate sign. The value in the sign position was not a plus (+) or a minus (-).

Programmer response: This error might have occurred because of a REDEFINES clause involving the sign position or a group move involving the sign position, or the position was never initialized. Check for the above cases.

System action: The application was terminated.

IWZ045S

Unable to invoke method *method-name* on line number *line number* in program *program-name*.

Explanation: The specific method is not supported for the class of the current object reference.

Programmer response: Check the indicated line number in the program to ensure that the class of the current object reference supports the method being invoked.

System action: The application was terminated.

IWZ047S

Unable to invoke method *method-name* on line number *line number* in class *class-name*.

Explanation: The specific method is not supported for the class of the current object reference.

Programmer response: Check the indicated line number in the class to ensure that the class of the current object reference supports the method being invoked.

System action: The application was terminated.

IWZ048W

A negative base was raised to a fractional power in an exponentiation expression. The absolute value of the base was used.

Explanation: A negative number raised to a fractional power occurred in a library routine.

The value of a negative number raised to a fractional power is undefined in COBOL. If a SIZE ERROR clause had appeared on the statement in question, the SIZE ERROR imperative would have been used. However, no SIZE ERROR clause was present, so the absolute value of the base was used in the exponentiation.

Programmer response: Ensure that the program variables in the failing statement have been set correctly.

System action: No system action was taken.

IWZ049W

A zero base was raised to a zero power in an exponentiation expression. The result was set to one.

Explanation: The value of zero raised to the power zero occurred in a library routine.

The value of zero raised to the power zero is undefined in COBOL. If a SIZE ERROR clause had appeared on the statement in question, the SIZE ERROR imperative would have been used. However, no SIZE ERROR clause was present, so the value returned was one.

Programmer response: Ensure that the program variables in the failing statement have been set correctly.

System action: No system action was taken.

IWZ050S

A zero base was raised to a negative power in an exponentiation expression.

Explanation: The value of zero raised to a negative power occurred in a library routine.

The value of zero raised to a negative number is not defined. If a SIZE ERROR clause had appeared on the statement in question, the SIZE ERROR imperative would have been used. However, no SIZE ERROR clause was present.

Programmer response: Ensure that the program variables in the failing statement have been set correctly.

System action: The application was terminated.

IWZ051W

No significant digits remain in a fixed-point exponentiation operation in program *program-name* due to excessive decimal positions specified in the operands or receivers.

Explanation: A fixed-point calculation produced a result that had no significant digits because the operands or receiver had too many decimal positions.

Programmer response: Modify the `PICTURE` clauses of the operands or the receiving numeric item as needed to have additional integer positions and fewer decimal positions.

System action: No system action was taken.

IWZ053S

An overflow occurred on conversion to floating point.

Explanation: A number was generated in the program that is too large to be represented in floating point.

Programmer response: You need to modify the program appropriately to avoid an overflow.

System action: The application was terminated.

IWZ054S

A floating point exception occurred.

Explanation: A floating-point calculation has produced an illegal result. Floating-point calculations are done using IEEE floating-point arithmetic, which can produce results called NaN (Not a Number). For example, the result of 0 divided by 0 is NaN.

Programmer response: Modify the program to test the arguments to this operation so that NaN is not produced.

System action: The application was terminated.

IWZ055W

An underflow occurred on conversion to floating point. The result was set to zero.

Explanation: On conversion to floating point, the negative exponent exceeded the limit of the hardware. The floating-point value was set to zero.

Programmer response: No action is necessary, although you may want to modify the program to avoid an underflow.

System action: No system action was taken.

IWZ058S

Exponent overflow occurred.

Explanation: Floating-point exponent overflow occurred in a library routine.

Programmer response: Ensure that the program variables in the failing statement have been set correctly.

System action: The application was terminated.

IWZ059W

An exponent with more than nine digits was truncated.

Explanation: Exponents in fixed point exponentiations may not contain more than nine digits. The exponent was truncated back to nine digits; some of the truncated digits were not 0.

Programmer response: No action is necessary, although you may want to adjust the exponent in the failing statement.

System action: No system action was taken.

IWZ060W

Truncation of high-order digit positions occurred.

Explanation: Code in a library routine has truncated an intermediate result (that is, temporary storage used during an arithmetic calculation) back to 30 digits; some of the truncated digits were not 0.

Programmer response: See the related concepts at the end of this section for a description of intermediate results.

System action: No system action was taken.

IWZ061S

Division by zero occurred.

Explanation: Division by zero occurred in a library routine. Division by zero is not defined. If a SIZE ERROR clause had appeared on the statement in question, the SIZE ERROR imperative would have been used. However, no SIZE ERROR clause was present.

Programmer response: Ensure that the program variables in the failing statement have been set correctly.

System action: The application was terminated.

IWZ063S

An invalid sign was detected in a numeric edited sending field in *program-name* on line number *line-number*.

Explanation: An attempt has been made to move a signed numeric edited field to a signed numeric or numeric edited receiving field in a MOVE statement. However, the sign position in the sending field contained a character that was not a valid sign character for the corresponding PICTURE.

Programmer response: Ensure that the program variables in the failing statement have been set correctly.

System action: The application was terminated.

IWZ064S

A recursive call to active program *program-name* in compilation unit *compilation-unit* was attempted.

Explanation: COBOL does not allow reinvocation of an internal program which has begun execution, but has not yet terminated. For example, if internal programs A and B are siblings of a containing program, and A calls B and B calls A, this message will be issued.

Programmer response: Examine your program to eliminate calls to active internal programs.

System action: The application was terminated.

IWZ065I

A CANCEL of active program *program-name* in compilation unit *compilation-unit* was attempted.

Explanation: An attempt was made to cancel an active internal program. For example, if internal programs A and B are siblings in a containing program and A calls B and B cancels A, this message will be issued.

Programmer response: Examine your program to eliminate cancellation of active internal programs.

System action: The application was terminated.

IWZ066S

The length of external data record *data-record* in program *program-name* did not match the existing length of the record.

Explanation: While processing External data records during program initialization, it was determined that an External data record was previously defined in another program in the run-unit, and the length of the record as specified in the current program was not the same as the previously defined length.

Programmer response: Examine the current file and ensure the External data records are specified correctly.

System action: The application was terminated.

IWZ071S

ALL subscripted table reference to table *table-name* by verb number *verb-number* on line *line-number* had an ALL subscript specified for an OCCURS DEPENDING ON dimension, and the object was less than or equal to 0.

Explanation: When the SSRANGE option is in effect, this message is issued to indicate that there are 0 occurrences of dimension subscripted by ALL.

The check is performed against the current value of the OCCURS DEPENDING ON object.

Programmer response: Ensure that ODO objects of ALL-subscripted dimensions of any subscripted items in the indicated statement are positive.

System action: The application was terminated.

IWZ072S

A reference modification start position value of *reference-modification-value* on line *line-number* referenced an area outside the region of data item *data-item*.

Explanation: The value of the starting position in a reference modification specification was less than 1, or was greater than the current length of the data item that was being reference modified. The starting position value must be a positive integer less than or equal to the number of characters in the reference modified data item.

Programmer response: Check the value of the starting position in the reference modification specification.

System action: The application was terminated.

IWZ073S

A nonpositive reference modification length value of *reference-modification-value* on line *line-number* was found in a reference to data item *data-item*.

Explanation: The length value in a reference modification specification was less than or equal to 0. The length value must be a positive integer.

Programmer response: Check the indicated line number in the program to ensure that any reference modified length values are (or will resolve to) positive integers.

System action: The application was terminated.

IWZ074S

A reference modification start position value of *reference-modification-value* and length value of *length* on line *line-number* caused reference to be made beyond the rightmost character of data item *data-item*.

Explanation: The starting position and length value in a reference modification specification combine to address an area beyond the end of the reference modified data item. The sum of the starting position and length value minus one must be less than or equal to the number of characters in the reference modified data item.

Programmer response: Check the indicated line number in the program to ensure that any reference modified start and length values are set such that a reference is not made beyond the rightmost character of the data item.

System action: The application was terminated.

IWZ075S

Inconsistencies were found in EXTERNAL file *file-name* in program *program-name*. The following file attributes did not match those of the established external file: *attribute-1 attribute-2 attribute-3 attribute-4 attribute-5 attribute-6 attribute-7*.

Explanation: One or more attributes of an external file did not match between two programs that defined it.

Programmer response: Correct the external file. For a summary of file attributes which must match between definitions of the same external file.

System Action: The application was terminated.

IWZ076W

The number of characters in the INSPECT REPLACING CHARACTERS BY data-name was not equal to one. The first character was used.

Explanation: A data item which appears in a `CHARACTERS` phrase within a `REPLACING` phrase in an `INSPECT` statement must be defined as being one character in length. Because of a reference modification specification for this data item, the resultant length value was not equal to one. The length value is assumed to be one.

Programmer response: You may correct the reference modification specifications in the failing `INSPECT` statement to ensure that the reference modification length is (or will resolve to) 1; programmer action is not required.

System action: No system action was taken.

IWZ077W

The lengths of the INSPECT data items were not equal. The shorter length was used.

Explanation: The two data items which appear in a `REPLACING` or `CONVERTING` phrase in an `INSPECT` statement must have equal lengths, except when the second such item is a figurative constant. Because of the reference modification for one or both of these data items, the resultant length values were not equal. The shorter length value is applied to both items, and execution proceeds.

Programmer response: You may adjust the operands of unequal length in the failing `INSPECT` statement; programmer action is not required.

System action: No system action was taken.

IWZ078S

ALL subscripted table reference to table *table-name* by verb number *verb-number* on line *line-number* will exceed the upper bound of the table.

Explanation: When the `SSRANGE` option is in effect, this message is issued to indicate that a multidimensional table with ALL specified as one or more of the subscripts will result in a reference beyond the upper limit of the table.

The range check was performed on the composite of the subscripts and the maximum occurrences for the ALL subscripted dimensions. For variable-length tables the address is outside the region of the table defined when all OCCURS DEPENDING ON objects are at their maximum values; the ODO object's current value is not considered. The check was not performed on individual subscripts.

Programmer response: Ensure that OCCURS DEPENDING ON objects as evaluated at run time do not exceed the maximum number of occurrences of the dimension for table items referenced in the failing statement.

System action: The application was terminated.

IWZ096C

Dynamic call of program *program-name* failed. Message variants include:

- **A load of module *module-name* failed with an error code of *error-code*.**
- **A load of module *module-name* failed with a return code of *return-code*.**
- **Dynamic call of program *program-name* failed. Insufficient resources.**
- **Dynamic call of program *program-name* failed. COBPATH not found in environment.**
- **Dynamic call of program *program-name* failed. Entry *entry-name* not found.**
- **Dynamic call failed. The name of the target program does not contain any valid characters.**
- **Dynamic call of program *program-name* failed. The load module *load-module* could not be found in the directories identified in the COBPATH environment variable.**

Explanation: A dynamic call failed due to one of the reasons listed in the message variants above. In the above, the value of *error-code* is the errno set by `load`.

Programmer response: Check that COBPATH is defined. Check that the module exists: AIX has graphical interfaces for showing directories and files. You can also use the `ls` command. Check that the name of the module to be loaded matches the name of the entry called. Check that the module to be loaded is built correctly using the appropriate `cob2` options.

System action: The application was terminated.

IWZ097S

Argument-1 for function *function-name* contained no digits.

Explanation: Argument-1 for the indicated function must contain at least 1 digit.

Programmer response: Adjust the number of digits in Argument-1 in the failing statement.

System action: The application was terminated.

IWZ100S

Argument-1 for function *function* was less than or equal to -1.

Explanation: An illegal value was used for Argument-1.

Programmer response: Ensure that argument-1 is greater than -1.

System action: The application was terminated.

IWZ103S

Argument-1 for function *function-name* was less than zero or greater than 99.

Explanation: An illegal value was used for Argument-1.

Programmer response: Check that the function argument is in the valid range.

System action: The application was terminated.

IWZ104S

Argument-1 for function *function-name* was less than zero or greater than 99999.

Explanation: An illegal value was used for Argument-1.

Programmer response: Check that the function argument is in the valid range.

System action: The application was terminated.

IWZ105S

Argument-1 for function *function-name* was less than zero or greater than 999999.

Explanation: An illegal value was used for Argument-1.

Programmer response: Check that the function argument is in the valid range.

System action: The application was terminated.

IWZ151S

Argument-1 for function *function-name* contained more than 18 digits.

Explanation: The total number of digits in Argument-1 of the indicated function exceeded 18 digits.

Programmer response: Adjust the number of digits in Argument-1 in the failing statement.

System action: The application was terminated.

IWZ152S

Invalid character *character* was found in column *column-number* in argument-1 for function *function-name*.

Explanation: A nondigit character other than a decimal point, comma, space or sign (+,-,CR,DB) was found in argument-1 for NUMVAL/NUMVAL-C function.

Programmer response: Correct argument-1 for NUMVAL or NUMVAL-C in the indicated statement.

System action: The application was terminated.

IWZ155S

Invalid character *character* was found in column *column-number* in argument-2 for function *function-name*.

Explanation: Illegal character was found in argument-2 for NUMVAL-C function.

Programmer response: Check that the function argument does follow the syntax rules.

System action: The application was terminated.

IWZ156S

Argument-1 for function *function-name* was less than zero or greater than 28.

Explanation: Input argument to function FACTORIAL is greater than 28 or less than 0.

Programmer response: Check that the function argument is in the valid range.

System action: The application was terminated.

IWZ157S

The length of Argument-1 for function *function-name* was not equal to 1.

Explanation: The length of input argument to ORD function is not 1.

Programmer response: Check that the function argument is only 1 byte long.

System action: The application was terminated.

IWZ158S

Argument-1 for function *function-name* was less than zero or greater than 29.

Explanation: Input argument to function FACTORIAL is greater than 29 or less than 0.

Programmer response: Check that the function argument is in the valid range.

System action: The application was terminated.

IWZ159S

Argument-1 for function *function-name* was less than 1 or greater than 3067671.

Explanation: The input argument to DATE-OF-INTEGER or DAY-OF-INTEGER function is less than 1 or greater than 3067671.

Programmer response: Check that the function argument is in the valid range.

System action: The application was terminated.

IWZ160S

Argument-1 for function *function-name* was less than 16010101 or greater than 99991231.

Explanation: The input argument to function INTEGER-OF-DATE is less than 16010101 or greater than 99991231.

Programmer response: Check that the function argument is in the valid range.

System action: The application was terminated.

IWZ161S

Argument-1 for function *function-name* was less than 1601001 or greater than 9999365.

Explanation: The input argument to function INTEGER-OF-DAY is less than 1601001 or greater than 9999365.

Programmer response: Check that the function argument is in the valid range.

System action: The application was terminated.

IWZ162S

Argument-1 for function *function-name* was less than 1 or greater than the number of positions in the program collating sequence.

Explanation: The input argument to function CHAR is less than 1 or greater than the highest ordinal position in the program collating sequence.

Programmer response: Check that the function argument is in the valid range.

System action: The application was terminated.

IWZ163S

Argument-1 for function *function-name* was less than zero.

Explanation: The input argument to function RANDOM is less than 0.

Programmer response: Correct the argument for function RANDOM in the failing statement.

System action: The application was terminated.

IWZ165S

A reference modification start position value of *start-position-value* on line *line number* referenced an area outside the region of the function result of *function-result*.

Explanation: The value of the starting position in a reference modification specification was less than 1, or was greater than the current length of the function result that was being reference modified. The starting position value must be a positive integer less than or equal to the number of characters in the reference modified function result.

Programmer response: Check the value of the starting position in the reference modification specification and the length of the actual function result.

System action: The application was terminated.

IWZ166S

A nonpositive reference modification length value of *length* on line *line-number* was found in a reference to the function result of *function-result*.

Explanation: The length value in a reference modification specification for a function result was less than or equal to 0. The length value must be a positive integer.

Programmer response: Check the length value and make appropriate correction.

System action: The application was terminated.

IWZ167S

A reference modification start position value of *start-position* and length value of *length* on line *line-number* caused reference to be made beyond the rightmost character of the function result of *function-result*.

Explanation: The starting position and length value in a reference modification specification combine to address an area beyond the end of the reference modified function result. The sum of the starting position and length value minus one must be less than or equal to the number of characters in the reference modified function result.

Programmer response: Check the length of the reference modification specification against the actual length of the function result and make appropriate corrections.

System action: The application was terminated.

IWZ168W

SYSPUNCH/SYSPCH will default to the system logical output device. The corresponding environment variable has not been set.

Explanation: COBOL environment names (such as SYSPUNCH/SYSPCH) are used as the environment variable names corresponding to the mnemonic names used on ACCEPT and DISPLAY statements. Set them equal to files, not existing directory names. To set environment variables, use the `export` command.

You can set environment variables either persistently or temporarily.

Programmer response: If you do not want SYSPUNCH/SYSPCH to default to the screen, set the corresponding environment variable.

System action: No system action was taken.

IWZ170S

Illegal data type for DISPLAY operand.

Explanation: An invalid data type was specified as the target of the DISPLAY statement.

Programmer response: Specify a valid data type. The following data types are **not** valid:

- Data items defined with `USAGE IS PROCEDURE-POINTER`
- Data items defined with `USAGE IS OBJECT REFERENCE`
- Data items or index names defined with `USAGE IS INDEX`

System action: The application was terminated.

IWZ171I

***string-name* is not a valid runtime option.**

Explanation: *string-name* is not a valid option.

Programmer response: `CHECK`, `DEBUG`, `ERRCOUNT`, `FILESYS`, `TRAP`, and `UPSI` are valid runtime options.

System action: *string-name* is ignored.

IWZ172I

The string *string-name* is not a valid suboption of the runtime option *option-name*.

Explanation: *string-name* was not in the set of recognized values.

Programmer response: Remove the invalid suboption *string* from the runtime option *option-name*.

System action: The invalid suboption is ignored.

IWZ173I

The suboption string *string-name* of the runtime option *option-name* must be *number of* characters long. The default will be used.

Explanation: The number of characters for the suboption string *string-name* of runtime option *option-name* is invalid.

Programmer response: If you do not want to accept the default, specify a valid character length.

System action: The default value will be used.

IWZ174I

The suboption string *string-name* of the runtime option *option-name* contains one or more invalid characters. The default will be used.

Explanation: At least one invalid character was detected in the specified suboption.

Programmer response: If you do not want to accept the default, specify valid characters.

System action: The default value will be used.

IWZ175S

There is no support for routine *routine-name* on this system.

Explanation: *routine-name* is not supported.

Programmer response:

System action: The application was terminated.

IWZ176S

Argument-1 for function *function-name* was greater than *decimal-value*.

Explanation: An illegal value for argument-1 was used.

Programmer response: Ensure argument-1 is less than or equal to *decimal-value*.

System action: The application was terminated.

IWZ177S

Argument-2 for function *function-name* was equal to *decimal-value*.

Explanation: An illegal value for argument-2 was used.

Programmer response: Ensure argument-1 is not equal to *decimal-value*.

System action: The application was terminated.

IWZ178S

Argument-1 for function *function-name* was less than or equal to *decimal-value*.

Explanation: An illegal value for Argument-1 was used.

Programmer response: Ensure that Argument-1 is greater than *decimal-value*.

System action: The application was terminated.

IWZ179S

Argument-1 for function *function-name* was less than *decimal-value*.

Explanation: An illegal value for Argument-1 was used.

Programmer response: Ensure that Argument-1 is equal to or greater than *decimal-value*.

System action: The application was terminated.

IWZ180S

Argument-1 for function *function-name* was not an integer.

Explanation: An illegal value for Argument-1 was used.

Programmer response: Ensure that Argument-1 is an integer.

System action: The application was terminated.

IWZ181I

An invalid character was found in the numeric string *string* of the runtime option *option-name*. The default will be used.

Explanation: *string* did not contain all decimal numeric characters.

Programmer response: If you do not want the default value, correct the runtime option's string to contain all numeric characters.

System action: The default will be used.

IWZ182I

The number *number* of the runtime option *option-name* exceeded the range of *min-range* to *max-range*. The default will be used.

Explanation: *number* exceeded the range of *min-range* to *max-range*.

Programmer response: Correct the runtime option's string to be within the valid range.

System action: The default will be used.

IWZ183S

The function name in _iwzCOBOLInit did a return.

Explanation: The run unit termination exit routine returned to the function that invoked the routine (the function specified in `function_code`).

Programmer response: Rewrite the function so that the run unit termination exit routine does a longjump or exit instead of return to the function.

System action: The application was terminated.

IWZ200S

Error detected during *I/O operation* for file *file-name*. File status is: *file-status*.

Explanation: An error was detected during a file I/O operation. No file status was specified for the file and no applicable error declarative is in effect for the file.

Programmer response: Correct the condition described in this message. You can specify the `FILE STATUS` clause for the file if you want to detect the error and take appropriate actions within your source program.

System action: The application was terminated.

IWZ200S

STOP or ACCEPT failed with an I/O error, *error-code*. The run unit is terminated.

Explanation: A STOP or ACCEPT statement failed.

Programmer response: Check that the STOP or ACCEPT refers to a legitimate file or terminal device.

System action: The application was terminated.

IWZ201C

```
Message variants include:                          Address Error.
Access Intent List Error.                          Command Check.
Concurrent Opens Exceeds Maximum.                  Duplicate File Name.
Cursor Not Selecting a Record Position.            End of File Condition.
Data Stream Syntax Error.                          Existing Condition.
Duplicate Key Different Index.                     File Handle Not Found.
Duplicate Key Same Index.                          Field Length Error.
Duplicate Record Number.                           File Not Found.
File Temporarily Not Available.                    File Damaged.
File system cannot be found.                       File is Full.
File Space Not Available.                          File In Use.
File Closed with Damage.                           Function Not Supported.
Invalid Key Definition.                            Invalid Access Method.
Invalid Base File Name.                            Invalid Data Record.
Key Update Not Allowed by Different Index.         Invalid Key Length.
Key Update Not Allowed by Same Index.              Invalid File Name.
No Update Intent on Record.                        Invalid Request.
Not Authorized to Use Access Method.               Invalid Flag.
Not Authorized to Directory.                       Object Not Supported.
Not Authorized to Function.                        Record Not Available.
Not authorized to File.                            Record Not Found.
Parameter Value Not Supported.                     Record Inactive.
Parameter Not Supported.                           Record Damaged.
Record Number Out of Bounds.                       Record In Use.
Record Length Mismatch.                            Update Cursor Error.
Resource Limits Reached in Target System.
Resource Limits Reached in Source System.
```

Explanation: An error was detected during a file I/O operation for a VSAM file. No file status was specified for the file and no applicable error declarative is in effect for the file.

Programmer response: Correct the condition described in this message. For details, see *VSAM in a Distributed Environment* for AIX.

System action: The application was terminated.

IWZ203W

The code page in effect is not a DBCS code page.

Explanation: References to DBCS data were made with a non-DBCS code page in effect.

Programmer response: For DBCS data, specify a valid DBCS code page. Valid DBCS code pages are:

Country or region	Code page
Japan	IBM-932
Korea	IBM-1363
People's Republic of China (Simplified)	IBM-1386
Taiwan (Traditional)	

Note: The code pages listed above might not be supported for a specific version or release of that platform.

System Action: No system action was taken.

IWZ204W

An error occurred during conversion from ASCII DBCS to EBCDIC DBCS.

Explanation: A Kanji or DBCS class test failed due to an error detected during the ASCII character string EBCDIC string conversion.

Programmer response: Verify that the locale in effect is consistent with the ASCII character string being tested. No action is likely to be required if the locale setting is correct. The class test is likely to indicate the string to be non-Kanji or non-DBCS correctly.

System action: No system action was taken.

IWZ221S

The ICU converter for the code page, *codepage value*, can not be opened. The error code is *error code value*.

Explanation: The ICU converter to convert between the code page and UTF-16 cannot be opened.

| **Programmer response:** Verify that the code-page value identifies a primary or alias
| code-page name that is supported by ICU conversion libraries (see *International*
| *Components for Unicode: Converter Explorer*). If the code-page value is valid, contact your IBM representative.

System action: The application was terminated.

IWZ222S

Data conversion via ICU failed with error code *error code value*.

Explanation: The data conversion through ICU failed.

Programmer response: Contact your IBM representative.

System action: The application was terminated.

IWZ223S

Close of ICU converter failed with error code *error code value*.

Explanation: The close of an ICU converter failed.

Programmer response: Contact your IBM representative.

System action: The application was terminated.

IWZ224S

ICU collator for the locale value, *locale value*, can not be opened. The error code is *error code value*.

Explanation: The ICU collator for the locale cannot be opened.

Programmer response: Contact your IBM representative.

System action: The application was terminated.

IWZ225S

Unicode case mapping function via ICU failed with error code *error code value*.

Explanation: The ICU case mapping function failed.

Programmer response: Contact your IBM representative.

System action: The application was terminated.

IWZ230W

The conversion table for the current code set, *ASCII codeset-id*, to the EBCDIC code set, *EBCDIC codeset-id*, is not available. The default ASCII to EBCDIC conversion table will be used.

Explanation: The application has a module that was compiled with the `CHAR(EBCDIC)` compiler option. At run time a translation table will be built to handle the conversion from the current ASCII code page to an EBCDIC code page specified by the EBCDIC_CODEPAGE environment variable. This error occurred because either a conversion table is not available for the specified code pages, or the specification of the EBCDIC_CODE page is invalid. Execution will continue with a default conversion table based on ASCII code page IBM-1252 and EBCDIC code page IBM-037.

Programmer response: Verify that the EBCDIC_CODEPAGE environment variable has a valid value.

If EBCDIC_CODEPAGE is not set, the default value, IBM-037, will be used. This is the default code page used by Enterprise COBOL for z/OS.

System action: No system action was taken.

IWZ230W

The EBCDIC code page specified, *EBCDIC codepage*, is not consistent with the locale *locale*, but will be used as requested.

Explanation: The application has a module that was compiled with the `CHAR(EBCDIC)` compiler option. This error occurred because the code page specified is not the same language as the current locale.

Programmer response: Verify that the EBCDIC_CODEPAGE environment variable is valid for this locale.

System action: No system action was taken.

IWZ230W

The EBCDIC code page specified, *EBCDIC codepage*, is not supported. The default EBCDIC code page, *EBCDIC codepage*, will be used.

Explanation: The application has a module that was compiled with the `CHAR(EBCDIC)` compiler option. This error occurred because the specification of the EBCDIC_CODEPAGE environment variable is invalid. Execution will continue with the default host code page that corresponds to the current locale.

Programmer response: Verify that the EBCDIC_CODEPAGE environment variable has a valid value.

System action: No system action was taken.

IWZ230S

The EBCDIC conversion table cannot be opened.

Explanation: The current system installation does not include the translation table for the default ASCII and EBCDIC code pages.

Programmer response: Reinstall the compiler and run time. If the problem still persists, call your IBM representative.

System action: The application was terminated.

IWZ230S

The EBCDIC conversion table cannot be built.

Explanation: The ASCII to EBCDIC conversion table has been opened, but the conversion failed.

Programmer response: Retry the execution from a new window.

System action: The application was terminated.

IWZ230S

The main program was compiled with both the `-host` flag and the `CHAR(NATIVE)` option, which are not compatible.

Explanation: Compilation with both the `-host` flag and the `CHAR(NATIVE)` option is not supported.

Programmer response: Either remove the `-host` flag, or remove the `CHAR(NATIVE)` option. The `-host` flag sets `CHAR(EBCDIC)`.

System action: The application was terminated.

IWZ231S

Query of current locale setting failed.

Explanation: A query of the execution environment failed to identify a valid locale setting. The current locale needs to be established to access appropriate message files and set the collating order. It is also used by the date/time services and for EBCDIC character support.

Programmer response: Check the settings for the following environment variable:

LANG
: This should be set to a locale that has been installed on your machine. Enter `locale -a` to get a list of the valid values. The default value is en_US.

System action: The application was terminated.

IWZ232W

Message variants include:

- **An error occurred during the conversion of data item *data-name* to EBCDIC in program *program-name* on line number *decimal-value*.**
- **An error occurred during the conversion of data item *data-name* to ASCII in program *program-name* on line number *decimal-value*.**
- **An error occurred during the conversion to EBCDIC for data item *data-name* in program *program-name* on line number *decimal-value*.**
- **An error occurred during the conversion to ASCII for data item *data-name* in program *program-name* on line number *decimal-value*.**
- **An error occurred during the conversion from ASCII to EBCDIC in program *program-name* on line number *decimal-value*.**
- **An error occurred during the conversion from EBCDIC to ASCII in program *program-name* on line number *decimal-value*.**

Explanation: The data in an identifier could not be converted between ASCII and EBCDIC formats as requested by the `CHAR(EBCDIC)` compiler option.

Programmer response: Check that the appropriate ASCII and EBCDIC locales are installed and selected. Check that the data in the identifier is valid and can be represented in both ASCII and EBCDIC format.

System action: No system action was taken. The data remains in its unconverted form.

IWZ233S

COBOL program *program-name* cannot be run because it was compiled with the `NOTHREAD` compiler option and linked with the multithreaded library.

Explanation: Non-CICS programs must be compiled with the `THREAD` compiler option if they are linked with the multithreaded library.

Programmer response: Compile the program with the `THREAD` compiler option, or use `cob2` (rather than `cob2_r` or `cob2_j`) to link it with the static library.

System action: The application was terminated.

IWZ240S

The base year for program *program-name* was outside the valid range of 1900 through 1999. The sliding window value *window-value* resulted in a base year of *base-year*.

Explanation: When the 100-year window was computed using the current year and the sliding window value specified with the YEARWINDOW compiler option, the base year of the 100-year window was outside the valid range of 1900 through 1999.

Programmer response: Examine the application design to determine if it will support a change to the YEARWINDOW option value. If the application can run with a change to the YEARWINDOW option value, then compile the program with an appropriate YEARWINDOW option value. If the application cannot run with a change to the YEARWINDOW option value, then convert all date fields to expanded dates and compile the program with NODATEPROC.

System action: The application was terminated.

IWZ241S

The current year was outside the 100-year window, *year-start* through *year-end*, for program *program-name*.

Explanation: The current year was outside the 100-year fixed window specified by the YEARWINDOW compiler option value.

For example, if a COBOL program is compiled with YEARWINDOW(1920), the 100-year window for the program is 1920 through 2019. When the program is run in the year 2020, this error message would occur since the current year is not within the 100-year window.

Programmer response: Examine the application design to determine if it will support a change to the YEARWINDOW option value. If the application can run with a change to the YEARWINDOW option value, then compile the program with an appropriate YEARWINDOW option value. If the application cannot run with a change to the YEARWINDOW option value, then convert all date fields to expanded dates and compile the program with NODATEPROC.

System action: The application was terminated.

IWZ242S

There was an invalid attempt to start an XML PARSE statement.

Explanation: An XML PARSE statement initiated by a COBOL program was already in progress when another XML PARSE statement was attempted by the same COBOL program. Only one XML PARSE statement can be active in a given invocation of a COBOL program.

Programmer response: Change the application so that it does not initiate another XML PARSE statement from within the same COBOL program.

System action: The application is terminated.

IWZ243S

There was an invalid attempt to end an XML PARSE statement.

Explanation: An XML PARSE statement initiated by a COBOL program was in progress and one of the following actions was attempted:
- A GOBACK or an EXIT PROGRAM statement was issued within the COBOL program that initiated the XML PARSE statement.
- A user handler associated with the program that initiated the XML PARSE statement moved the condition handler resume cursor and resumed the application.

Programmer response: Change the application so that it does not use one of the above methods to end the XML PARSE statement.

System action: The application is terminated.

IWZ250S

Internal error: the call to JNI_GetCreatedJavaVMs returned an error, return code *nn*.

Explanation: The call to the JNI_GetCreatedJavaVMs function returned an error with a return code of *nn*.

Programmer response: Contact your IBM representative.

System action: Application execution is terminated.

IWZ251S

Internal error: *n* active Java VMs were detected, when only 1 was expected.

Explanation: Multiple active Java VMs were detected, when only one Java VM was expected.

Programmer response: Contact your IBM representative.

System action: Application execution is terminated.

IWZ253S

More than *nn* JVM initialization options was specified.

Explanation: In the COBJVMINITOPTIONS environment variable, you specified more Java initialization options than the maximum allowable number of options. The limit is 256.

Programmer response: Specify fewer options in the COBJVMINITOPTIONS environment variable.

System action: Application execution is terminated.

IWZ254S

Internal error: the call to JNI_CreateJavaVM returned an error.

Explanation: The call to JNI_CreateJavaVM returned an error.

Programmer response: Contact your IBM representative.

System action: Application execution is terminated.

IWZ255S

Internal error: the call to GetEnv returned code *nn* because the current thread is not attached to the JVM.

Explanation: The call to GetEnv returned code *nn* because the current thread is not attached to the JVM.

Programmer response: Contact your IBM representative.

System action: Application execution is terminated.

IWZ256S

Internal error: the call to GetEnv returned code *nn* because the JVM version is not supported.

Explanation: The call to GetEnv returned code *nn* because the JVM version is not supported.

Programmer response: Contact your IBM representative.

System action: Application execution is terminated.

IWZ257S

Internal error: the call to GetEnv returned the unrecognized return code *nn*.

Explanation: The call to GetEnv returned the unrecognized return code *nn*.

Programmer response: Contact your IBM representative.

System action: Application execution is terminated.

IWZ258S

Internal error: GetByteArrayElements was unable to acquire a pointer to the instance data.

Explanation: The GetByteArrayElements service was unable to acquire a pointer to the instance data.

Programmer response: Contact your IBM representative.

System action: Application execution is terminated.

IWZ259S

Unable to acquire a direct pointer to instance data. The installed JVM does not support pinned byte arrays.

Explanation: You are running with an unsupported JVM, such as Sun 1.4.1, that doesn't support pinned byte arrays. For more information about pinning, see the related reference about the Java Native Interface at the end of this section.

Programmer response: Run a supported JVM.

System action: Application execution is terminated.

IWZ260S

The Java class *name* could not be found.

Explanation: Your program references a class name that is not defined or specified in the CLASSPATH environment variable.

Programmer response: Check the program that references *name*, and either correct the reference or provide the missing *name*.class.

System action: Application execution is terminated.

IWZ813S

Insufficient storage was available to satisfy a get storage request.

Explanation: There was not enough free storage available to satisfy a get storage or reallocate request. This message indicates that storage management could not obtain sufficient storage from the operating system.

Programmer response: Ensure that you have sufficient storage available to run your application.

System action: No storage is allocated.

Symbolic feedback code: CEE0PD

IWZ901S

Message variants include:

- **Program exits due to severe or critical error.**
- **Program exits: more than ERRCOUNT errors occurred.**

Explanation: Every severe or critical message is followed by an IWZ901 message. An IWZ901 message is also issued if you used the ERRCOUNT runtime option and the number of warning messages exceeds ERRCOUNT.

Programmer response: See the severe or critical message, or increase ERRCOUNT.

System action: The application was terminated.

IWZ902S

The system detected a decimal-divide exception.

Explanation: An attempt to divide a number by 0 was detected.

Programmer response: Modify the program. For example, add ON SIZE ERROR to the flagged statement.

System action: The application was terminated.

IWZ903S

The system detected a data exception.

Explanation: An operation on packed-decimal or zoned decimal data failed because the data contained an invalid value.

Programmer response: Verify the data is valid packed-decimal or zoned decimal data.

System action: The application was terminated.

IWZ907S

Message variants include:
- **Insufficient storage.**
- **Insufficient storage. Cannot get *number-bytes* bytes of space for *storage*.**

Explanation: The runtime library requested virtual memory space and the operating system denied the request.

Programmer response: Your program uses a large amount of virtual memory and it ran out of space. The problem is usually not due to a particular statement, but is associated with the program as a whole. Look at your use of OCCURS clauses and reduce the size of your tables.

System action: The application was terminated.

IWZ993W

Insufficient storage. Cannot find space for message *message-number*.

Explanation: The runtime library requested virtual memory space and the operating system denied the request.

Programmer response: Your program uses a large amount of virtual memory and it ran out of space. The problem is usually not due to a particular statement, but is associated with the program as a whole. Look at your use of OCCURS clauses and reduce the size of your tables.

System action: No system action was taken.

IWZ994W

Cannot find message *message-number* in message-catalog.

Explanation: The runtime library cannot find either the message catalog or a particular message in the message catalog.

Programmer response: Check that the COBOL library and messages were correctly installed and that LANG and NLSPATH are specified correctly.

System action: No system action was taken.

IWZ995C

Message variants include:

- ***system exception* signal received while executing routine *routine-name* at offset 0x*offset-value*.**
- ***system exception* signal received while executing code at location 0x*offset-value*.**
- ***system exception* signal received. The location could not be determined.**

Explanation: The operating system has detected an illegal action, such as an attempt to store into a protected area of memory or the operating system has detected that you pressed the interrupt key (typically the Control-C key, but it can be reconfigured).

Programmer response: If the signal was due to an illegal action, run the program under the debugger and it will give you more precise information as to where the error occurred. An example of this type of error is a pointer with an illegal value.

System action: The application was terminated.

IWZ2502S

The UTC/GMT was not available from the system.

Explanation: A call to CEEUTC or CEEGMT failed because the system clock was in an invalid state. The current time could not be determined.

Programmer response: Notify systems support personnel that the system clock is in an invalid state.

System action: All output values are set to 0.

Symbolic feedback code: CEE2E6

IWZ2503S

The offset from UTC/GMT to local time was not available from the system.

Explanation: A call to CEEGMTO failed because either (1) the current operating system could not be determined, or (2) the time zone field in the operating system control block appears to contain invalid data.

Programmer response: Notify systems support personnel that the local time offset stored in the operating system appears to contain invalid data.

System action: All output values are set to 0.

Symbolic feedback code: CEE2E7

IWZ2505S

The input_seconds value in a call to CEEDATM or CEESECI was not within the supported range.

Explanation: The input_seconds value passed in a call to CEEDATM or CEESECI was not a floating-point number between 86,400.0 and 265,621,679,999.999 The input parameter should represent the number of seconds elapsed since 00:00:00 on 14 October 1582, with 00:00:00.000 15 October 1582 being the first supported date/time, and 23:59:59.999 31 December 9999 being the last supported date/time.

Programmer response: Verify that input parameter contains a floating-point value between 86,400.0 and 265,621,679,999.999.

System action: For CEEDATM, the output value is set to blanks. For CEESECI, all output parameters are set to 0.

Symbolic feedback code: CEE2E9

IWZ2506S

An era (<JJJJ>, <CCCC>, or <CCCCCCCC>) was was used in a picture string passed to CEEDATM, but the input number-of-seconds value was not within the supported range. The era could not be determined.

Explanation: In a CEEDATM call, the picture string indicates that the input value is to be converted to an era; however the input value that was specified lies outside the range of supported eras.

Programmer response: Verify that the input value contains a valid number-of-seconds value within the range of supported eras.

System action: The output value is set to blanks.

IWZ2507S

Insufficient data was passed to CEEDAYS or CEESECS. The Lilian value was not calculated.

Explanation: The picture string passed in a CEEDAYS or CEESECS call did not contain enough information. For example, it is an error to use the picture string 'MM/DD' (month and day only) in a call to CEEDAYS or CEESECS, because the year value is missing. The minimum information required to calculate a Lilian value is either (1) month, day and year, or (2) year and Julian day.

Programmer response: Verify that the picture string specified in a call to CEEDAYS or CEESECS specifies, as a minimum, the location in the input string of either (1) the year, month, and day, or (2) the year and Julian day.

System action: The output value is set to 0.

Symbolic feedback code: CEE2EB

IWZ2508S

The date value passed to CEEDAYS or CEESECS was invalid.

Explanation: In a CEEDAYS or CEESECS call, the value in the DD or DDD field is not valid for the given year and/or month. For example, 'MM/DD/YY' with '02/29/90', or 'YYYY.DDD' with '1990.366' are invalid because 1990 is not a leap year. This code may also be returned for any nonexistent date value such as June 31st, January 0.

Programmer response: Verify that the format of the input data matches the picture string specification and that input data contains a valid date.

System action: The output value is set to 0.

Symbolic feedback code: CEE2EC

IWZ2509S

The era passed to CEEDAYS or CEESECS was not recognized.

Explanation: The value in the <JJJJ>, <CCCC>, or <CCCCCCCC> field passed in a call to CEEDAYS or CEESECS does not contain a supported era name.

Programmer response: Verify that the format of the input data matches the picture string specification and that the spelling of the era name is correct. Note that the era name must be a proper DBCS string where the '<' position must contain the first byte of the era name.

System action: The output value is set to 0.

IWZ2510S

The hours value in a call to CEEISEC or CEESECS was not recognized.

Explanation: (1) In a CEEISEC call, the hours parameter did not contain a number between 0 and 23, or (2) in a CEESECS call, the value in the HH (hours) field does not contain a number between 0 and 23, or the "AP" (a.m./p.m.) field is present and the HH field does not contain a number between 1 and 12.

Programmer response: For CEEISEC, verify that the hours parameter contains an integer between 0 and 23. For CEESECS, verify that the format of the input data matches the picture string specification, and that the hours field contains a value between 0 and 23, (or 1 and 12 if the ″AP″ field is used).

System action: The output value is set to 0.

Symbolic feedback code: CEE2EE

IWZ2511S

The day parameter passed in a CEEISEC call was invalid for year and month specified.

Explanation: The day parameter passed in a CEEISEC call did not contain a valid day number. The combination of year, month, and day formed an invalid date value. Examples: year=1990, month=2, day=29; or month=6, day=31; or day=0.

Programmer response: Verify that the day parameter contains an integer between 1 and 31, and that the combination of year, month, and day represents a valid date.

System action: The output value is set to 0.

Symbolic feedback code: CEE2EF

IWZ2512S

The Lilian date value passed in a call to CEEDATE or CEEDYWK was not within the supported range.

Explanation: The Lilian day number passed in a call to CEEDATE or CEEDYWK was not a number between 1 and 3,074,324.

Programmer response: Verify that the input parameter contains an integer between 1 and 3,074,324.

System action: The output value is set to blanks.

Symbolic feedback code: CEE2EG

IWZ2513S

The input date passed in a CEEISEC, CEEDAYS, or CEESECS call was not within the supported range.

Explanation: The input date passed in a CEEISEC, CEEDAYS, or CEESECS call was earlier than 15 October 1582, or later than 31 December 9999.

Programmer response: For CEEISEC, verify that the year, month, and day parameters form a date greater than or equal to 15 October 1582. For CEEDAYS and CEESECS, verify that the format of the input date matches the picture string specification, and that the input date is within the supported range.

System action: The output value is set to 0.

Symbolic feedback code: CEE2EH

IWZ2514S

The year value passed in a CEEISEC call was not within the supported range.

Explanation: The year parameter passed in a CEEISEC call did not contain a number between 1582 and 9999.

Programmer response: Verify that the year parameter contains valid data, and that the year parameter includes the century, for example, specify year 1990, not year 90.

System action: The output value is set to 0.

Symbolic feedback code: CEE2EI

IWZ2515S

The milliseconds value in a CEEISEC call was not recognized.

Explanation: In a CEEISEC call, the milliseconds parameter (*input_milliseconds*) did not contain a number between 0 and 999.

Programmer response: Verify that the milliseconds parameter contains an integer between 0 and 999.

System action: The output value is set to 0.

Symbolic feedback code: CEE2EJ

IWZ2516S

The minutes value in a CEEISEC call was not recognized.

Explanation: (1) In a CEEISEC call, the minutes parameter (*input_minutes*) did not contain a number between 0 and 59, or (2) in a CEESECS call, the value in the MI (minutes) field did not contain a number between 0 and 59.

Programmer response: For CEEISEC, verify that the minutes parameter contains an integer between 0 and 59. For CEESECS, verify that the format of the input data matches the picture string specification, and that the minutes field contains a number between 0 and 59.

System action: The output value is set to 0.

Symbolic feedback code: CEE2EK

IWZ2517S

The month value in a CEEISEC call was not recognized.

Explanation: (1) In a CEEISEC call, the month parameter (*input_month*) did not contain a number between 1 and 12, or (2) in a CEEDAYS or CEESECS call, the value in the MM field did not contain a number between 1 and 12, or the value in the MMM, MMMM, etc. field did not contain a correctly spelled month name or month abbreviation in the currently active National Language.

Programmer response: For CEEISEC, verify that the month parameter contains an integer between 1 and 12. For CEEDAYS and CEESECS, verify that the format of the input data matches the picture string specification. For the MM field, verify that the input value is between 1 and 12. For spelled-out month names (MMM, MMMM, etc.), verify that the spelling or abbreviation of the month name is correct in the currently active National Language.

System action: The output value is set to 0.

Symbolic feedback code: CEE2EL

IWZ2518S

An invalid picture string was specified in a call to a date/time service.

Explanation: The picture string supplied in a call to one of the date/time services was invalid. Only one era character string can be specified.

Programmer response: Verify that the picture string contains valid data. If the picture string contains more than one era descriptor, such as both <JJJJ> and <CCCC>, then change the picture string to use only one era.

System action: The output value is set to 0.

Symbolic feedback code: CEE2EM

IWZ2519S

The seconds value in a CEEISEC call was not recognized.

Explanation: (1) In a CEEISEC call, the seconds parameter (*input_seconds*) did not contain a number between 0 and 59, or (2) in a CEESECS call, the value in the SS (seconds) field did not contain a number between 0 and 59.

Programmer response: For CEEISEC, verify that the seconds parameter contains an integer between 0 and 59. For CEESECS, verify that the format of the input data matches the picture string specification, and that the seconds field contains a number between 0 and 59.

System action: The output value is set to 0.

Symbolic feedback code: CEE2EN

IWZ2520S

CEEDAYS detected nonnumeric data in a numeric field, or the date string did not match the picture string.

Explanation: The input value passed in a CEEDAYS call did not appear to be in the format described by the picture specification, for example, nonnumeric characters appear where only numeric characters are expected.

Programmer response: Verify that the format of the input data matches the picture string specification and that numeric fields contain only numeric data.

System action: The output value is set to 0.

Symbolic feedback code: CEE2EO

IWZ2521S

The <JJJJ>, <CCCC>, or <CCCCCCCC> year-within-era value passed to CEEDAYS or CEESECS was zero.

Explanation: In a CEEDAYS or CEESECS call, if the YY or ZYY picture token is specified, and if the picture string contains one of the era tokens such as <CCCC> or <JJJJ>, then the year value must be greater than or equal to 1 and must be a valid year value for the era. In this context, the YY or ZYY field means year within era.

Programmer response: Verify that the format of the input data matches the picture string specification and that the input data is valid.

System action: The output value is set to 0.

IWZ2522S

An era (<JJJJ>, <CCCC>, or <CCCCCCCC>) was used in a picture string passed to CEEDATE, but the Lilian date value was not within the supported range. The era could not be determined.

Explanation: In a CEEDATE call, the picture string indicates that the Lilian date is to be converted to an era, but the Lilian date lies outside the range of supported eras.

Programmer response: Verify that the input value contains a valid Lilian day number within the range of supported eras.

System action: The output value is set to blanks.

IWZ2525S

CEESECS detected nonnumeric data in a numeric field, or the timestamp string did not match the picture string.

Explanation: The input value passed in a CEESECS call did not appear to be in the format described by the picture specification. For example, nonnumeric characters appear where only numeric characters are expected, or the a.m./p.m. field (AP, A.P., etc.) did not contain the strings 'AM' or 'PM'.

Programmer response: Verify that the format of the input data matches the picture string specification and that numeric fields contain only numeric data.

System action: The output value is set to 0.

Symbolic feedback code: CEE2ET

IWZ2526S

The date string returned by CEEDATE was truncated.

Explanation: In a CEEDATE call, the output string was not large enough to contain the formatted date value.

Programmer response: Verify that the output string data item is large enough to contain the entire formatted date. Ensure that the output parameter is at least as long as the picture string parameter.

System action: The output value is truncated to the length of the output parameter.

Symbolic feedback code: CEE2EU

IWZ2527S

The timestamp string returned by CEEDATM was truncated.

Explanation: In a CEEDATM call, the output string was not large enough to contain the formatted timestamp value.

Programmer response: Verify that the output string data item is large enough to contain the entire formatted timestamp. Ensure that the output parameter is at least as long as the picture string parameter.

System action: The output value is truncated to the length of the output parameter.

Symbolic feedback code: CEE2EV

IWZ2531S

The local time was not available from the system.

Explanation: A call to CEELOCT failed because the system clock was in an invalid state. The current time cannot be determined.

Programmer response: Notify systems support personnel that the system clock is in an invalid state.

System action: All output values are set to 0.

Symbolic feedback code: CEE2F3

IWZ2533S

The value passed to CEESCEN was not between 0 and 100.

Explanation: The *century_start* value passed in a CEESCEN call was not between 0 and 100, inclusive.

Programmer response: Ensure that the input parameter is within range.

System action: No system action is taken; the 100-year window assumed for all two-digit years is unchanged.

Symbolic feedback code: CEE2F5

IWZ2534W

Insufficient field width was specified for a month or weekday name in a call to CEEDATE or CEEDATM. Output set to blanks.

Explanation: The CEEDATE or CEEDATM callable services issues this message whenever the picture string contained MMM, MMMMMZ, WWW, Wwww, etc., requesting a spelled out month name or weekday name, and the month name currently being formatted contained more characters than can fit in the indicated field.

Programmer response: Increase the field width by specifying enough Ms or Ws to contain the longest month or weekday name being formatted.

System action: The month name and weekday name fields that are of insufficient width are set to blanks. The rest of the output string is unaffected. Processing continues.

Symbolic feedback code: CEE2F6

RELATED CONCEPTS
Appendix D, “Intermediate results and arithmetic precision,” on page 583

RELATED TASKS
“Setting environment variables” on page 209
“Generating a list of compiler error messages” on page 225

RELATED REFERENCES
The Java Native Interface

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 grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
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:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, 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 Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Trademarks

Glossary

The terms in this glossary are defined in accordance with their meaning in COBOL. These terms might or might not have the same meaning in other languages.

This glossary includes terms and definitions from the following publications:

- *ANSI INCITS 23-1985, Programming languages - COBOL*, as amended by *ANSI INCITS 23a-1989, Programming Languages - COBOL - Intrinsic Function Module for COBOL*, and *ANSI INCITS 23b-1993, Programming Languages - Correction Amendment for COBOL*
- *ANSI X3.172-2002, American National Standard Dictionary for Information Systems*

American National Standard definitions are preceded by an asterisk (*).

This glossary includes definitions developed by Sun Microsystems, Inc. for their Java and J2EE glossaries. When Sun is the source of a definition, that is indicated.

A

* **abbreviated combined relation condition**
The combined condition that results from the explicit omission of a common subject or a common subject and common relational operator in a consecutive sequence of relation conditions.

abend Abnormal termination of a program.

* **access mode**
The manner in which records are to be operated upon within a file.

* **actual decimal point**
The physical representation, using the decimal point characters period (.) or comma (,), of the decimal point position in a data item.

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.

alphabetic data item
A data item that is described with a `PICTURE` character string that contains only the symbol `A`. An alphabetic data item has `USAGE DISPLAY`.

* **alphanumeric character**
Any character in the single-byte character set of the computer.

alphanumeric data item
A general reference to a data item that is described implicitly or explicitly as `USAGE DISPLAY`, and that has category alphanumeric, alphanumeric-edited, or numeric-edited.

alphanumeric-edited data item
A data item that is described by a `PICTURE` character string that contains at least one instance of the symbol `A` or `X` and at least one of the simple insertion symbols `B`, `0`, or `/`. An alphanumeric-edited data item has `USAGE DISPLAY`.

* **alphanumeric function**
A function whose value is composed of a string of one or more characters from the alphanumeric character set of the computer.

alphanumeric group item
A group item that is defined without a `GROUP-USAGE NATIONAL` clause. For operations such as `INSPECT`, `STRING`, and `UNSTRING`, an alphanumeric group item is processed as though all its content were described as `USAGE DISPLAY` regardless of

the actual content of the group. For operations that require processing of the elementary items within a group, such as MOVE CORRESPONDING, ADD CORRESPONDING, or INITIALIZE, an alphanumeric group item is processed using group semantics.

alphanumeric literal
A literal that has an opening delimiter from the following set: ', ", X', X", Z', or Z". The string of characters can include any character in the character set of the computer.

*** alternate record key**
A key, other than the prime record key, whose contents identify a record within an indexed file.

ANSI (American National Standards Institute)
An organization that consists of producers, consumers, and general-interest groups and establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

argument
(1) An identifier, a literal, an arithmetic expression, or a function-identifier that specifies a value to be used in the evaluation of a function. (2) An operand of the USING phrase of a CALL or INVOKE statement, used for passing values to a called program or an invoked method.

*** arithmetic expression**
An identifier of a numeric elementary item, a numeric literal, such identifiers and literals separated by arithmetic operators, two arithmetic expressions separated by an arithmetic operator, or an arithmetic expression enclosed in parentheses.

*** arithmetic operation**
The process caused by the execution of an arithmetic statement, or the evaluation of an arithmetic expression, that results in a mathematically correct solution to the arguments presented.

*** arithmetic operator**
A single character, or a fixed two-character combination that belongs to the following set:

Character	**Meaning**
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

*** arithmetic statement**
A statement that causes an arithmetic operation to be executed. The arithmetic statements are ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT.

array An aggregate that consists of data objects, each of which can be uniquely referenced by subscripting. An array is roughly analogous to a COBOL table.

*** ascending key**
A key upon the values of which data is ordered, starting with the lowest value of the key up to the highest value of the key, in accordance with the rules for comparing data items.

ASCII
American National Standard Code for Information Interchange. The standard code uses a coded character set that is based on 7-bit coded characters (8 bits including parity check). The standard is used for information interchange between data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters.

IBM has defined an extension to ASCII (characters 128-255).

ASCII-based multibyte code page
A UTF-8, EUC, or ASCII DBCS code page. Each ASCII-based multibyte code page includes both single-byte and multibyte characters. The encoding of the single-byte characters is the ASCII encoding.

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

big-endian
The default format that the mainframe and the AIX workstation use to store binary data and UTF-16 characters. In this format, the least significant byte of a binary data item is at the highest address and the least significant byte of a UTF-16 character is at the highest address. Compare with *little-endian*.

binary item
A numeric data item that is represented in binary notation (on the base 2 numbering system). The decimal equivalent consists of the decimal digits 0 through 9, plus an operational sign. The leftmost bit of the item is the operational sign.

binary search
A dichotomizing search in which, at each step of the search, the set of data elements is divided by two; some appropriate action is taken in the case of an odd number.

*** block**
A physical unit of data that is normally composed of one or more logical records. For mass storage files, a block can contain a portion of a logical record. The size of a block has no direct relationship to the size of the file within which the block is contained or to the size of the logical records that are either contained within the block or that overlap the block. Synonymous with *physical record*.

breakpoint
A place in a computer program, usually specified by an instruction, where external intervention or a monitor program can interrupt the program as it runs.

buffer
A portion of storage that is used to hold input or output data temporarily.

built-in function
See *intrinsic function*.

byte
A string that consists of a certain number of bits, usually eight, treated as a unit, and representing a character or a control function.

byte order mark (BOM)
A Unicode character that can be used at the start of UTF-16 or UTF-32 text to indicate the byte order of subsequent text; the byte order can be either big-endian or little-endian.

bytecode
Machine-independent code that is generated by the Java compiler and executed by the Java interpreter. (Sun)

C

callable services
In Language Environment, a set of services that a COBOL program can invoke by using the conventional Language Environment-defined call interface. All programs that share the Language Environment conventions can use these services.

called program
A program that is the object of a CALL statement. At run time the called program and calling program are combined to produce a *run unit*.

*** calling program**
A program that executes a CALL to another program.

case structure
A program-processing logic in which a series of conditions is tested in order to choose between a number of resulting actions.

cataloged procedure
A set of job control statements that are placed in a partitioned data set called the procedure library (SYS1.PROCLIB). You can use cataloged procedures to save time and reduce errors in coding JCL.

CCSID
See *coded character set identifier*.

century window
A 100-year interval within which any two-digit year is unique. Several types of century window are available to COBOL programmers:

- For windowed date fields, you use the `YEARWINDOW` compiler option.
- For the windowing intrinsic functions `DATE-TO-YYYYMMDD`, `DAY-TO-YYYYDDD`, and `YEAR-TO-YYYY`, you specify the century window with *argument-2*.

*** character**
The basic indivisible unit of the language.

character encoding unit
A unit of data that corresponds to one code point in a coded character set. One or more character encoding units are used to represent a character in a coded character set. Also known as *encoding unit*.

For `USAGE NATIONAL`, a character encoding unit corresponds to one 2-byte code point of UTF-16.

For `USAGE DISPLAY`, a character encoding unit corresponds to a byte.

For `USAGE DISPLAY-1`, a character encoding unit corresponds to a 2-byte code point in the DBCS character set.

character position
The amount of physical storage or presentation space required to hold or present one character. The term applies to any class of character. For specific classes of characters, the following terms apply:

- *Alphanumeric character position*, for characters represented in `USAGE DISPLAY`
- *DBCS character position*, for DBCS characters represented in `USAGE DISPLAY-1`
- *National character position*, for characters represented in `USAGE NATIONAL`; synonymous with *character encoding unit* for UTF-16

character set
A collection of elements that are used to represent textual information, but for which no coded representation is assumed. See also *coded character set*.

character string
A sequence of contiguous characters that form a COBOL word, a literal, a `PICTURE` character string, or a comment-entry. A character string must be delimited by separators.

checkpoint
A point at which information about the status of a job and the system can be recorded so that the job step can be restarted later.

*** class**
The entity that defines common behavior and implementation for zero, one, or more objects. The objects that share the same implementation are considered to be objects of the same class. Classes can be defined hierarchically, allowing one class to inherit from another.

*** class condition**
The proposition (for which a truth value can be determined) that the content of an item is wholly alphabetic, is wholly numeric, is wholly DBCS, is wholly Kanji, or consists exclusively of the characters that are listed in the definition of a class-name.

*** class definition**
The COBOL source unit that defines a class.

class hierarchy
A tree-like structure that shows relationships among object classes. It places one class at the top and one or more layers of classes below it. Synonymous with *inheritance hierarchy*.

*** class identification entry**
An entry in the `CLASS-ID` paragraph of the `IDENTIFICATION DIVISION`; this entry contains clauses that specify the class-name and assign selected attributes to the class definition.

class-name (object-oriented)
The name of an object-oriented COBOL class definition.

*** class-name (of data)**
A user-defined word that is defined in the `SPECIAL-NAMES` paragraph of the `ENVIRONMENT DIVISION`; this word assigns a name to the proposition (for which a truth value can be defined) that the content of a data item consists exclusively of the characters that are listed in the definition of the class-name.

class object
The runtime object that represents a class.

*** clause**
An ordered set of consecutive COBOL character strings whose purpose is to specify an attribute of an entry.

client In object-oriented programming, a program or method that requests services from one or more methods in a class.

*** COBOL character set**
The set of characters used in writing COBOL syntax. The complete COBOL character set consists of the characters listed below:

Character	**Meaning**
0,1, . . . ,9	Digit
A,B, . . . ,Z	Uppercase letter
a,b, . . . ,z	Lowercase letter
	Space
+	Plus sign
-	Minus sign (hyphen)
*	Asterisk
/	Slant (virgule, slash)
=	Equal sign
$	Currency sign
,	Comma (decimal point)
;	Semicolon
.	Period (decimal point, full stop)
"	Quotation mark
(	Left parenthesis
)	Right parenthesis
>	Greater than symbol
<	Less than symbol
:	Colon

*** COBOL word**
See *word*.

code page
An assignment of graphic characters and control function meanings to all code points. For example, one code page could assign characters and meanings to 256 code points for 8-bit code, and another code page could assign characters and meanings to 128 code points for 7-bit code. For example, one of the IBM code pages for English on the workstation is IBM-1252 and on the host is IBM-1047.

code point
A unique bit pattern that is defined in a coded character set (code page). Graphic symbols and control characters are assigned to code points.

coded character set
A set of unambiguous rules that establish a character set and the relationship between the characters of the set and their coded representation. Examples of coded character sets are the character sets as represented by ASCII or EBCDIC code pages or by the UTF-16 encoding scheme for Unicode.

coded character set identifier (CCSID)
An IBM-defined number in the range 1 to 65,535 that identifies a specific code page.

*** collating sequence**
The sequence in which the characters that are acceptable to a computer are ordered for purposes of sorting, merging, comparing, and for processing indexed files sequentially.

*** column**
A byte position within a print line or within a reference format line. The columns are numbered from 1, by 1, starting at the leftmost position of the line and extending to the rightmost position of the line. A column holds one single-byte character.

*** combined condition**
A condition that is the result of connecting two or more conditions with the `AND` or the `OR` logical operator. See also *condition* and *negated combined condition*.

*** comment-entry**
An entry in the `IDENTIFICATION DIVISION` that can be any combination of characters from the character set of the computer.

*** comment line**
A source program line represented by an asterisk (*) in the indicator area of the line and any characters from the character set of the computer in area A and area B of that line. The comment line serves only for documentation. A special form of

comment line represented by a slant (/) in the indicator area of the line and any characters from the character set of the computer in area A and area B of that line causes page ejection before printing the comment.

*** common program**
: A program that, despite being directly contained within another program, can be called from any program directly or indirectly contained in that other program.

compatible date field
: The meaning of the term *compatible*, when applied to date fields, depends on the COBOL division in which the usage occurs:

- `DATA DIVISION`: Two date fields are compatible if they have identical `USAGE` and meet at least one of the following conditions:
  - They have the same date format.
  - Both are windowed date fields, where one consists only of a windowed year, `DATE FORMAT YY`.
  - Both are expanded date fields, where one consists only of an expanded year, `DATE FORMAT YYYY`.
  - One has `DATE FORMAT YYXXXX`, and the other has `YYXX`.
  - One has `DATE FORMAT YYYYXXXX`, and the other has `YYYYXX`.

  A windowed date field can be subordinate to a data item that is an expanded date group. The two date fields are compatible if the subordinate date field has `USAGE DISPLAY`, starts two bytes after the start of the group expanded date field, and the two fields meet at least one of the following conditions:
  - The subordinate date field has a `DATE FORMAT` pattern with the same number of Xs as the `DATE FORMAT` pattern of the group date field.
  - The subordinate date field has `DATE FORMAT YY`.
  - The group date field has `DATE FORMAT YYYYXXXX` and the subordinate date field has `DATE FORMAT YYXX`.
- `PROCEDURE DIVISION`: Two date fields are compatible if they have the same date format except for the year part, which can be windowed or expanded. For example, a windowed date field with `DATE FORMAT YYXXX` is compatible with:
  - Another windowed date field with `DATE FORMAT YYXXX`
  - An expanded date field with `DATE FORMAT YYYYXXX`

*** compile**
: (1) To translate a program expressed in a high-level language into a program expressed in an intermediate language, assembly language, or a computer language. (2) To prepare a machine-language program from a computer program written in another programming language by making use of the overall logic structure of the program, or generating more than one computer instruction for each symbolic statement, or both, as well as performing the function of an assembler.

*** compile time**
: The time at which COBOL source code is translated, by a COBOL compiler, to a COBOL object program.

compiler
: A program that translates source code written in a higher-level language into machine-language object code.

compiler-directing statement
: A statement that causes the compiler to take a specific action during compilation. The standard compiler-directing statements are `COPY`, `REPLACE`, and `USE`.

compiler directive
: A directive that causes the compiler to take a specific action during compilation. COBOL for AIX has one compiler directive, `CALLINTERFACE`. You can code `CALLINTERFACE` directives within a program to use specific interface conventions for specific `CALL` statements.

*** complex condition**
: A condition in which one or more logical operators act upon one or more conditions. See also *condition*, *negated simple condition*, and *negated combined condition*.

complex ODO
Certain forms of the `OCCURS DEPENDING ON` clause:

- Variably located item or group: A data item described by an `OCCURS` clause with the `DEPENDING ON` option is followed by a nonsubordinate data item or group. The group can be an alphanumeric group or a national group.
- Variably located table: A data item described by an `OCCURS` clause with the `DEPENDING ON` option is followed by a nonsubordinate data item described by an `OCCURS` clause.
- Table with variable-length elements: A data item described by an `OCCURS` clause contains a subordinate data item described by an `OCCURS` clause with the `DEPENDING ON` option.
- Index name for a table with variable-length elements.
- Element of a table with variable-length elements.

component
(1) A functional grouping of related files. (2) In object-oriented programming, a reusable object or program that performs a specific function and is designed to work with other components and applications. JavaBeans™ is Sun Microsystems, Inc.'s architecture for creating components.

* **computer-name**
A system-name that identifies the computer where the program is to be compiled or run.

condition (exception)
Any alteration to the normal programmed flow of an application. Conditions can be detected by the hardware or the operating system and result in an interrupt. They can also be detected by language-specific generated code or language library code.

condition (expression)
A status of data at run time for which a truth value can be determined. Where used in this information in or in reference to "condition" (*condition-1*, *condition-2*,. . .) of a general format, the term refers to a conditional expression that consists of either a simple condition optionally parenthesized or a combined condition (consisting of the syntactically correct combination of simple conditions, logical operators, and parentheses) for which a truth value can be determined. See also *simple condition*, *complex condition*, *negated simple condition*, *combined condition*, and *negated combined condition*.

* **conditional expression**
A simple condition or a complex condition specified in an `EVALUATE`, `IF`, `PERFORM`, or `SEARCH` statement. See also *simple condition* and *complex condition*.

* **conditional phrase**
A phrase that specifies the action to be taken upon determination of the truth value of a condition that results from the execution of a conditional statement.

* **conditional statement**
A statement that specifies that the truth value of a condition is to be determined and that the subsequent action of the object program depends on this truth value.

* **conditional variable**
A data item one or more values of which has a condition-name assigned to it.

* **condition-name**
A user-defined word that assigns a name to a subset of values that a conditional variable can assume; or a user-defined word assigned to a status of an implementor-defined switch or device.

* **condition-name condition**
The proposition (for which a truth value can be determined) that the value of a conditional variable is a member of the set of values attributed to a condition-name associated with the conditional variable.

* **`CONFIGURATION SECTION`**
A section of the `ENVIRONMENT DIVISION` that describes overall specifications of source and object programs and class definitions.

CONSOLE
A COBOL environment-name associated with the operator console.

contained program
A COBOL program that is nested within another COBOL program.

*** contiguous items**
: Items that are described by consecutive entries in the `DATA DIVISION`, and that bear a definite hierarchic relationship to each other.

copybook
: A file or library member that contains a sequence of code that is included in the source program at compile time using the `COPY` statement. The file can be created by the user, supplied by COBOL, or supplied by another product. Synonymous with *copy file*.

*** counter**
: A data item used for storing numbers or number representations in a manner that permits these numbers to be increased or decreased by the value of another number, or to be changed or reset to zero or to an arbitrary positive or negative value.

cross-reference listing
: The portion of the compiler listing that contains information on where files, fields, and indicators are defined, referenced, and modified in a program.

currency-sign value
: A character string that identifies the monetary units stored in a numeric-edited item. Typical examples are $, USD, and EUR. A currency-sign value can be defined by either the `CURRENCY` compiler option or the `CURRENCY SIGN` clause in the `SPECIAL-NAMES` paragraph of the `ENVIRONMENT DIVISION`. If the `CURRENCY SIGN` clause is not specified and the `NOCURRENCY` compiler option is in effect, the dollar sign ($) is used as the default currency-sign value. See also *currency symbol*.

currency symbol
: A character used in a `PICTURE` clause to indicate the position of a currency sign value in a numeric-edited item. A currency symbol can be defined by either the `CURRENCY` compiler option or the `CURRENCY SIGN` clause in the `SPECIAL-NAMES` paragraph of the `ENVIRONMENT DIVISION`. If the `CURRENCY SIGN` clause is not specified and the `NOCURRENCY` compiler option is in effect, the dollar sign ($) is used as the default currency sign value and currency symbol. Multiple currency symbols and currency sign values can be defined. See also *currency sign value*.

*** current record**
: In file processing, the record that is available in the record area associated with a file.

D

*** data clause**
: A clause, appearing in a data description entry in the `DATA DIVISION` of a COBOL program, that provides information describing a particular attribute of a data item.

*** data description entry**
: An entry in the `DATA DIVISION` of a COBOL program that is composed of a level-number followed by a data-name, if required, and then followed by a set of data clauses, as required.

`DATA DIVISION`
: The division of a COBOL program or method that describes the data to be processed by the program or method: the files to be used and the records contained within them; internal working-storage records that will be needed; data to be made available in more than one program in the COBOL run unit.

*** data item**
: A unit of data (excluding literals) defined by a COBOL program or by the rules for function evaluation.

*** data-name**
: A user-defined word that names a data item described in a data description entry. When used in the general formats, data-name represents a word that must not be reference-modified, subscripted, or qualified unless specifically permitted by the rules for the format.

date field
: Any of the following:
- A data item whose data description entry includes a `DATE FORMAT` clause.
- A value returned by one of the following intrinsic functions:

```
DATE-OF-INTEGER
DATE-TO-YYYYMMDD
DATEVAL
```

```
DAY-OF-INTEGER
DAY-TO-YYYYDDD
YEAR-TO-YYYY
YEARWINDOW
```

- The conceptual data items DATE, DATE YYYYMMDD, DAY, and DAY YYYYDDD of the `ACCEPT` statement.
- The result of certain arithmetic operations. For details, see Arithmetic with date fields (*COBOL for AIX Language Reference*).

The term *date field* refers to both *expanded date field* and *windowed date field*. See also *nondate*.

date format
The date pattern of a date field, specified in either of the following ways:

- Explicitly, by the `DATE FORMAT` clause or `DATEVAL` intrinsic function argument-2
- Implicitly, by statements and intrinsic functions that return date fields. For details, see Date field (*COBOL for AIX Language Reference*).

DBCS
See *double-byte character set (DBCS)*.

DBCS character
Any character defined in IBM's double-byte character set.

DBCS character position
See *character position*.

DBCS data item
A data item that is described by a `PICTURE` character string that contains at least one symbol `G`, or, when the `NSYMBOL(DBCS)` compiler option is in effect, at least one symbol `N`. A DBCS data item has `USAGE DISPLAY-1`.

* **debugging line**
Any line with a D in the indicator area of the line.

* **debugging section**
A section that contains a `USE FOR DEBUGGING` statement.

* **declarative sentence**
A compiler-directing sentence that consists of a single `USE` statement terminated by the separator period.

* **declaratives**
A set of one or more special-purpose sections, written at the beginning of the `PROCEDURE DIVISION`, the first of which is preceded by the key word `DECLARATIVE` and the last of which is followed by the key words `END DECLARATIVES`. A declarative is composed of a section header, followed by a `USE` compiler-directing sentence, followed by a set of zero, one, or more associated paragraphs.

* **de-edit**
The logical removal of all editing characters from a numeric-edited data item in order to determine the unedited numeric value of the item.

* **delimited scope statement**
Any statement that includes its explicit scope terminator.

* **delimiter**
A character or a sequence of contiguous characters that identify the end of a string of characters and separate that string of characters from the following string of characters. A delimiter is not part of the string of characters that it delimits.

* **descending key**
A key upon the values of which data is ordered starting with the highest value of key down to the lowest value of key, in accordance with the rules for comparing data items.

digit
Any of the numerals from 0 through 9. In COBOL, the term is not used to refer to any other symbol.

* **digit position**
The amount of physical storage required to store a single digit. This amount can vary depending on the usage specified in the data description entry that defines the data item.

* **direct access**
The facility to obtain data from storage devices or to enter data into a storage device in such a way that the process depends only on the location of that data and not on a reference to data previously accessed.

display floating-point data item
A data item that is described implicitly or explicitly as `USAGE DISPLAY` and that has a `PICTURE` character string that describes an external floating-point data item.

*** division**
A collection of zero, one, or more sections or paragraphs, called the division body, that are formed and combined in accordance with a specific set of rules. Each division consists of the division header and the related division body. There are four divisions in a COBOL program: Identification, Environment, Data, and Procedure.

*** division header**
A combination of words followed by a separator period that indicates the beginning of a division. The division headers are:

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
```

do construct
In structured programming, a `DO` statement is used to group a number of statements in a procedure. In COBOL, an inline `PERFORM` statement functions in the same way.

do-until
In structured programming, a do-until loop will be executed at least once, and until a given condition is true. In COBOL, a `TEST AFTER` phrase used with the `PERFORM` statement functions in the same way.

do-while
In structured programming, a do-while loop will be executed if, and while, a given condition is true. In COBOL, a `TEST BEFORE` phrase used with the `PERFORM` statement functions in the same way.

document type definition (DTD)
The grammar for a class of XML documents. See *XML type definition*.

double-byte character set (DBCS)
A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets. Because each character requires 2 bytes, entering, displaying, and printing DBCS characters requires hardware and supporting software that are DBCS-capable.

*** dynamic access**
An access mode in which specific logical records can be obtained from or placed into a mass storage file in a nonsequential manner and obtained from a file in a sequential manner during the scope of the same `OPEN` statement.

dynamic CALL
A `CALL` *literal* statement in a program that has been compiled with the `DYNAM` option, or a `CALL` *identifier* statement in a program.

*** EBCDIC (Extended Binary-Coded Decimal Interchange Code)**
A coded character set based on 8-bit coded characters.

EBCDIC character
Any one of the symbols included in the EBCDIC (Extended Binary-Coded-Decimal Interchange Code) set.

edited data item
A data item that has been modified by suppressing zeros or inserting editing characters or both.

*** editing character**
A single character or a fixed two-character combination belonging to the following set:

Character	**Meaning**
	Space
0	Zero
+	Plus
-	Minus
CR	Credit
DB	Debit
Z	Zero suppress
*	Check protect
$	Currency sign
,	Comma (decimal point)
.	Period (decimal point)
/	Slant (virgule, slash)

element (text element)
One logical unit of a string of text, such as the description of a single data item or verb, preceded by a unique code identifying the element type.

*** elementary item**
A data item that is described as not being further logically subdivided.

encapsulation
In object-oriented programming, the technique that is used to hide the inherent details of an object. The object provides an interface that queries and manipulates the data without exposing its underlying structure. Synonymous with *information hiding*.

encoding unit
See *character encoding unit*.

end class marker
A combination of words, followed by a separator period, that indicates the end of a COBOL class definition. The end class marker is:

`END CLASS` *`class-name`*`.`

end method marker
A combination of words, followed by a separator period, that indicates the end of a COBOL method definition. The end method marker is:

`END METHOD` *`method-name`*`.`

*** end of `PROCEDURE DIVISION`**
The physical position of a COBOL source program after which no further procedures appear.

*** end program marker**
A combination of words, followed by a separator period, that indicates the end of a COBOL source program. The end program marker is:

`END PROGRAM` *`program-name`*`.`

*** entry**
Any descriptive set of consecutive clauses terminated by a separator period and written in the `IDENTIFICATION DIVISION`, `ENVIRONMENT DIVISION`, or `DATA DIVISION` of a COBOL program.

*** environment clause**
A clause that appears as part of an `ENVIRONMENT DIVISION` entry.

`ENVIRONMENT DIVISION`
One of the four main component parts of a COBOL program, class definition, or method definition. The `ENVIRONMENT DIVISION` describes the computers where the source program is compiled and those where the object program is run. It provides a linkage between the logical concept of files and their records, and the physical aspects of the devices on which files are stored.

environment-name
A name, specified by IBM, that identifies system logical units, printer and card punch control characters, report codes, program switches or all of these. When an environment-name is associated with a mnemonic-name in the `ENVIRONMENT DIVISION`, the mnemonic-name can be substituted in any format in which such substitution is valid.

environment variable
Any of a number of variables that define some aspect of the computing environment, and are accessible to programs that operate in that environment. Environment variables can affect the behavior of programs that are sensitive to the environment in which they operate.

execution time
See *run time*.

execution-time environment
See *runtime environment*.

expanded date field
A date field containing an expanded (four-digit) year. See also *date field* and *expanded year*.

expanded year
A date field that consists only of a four-digit year. Its value includes the century: for example, 1998. Compare with *windowed year*.

*** explicit scope terminator**
A reserved word that terminates the scope of a particular `PROCEDURE DIVISION` statement.

exponent
A number that indicates the power to which another number (the base) is to be raised. Positive exponents denote multiplication; negative exponents denote division; and fractional exponents denote a root of a quantity. In COBOL, an exponential expression is indicated with the symbol ** followed by the exponent.

*** expression**
An arithmetic or conditional expression.

*** extend mode**
The state of a file after execution of an `OPEN` statement, with the `EXTEND` phrase specified for that file, and before the execution of a `CLOSE` statement, without the `REEL` or `UNIT` phrase for that file.

Extensible Markup Language
See *XML*.

extensions
COBOL syntax and semantics supported by IBM compilers in addition to those described in Standard COBOL 85.

external code page
| For ASCII or UTF-8 XML documents, the code page indicated by the current runtime locale. For EBCDIC XML documents, either:

- The code page specified in the EBCDIC_CODEPAGE environment variable
- The default EBCDIC code page selected for the current runtime locale if the EBCDIC_CODEPAGE environment variable is not set

*** external data**
The data that is described in a program as external data items and external file connectors.

*** external data item**
A data item that is described as part of an external record in one or more programs of a run unit and that can be referenced from any program in which it is described.

*** external data record**
A logical record that is described in one or more programs of a run unit and whose constituent data items can be referenced from any program in which they are described.

external decimal data item
See *zoned decimal data item* and *national decimal data item*.

*** external file connector**
A file connector that is accessible to one or more object programs in the run unit.

external floating-point data item
See *display floating-point data item* and *national floating-point data item*.

external program
The outermost program. A program that is not nested.

*** external switch**
A hardware or software device, defined and named by the implementor, which is used to indicate that one of two alternate states exists.

F

factory data
Data that is allocated once for a class and shared by all instances of the class. Factory data is declared in the `WORKING-STORAGE SECTION` of the `DATA DIVISION` in the `FACTORY` paragraph of the class definition, and is equivalent to Java private static data.

factory method
A method that is supported by a class independently of an object instance. Factory methods are declared in the `FACTORY` paragraph of the class definition, and are equivalent to Java public static methods. They are typically used to customize the creation of objects.

*** figurative constant**
A compiler-generated value referenced through the use of certain reserved words.

*** file** A collection of logical records.

*** file attribute conflict condition**
An unsuccessful attempt has been made to execute an input-output operation on a file and the file attributes, as specified for that file in the program, do not match the fixed attributes for that file.

*** file clause**
A clause that appears as part of any of the following `DATA DIVISION` entries: file description entry (`FD` entry) and sort-merge file description entry (`SD` entry).

*** file connector**
A storage area that contains information about a file and is used as the linkage between a file-name and a physical file and between a file-name and its associated record area.

*** file control entry**
: A SELECT clause and all its subordinate clauses that declare the relevant physical attributes of a file.

FILE-CONTROL paragraph
: A paragraph in the ENVIRONMENT DIVISION in which the data files for a given source unit are declared.

*** file description entry**
: An entry in the FILE SECTION of the DATA DIVISION that is composed of the level indicator FD, followed by a file-name, and then followed by a set of file clauses as required.

*** file-name**
: A user-defined word that names a file connector described in a file description entry or a sort-merge file description entry within the FILE SECTION of the DATA DIVISION.

*** file organization**
: The permanent logical file structure established at the time that a file is created.

file position indicator
: A conceptual entity that contains the value of the current key within the key of reference for an indexed file, or the record number of the current record for a sequential file, or the relative record number of the current record for a relative file, or indicates that no next logical record exists, or that an optional input file is not 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 point
: A format for representing numbers in which a real number is represented by a pair of distinct numerals. In a floating-point representation, the real number is the product of the fixed-point part (the first numeral) and a value obtained by raising the implicit floating-point base to a power denoted by the exponent (the second numeral). For example, a floating-point representation of the number 0.0001234 is 0.1234 -3, where 0.1234 is the mantissa and -3 is the exponent.

floating-point data item
: A numeric data item that contains a fraction and an exponent. Its value is obtained by multiplying the fraction by the base of the numeric data item raised to the power that the exponent specifies.

*** format**
: A specific arrangement of a set of data.

*** function**
: A temporary data item whose value is determined at the time the function is referenced during the execution of a statement.

*** function-identifier**
: A syntactically correct combination of character strings and separators that references a function. The data item

represented by a function is uniquely identified by a function-name with its arguments, if any. A function-identifier can include a reference-modifier. A function-identifier that references an alphanumeric function can be specified anywhere in the general formats that an identifier can be specified, subject to certain restrictions. A function-identifier that references an integer or numeric function can be referenced anywhere in the general formats that an arithmetic expression can be specified.

function-name
A word that names the mechanism whose invocation, along with required arguments, determines the value of a function.

function-pointer data item
A data item in which a pointer to an entry point can be stored. A data item defined with the `USAGE IS FUNCTION-POINTER` clause contains the address of a function entry point. Typically used to communicate with C and Java programs.

G

garbage collection
The automatic freeing by the Java runtime system of the memory for objects that are no longer referenced.

*** global name**
A name that is declared in only one program but that can be referenced from the program and from any program contained within the program. Condition-names, data-names, file-names, record-names, report-names, and some special registers can be global names.

group item
(1) A data item that is composed of subordinate data items. See *alphanumeric group item* and *national group item*. (2) When not qualified explicitly or by context as a national group or an alphanumeric group, the term refers to groups in general.

grouping separator
A character used to separate units of digits in numbers for ease of reading. The default is the character comma.

H

header label
(1) A file label or data-set label that precedes the data records on a unit of recording media. (2) Synonym for *beginning-of-file label*.

hide To redefine a factory or static method (inherited from a parent class) in a subclass.

hierarchical file system
A collection of files and directories that are organized in a hierarchical structure and can be accessed by using z/OS UNIX.

*** high-order end**
The leftmost character of a string of characters.

| **host alphanumeric data item**
| (Of XML documents) A category alphanumeric data item whose data description entry does not contain the `NATIVE` phrase, and that was compiled with the `CHAR(EBCDIC)` option in effect. The encoding for the data item is the EBCDIC code page in effect. This code page is determined from the EBCDIC_CODEPAGE environment variable, if set, otherwise from the default code page associated with the runtime locale.

I

IBM COBOL extension
COBOL syntax and semantics supported by IBM compilers in addition to those described in Standard COBOL 85.

ICU See *International Components for Unicode (ICU)*.

`IDENTIFICATION DIVISION`
One of the four main component parts of a COBOL program, class definition, or method definition. The `IDENTIFICATION DIVISION` identifies the program, 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.

IGZCBSO
The COBOL for AIX bootstrap routine. It must be link-edited with any module that contains a COBOL for AIX program.

*** imperative statement**
A statement that either begins with an imperative verb and specifies an unconditional action to be taken or is a conditional statement that is delimited by its explicit scope terminator (delimited scope statement). An imperative statement can consist of a sequence of imperative statements.

*** implicit scope terminator**
A separator period that terminates the scope of any preceding unterminated statement, or a phrase of a statement that by its occurrence indicates the end of the scope of any statement contained within the preceding phrase.

*** index**
A computer storage area or register, the content of which represents the identification of a particular element in a table.

*** index data item**
A data item in which the values associated with an index-name can be stored in a form specified by the implementor.

indexed data-name
An identifier that is composed of a data-name, followed by one or more index-names enclosed in parentheses.

*** indexed file**
A file with indexed organization.

*** indexed organization**
The permanent logical file structure in which each record is identified by the value of one or more keys within that record.

indexing
Synonymous with *subscripting* using index-names.

*** index-name**
A user-defined word that names an index associated with a specific table.

inheritance
A mechanism for using the implementation of a class as the basis for another class. By definition, the inheriting class conforms to the inherited classes. COBOL for AIX does not support *multiple inheritance*; a subclass has exactly one immediate superclass.

inheritance hierarchy
See *class hierarchy*.

*** initial program**
A program that is placed into an initial state every time the program is called in a run unit.

*** initial state**
The state of a program when it is first called in a run unit.

inline
In a program, instructions that are executed sequentially, without branching to routines, subroutines, or other programs.

*** input file**
A file that is opened in the input mode.

*** input mode**
The state of a file after execution of an `OPEN` statement, with the `INPUT` phrase specified, for that file and before the execution of a `CLOSE` statement, without the `REEL` or `UNIT` phrase for that file.

*** input-output file**
A file that is opened in the `I-O` mode.

*** `INPUT-OUTPUT SECTION`**
The section of the `ENVIRONMENT DIVISION` that names the files and the external media required by an object program or method and that provides information required for transmission and handling of data at run time.

*** input-output statement**
A statement that causes files to be processed by performing operations on individual records or on the file as a unit. The input-output statements are `ACCEPT` (with the identifier phrase), `CLOSE`, `DELETE`, `DISPLAY`, `OPEN`, `READ`, `REWRITE`, `SET` (with the `TO ON` or `TO OFF` phrase), `START`, and `WRITE`.

*** input procedure**
A set of statements, to which control is given during the execution of a SORT statement, for the purpose of controlling the release of specified records to be sorted.

instance data
Data that defines the state of an object. The instance data introduced by a class is defined in the WORKING-STORAGE SECTION of the DATA DIVISION in the OBJECT paragraph of the class definition. The state of an object also includes the state of the instance variables introduced by classes that are inherited by the current class. A separate copy of the instance data is created for each object instance.

*** integer**
(1) A numeric literal that does not include any digit positions to the right of the decimal point. (2) A numeric data item defined in the DATA DIVISION that does not include any digit positions to the right of the decimal point. (3) A numeric function whose definition provides that all digits to the right of the decimal point are zero in the returned value for any possible evaluation of the function.

integer function
A function whose category is numeric and whose definition does not include any digit positions to the right of the decimal point.

interlanguage communication (ILC)
The ability of routines written in different programming languages to communicate. ILC support lets you readily build applications from component routines written in a variety of languages.

intermediate result
An intermediate field that contains the results of a succession of arithmetic operations.

*** internal data**
The data that is described in a program and excludes all external data items and external file connectors. Items described in the LINKAGE SECTION of a program are treated as internal data.

*** internal data item**
A data item that is described in one program in a run unit. An internal data item can have a global name.

internal decimal data item
A data item that is described as USAGE PACKED-DECIMAL or USAGE COMP-3, and that has a PICTURE character string that defines the item as numeric (a valid combination of symbols 9, S, P, or V). Synonymous with *packed-decimal data item*.

*** internal file connector**
A file connector that is accessible to only one object program in the run unit.

internal floating-point data item
A data item that is described as USAGE COMP-1 or USAGE COMP-2. COMP-1 defines a single-precision floating-point data item. COMP-2 defines a double-precision floating-point data item. There is no PICTURE clause associated with an internal floating-point data item.

International Components for Unicode (ICU)
An open-source development project sponsored, supported, and used by IBM. ICU libraries provide robust and full-featured Unicode services on a wide variety of platforms, including AIX.

*** intrarecord data structure**
The entire collection of groups and elementary data items from a logical record that a contiguous subset of the data description entries defines. These data description entries include all entries whose level-number is greater than the level-number of the first data description entry describing the intra-record data structure.

intrinsic function
A predefined function, such as a commonly used arithmetic function, called by a built-in function reference.

*** invalid key condition**
A condition, at run time, caused when a specific value of the key associated with an indexed or relative file is determined to be not valid.

*** I-O-CONTROL**
The name of an ENVIRONMENT DIVISION paragraph in which object program requirements for rerun points, sharing of same areas by several data files, and multiple file storage on a single input-output device are specified.

*** I-O-CONTROL entry**
An entry in the I-O-CONTROL paragraph of the ENVIRONMENT DIVISION; this entry contains clauses that provide information required for the transmission and handling of data on named files during the execution of a program.

*** I-O mode**
The state of a file after execution of an OPEN statement, with the I-O phrase specified, for that file and before the execution of a CLOSE statement without the REEL or UNIT phase for that file.

*** I-O status**
A conceptual entity that contains the two-character value indicating the resulting status of an input-output operation. This value is made available to the program through the use of the FILE STATUS clause in the file control entry for the file.

is-a
A relationship that characterizes classes and subclasses in an inheritance hierarchy. Subclasses that have an is-a relationship to a class inherit from that class.

iteration structure
A program processing logic in which a series of statements is repeated while a condition is true or until a condition is true.

J

J2EE
See *Java 2 Platform, Enterprise Edition (J2EE)*.

Java 2 Platform, Enterprise Edition (J2EE)
An environment for developing and deploying enterprise applications, defined by Sun Microsystems, Inc. The J2EE platform consists of a set of services, application programming interfaces (APIs), and protocols that provide the functionality for developing multitiered, Web-based applications. (Sun)

Java Native Interface (JNI)
A programming interface that 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.

JVM
See *Java virtual machine (JVM)*.

K

K
When referring to storage capacity, two to the tenth power; 1024 in decimal notation.

*** key**
A data item that identifies the location of a record, or a set of data items that serve to identify the ordering of data.

*** key of reference**
The key, either prime or alternate, currently being used to access records within an indexed file.

*** keyword**
A reserved word or function-name whose presence is required when the format in which the word appears is used in a source program.

kilobyte (KB)
One kilobyte equals 1024 bytes.

L

*** language-name**
A system-name that specifies a particular programming language.

last-used state
A state that a program is in if its internal values remain the same as when the program was exited (the values are not reset to their initial values).

*** letter**
A character belonging to one of the following two sets:

1. Uppercase letters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z
2. Lowercase letters: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z

*** level indicator**
Two alphabetic characters that identify a specific type of file or a position in a hierarchy. The level indicators in the DATA DIVISION are: CD, FD, and SD.

*** level-number**
A user-defined word (expressed as a two-digit number) that indicates the

hierarchical position of a data item or the special properties of a data description entry. Level-numbers in the range from 1 through 49 indicate the position of a data item in the hierarchical structure of a logical record. Level-numbers in the range 1 through 9 can be written either as a single digit or as a zero followed by a significant digit. Level-numbers 66, 77, and 88 identify special properties of a data description entry.

* **library-name**
: A user-defined word that names a COBOL library that the compiler is to use for compiling a given source program.

* **library text**
: A sequence of text words, comment lines, the separator space, or the separator pseudo-text delimiter in a COBOL library.

Lilian date
: The number of days since the beginning of the Gregorian calendar. Day one is Friday, October 15, 1582. The Lilian date format is named in honor of Luigi Lilio, the creator of the Gregorian calendar.

* **linage-counter**
: A special register whose value points to the current position within the page body.

link
: (1) The combination of the link connection (the transmission medium) and two link stations, one at each end of the link connection. A link can be shared among multiple links in a multipoint or token-ring configuration. (2) To interconnect items of data or portions of one or more computer programs; for example, linking object programs by a linkage editor to produce a shared library.

`LINKAGE SECTION`
: The section in the `DATA DIVISION` of the called program or invoked method that describes data items available from the calling program or invoking method. Both the calling program or invoking method and the called program or invoked method can refer to these data items.

literal
: A character string whose value is specified either by the ordered set of characters comprising the string or by the use of a figurative constant.

little-endian
: The default format that Intel processors use to store binary data and UTF-16 characters. In this format, the most significant byte of a binary data item is at the highest address and the most significant byte of a UTF-16 character is at the highest address. Compare with *big-endian*.

locale
: A set of attributes for a program execution environment that indicates culturally sensitive considerations, such as character code page, collating sequence, date and time format, monetary value representation, numeric value representation, or language.

* **`LOCAL-STORAGE SECTION`**
: The section of the `DATA DIVISION` that defines storage that is allocated and freed on a per-invocation basis, depending on the value assigned in the `VALUE` clauses.

* **logical operator**
: One of the reserved words AND, OR, or NOT. In the formation of a condition, either AND, or OR, or both can be used as logical connectives. NOT can be used for logical negation.

* **logical record**
: The most inclusive data item. The level-number for a record is 01. A record can be either an elementary item or a group of items. Synonymous with *record*.

* **low-order end**
: The rightmost character of a string of characters.

M

main program
: In a hierarchy of programs and subroutines, the first program that receives control when the programs are run within a process.

makefile
: A text file that contains a list of the files for your application. The make utility uses this file to update the target files with the latest changes.

* **mass storage**
: A storage medium in which data can be organized and maintained in both a sequential manner and a nonsequential manner.

*** mass storage device**
A device that has a large storage capacity, such as a magnetic disk.

*** mass storage file**
A collection of records that is stored in a mass storage medium.

MBCS See *multibyte character set (MBCS)*.

*** megabyte (MB)**
One megabyte equals 1,048,576 bytes.

*** merge file**
A collection of records to be merged by a `MERGE` statement. The merge file is created and can be used only by the merge function.

method
Procedural code that defines an operation supported by an object and that is executed by an `INVOKE` statement on that object.

*** method definition**
The COBOL source code that defines a method.

*** method identification entry**
An entry in the `METHOD-ID` paragraph of the `IDENTIFICATION DIVISION`; this entry contains a clause that specifies the method-name.

method invocation
A communication from one object to another that requests the receiving object to execute a method.

method-name
The name of an object-oriented operation. When used to invoke the method, the name can be an alphanumeric or national literal or a category alphanumeric or category national data item. When used in the `METHOD-ID` paragraph to define the method, the name must be an alphanumeric or national literal.

*** mnemonic-name**
A user-defined word that is associated in the `ENVIRONMENT DIVISION` with a specified implementor-name.

module definition file
A file that describes the code segments within a load module.

multibyte character set (MBCS)
A coded character set that is composed of characters represented in a varying number of bytes. Examples are: EUC (Extended Unix Code), UTF-8, and character sets composed of a mixture of single-byte and double-byte EBCDIC or ASCII characters.

multitasking
A mode of operation that provides for the concurrent, or interleaved, execution of two or more tasks.

multithreading
Concurrent operation of more than one path of execution within a computer. Synonymous with *multiprocessing*.

N

name A word (composed of not more than 30 characters) that defines a COBOL operand.

| **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 position
See *character position*.

national data item
A data item of category national, national-edited, or numeric-edited of `USAGE NATIONAL`.

national decimal data item
An external decimal data item that is described implicitly or explicitly as `USAGE NATIONAL` and that contains a valid combination of `PICTURE` symbols `9`, `S`, `P`, and `V`.

national-edited data item
A data item that is described by a `PICTURE` character string that contains at least one instance of the symbol `N` and at least one of the simple insertion symbols `B`, `0`, or `/`. A national-edited data item has `USAGE NATIONAL`.

national floating-point data item
An external floating-point data item that is described implicitly or explicitly as `USAGE NATIONAL` and that has a `PICTURE` character string that describes a floating-point data item.

national group item
A group item that is explicitly or implicitly described with a GROUP-USAGE NATIONAL clause. A national group item is processed as though it were defined as an elementary data item of category national for operations such as INSPECT, STRING, and UNSTRING. This processing ensures correct padding and truncation of national characters, as contrasted with defining USAGE NATIONAL data items within an alphanumeric group item. For operations that require processing of the elementary items within a group, such as MOVE CORRESPONDING, ADD CORRESPONDING, and INITIALIZE, a national group is processed using group semantics.

| **native alphanumeric data item**
| (Of XML documents) A category alphanumeric data item that is described with the NATIVE phrase, or that was compiled with the CHAR(NATIVE) option in effect. The encoding for the data item is the ASCII or UTF-8 code page of the runtime locale in effect.

* **native character set**
The implementor-defined character set associated with the computer specified in the OBJECT-COMPUTER paragraph.

* **native collating sequence**
The implementor-defined collating sequence associated with the computer specified in the OBJECT-COMPUTER paragraph.

native method
A Java method with an implementation that is written in another programming language, such as COBOL.

* **negated combined condition**
The NOT logical operator immediately followed by a parenthesized combined condition. See also *condition* and *combined condition*.

* **negated simple condition**
The NOT logical operator immediately followed by a simple condition. See also *condition* and *simple condition*.

nested program
A program that is directly contained within another program.

* **next executable sentence**
The next sentence to which control will be transferred after execution of the current statement is complete.

* **next executable statement**
The next statement to which control will be transferred after execution of the current statement is complete.

* **next record**
The record that logically follows the current record of a file.

* **noncontiguous items**
Elementary data items in the WORKING-STORAGE SECTION and LINKAGE SECTION that bear no hierarchic relationship to other data items.

nondate
Any of the following items:

- A data item whose date description entry does not include the DATE FORMAT clause
- A literal
- A date field that has been converted using the UNDATE function
- A reference-modified date field
- The result of certain arithmetic operations that can include date field operands; for example, the difference between two compatible date fields

null A figurative constant that is used to assign, to pointer data items, the value of an address that is not valid. NULLS can be used wherever NULL can be used.

* **numeric character**
A character that belongs to the following set of digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

numeric data item
(1) A data item whose description restricts its content to a value represented by characters chosen from the digits 0 through 9. If signed, the item can also contain a +, -, or other representation of an operational sign. (2) A data item of category numeric, internal floating-point, or external floating-point. A numeric data item can have USAGE DISPLAY, NATIONAL, PACKED-DECIMAL, BINARY, COMP, COMP-1, COMP-2, COMP-3, COMP-4, or COMP-5.

numeric-edited data item
A data item that contains numeric data in a form suitable for use in printed output. The data item can consist of external decimal digits from 0 through 9, the

decimal separator, commas, the currency sign, sign control characters, and other editing characters. A numeric-edited item can be represented in either `USAGE DISPLAY` or `USAGE NATIONAL`.

*** numeric function**
A function whose class and category are numeric but that for some possible evaluation does not satisfy the requirements of integer functions.

*** numeric literal**
A literal composed of one or more numeric characters that can contain a decimal point or an algebraic sign, or both. The decimal point must not be the rightmost character. The algebraic sign, if present, must be the leftmost character.

O

object
An entity that has state (its data values) and operations (its methods). An object is a way to encapsulate state and behavior. Each object in the class is said to be an instance of the class.

object code
Output from a compiler or assembler that is itself executable machine code or is suitable for processing to produce executable machine code.

*** `OBJECT-COMPUTER`**
The name of an `ENVIRONMENT DIVISION` paragraph in which the computer environment, where the object program is run, is described.

*** object computer entry**
An entry in the `OBJECT-COMPUTER` paragraph of the `ENVIRONMENT DIVISION`; this entry contains clauses that describe the computer environment in which the object program is to be executed.

object instance
See *object*.

*** object of entry**
A set of operands and reserved words, within a `DATA DIVISION` entry of a COBOL program, that immediately follows the subject of the entry.

object-oriented programming
A programming approach based on the concepts of encapsulation and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates on the data objects that comprise the problem and how they are manipulated, not on how something is accomplished.

object program
A set or group of executable machine-language instructions and other material designed to interact with data to provide problem solutions. In this context, an object program is generally the machine language result of the operation of a COBOL compiler on a source program or class definition. Where there is no danger of ambiguity, the word *program* can be used in place of *object program*.

object reference
A value that identifies an instance of a class. If the class is not specified, the object reference is universal and can apply to instances of any class.

*** object time**
The time at which an object program is executed. Synonymous with *run time*.

*** obsolete element**
A COBOL language element in Standard COBOL 85 that was deleted from Standard COBOL 2002.

ODBC
See *Open Database Connectivity (ODBC)*.

ODO object
In the example below, `X` is the object of the `OCCURS DEPENDING ON` clause (ODO object).

```
WORKING-STORAGE SECTION
01  TABLE-1.
    05  X                   PICS9.
    05  Y OCCURS 3 TIMES
          DEPENDING ON X    PIC X.
```

The value of the ODO object determines how many of the ODO subject appear in the table.

ODO subject
In the example above, `Y` is the subject of the `OCCURS DEPENDING ON` clause (ODO subject). The number of `Y` ODO subjects that appear in the table depends on the value of `X`.

Open Database Connectivity (ODBC)
A specification for an application programming interface (API) that provides access to data in a variety of databases and file systems.

*** open mode**
The state of a file after execution of an OPEN statement for that file and before the execution of a CLOSE statement without the REEL or UNIT phrase for that file. The particular open mode is specified in the OPEN statement as either INPUT, OUTPUT, I-O, or EXTEND.

*** operand**
(1) The general definition of operand is "the component that is operated upon." (2) For the purposes of this document, any lowercase word (or words) that appears in a statement or entry format can be considered to be an operand and, as such, is an implied reference to the data indicated by the operand.

operation
A service that can be requested of an object.

*** operational sign**
An algebraic sign that is associated with a numeric data item or a numeric literal, to indicate whether its value is positive or negative.

optional file
A file that is declared as being not necessarily available each time the object program is run.

*** optional word**
A reserved word that is included in a specific format only to improve the readability of the language. Its presence is optional to the user when the format in which the word appears is used in a source unit.

*** output file**
A file that is opened in either output mode or extend mode.

*** output mode**
The state of a file after execution of an OPEN statement, with the OUTPUT or EXTEND phrase specified, for that file and before the execution of a CLOSE statement without the REEL or UNIT phrase for that file.

*** output procedure**
A set of statements to which control is given during execution of a SORT statement after the sort function is completed, or during execution of a MERGE statement after the merge function reaches a point at which it can select the next record in merged order when requested.

overflow condition
A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage.

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.

*** phrase**
An ordered set of one or more consecutive COBOL character strings that form a portion of a COBOL procedural statement or of a COBOL clause.

*** physical record**
See *block*.

pointer data item
A data item in which address values can be stored. Data items are explicitly defined as pointers with the USAGE IS POINTER clause. ADDRESS OF special registers are implicitly defined as pointer data items. Pointer data items can be compared for equality or moved to other pointer data items.

port (1) To modify a computer program to enable it to run on a different platform. (2) In the Internet suite of protocols, a specific logical connector between the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP) and a higher-level protocol or application. A port is identified by a port number.

portability
The ability to transfer an application program from one application platform to another with relatively few changes to the source program.

preinitialization
The initialization of the COBOL runtime environment in preparation for multiple calls from programs, especially non-COBOL programs. The environment is not terminated until an explicit termination.

*** prime record key**
A key whose contents uniquely identify a record within an indexed file.

*** priority-number**
A user-defined word that classifies sections in the PROCEDURE DIVISION for purposes of segmentation. Segment numbers can contain only the characters 0 through 9. A segment number can be expressed as either one or two digits.

private
As applied to factory data or instance data, accessible only by methods of the class that defines the data.

*** procedure**
A paragraph or group of logically successive paragraphs, or a section or group of logically successive sections, within the PROCEDURE DIVISION.

*** procedure branching statement**
A statement that causes the explicit transfer of control to a statement other than the next executable statement in the sequence in which the statements are written in the source code. The procedure branching statements are: ALTER, CALL, EXIT, EXIT PROGRAM, GO TO, MERGE (with the OUTPUT PROCEDURE phrase), PERFORM and SORT (with the INPUT PROCEDURE or OUTPUT PROCEDURE phrase), XML PARSE.

PROCEDURE DIVISION
The COBOL division that contains instructions for solving a problem.

procedure integration
One of the functions of the COBOL optimizer is to simplify calls to performed procedures or contained programs.

PERFORM procedure integration is the process whereby a PERFORM statement is replaced by its performed procedures. Contained program procedure integration is the process where a call to a contained program is replaced by the program code.

*** procedure-name**
A user-defined word that is used to name a paragraph or section in the PROCEDURE DIVISION. It consists of a paragraph-name (which can be qualified) or a section-name.

procedure-pointer data item
A data item in which a pointer to an entry point can be stored. A data item defined with the USAGE IS PROCEDURE-POINTER clause contains the address of a procedure entry point. Typically used to communicate with COBOL programs.

process
The course of events that occurs during the execution of all or part of a program. Multiple processes can run concurrently, and programs that run within a process can share resources.

program
(1) A sequence of instructions suitable for processing by a computer. Processing may include the use of a compiler to prepare the program for execution, as well as a runtime environment to execute it. (2) A logical assembly of one or more interrelated modules. Multiple copies of the same program can be run in different processes.

*** program identification entry**
In the PROGRAM-ID paragraph of the IDENTIFICATION DIVISION, an entry that contains clauses that specify the program-name and assign selected program attributes to the program.

*** program-name**
In the IDENTIFICATION DIVISION and the end program marker, a user-defined word or alphanumeric literal that identifies a COBOL source program.

project
The complete set of data and actions that are required to build a target, such as a dynamic link library (DLL) or other executable (EXE).

*** pseudo-text**
A sequence of text words, comment lines, or the separator space in a source program or COBOL library bounded by, but not including, pseudo-text delimiters.

*** pseudo-text delimiter**
Two contiguous equal sign characters (==) used to delimit pseudo-text.

*** punctuation character**
A character that belongs to the following set:

Character	**Meaning**
,	Comma
;	Semicolon
:	Colon
.	Period (full stop)
"	Quotation mark
(	Left parenthesis
)	Right parenthesis
	Space
=	Equal sign

Q

*** qualified data-name**
An identifier that is composed of a data-name followed by one or more sets of either of the connectives OF and IN followed by a data-name qualifier.

*** qualifier**
(1) A data-name or a name associated with a level indicator that is used in a reference either together with another data-name (which is the name of an item that is subordinate to the qualifier) or together with a condition-name. (2) A section-name that is used in a reference together with a paragraph-name specified in that section. (3) A library-name that is used in a reference together with a text-name associated with that library.

R

*** random access**
An access mode in which the program-specified value of a key data item identifies the logical record that is

obtained from, deleted from, or placed into a relative or indexed file.

*** record**
See *logical record*.

*** record area**
A storage area allocated for the purpose of processing the record described in a record description entry in the FILE SECTION of the DATA DIVISION. In the FILE SECTION, the current number of character positions in the record area is determined by the explicit or implicit RECORD clause.

*** record description**
See *record description entry*.

*** record description entry**
The total set of data description entries associated with a particular record. Synonymous with *record description*.

record key
A key whose contents identify a record within an indexed file.

*** record-name**
A user-defined word that names a record described in a record description entry in the DATA DIVISION of a COBOL program.

*** record number**
The ordinal number of a record in the file whose organization is sequential.

recording mode
The format of the logical records in a file. Recording mode can be F (fixed length), V (variable length), S (spanned), or U (undefined).

recursion
A program calling itself or being directly or indirectly called by one of its called programs.

recursively capable
A program is recursively capable (can be called recursively) if the RECURSIVE attribute is on the PROGRAM-ID statement.

reel A discrete portion of a storage medium, the dimensions of which are determined by each implementor that contains part of a file, all of a file, or any number of files. Synonymous with *unit* and *volume*.

reentrant
The attribute of a program or routine that lets more than one user share a single copy of a load module.

*** reference format**
A format that provides a standard method for describing COBOL source programs.

reference modification
A method of defining a new category alphanumeric, category DBCS, or category national data item by specifying the leftmost character and length relative to the leftmost character position of a USAGE DISPLAY, DISPLAY-1, or NATIONAL data item.

*** reference-modifier**
A syntactically correct combination of character strings and separators that defines a unique data item. It includes a delimiting left parenthesis separator, the leftmost character position, a colon separator, optionally a length, and a delimiting right parenthesis separator.

*** relation**
See *relational operator* or *relation condition*.

*** relation character**
A character that belongs to the following set:

Character	Meaning
>	Greater than
<	Less than
=	Equal to

*** relation condition**
The proposition (for which a truth value can be determined) that the value of an arithmetic expression, data item, alphanumeric literal, or index-name has a specific relationship to the value of another arithmetic expression, data item, alphanumeric literal, or index name. See also *relational operator*.

*** relational operator**
A reserved word, a relation character, a group of consecutive reserved words, or a group of consecutive reserved words and relation characters used in the construction of a relation condition. The permissible operators and their meanings are:

Character	Meaning
IS GREATER THAN	Greater than
IS >	Greater than
IS NOT GREATER THAN	Not greater than
IS NOT >	Not greater than

Character	Meaning
`IS LESS THAN`	Less than
`IS <`	Less than
`IS NOT LESS THAN`	Not less than
`IS NOT <`	Not less than
`IS EQUAL TO`	Equal to
`IS =`	Equal to
`IS NOT EQUAL TO`	Not equal to
`IS NOT =`	Not equal to
`IS GREATER THAN OR EQUAL TO`	Greater than or equal to
`IS >=`	Greater than or equal to
`IS LESS THAN OR EQUAL TO`	Less than or equal to
`IS <=`	Less than or equal to

*** relative file**
A file with relative organization.

*** relative key**
A key whose contents identify a logical record in a relative file.

*** relative organization**
The permanent logical file structure in which each record is uniquely identified by an integer value greater than zero, which specifies the logical ordinal position of the record in the file.

*** relative record number**
The ordinal number of a record in a file whose organization is relative. This number is treated as a numeric literal that is an integer.

*** reserved word**
A COBOL word that is specified in the list of words that can be used in a COBOL source program, but that must not appear in the program as a user-defined word or system-name.

*** resource**
A facility or service, controlled by the operating system, that an executing program can use.

*** resultant identifier**
A user-defined data item that is to contain the result of an arithmetic operation.

routine
A set of statements in a COBOL program that causes the computer to perform an operation or series of related operations.

*** routine-name**
A user-defined word that identifies a procedure written in a language other than COBOL.

RSD file system
The record sequential delimited file system is a workstation file system that supports sequential files. An RSD file supports all COBOL data types in fixed-length records, can be edited by most file editors, and can be read by programs written in other languages.

*** run time**
The time at which an object program is executed. Synonymous with *object time*.

runtime environment
The environment in which a COBOL program executes.

*** run unit**
A stand-alone object program, or several object programs, that interact by means of COBOL `CALL` or `INVOKE` statements and function at run time as an entity.

S

SBCS See *single-byte character set (SBCS)*.

scope terminator
A COBOL reserved word that marks the end of certain `PROCEDURE DIVISION` statements. It can be either explicit (`END-ADD`, for example) or implicit (separator period).

*** section**
A set of zero, one, or more paragraphs or entities, called a section body, the first of which is preceded by a section header. Each section consists of the section header and the related section body.

*** section header**
A combination of words followed by a separator period that indicates the beginning of a section in any of these divisions: `ENVIRONMENT`, `DATA`, or `PROCEDURE`. In the `ENVIRONMENT DIVISION` and `DATA DIVISION`, a section header is composed of reserved words followed by a separator period. The permissible section headers in the `ENVIRONMENT DIVISION` are:

```
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
```

The permissible section headers in the DATA DIVISION are:

```
FILE SECTION.
WORKING-STORAGE SECTION.
LOCAL-STORAGE SECTION.
LINKAGE SECTION.
```

In the PROCEDURE DIVISION, a section header is composed of a section-name, followed by the reserved word SECTION, followed by a separator period.

*** section-name**
A user-defined word that names a section in the PROCEDURE DIVISION.

selection structure
A program processing logic in which one or another series of statements is executed, depending on whether a condition is true or false.

*** sentence**
A sequence of one or more statements, the last of which is terminated by a separator period.

*** separately compiled program**
A program that, together with its contained programs, is compiled separately from all other programs.

*** separator**
A character or two contiguous characters used to delimit character strings.

*** separator comma**
A comma (,) followed by a space used to delimit character strings.

*** separator period**
A period (.) followed by a space used to delimit character strings.

*** separator semicolon**
A semicolon (;) followed by a space used to delimit character strings.

sequence structure
A program processing logic in which a series of statements is executed in sequential order.

*** sequential access**
An access mode in which logical records are obtained from or placed into a file in a consecutive predecessor-to-successor logical record sequence determined by the order of records in the file.

*** sequential file**
A file with sequential organization.

*** sequential organization**
The permanent logical file structure in which a record is identified by a predecessor-successor relationship established when the record is placed into the file.

serial search
A search in which the members of a set are consecutively examined, beginning with the first member and ending with the last.

77-level-description-entry
A data description entry that describes a noncontiguous data item that has level-number 77.

SFS file system
The Encina Structured File Server file system is a record-oriented file system that supports sequential and key-indexed file access. SFS files can be shared with PL/I applications.

shared library
A library created by the linker that contains at least one subroutine that can be used by multiple processes. Programs and subroutines are linked as usual, but the code common to different subroutines is combined in one library file that can be loaded at run time and shared by many programs. A key to identify the shared library file is in the header of each subroutine.

*** sign condition**
The proposition (for which a truth value can be determined) that the algebraic value of a data item or an arithmetic expression is either less than, greater than, or equal to zero.

signature
(1) The name of an operation and its parameters. (2) The name of a method and the number and types of its formal parameters.

*** simple condition**
Any single condition chosen from this set:
- Relation condition
- Class condition
- Condition-name condition
- Switch-status condition
- Sign condition

See also *condition* and *negated simple condition*.

single-byte character set (SBCS)
A set of characters in which each character is represented by a single byte. See also *ASCII* and *EBCDIC (Extended Binary-Coded Decimal Interchange Code)*.

slack bytes
Bytes inserted between data items or records to ensure correct alignment of some numeric items. Slack bytes contain no meaningful data. In some cases, they are inserted by the compiler; in others, it is the responsibility of the programmer to insert them. The SYNCHRONIZED clause instructs the compiler to insert slack bytes when they are needed for proper alignment. Slack bytes between records are inserted by the programmer.

*** sort file**
A collection of records to be sorted by a SORT statement. The sort file is created and can be used by the sort function only.

*** sort-merge file description entry**
An entry in the FILE SECTION of the DATA DIVISION that is composed of the level indicator SD, followed by a file-name, and then followed by a set of file clauses as required.

*** SOURCE-COMPUTER**
The name of an ENVIRONMENT DIVISION paragraph in which the computer environment, where the source program is compiled, is described.

*** source computer entry**
An entry in the SOURCE-COMPUTER paragraph of the ENVIRONMENT DIVISION; this entry contains clauses that describe the computer environment in which the source program is to be compiled.

*** source item**
An identifier designated by a SOURCE clause that provides the value of a printable item.

source program
Although a source program can be represented by other forms and symbols, in this document the term always refers to a syntactically correct set of COBOL statements. A COBOL source program commences with the IDENTIFICATION DIVISION or a COPY statement and terminates with the end program marker, if specified, or with the absence of additional source program lines.

source unit
A unit of COBOL source code that can be separately compiled: a program or a class definition. Also known as a *compilation unit*.

*** special character**
A character that belongs to the following set:

Character	Meaning
+	Plus sign
-	Minus sign (hyphen)
*	Asterisk
/	Slant (virgule, slash)
=	Equal sign
$	Currency sign
,	Comma (decimal point)
;	Semicolon
.	Period (decimal point, full stop)
"	Quotation mark
(	Left parenthesis
)	Right parenthesis
>	Greater than symbol
<	Less than symbol
:	Colon

SPECIAL-NAMES
The name of an ENVIRONMENT DIVISION paragraph in which environment-names are related to user-specified mnemonic-names.

*** special names entry**
An entry in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION; this entry provides means for specifying the currency sign; choosing the decimal point; specifying symbolic characters; relating implementor-names to user-specified mnemonic-names; relating alphabet-names to character sets or collating sequences; and relating class-names to sets of characters.

*** special registers**
Certain compiler-generated storage areas whose primary use is to store information produced in conjunction with the use of a specific COBOL feature.

Standard COBOL 85
The COBOL language defined by the following standards:

- *ANSI INCITS 23-1985, Programming languages - COBOL*, as amended by *ANSI INCITS 23a-1989, Programming Languages - COBOL - Intrinsic Function Module for COBOL*
- *ISO 1989:1985, Programming languages - COBOL*, as amended by *ISO/IEC 1989/AMD1:1992, Programming languages - COBOL: Intrinsic function module*

*** statement**
A syntactically valid combination of words, literals, and separators, beginning with a verb, written in a COBOL source program.

STL file system
The standard language file system is the native workstation file system for COBOL and PL/I. This system supports sequential, relative, and indexed files.

structured programming
A technique for organizing and coding a computer program in which the program comprises a hierarchy of segments, each segment having a single entry point and a single exit point. Control is passed downward through the structure without unconditional branches to higher levels of the hierarchy.

*** subclass**
A class that inherits from another class. When two classes in an inheritance relationship are considered together, the subclass is the inheritor or inheriting class; the superclass is the inheritee or inherited class.

*** subject of entry**
An operand or reserved word that appears immediately following the level indicator or the level-number in a `DATA DIVISION` entry.

*** subprogram**
See *called program*.

*** subscript**
An occurrence number that is represented by either an integer, a data-name optionally followed by an integer with the operator + or -, or an index-name optionally followed by an integer with the operator + or -, that identifies a particular element in a table. A subscript can be the word `ALL` when the subscripted identifier is used as a function argument for a function allowing a variable number of arguments.

*** subscripted data-name**
An identifier that is composed of a data-name followed by one or more subscripts enclosed in parentheses.

substitution character
A character that is used in a conversion from a source code page to a target code page to represent a character that is not defined in the target code page.

*** superclass**
A class that is inherited by another class. See also *subclass*.

surrogate pair
In the UTF-16 format of Unicode, a pair of encoding units that together represents a single Unicode graphic character. The first unit of the pair is called a *high surrogate* and the second a *low surrogate*. The code value of a high surrogate is in the range X'D800' through X'DBFF'. The code value of a low surrogate is in the range X'DC00' through X'DFFF'. Surrogate pairs provide for more characters than the 65,536 characters that fit in the Unicode 16-bit coded character set.

switch-status condition
The proposition (for which a truth value can be determined) that an UPSI switch, capable of being set to an on or off status, has been set to a specific status.

*** symbolic-character**
A user-defined word that specifies a user-defined figurative constant.

syntax
(1) The relationship among characters or groups of characters, independent of their meanings or the manner of their interpretation and use. (2) The structure of expressions in a language. (3) The rules governing the structure of a language. (4) The relationship among symbols. (5) The rules for the construction of a statement.

*** system-name**
A COBOL word that is used to communicate with the operating environment.

T

*** table**
A set of logically consecutive items of data that are defined in the DATA DIVISION by means of the OCCURS clause.

*** table element**
A data item that belongs to the set of repeated items comprising a table.

*** text-name**
A user-defined word that identifies library text.

*** text word**
A character or a sequence of contiguous characters between margin A and margin R in a COBOL library, source program, or pseudo-text that is any of the following characters:

- A separator, except for space; a pseudo-text delimiter; and the opening and closing delimiters for 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 file or data-set label that follows the data records on a unit of recording medium. (2) Synonym for *end-of-file label*.

troubleshoot
To detect, locate, and eliminate problems in using computer software.

*** truth value**
The representation of the result of the evaluation of a condition in terms of one of two values: true or false.

typed object reference
A data-name that can refer only to an object of a specified class or any of its subclasses.

U

*** unary operator**
A plus (+) or a minus (-) sign that precedes a variable or a left parenthesis in an arithmetic expression and that has the effect of multiplying the expression by +1 or -1, respectively.

Unicode
A universal character encoding standard that supports the interchange, processing, and display of text that is written in any of the languages of the modern world. There are multiple encoding schemes to represent Unicode, including UTF-8, UTF-16, and UTF-32. COBOL for AIX 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 COBOL for AIX, 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.

*** unsuccessful execution**
The attempted execution of a statement that does not result in the execution of all the operations specified by that statement. The unsuccessful execution of a statement does not affect any data referenced by that statement, but can affect status indicators.

UPSI switch
A program switch that performs the functions of a hardware switch. Eight are provided: UPSI-0 through UPSI-7.

| **URI** See *Uniform Resource Identifier (URI)*.

*** user-defined word**
A COBOL word that must be supplied by the user to satisfy the format of a clause or statement.

V

*** variable**
A data item whose value can be changed by execution of the object program. A variable used in an arithmetic expression must be a numeric elementary item.

variable-length item
A group item that contains a table described with the `DEPENDING` phrase of the `OCCURS` clause.

*** variable-length record**
A record associated with a file whose file description or sort-merge description entry permits records to contain a varying number of character positions.

*** variable-occurrence data item**
A variable-occurrence data item is a table element that is repeated a variable number of times. Such an item must contain an `OCCURS DEPENDING ON` clause in its data description entry or be subordinate to such an item.

*** variably located group**
A group item following, and not subordinate to, a variable-length table in the same record. The group item can be an alphanumeric group or a national group.

*** variably located item**
A data item following, and not subordinate to, a variable-length table in the same record.

*** verb**
A word that expresses an action to be taken by a COBOL compiler or object program.

volume
A module of external storage. For tape devices it is a reel; for direct-access devices it is a unit.

VSAM file system
A file system that supports COBOL sequential, relative, and indexed organizations.

W

Web service
A modular application that performs specific tasks and is accessible through open protocols like HTTP and SOAP.

white space
Characters that introduce space into a document. They are:

- Space
- Horizontal tabulation
- Carriage return
- Line feed
- Next line

as named in the Unicode Standard.

windowed date field
A date field containing a windowed (two-digit) year. See also *date field* and *windowed year*.

windowed year
A date field that consists only of a two-digit year. This two-digit year can be interpreted using a century window. For example, 08 could be interpreted as 2008. See also *century window*. Compare with *expanded year*.

*** word**
A character string of not more than 30 characters that forms a user-defined word, a system-name, a reserved word, or a function-name.

*** `WORKING-STORAGE SECTION`**
The section of the `DATA DIVISION` that describes working-storage data items, composed either of noncontiguous items or working-storage records or of both.

workstation
A generic term for computers used by end users including personal computers, 3270 terminals, intelligent workstations, and UNIX terminals. Often a workstation is connected to a mainframe or to a network.

wrapper
An object that provides an interface between object-oriented code and

procedure-oriented code. Using wrappers 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 type definition
An XML element that contains or points to markup declarations that provide a grammar for a class of documents. This grammar is known as a document type definition, or DTD.

Y

year field expansion
Explicit expansion of date fields that contain two-digit years to contain four-digit years in files and databases, and then use of these fields in expanded form in programs. This is the only method for assuring reliable date processing for applications that have used two-digit years.

Z

zoned decimal data item
An external decimal data item that is described implicitly or explicitly as USAGE DISPLAY and that contains a valid combination of PICTURE symbols 9, S, P, and V. The content of a zoned decimal data item is represented in characters 0 through 9, optionally with a sign. If the PICTURE string specifies a sign and the SIGN IS SEPARATE clause is specified, the sign is represented as characters + or -. If SIGN IS SEPARATE is not specified, the sign is one hexadecimal digit that overlays the first 4 bits of the sign position (leading or trailing).

List of resources

COBOL for AIX

Installation Guide, GC23-8569

Language Reference, SC23-8567

Programming Guide, SC23-8566

VSAM File System Reply Messages, SC23-8568

Support

If you have a problem using COBOL for AIX, see the following site, which provides up-to-date support information: www.ibm.com/software/awdtools/cobol/aix/support/.

Related publications

AIX

Commands Reference: Volume 1, a - c, SC23-5243

Commands Reference: Volume 2, d - h, SC23-5244

Commands Reference: Volume 3, i - m, SC23-5245

Commands Reference: Volume 4, n - r, SC23-5246

Commands Reference: Volume 5, s - u, SC23-5247

Commands Reference: Volume 6, v - z, SC23-5248

General Programming Concepts: Writing and Debugging Programs, SC23-5259

XL C/C++ for AIX

Optimization and Programming Guide, SC23-8887

DB2 for Linux, UNIX, and Windows

Command Reference, SC23-5846

SQL Reference Volume 1, SC23-5861

SQL Reference Volume 2, SC23-5862

See also the Information Center: publib.boulder.ibm.com/infocenter/db2luw/v9r5/index.jsp.

Java

IBM SDK for Java - Tools Documentation, publib.boulder.ibm.com/infocenter/javasdk/tools/index.jsp

Java 2 SDK, Standard Edition Documentation, java.sun.com/j2se/1.4.2/docs/

The Java Language Specification, Third Edition, by Gosling et al., java.sun.com/docs/books/jls/

The Java Native Interface, java.sun.com/j2se/1.4.2/docs/guide/jni/

Java Technology Edition SDK User Guides, www.ibm.com/developerworks/java/jdk/aix/service.html

TXSeries for Multiplatforms

CICS Administration Reference, SC34-6641

CICS Application Programming Guide, SC34-6634

CICS Application Programming Reference, SC34-6640

CICS Problem Determination Guide, SC34-6636

Encina Administration Guide Volume 2: Server Administration, SC09-4474

Encina SFS Programming Guide, SC09-4483

Unicode and character representation

Unicode, www.unicode.org/

International Components for Unicode: Converter Explorer, demo.icu-project.org/icu-bin/convexp/

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

_iwzGetCCSID: convert code-page ID to CCSID
- example 204
- syntax 203

_iwzGetLocaleCP: get locale and EBCDIC code-page values
- example 204
- syntax 203

-? cob2 option 229
-# cob2 option 221, 229
-bD cob2 option, specifying heap space with 230
-c cob2 option 227
-cmain cob2 option 228
-comprc_ok cob2 option 227
-Fxxx cob2 option 228
-g cob2 option
- for debugging 228

-host cob2 option
- effect on command-line arguments 495
- effect on compiler options 227
- for host data format 227

-host cob2_j option
- with OO applications 573

-I cob2 option
- searching copybooks 227

-main cob2 option
- specifying main program 228

-o cob2 option
- specifying main program 228

-p cob2 option
- analyzing performance 228

-pg cob2 option 228
-q cob2 option 227
-q32 cob2 option
- description 228
- selects 32-bit Java SDK 241

-q64 cob2 option
- description 228
- selects 64-bit Java SDK 241

-S cob2 option
- specifying stack space 231

-v cob2 option 229
! character, hexadecimal values 373
? cob2 option 229
.adt file 246
.lst file suffix 225
.profile file, setting environment variables in 209
.wlist file 269
*CBL statement 295
*CONTROL statement 295
[character, hexadecimal values 373
] character, hexadecimal values 373
>>CALLINT statement
- description 295

| character, hexadecimal values 373
# character, hexadecimal values 373

Numerics

64-bit mode
- -q64 compiler option
  - description 228
  - selects 64-bit Java SDK 241
- ADDR compiler option 246
- file-system summary 121
- interlanguage communication 471
- migration from 32-bit mode 570
- programming requirements 247
- restrictions
  - can't mix 64-bit and 32-bit COBOL programs 470
  - CICS 337
  - DB2 571
  - overview 247
  - SFS files 117
  - summary 571

A

abends, using ERRCOUNT runtime option to induce 302
ACCEPT statement
- assigning input 34
- under CICS 338
- using in GUI applications 212

ACCEPT statement, environment variables used in 212
accessibility
- assistive technologies xix
- keyboard navigation xix
- of COBOL for AIX xix
- of this document xix

accessing files using environment variables 212
active locale 193
ADATA compiler option 246
adding alternate indexes 138
adding records to files
- effect of STL and VSAM format 121
- overview 133
- randomly or dynamically 134
- sequentially 133

ADDR compiler option 246
ADDRESS OF special register
- 32-bit mode versus 64-bit mode 571
- size depends on ADDR 247
- use in CALL statement 482

addresses
- incrementing 487
- NULL value 486
- passing between programs 486
- passing entry-point addresses 489

AIX, porting to 462
alignment depending on ADDR 247
ALL subscript
- examples 79
- processing table elements iteratively 78

ALL subscript *(continued)*
- table elements as function arguments 54

allocating heap space 230
allocating stack space 231
ALPHABET clause, establishing collating sequence with 8
alphabetic data
- comparing to national 189
- MOVE statement with 32

alphanumeric comparison 86
alphanumeric data
- comparing
  - effect of collating sequence 200
  - effect of ZWB 293
  - to national 189
- converting
  - to national with MOVE 182
  - to national with NATIONAL-OF 183
- MOVE statement with 32

alphanumeric date fields, contracting 539
alphanumeric group item
- a group without GROUP-USAGE NATIONAL 25
- definition 24

alphanumeric literals
- control characters within 26
- description 25
- with multibyte content 191

alphanumeric-edited data
- initializing
  - example 29
  - using INITIALIZE 68
- MOVE statement with 32

alternate collating sequence
- choosing 150
- example 9

alternate file system
- file system ID 212
- using environment variables 212

alternate index
- adding 138

alternate index file
- specifying data volume for 136
- specifying file name for 114

alternate index, definition 124
ANNUITY intrinsic function 55
APOST compiler option 278
applications, porting
- differences between platforms 457
- language differences 457
- mainframe to workstation
  - choosing compiler options 457
  - running mainframe applications on the workstation 458
- using COPY to isolate platform-specific code 457
- Windows to AIX 462

applications, porting *(continued)*
  workstation to mainframe
    workstation-only language features 462
    workstation-only names 462
arguments
  describing in calling program 483
  passing between COBOL and C 476
  passing between COBOL and C, example 475
  passing BY VALUE 483
  passing from COBOL to C, example 474
  passing from COBOL to C++, example 477
  specifying OMITTED 484
  testing for OMITTED arguments 484
  to main program 495
ARITH compiler option
  description 248
arithmetic
  calculation on dates
    convert date to COBOL integer format (CEECBLDY) 600
    convert date to Lilian format (CEEDAYS) 611
    convert timestamp to number of seconds (CEESECS) 632
    get current Greenwich Mean Time (CEEGMT) 617
  COMPUTE statement simpler to code 51
  error handling 158
  with intrinsic functions 53
arithmetic comparisons 57
arithmetic evaluation
  conversions and precision 49
  data format conversion 48
  examples 56, 58
  fixed-point contrasted with floating-point 56
  intermediate results 583
  performance tips 546
  precedence 52, 585
  precision 583
arithmetic expression
  as reference modifier 102
  description of 52
  in nonarithmetic statement 591
  in parentheses 52
  with MLE 533
arithmetic operation
  with MLE 532, 533
arrays
  COBOL 37
  Java
    declaring 449
    manipulating 450
ASCII
  code pages supported in XML documents 371
  converting to EBCDIC 107
  multibyte portability 460
  SBCS portability 458
assembler
  from LIST option 550
assembler *(continued)*
  programs
    listing of 269, 550
assembler language programs
  debugging 328
ASSIGN clause 10
assigning values 27
assignment-name environment variable 212
assistive technologies xix
assumed century window for nondates 530
AT END (end-of-file) 160
attribute methods 411
avoiding coding errors 543
awk command for padding records 120

B

backward branches, avoid 544
Base class
  equating to java.lang.Object 403
  using for java.lang.Object 402
base file, Encina SFS 114
base locator 322
batch compilation 279
batch debugging, activating 302
Bibliography 807
big-endian
  representation of integers 462
  representation of national characters 463
big-endian, converting to little-endian 173
BINARY compiler option 249
binary data item
  general description 44
  intermediate results 588
  synonyms 42
  using efficiently 44, 546
binary data, data representation 249
binary search
  description 77
  example 78
BLANK WHEN ZERO clause
  coded for numeric data 175
  example with numeric-edited data 41
branch, implicit 90
BY CONTENT 481
BY REFERENCE 481
BY VALUE
  description 481
  restrictions 483
  valid data types 483
byte order mark not generated 384

C

C
  functions called from COBOL, example 474
  functions calling COBOL, example 475, 476
C/C++
  and COBOL 471
C/C++ *(continued)*
  communicating with COBOL
    overview 471
    restriction 471
  data types, correspondence with COBOL 473
  multiple calls to a COBOL program 472
C++
  function called from COBOL, example 477
cache 139
caching, client-side
  description 139
  environment variable for 212
  insert caching 139
  read caching 139
CALL identifier
  example call of a shared library 501
call interface conventions
  indicating with CALLINT 250
  with ODBC 357
CALL statement
  BY CONTENT 481
  BY REFERENCE 481
  BY VALUE
    description 481
    restrictions 483
  CALL identifier 470
  CALL literal 470
  effect of CALLINT option 249
  exception condition 165
  for error handling 165
  handling of program-name in 276
  overflow condition 165
  RETURNING 490
  to invoke date and time services 559
  USING 483
  with DYNAM 258
  with ON EXCEPTION 165
  with ON OVERFLOW 19, 165
callable services
  _iwzGetCCSID: convert code-page ID to CCSID 203
  _iwzGetLocaleCP: get locale and EBCDIC code-page values 203
  CEECBLDY: convert date to COBOL integer format 600
  CEEDATE: convert Lilian date to character format 604
  CEEDATM: convert seconds to character timestamp 608
  CEEDAYS: convert date to Lilian format 611
  CEEDYWK: calculate day of week from Lilian date 615
  CEEGMT: get current Greenwich Mean Time 617
  CEEGMTO: get offset from Greenwich Mean Time 619
  CEEISEC: convert integers to seconds 621
  CEELOCT: get current local time 623
  CEEQCEN: query the century window 625
  CEESCEN: set the century window 627

callable services *(continued)*
  CEESECI: convert seconds to integers 629
  CEESECS: convert timestamp to number of seconds 632
  CEEUTC: get Coordinated Universal Time 636
  IGZEDT4: get current date with four-digit year 636
CALLINT compiler option
  description 249
CALLINT statement
  description 295
calls
  between COBOL and C/C++ under CICS 341
  dynamic 470
  exception condition 165
  LINKAGE SECTION 484
  OMITTED arguments 484
  overflow condition 165
  passing arguments 483
  passing data 481
  receiving parameters 483
  recursive 478
  static 470
  to date and time services 559
  to JNI services 443
CANCEL statement
  handling of program-name in 276
case structure, EVALUATE statement for 83
CBL statement
  description 295
  specifying compiler options 221
CCSID
  conflict in XML documents 376
  definition 173
  in PARSE statement 363
  of XML documents 371
  of XML documents to be parsed 363
CEECBLDY: convert date to COBOL integer format
  example 601
  syntax 600
CEEDATE: convert Lilian date to character format
  example 605
  syntax 604
  table of sample output 607
CEEDATM: convert seconds to character timestamp
  CEESECI 629
  example 609
  syntax 608
  table of sample output 610
CEEDAYS: convert date to Lilian format
  example 613
  syntax 611
CEEDYWK: calculate day of week from Lilian date
  example 615
  syntax 615
CEEGMT: get current Greenwich Mean Time
  example 618
  syntax 617
CEEGMTO: get offset from Greenwich Mean Time
  example 620
  syntax 619
CEEISEC: convert integers to seconds
  example 622
  syntax 621
CEELOCT: get current local time
  example 625
  syntax 623
CEEQCEN: query the century window
  example 626
  syntax 625
CEESCEN: set the century window
  example 628
  syntax 627
CEESECI: convert seconds to integers
  example 630
  syntax 629
CEESECS: convert timestamp to number of seconds
  example 634
  syntax 632
century window
  assumed for nondates 530
  CEECBLDY 602
  CEEDAYS 613
  CEEQCEN 625
  CEESCEN 627
  CEESECS 634
  example of querying and changing 565
  fixed 523
  overview 565
  sliding 523
chained-list processing
  example 487
  overview 486
changing
  characters to numbers 105
  file-name 11
  title on source listing 6
CHAR compiler option
  description 250
  effect on XML document encodings 371
  multibyte portability 460
  SBCS portability 458
CHAR intrinsic function, example 108
character set, definition 173
character timestamp
  converting Lilian seconds to (CEEDATM) 608
    example 609
  converting to COBOL integer format (CEECBLDY) 600
    example 603
  converting to Lilian seconds (CEESECS) 632
    example 632
CHECK runtime option 301
  performance considerations 552
  reference modification 101
checking errors, flagging at run time 301
checking for valid data
  conditional expressions 86
Chinese GB 18030 data processing 186
CICS
  accessing CICS files from non-CICS applications 340
  calls between COBOL and C/C++ 341
  coding programs 337
  commands relevant to COBOL 337
  compiler options 341
  debugging programs 342
  developing COBOL programs for 337
  DFHCOMMAREA parameter for dynamic calls 339
  DFHEIBLK parameter for dynamic calls 339
  dynamic calls under 339
  host data format not supported 338
  integrated translator
    advantages 342
    overview 342
  portability considerations 338
  restrictions
    applications must be 32-bit 337
    DYNAM compiler option 259
    nested programs 337
    OO programs 337, 397
    preinitialization 513
    separate translator 342
  runtime options 341
  separate translator
    restrictions 342
  shared libraries 340
  system date, getting 338
  TRAP runtime option, effect of 304
CICS compiler option
  description 252
  effect of cicstcl -p command 342
  enables integrated translator 342
  multioption interaction 245
  specifying suboptions 253
cicsmap command 337
cicstcl command 337
  -p flag for integrated translator 342
cicsterm command 337
class
  defining 400
  definition of 397
  factory data 430
  instance data 404
  instantiating
    COBOL 423
    Java 423
  name
    external 403, 415
    in a program 402
  object, obtaining reference with JNI 444
  user-defined 10
class condition
  testing
    for DBCS 192
    for Kanji 191
    for numeric 51
    overview 86
  validating data 309

CLASSPATH environment variable
  description 212
  specifying location of Java classes 239
client
  defining 414
  definition of 414
client-side caching
  description 139
  environment variable for 212
  insert caching 139
  read caching 139
clustered files 117
cob2 command
  command-line argument format 495
  description 219
  examples using, for compiling 220
  examples using, for linking 231
  modifying configuration file 222
  options
    -# 221
    -host 495
    -q abbreviation for ADDR 247
    description 227
    modifying defaults 221
  stanza used 222
cob2 stanza 222
cob2_j command
  command-line argument format 495
  description 219
  for compiling OO applications 237
  for linking OO applications 238
  for OO applications 219
  options
    -host 573
    -q abbreviation for ADDR 247
    -q32 selects 32-bit SDK 241
    -q64 selects 64-bit SDK 241
    description 227
  stanza used 222
cob2_j stanza 222
cob2_r command
  description 219
  for threaded or CICS applications 219
  options 227
  stanza used 222
  threading example 510
cob2_r stanza 222
cob2.cfg configuration file 221
COBCPYEXT environment variable 210
COBJVMINITOPTIONS environment variable
  description 212
  specifying JVM options 240
COBLSTDIR environment variable 210
COBMSGS environment variable 212
COBOL
  and C/C++ 471
  and Java
    communicating between 443
    compatibility 569
    compiling 237
    linking 238
    running 239
    structuring applications 439
COBOL *(continued)*
  called by C functions, example 475, 476
  calling C functions, example 474
  calling C++ function, example 477
  data types, correspondence with C/C++ 473
  object-oriented
    compiling 237
    linking 238
    running 239
COBOL client
  example 433
  example of passing object references 420
COBOL for AIX
  accessibility xix
  compatibility 569
  runtime messages 725
COBOL terms 23
COBOPT environment variable 210
COBPATH environment variable
  CICS dynamic calls 339
  description 212
COBRTOPT environment variable 212
code
  copy 557
  optimized 551
code page
  accessing 203
  ASCII 195
  conflict in XML documents 376
  definition 173
  EBCDIC 195
  EUC 195
  euro currency support 59
  for alphabetic data item 194
  for alphanumeric data item 194
  for DBCS data item 194
  for national data item 194
  hexadecimal values of special characters 373
  overriding 184
  querying 203
  specifying 372
  system default 196
  using characters from 194
  UTF-8 195
  valid 197
code point, definition 173
coded character set
  definition 173
  in XML documents 371
coding
  class definition 400
  clients 414
  condition tests 87
  constructor methods 431
  DATA DIVISION 11
  decisions 81
  efficiently 543
  ENVIRONMENT DIVISION 7
  EVALUATE statement 83
  factory definition 429
  factory methods 431
  file input/output 122
coding *(continued)*
  for files
    example 127
    overview 127
  for SFS files, example 137
  IDENTIFICATION DIVISION 5
  IF statement 81
  input/output
    example 127
    overview 127
  instance methods 405, 428
  interoperable data types with Java 448
  loops 89
  OO programs
    overview 397
  PROCEDURE DIVISION 16
  programs to run under CICS 337
    overview 337
    system date, getting 338
  programs to run under DB2
    overview 331
  restrictions under CICS 337
  simplifying 557
  SQL statements 332
  subclasses
    example 428
    overview 425
  tables 61
  techniques 11, 543
  test conditions 87
collating sequence
  alphanumeric 200
  alternate
    choosing 150
    example 9
  ASCII 8
  binary for national keys 149
  binary for national sort or merge keys 202
  COLLSEQ effect on alphanumeric and DBCS operands 253
  controlling 199
  DBCS 201
  EBCDIC 8
  HIGH-VALUE 8
  intrinsic functions and 202
  ISO 7-bit code 8
  LOW-VALUE 8
  MERGE 8, 150
  national 202
  NATIVE 8
  NCOLLSEQ effect on national operands 273
  nonnumeric comparisons 8
  ordinal position of a character 107
  portability considerations 459
  SEARCH ALL 8
  SORT 8, 150
  specifying 8
  STANDARD-1 8
  STANDARD-2 8
  symbolic characters in the 9
COLLATING SEQUENCE phrase
  does not apply to national keys 149
  effect on sort and merge keys 200

COLLATING SEQUENCE phrase *(continued)*
overrides PROGRAM COLLATING SEQUENCE clause 8, 150
portability considerations 459
use in SORT or MERGE 150
COLLSEQ compiler option
description 253
effect on alphanumeric collating sequence 199
effect on DBCS collating sequence 201
portability considerations 459
columns in tables 61
command prompt, defining environment variables 209
command-line arguments
example with -host option 496
example without -host option 495
using 495
COMMON attribute 6, 466
COMP (COMPUTATIONAL) 44
COMP-1 (COMPUTATIONAL-1)
format 46
performance tips 547
COMP-2 (COMPUTATIONAL-2)
format 46
performance tips 547
COMP-3 (COMPUTATIONAL-3) 45
COMP-4 (COMPUTATIONAL-4) 44
COMP-5 (COMPUTATIONAL-5) 45
comparing data items
alphanumeric
effect of collating sequence 200
effect of COLLSEQ 253
date fields 527
DBCS
effect of collating sequence 201
effect of COLLSEQ 253
literals 191
to alphanumeric groups 201
to national 201
national
effect of collating sequence 202
effect of NCOLLSEQ 187
overview 186
to alphabetic, alphanumeric, or DBCS 189
to alphanumeric groups 189
to numeric 188
two operands 187
object references 417
zoned decimal and alphanumeric, effect of ZWB 293
compatibility
32-bit mode and 64-bit mode 570
data-item size 569
dates
in comparisons 527
in DATA DIVISION 527
in PROCEDURE DIVISION 527
FILEMODE 121
Java and COBOL 569
object code 569
object-oriented syntax 569
SFS (Encina) files 570
small-format files 121
compatibility *(continued)*
XML generation 570
compatibility mode 39, 583
compilation
statistics 319
tailoring 222
COMPILE compiler option
description 255
use NOCOMPILE to find syntax errors 312
compile-time considerations
compiler-directed errors 225
compiling programs 227
compiling without linking 227
display cob2 help 229
display compile and link steps 229
error messages
determining what severity level to produce 264
severity levels 225
executing compile and link steps after display 229
using a nondefault configuration file 228
compiler
calculation of intermediate results 584
date-related messages, analyzing 537
generating list of error messages 225
invoking 219
limits
DATA DIVISION 12
messages
choosing severity to be flagged 313
determining what severity level to produce 264
embedding in source listing 313
severity levels 225
compiler listings
getting 317
specifying output directory 210
compiler options
abbreviations 243
ADATA 246
ADDR 246
APOST 278
ARITH 248
BINARY 249
CALLINT 249
CHAR 250
CICS 252
COLLSEQ 253
COMPILE 255
conflicting 245
CURRENCY 255
DATEPROC 256
DIAGTRUNC 257
DYNAM 258, 552
ENTRYINT 259
EXIT 259
FLAG 264, 313
FLAGSTD 265
FLOAT 267
for CICS 341
for debugging
overview 311
compiler options *(continued)*
for debugging *(continued)*
TEST restriction 310
THREAD restriction 310
for portability 457
LIB 268
LINECOUNT 268
LIST 269, 317
LSTFILE 269
MAP 270, 316
MAXMEM 271
MDECK 272
NCOLLSEQ 273
NOCOMPILE 312
NSYMBOL 273
NUMBER 274, 318
on compiler invocation 319
OPTIMIZE 275, 550
performance considerations 552
PGMNAME 276
PROFILE 277
QUOTE 278
SEPOBJ 278
SEQUENCE 280
SIZE 280
SOSI 281
SOURCE 282, 317
SPACE 283
specifying
cob2 command 219
cob2_j command 219
cob2_r command 219
environment variable 210
in shell script 220
using COBOPT 210
specifying with PROCESS (CBL) 221
SPILL 284
SQL
coding suboptions 333
description 284
SSRANGE 285, 312
performance considerations 552
status 319
table of 243
TERMINAL 286
TEST
description 286
performance considerations 552
use for debugging 316
THREAD
debugging restriction 310
description 287
performance considerations 552
TRUNC 288
performance considerations 552
VBREF 290, 317
WSCLEAR 291
XREF 291, 315
YEARWINDOW 292
ZWB 293
compiler-directing statements
description 295
overview 19
compiling
for SFS file system 220
OO applications
cob2_j command 237

compiling *(continued)*
  OO applications *(continued)*
    example 239
  programs 218
  tailoring the configuration file 222
compiling and linking
  OO applications
    cob2_j command 238
    example 239
completion code
  merge 151
  sort 151
complex OCCURS DEPENDING ON
  basic forms of 593
  complex ODO item 593
  variably located data item 593
  variably located group 593
computation
  arithmetic data items 546
  constant data items 545
  duplicate 545
  of indexes 66
  of subscripts 549
COMPUTATIONAL (COMP) 44
COMPUTATIONAL-1 (COMP-1)
  format 46
  performance tips 547
COMPUTATIONAL-2 (COMP-2)
  format 46
  performance tips 547
COMPUTATIONAL-3 (COMP-3)
  date fields, potential problems 539
  description 45
COMPUTATIONAL-4 (COMP-4) 44
COMPUTATIONAL-5 (COMP-5) 45
COMPUTE statement
  assigning arithmetic results 34
  simpler to code 51
computer, describing 7
concatenating data items (STRING) 93
condition handling
  date and time services and 560
  effect of ERRCOUNT 302
condition testing 87
condition-name 529
conditional expression
  EVALUATE statement 81
  IF statement 81
  PERFORM statement 91
conditional statement
  overview 18
  with NOT phrase 19
  with object references 417
configuration file
  default 221
  modifying 222
  stanzas 223
  tailoring 222
CONFIGURATION SECTION 7
conflicting compiler options 245
conformance requirements
  example of passing object references in INVOKE 420
  RETURNING phrase of INVOKE 421
  USING phrase of INVOKE 419
constants
  computations 545
constants *(continued)*
  data items 545
  definition 26
  figurative, definition 26
contained program integration 551
continuation
  of program 158
  syntax checking 255
CONTINUE statement 81
contracting alphanumeric dates 539
control
  in nested programs 466
  program flow 81
  transfer 465
CONTROL statement 295
convert character format to Lilian date (CEEDAYS) 611
convert Lilian date to character format (CEEDATE) 604
converting data items
  between code pages 107
  between data formats 48
  precision 49
  reversing order of characters 105
  to alphanumeric
    with DISPLAY 36
    with DISPLAY-OF 184
  to Chinese GB 18030 from national 186
  to integers with INTEGER, INTEGER-PART 102
  to national
    from Chinese GB 18030 186
    from UTF-8 185
    with ACCEPT 35
    with MOVE 182
    with NATIONAL-OF 183
  to numbers with NUMVAL, NUMVAL-C 105
  to uppercase or lowercase
    with INSPECT 104
    with intrinsic functions 105
  to UTF-8 from national 185
  with INSPECT 103
  with intrinsic functions 104
converting files to expanded date form, example 525
CONVERTING phrase (INSPECT), example 104
coprocessor, DB2
  overview 331
  using SQL INCLUDE with 332
copy code, obtaining from user-supplied module 260
copy libraries
  example 558
COPY name
  file suffixes searched 210
COPY statement
  description 295
  example 558
  nested 263, 557
  search rules 295
  using for portability 457
copybook cross-reference, description 315
copybooks
  cross-reference 325
  library-name environment variable 210
  odbc3
    description 350
    example 351
  odbc3d
    description 350
    example 351
    source 355
  odbc3p
    description 350
    example 351
    source 352
  search rules 295
  searching for 227
  specifying search paths with SYSLIB 210
  using 557
  with ODBC 350
COUNT IN phrase
  UNSTRING 95
  XML GENERATE 384
counting
  characters (INSPECT) 103
  generated XML characters 380
creating
  alternate index files 136
  objects 422
  SFS files
    environment variables for 136
    sfsadmin command for 138
  variable-length tables 73
cross-reference
  COPY/BASIS 325
  COPY/BASIS statements 317
  copybooks 317
  data and procedure-names 315
  embedded 317
  list 291
  program-name 325
  special definition symbols 326
  text-names and file names 315
  verb list 290
  verbs 317
cultural conventions, definition 193
CURRENCY compiler option 255
currency signs
  euro 59
  hexadecimal literals 59
  multiple-character 59
  using 58
CURRENT-DATE intrinsic function
  example 55
  under CICS 339
customer support 807
customizing
  setting environment variables 209

D

data
  concatenating (STRING) 93
  efficient execution 543
  format conversion of 48
  format, numeric types 42

data *(continued)*
grouping 485
incompatible 51
naming 13
numeric 39
passing 481
record size 13
splitting (UNSTRING) 95
validating 51
data and procedure-name cross-reference, description 315
data areas, dynamic 258
data definition 321
data definition attribute codes 321
data description entry 12
DATA DIVISION
client 416
coding 11
description 11
factory data 430
factory method 431
FD entry 11
FILE SECTION 11
GROUP-USAGE NATIONAL clause 62
instance data 404, 427
instance method 407
limits 12
LINKAGE SECTION 11, 15
listing 316
LOCAL-STORAGE SECTION 11
mapping of items 270, 316
OCCURS clause 61
OCCURS DEPENDING ON (ODO) clause 73
REDEFINES clause 69
restrictions 12
USAGE clause at the group level 25
USAGE IS INDEX clause 66
USAGE NATIONAL clause at the group level 178
WORKING-STORAGE SECTION 11
data item
alignment depends on ADDR 247
coding Java types 448
common, in subprogram linkage 483
concatenating (STRING) 93
converting characters (INSPECT) 103
converting characters to numbers 105
converting to uppercase or lowercase 105
converting with intrinsic functions 104
counting characters (INSPECT) 103
elementary, definition 24
evaluating with intrinsic functions 107
finding the smallest or largest item 108
group, definition 24
index, referring to table elements with 64
initializing, examples of 28
numeric 39
reference modification 99
referring to a substring 99
data item *(continued)*
replacing characters (INSPECT) 103
reversing characters 105
size compatibility 569
splitting (UNSTRING) 95
unused 275
variably located 593
data manipulation
character data 93
DATA RECORDS clause 13
data representation
compiler option affecting 249
portability 458
data types, correspondence between COBOL and C/C++ 473
data-name
cross-reference 324
in MAP listing 321
date and time
format
converting from character format to COBOL integer format (CEECBLDY) 600
converting from character format to Lilian format (CEEDAYS) 611
converting from integers to seconds (CEEISEC) 621
converting from Lilian format to character format (CEEDATE) 604
converting from seconds to character timestamp (CEEDATM) 608
converting from seconds to integers (CEESECI) 629
converting from timestamp to number of seconds (CEESECS) 632
getting date and time (CEELOCT) 623
intrinsic functions 600
services
CEECBLDY: convert date to COBOL integer format 600
CEEDATE: convert Lilian date to character format 604
CEEDATM: convert seconds to character timestamp 608
CEEDAYS: convert date to Lilian format 611
CEEDYWK: calculate day of week from Lilian date 615
CEEGMT: get current Greenwich Mean Time 617
CEEGMTO: get offset from Greenwich Mean Time 619
CEEISEC: convert integers to seconds 621
CEELOCT: get current local time 623
CEEQCEN: query the century window 625, 626
CEESCEN: set the century window 627
CEESECI: convert seconds to integers 629
date and time *(continued)*
services *(continued)*
CEESECS: convert timestamp to number of seconds 632
CEEUTC: get Coordinated Universal Time 636
condition feedback 561
condition handling 560
examples of using 560
feedback code 559
invoking with a CALL statement 559
list of 599
overview 599
performing calculations with 560
picture strings 562
return code 559
RETURN-CODE special register 559
syntax 632
date arithmetic 533
date comparisons 527
date field expansion
advantages 522
description 525
date fields, potential problems with 538
DATE FORMAT clause
cannot use with national data 520
use for automatic date recognition 519
date information, formatting 212
date operations
finding date of compilation 111
intrinsic functions for 37
date processing with internal bridges, advantages 522
date windowing
advantages 522
example 523, 529
how to control 535
MLE approach 522
when not supported 528
DATE-COMPILED paragraph 5
DATE-OF-INTEGER intrinsic function 55
DATEPROC compiler option
analyzing warning-level messages 537
description 256
DATEVAL intrinsic function
example 536
using 535
day of week, calculating with CEEDYWK 615
DB2
64-bit mode considerations 571
bind file name 334
coding considerations 331
coprocessor
overview 331
using SQL INCLUDE with 332
ignored options 334
options 333
package name 334
precompiler requires NODYNAM 331

DB2 *(continued)*
  SQL statements
    coding 332
    overview 331
    return codes 333
    SQL INCLUDE 332
    using binary data in 333
DB2DBDFT environment variable 210, 333
DB2INCLUDE environment variable 332
DBCS comparison 86
DBCS data
  comparing
    effect of collating sequence 201
    literals 191
    to alphanumeric groups 201
    to national 189, 201
  converting
    to national, overview 192
  declaring 190
  encoding and storage 181
  literals
    comparing 191
    description 26
    maximum length 191
    using 190
  MOVE statement with 32
  testing for 191
DEBUG runtime option 302
debugging
  activating batch features 302
  assembler 328
  CICS programs 342
  compiler options for
    overview 311
    TEST restriction 310
    THREAD restriction 310
  idebug command 328
  overview 307
  producing symbolic information 228
  runtime options for 310
  user exits 328
  using COBOL language features 307
  using the debugger 316
  with message offset information 327
debugging, language features
  class test 309
  debugging lines 310
  debugging statements 310
  declaratives 309
  DISPLAY statements 307
  file status keys 308
  INITIALIZE statements 309
  scope terminators 308
  SET statements 309
  WITH DEBUGGING MODE clause 310
declarative procedures
  EXCEPTION/ERROR 160
  USE FOR DEBUGGING 309
defining
  files
    example 127
    overview 127
  SFS files, example 137
deleting records from files 135
delimited scope statement
  description of 18
  nested 20
depth in tables 63
DESC suboption of CALLINT compiler option 250
DESCRIPTOR suboption of CALLINT compiler option 250
DFHCOMMAREA parameter
  use in CICS dynamic calls 339
DFHEIBLK parameter
  use in CICS dynamic calls 339
diagnostics, program 319
DIAGTRUNC compiler option 257
differences from host COBOL 573
direct-access
  direct indexing 66
directories
  adding a path to 227
  for error listing file 225
  specifying path for shared libraries 212
dirty read 140
DISPLAY (USAGE IS)
  encoding and storage 181
  external decimal 43
  floating point 44
display floating-point data (USAGE DISPLAY) 44
DISPLAY statement
  displaying data values 35
  using in debugging 307
  using in GUI applications 212
DISPLAY statement, environment variables used in 212
DISPLAY-1 (USAGE IS)
  encoding and storage 181
DISPLAY-OF intrinsic function
  example with Chinese data 186
  example with Greek data 184
  example with UTF-8 data 185
  using 184
  with XML documents 372
do loop 91
do-until 91
do-while 91
document encoding declaration 372
documentation of program 7
dspcat command, listing messages with 725
dumps, TRAP(OFF) side effect 304
duplicate computations, grouping 545
DYNAM compiler option
  description 258
  effect on CALL literal 471
  performance considerations 552
dynamic calls
  cannot use for DB2 APIs 331
  under CICS 339
dynamic linking
  definition 470
  resolution of shared library references 498
dynamic loading, requirements 212

E

E-level error message 225, 313
EBCDIC
  code pages supported in XML documents 371
  converting to ASCII 107
  multibyte portability 460
  SBCS portability 458
EBCDIC_CODEPAGE environment variable
  setting 212
efficiency of coding 543
embedded cross-reference
  description 317
  example 326
embedded error messages 313
embedded MAP summary 316, 321
embedded SQL
  advantages of 345
  comparison with ODBC 345
Encina SFS file system
  accessing SFS files
    example 137
    overview 136
  description 117
  fully qualified file names 117
  improving performance 139
  nonhierarchical 117
  restriction 117
  system administration of 117
Encina SFS files
  accessing 340
    example 137
    overview 136
  adding alternate indexes 138
  alternate index file name 114
  base file name 114
  COBOL coding example 137
  compatibility 570
  compiling for 220
  creating alternate index files 136
  creating SFS files
    environment variables for 136
    sfsadmin command for 138
  determining available data volumes 136
  file names 117
  fully qualified name 137
  identifying 113
  improving performance 139
  nontransactional access 117
  organization 117
  primary and secondary indexes 117
  processing 115
  specifying data volume for 136
Encina SFS server
  fully qualfied name 114
  specifying server name 136
ENCINA_CDS_ROOT environment variable 212
ENCINA_SFS_DATA_VOLUME environment variable 212
ENCINA_SFS_INDEX_VOLUME environment variable 212
ENCINA_VSAM_AUTO_FLUSH environment variable 212

ENCINA_VSAM_CACHE environment variable 212
encoding
- conflict in XML documents 376
- controlling in generated XML output 383
- description 181
- language characters 173
- of XML documents 371
- of XML documents to be parsed 363

encoding declaration
- preferable to omit 372
- specifying 373

end-of-file phrase (AT END) 160
enhancing XML output
- example of converting hyphens to underscores 393
- example of modifying data definitions 390
- rationale and techniques 389

entry point
- alternate in ENTRY statement 489
- definition 497
- function-pointer data item 489
- passing addresses of 489
- procedure-pointer data item 489

ENTRY statement
- for alternate entry points 489
- handling of program-name in 276

ENTRYINT compiler option
- description 259

environment differences, zSeries and the workstation 460
ENVIRONMENT DIVISION
- class 402
- client 415
- collating sequence coding 8
- CONFIGURATION SECTION 7
- description 7
- INPUT-OUTPUT SECTION 7
- instance method 407
- subclass 427

environment variables
- accessing 209
- accessing files with 212
- assignment-name 212
- CLASSPATH
  - description 212
  - specifying location of Java classes 239
- COBCPYEXT 210
- COBJVMINITOPTIONS
  - description 212
  - specifying JVM options 240
- COBLSTDIR 210
- COBMSGS 212
- COBOPT 210
- COBPATH
  - CICS dynamic calls 339
  - description 212
- COBRTOPT 212
- compiler 210
- DB2DBDFT 210
- definition 209
- EBCDIC_CODEPAGE 212
- ENCINA_CDS_ROOT 212
- ENCINA_SFS_DATA_VOLUME 212

environment variables *(continued)*
- ENCINA_SFS_INDEX_VOLUME 212
- ENCINA_VSAM_AUTO_FLUSH 212
- ENCINA_VSAM_CACHE 212
- example of setting and accessing 217
- LANG 212
- LC_ALL 195, 212
- LC_COLLATE 195, 212
- LC_CTYPE 195, 212
- LC_MESSAGES 195, 212
- LC_TIME 195, 212
- LIBPATH
  - description 212
  - specifying location for COBOL classes 239
- library-name 210, 295
- LOCPATH 212
- NLSPATH 212
- PATH
  - description 212
- precedence 209
- runtime 211
- setting
  - in .profile 209
  - in command shell 209
  - in program 209
  - locale 196
  - overview 209
- SYSIN, SYSIPT, SYSOUT, SYSLIST, SYSLST, CONSOLE, SYSPUNCH, SYSPCH 212
- SYSLIB 210
  - specifying location of JNI.cpy 237
- text-name 210, 295
- TMP 212
- TZ 212

environment-name 7
environment, preinitializing
- example 515
- for C/C++ program 472
- overview 513

ERRCOUNT runtime option 302
ERRMSG, for generating list of error messages 225
error
- arithmetic 158
- compiler options, conflicting 245
- flagging at run time 301
- handling 157
- message table
  - example using indexing 72
  - example using subscripting 71

error messages
- compiler
  - choosing severity to be flagged 313
  - correcting source 224
  - determining what severity level to produce 264
  - embedding in source listing 313
  - format 226
  - generating a list of 225
  - location in listing 226
  - severity levels 225
- compiler-directed 225
- runtime
  - format 725

error messages *(continued)*
- runtime *(continued)*
  - incomplete or abbreviated 234
  - list of 725
  - listing with dspcat command 725
- setting national language 212

EUC code page 195
euro currency sign 59
EVALUATE statement
- case structure 83
- coding 83
- contrasted with nested IFs 84, 85
- example that tests several conditions 85
- example with multiple WHEN phrases 85
- example with THRU phrase 84
- performance 84
- structured programming 544
- testing multiple values, example 88, 89
- use to test multiple conditions 81

evaluating data item contents
- class test
  - for numeric 51
  - overview 86
- INSPECT statement 103
- intrinsic functions 107

example
- _iwzGetCCSID: convert code-page ID to CCSID 204
- _iwzGetLocaleCP: get locale and EBCDIC code-page values 204

examples
- CEECBLDY: convert date to COBOL integer format 603
- CEEDATE: convert Lilian date to character format 605
- CEEDATM: convert seconds to character format 609
- CEEDAYS: convert date to Lilian format 613
- CEEDYWK: calculate day of week from Lilian date 615
- CEEGMT: get current GMT 618
- CEEGMTO: get offset from Greenwich Mean Time 620
- CEEISEC: convert integers to seconds 622
- CEELOCT: get current local time 625
- CEEQCEN: query century window 626
- CEESCEN: set century window 628
- CEESECI: convert seconds to integers 630
- CEESECS: convert timestamp to number of seconds 634
- IGZEDT4: get current date with four-digit year 636

exception condition
- CALL 165
- XML GENERATE 384
- XML PARSE 376

exception handling
- with Java 445

EXCEPTION XML event 376

EXCEPTION/ERROR declarative
  description 160
  file status key 162
exceptions, intercepting 304
EXIT compiler option
  ADEXIT suboption 264
  character string formats 261
  description 259
  INEXIT suboption 262
  LIBEXIT suboption 262
  PRTEXIT suboption 263
exit modules
  called for SYSADATA data set 264
  debugging 328
  loading and invoking 262
  when used in place of library-name 262
  when used in place of SYSLIB 262
  when used in place of SYSPRINT 263
EXIT PROGRAM statement
  in main program 466
  in subprogram 466
explicit scope terminator 19
exponentiation
  evaluated in fixed-point arithmetic 586
  evaluated in floating-point arithmetic 591
  performance tips 547
export command
  defining environment variables 209
  path search order 209
export file 232
extended mode 39, 583
external class-name 403, 415
EXTERNAL clause
  example for files 492
  for data items 491
  for sharing files 12, 491
external code page, definition 372
external data
  sharing 491
external decimal data
  national 43
  zoned 43
external file 491
external floating-point data
  display 44
  national 44

F

factoring expressions 544
factory data
  defining 430
  definition of 397
  making it accessible 430
  private 430
factory definition, coding 429
factory methods
  defining 431
  definition of 397
  hiding 432
  invoking 432
  using to wrap procedural programs 438
FACTORY paragraph
  factory data 430
  factory methods 431
factory section, defining 429
FD (file description) entry 13
feedback token
  date and time services and 561
figurative constants
  definition 26
  HIGH-VALUE restriction 176
  national-character 176
file access mode
  dynamic 125
  for indexed files 124
  for line-sequential files 124
  for relative files 124
  for sequential files 123
  random 125
  sequential 125
  summary table of 123
file conversion
  with millennium language extensions 525
file description (FD) entry 13
file organization
  indexed 124
  line-sequential 124
  overview 122
  relative 124
  sequential 123
file position indicator 129, 132
FILE SECTION
  DATA RECORDS clause 13
  description 12
  EXTERNAL clause 12
  FD entry 13
  GLOBAL clause 12
  RECORD CONTAINS clause 13
  record description 12
  RECORD IS VARYING 13
  RECORDING MODE clause 13
  VALUE OF 13
FILE STATUS clause
  example 165
  file loading 133
  using 161
  with status code
    example 163
    overview 162
file status code
  02 132
  39 163
  92 135
  using 159
file status key
  checking for I/O errors 161
  checking for successful OPEN 161, 162
  error handling 308
  setting up 126
  used with status code
    example 163
    overview 162
file suffixes
  for error messages listing 225
FILE-CONTROL paragraph, example 7
file-system support
  Encina SFS 303
  RSD 303
  SFS (Encina) 303
  STL 303
  summary 121
  using FILEMODE runtime option 303
  using FILESYS runtime option 303
  VSAM 303
FILEMODE runtime option
  compatibility 121
  description 303
files
  32-bit mode versus 64-bit mode 571
  accessing Encina SFS 136
  accessing using environment variables 212
  adding records to 133
  associating program files to external files 7
  changing name 11
  clustered 117
  COBOL coding
    example 127
    overview 127
  comparison of file organizations 123
  deleting records from 135
  describing 12
  Encina SFS
    accessing 340
    identifying 113
  external 491
  file position indicator 129, 132
  FILEMODE runtime option, effect of 303
  FILESYS runtime option, effect of 303
  identifying 113
  identifying to the operating system 10
  large format 121
  linker
    export 232
    import 232
    library 232
  multiple, compiling 219
  opening 129
  padding records with blanks 120
  processing
    Encina SFS files 115
    RSD files 115
    SFS (Encina) files 115
    STL files 115
    VSAM files 115
  reading records from 132
  replacing records in 134
  RSD 113
  SFS (Encina)
    accessing 340
    identifying 113
  small format
    compatibility 121
    definition 121
  STL
    identifying 113
    large format 121

files *(continued)*
  STL *(continued)*
    size affected by FILEMODE 303
    small format 121
  summary 121
  TRAP runtime option, effect of 304
  updating records in 135
  usage explanation 11
  VSAM
    identifying 113
    large format 121
    size affected by FILEMODE 303
    small format 121
FILESYS runtime option 303
fixed century window 523
fixed-point arithmetic
  comparisons 57
  evaluation 56
  example evaluations 58
  exponentiation 586
fixed-point data
  binary 44
  conversions and precision 49
  conversions between fixed- and floating-point 49
  external decimal 43
  intermediate results 585
  packed-decimal 45
  planning use of 546
FLAG compiler option
  compiler output 314
  description 264
  using 313
flags and switches 87
FLAGSTD compiler option 265
FLOAT compiler option 267
floating-point arithmetic
  comparisons 57
  evaluation 56
  example evaluations 58
  exponentiation 591
floating-point data
  conversions and precision 49
  conversions between fixed- and floating-point 49
  external 44
  intermediate results 590
  internal
    format 46
    performance tips 547
  performance considerations 460
  planning use of 546
  portability 459
four-digit years 562
freeing object instances 424
full date field expansion, advantages 522
function-pointer data item
  32-bit mode versus 64-bit mode 571
  addressing JNI services 651
  definition 489
  passing parameters to callable services 489
  size depends on ADDR 247

G

garbage collection 424
GB 18030 data
  converting to or from national 186
  processing 186
generating XML output
  example 385
  overview 379
get and set methods 411
getenv() to access environment variables 209
GETMAIN, saving address of 261
GLOBAL clause for files 12, 16
global names 469
Glossary 775
GOBACK statement
  in main program 466
  in subprogram 466
Greenwich Mean Time (GMT)
  getting offset to local time (CEEGMTO) 619
  return Lilian date and Lilian seconds (CEEGMT) 617
Gregorian character string
  returning local time as a (CEELOCT) 623
    example 625
group item
  32-bit mode versus 64-bit mode 571
  cannot subordinate alphanumeric group within national group 179
  comparing to national data 189
  definition 24
  for defining tables 61
  group move contrasted with elementary move 33, 180
  initializing
    using a VALUE clause 70
    using INITIALIZE 30, 68
  MOVE statement with 33
  passing as an argument 485
  treated as a group item
    example with INITIALIZE 68
    in INITIALIZE 31
  variably located 593
group move contrasted with elementary move 33, 180
GROUP-USAGE NATIONAL clause
  communicating with Java 448
  defining a national group 179
  defining tables 62
  example of declaring a national group 24
  initializing a national group 31
grouping data to pass as an argument 485

H

header on listing 6
heap space, allocating with cob2 command 230
help files
  setting national language 212
  specifying path name 212
hexadecimal
  portability 459
hexadecimal literals
  as currency sign 59
  national
    description 26
    using 175
hiding factory methods 432

I

I-level message 225, 313
idebug command, example 328
IDENTIFICATION DIVISION
  class 402
  CLASS-ID paragraph 402, 426
  client 414
  coding 5
  DATE-COMPILED paragraph 5
  listing header example 6
  method 406
  PROGRAM-ID paragraph 5
  required paragraphs 5
  subclass 426
  TITLE statement 6
IEEE
  portability 459
IF statement
  coding 81
  nested 82
  use EVALUATE instead for multiple conditions 82
  with null branch 81
IGZEDT4: get current date with four-digit year 636
imperative statement, list 18
implicit scope terminator 19
import file 232
incrementing addresses 487
index
  assigning a value to 66
  computation of element displacement, example 64
  creating with OCCURS INDEXED BY clause 66
  definition 64
  in Encina SFS files 117
  incrementing or decrementing 66
  initializing 67
  key, detecting faulty 164
  range checking 312
  referencing other tables with 66
index data item
  32-bit mode versus 64-bit mode 571
  cannot use as subscript or index 67
  creating with USAGE IS INDEX clause 66
  size depends on ADDR 247
indexed file organization 124
indexed files
  Encina SFS files 117
  file access mode 124
indexing
  computation of element displacement, example 64
  definition 64
  example 72

indexing *(continued)*
preferred to subscripting 547
tables 66
inheritance hierarchy, definition of 399
INITIAL attribute 466
effect on nested programs 6
setting programs to initial state 6
INITIALIZE statement
examples 28
loading group values 30
loading national group values 31
loading table values 68
REPLACING phrase 68
using for debugging 309
initializing
a group item
using a VALUE clause 70
using INITIALIZE 30, 68
a national group item
using a VALUE clause 70
using INITIALIZE 31, 68
a structure using INITIALIZE 30
a table
all occurrences of an element 70
at the group level 70
each item individually 69
using INITIALIZE 68
using PERFORM VARYING 92
examples 28
instance data 422
the runtime environment
example 515
overview 513
variable-length group 75
inline PERFORM
example 90
overview 90
input
from files 113
overview 122
input procedure
coding 146
example 150
requires RELEASE or RELEASE FROM 146
restrictions 148
INPUT-OUTPUT SECTION 7
input/output
checking for errors 161
coding
example 127
overview 127
GUI applications 212
introduction 113
logic flow after error 158
processing errors
VSAM files 159
input/output coding
AT END (end-of-file) phrase 160
checking for successful operation 161
checking status code
example 163
overview 162
detecting faulty index key 164
error handling techniques 158
EXCEPTION/ERROR declaratives 160
insert cache 139
INSPECT statement
examples 103
using 103
inspecting data (INSPECT) 103
instance
creating 422
definition of 397
deleting 424
instance data
defining 404, 427
definition of 397
initializing 422
making it accessible 411
private 404
instance methods
defining 405, 428
definition of 397
invoking overridden 422
overloading 410
overriding 409
INTEGER intrinsic function, example 102
INTEGER-OF-DATE intrinsic function 55
INTEGER-PART intrinsic function 102
integers
converting Lilian seconds to (CEESECI) 629
integrated CICS translator
advantages 342
cicstcl -p command 342
overview 342
interlanguage communication
between COBOL and C/C++
overview 471
restriction 471
between COBOL and Java 443
restriction 571
intermediate results 583
internal bridges
advantages 522
example 524
for date processing 523
internal floating-point data (COMP-1, COMP-2) 46
interoperable data types with Java 448
intrinsic functions
as reference modifiers 102
collating sequence, effect of 202
compatibility with CEELOCT 624
converting alphanumeric data items with 104
converting national data items with 104
date and time 600
DATEVAL
example 536
using 535
evaluating data items 107
example of
ANNUITY 55
CHAR 108
CURRENT-DATE 55
DISPLAY-OF 184
INTEGER 102
INTEGER-OF-DATE 55
intrinsic functions *(continued)*
example of *(continued)*
LENGTH 54, 109, 110
LOG 56
LOWER-CASE 105
MAX 54, 79, 108, 109
MEAN 56
MEDIAN 56, 79
MIN 102
NATIONAL-OF 184
NUMVAL 105
NUMVAL-C 54, 105
ORD 107
ORD-MAX 79, 108
PRESENT-VALUE 55
RANGE 56, 79
REM 56
REVERSE 105
SQRT 56
SUM 79
UPPER-CASE 105
WHEN-COMPILED 111
finding date of compilation 111
finding largest or smallest item 108
finding length of data items 110
intermediate results 588, 591
introduction to 36
nesting 37
numeric functions
examples of 53
integer, floating-point, mixed 53
nested 54
special registers as arguments 54
table elements as arguments 54
uses for 53
processing table elements 78
UNDATE
example 536
using 536
INVALID KEY phrase
description 164
example 165
INVOKE statement
RETURNING phrase 421
USING phrase 419
using to create objects 422
using to invoke methods 418
with ON EXCEPTION 419, 433
with PROCEDURE DIVISION RETURNING 490
invoking
compiler and linker 219
date and time services 559
factory or static methods 432
instance methods 418
iwzGetSortErrno, obtaining sort or merge error number with 151

J

Java
and COBOL
communicating between 443
compatibility 569
compiling 237
linking 238
running 239

Java *(continued)*
  and COBOL *(continued)*
    structuring applications 439
  array classes 448
  arrays
    declaring 449
    example 452
    manipulating 450
  boolean array 449
  boolean type 448
  byte array 449
  byte type 448
  char array 449
  char type 448
  class types 448
  double array 450
  double type 448
  example
    exception handling 445
    processing an array 452
  exception
    catching 445
    example 445
    handling 445
    throwing 445
  float array 450
  float type 448
  global references
    JNI services for 447
    managing 446
    object 446
    passing 446
  int array 449
  int type 448
  interoperability 443
  interoperable data types, coding 448
  jstring class 448
  libraries
    specified in cob2.cfg 222
    updating 237
  local references
    deleting 446
    freeing 447
    JNI services for 447
    managing 446
    object 446
    passing 446
    per multithreading 446
    saving 446
  long array 449
  long type 448
  methods
    access control 447
  object array 449
  running with COBOL
    overview 239
    SDK support 241
  SDK support 241
  sharing data with 448
  short array 449
  short type 448
  string array 449
  strings
    declaring 449
    manipulating 453
Java virtual machine
  exceptions 445
Java virtual machine *(continued)*
  initializing 240
  object references 446
java.lang.Object
  referring to as Base 402
javac command
  compiling Java class definitions 237
  recompile for Java 5 or Java 6 569
JNI
  accessing services 443
  comparing object references 417
  converting local references to global 422
  environment structure 443
    addressability for 444
  exception handling services 445
  Java array services 450
  Java string services 453
  libraries
    specified in cob2.cfg 222
    updating 237
  obtaining class object reference 444
  restrictions when using 444
  Unicode services 453
  UTF-8 services 453
JNI.cpy
  for compiling 237
  for JNINativeInterface 443
  listing 651
JNIEnvPtr special register
  32-bit mode versus 64-bit mode 571
  size depends on ADDR 247
  use for JNI callable services 444
JNINativeInterface
  environment structure 443
  JNI.cpy 443
jstring Java class 448

K

Kanji comparison 86
Kanji data, testing for 191
keyboard navigation xix
keys
  for binary search 77
  for merging
    default 201
    defining 149
    overview 144
  for sorting
    default 201
    defining 149
    overview 143
  permissible data types
    in MERGE statement 149
    in OCCURS clause 62
    in SORT statement 149
  to specify order of table elements 62

L

LANG environment variable 212
largest or smallest item, finding 108
last-used state
  subprograms with EXIT PROGRAM or GOBACK 466
lazy write
  enabling 141
  environment variable for 212
LC_ALL 195
LC_ALL environment variable 212
LC_COLLATE 195
LC_COLLATE environment variable 212
LC_CTYPE 195
LC_CTYPE environment variable 212
LC_MESSAGES 195
LC_MESSAGES environment variable 212
LC_TIME 195
LC_TIME environment variable 212
LENGTH intrinsic function
  32-bit mode versus 64-bit mode 570
  compared with LENGTH OF special register 110
  example 54, 110
  result size depends on ADDR 247
  using 107
  variable-length results 109
  with national data 110
length of data items, finding 110
LENGTH OF special register
  32-bit mode versus 64-bit mode 570
  passing 482
  size depends on ADDR 247
  using 110
level-88 item
  conditional expressions 86
  for windowed date fields 529
  restriction 530
  setting switches off, example 89
  setting switches on, example 88
  switches and flags 87
  testing multiple values, example 88
  testing single values, example 87
level-number 321
LIB compiler option 268
libjava.a 238
LIBPATH environment variable
  description 212
  specifying location for COBOL classes 239
library components, redistributing 235
library file 232
library text
  specifying path for 210, 295
library-name
  alternative if not specified 227
  specifying path for library text 210, 295
library-name, when not used 262
Lilian date
  calculate day of week from (CEEDYWK) 615
  convert date to (CEEDAYS) 611
  convert date to COBOL integer format (CEECBLDY) 600
  convert output_seconds to (CEEISEC) 621
  convert to character format (CEEDATE) 604
  get current local date or time as a (CEELOCT) 623
  get GMT as a (CEEGMT) 617

Lilian date *(continued)*
  using as input to CEESECI 630
limits of the compiler
  DATA DIVISION 12
  data-size compatibility 569
  user data 12
line number 320
line-sequential files
  file access mode 124
  organization 124
LINECOUNT compiler option 268
linkage conventions
  compiler directive CALLINT for 295
  compiler option CALLINT for 249
LINKAGE SECTION
  coding 484
  for describing parameters 483
  with recursive calls 16
  with the THREAD option 16
linkages, data 473
linked-list processing, example 487
linker
  errors 234
  file defaults 233
  files
    export 232
    import 232
    library 232
  invoking 219, 229
  resolution of references to shared libraries 498
  search rules 233
  specifying options 229, 230
linking
  examples 231
  programs 229
  static 497
linking OO applications
  cob2_j command 238
  example 239
LIST compiler option
  description 269
  getting output 317
  use in debugging 327
List of resources 807
listings
  data and procedure-name cross-reference 315
  embedded error messages 313
  generating a short listing 317
  line numbers, user-supplied 318
  sorted cross-reference of program-names 325
  sorted cross-reference of text-names 325
  terms used in MAP output 322
  text-name cross-reference 315
literals
  alphanumeric
    control characters within 26
    description 25
    with multibyte content 191
  DBCS
    description 26
    maximum length 191
    using 190
  definition 25
literals *(continued)*
  hexadecimal
    using 175
  national
    description 26
    using 175
  numeric 26
  using 25
little-endian
  representation of integers 462
  representation of national characters 463
little-endian, converting to big-endian 173
loading a table dynamically 67
local names 469
local references, converting to global 422
local time
  getting (CEELOCT) 623
LOCAL-STORAGE SECTION
  client 416
  comparison with WORKING-STORAGE
    example 14
    OO client 416
    overview 13
locale
  accessing 203
  and messages 197
  cultural conventions, definition 193
  default 196
  definition 193
  effect of COLLSEQ compiler option 200
  effect of PROGRAM COLLATING SEQUENCE 200
  locale-based collating 199
  querying 203
  shown in listing 315, 319
  specifying 212
  supported values 197
  value syntax 196
locale information database
  specifying search path name 212
LOCPATH environment variable 212
LOG intrinsic function 56
loops
  coding 89
  conditional 91
  do 91
  in a table 92
  performed an explicit number of times 91
LOWER-CASE intrinsic function 105
lowercase, converting to 105
lst file suffix 225
LSTFILE compiler option 269

M

main entry point
  specifying with cob2 228
main program
  and subprograms 465
  arguments to 495
  specifying with cob2 228
MAP compiler option
  description 270
  embedded MAP summary 316
  example 320, 323
  nested program map 316
    example 323
  symbols used in output 322
  terms used in output 322
  using 316
mapping of DATA DIVISION items 316
mathematics
  intrinsic functions 53, 56
MAX intrinsic function
  example table calculation 79
  example with functions 54
  using 108
MAXMEM compiler option 271
MDECK compiler option
  description 272
  multioption interaction 245
MEAN intrinsic function
  example statistics calculation 56
  example table calculation 79
MEDIAN intrinsic function
  example statistics calculation 56
  example table calculation 79
merge
  alternate collating sequence 150
  completion code 151
  criteria 149
  description 143
  determining success 151
  diagnostic message 151
  error number
    list of possible values 152
    obtaining with iwzGetSortErrno 151
  files, describing 145
  keys
    default 201
    defining 149
    overview 144
  process 143
  terminating 154
MERGE statement
  ASCENDING|DESCENDING KEY phrase 149
  COLLATING SEQUENCE phrase 8, 150
  description 148
  GIVING phrase 148
  overview 143
  USING phrase 148
MERGE work files 212
message catalogs
  specifying path name 212
messages
  compiler
    choosing severity to be flagged 313
    date-related 537
    determining what severity level to produce 264
    embedding in source listing 313
    generating a list of 225
    millennium language extensions 537

messages *(continued)*
compiler *(continued)*
severity levels 225
compiler-directed 225
national language support 197
offset information 327
runtime
format 725
incomplete or abbreviated 234
list of 725
listing with dspcat command 725
setting national language 212
specifying file for 212
TRAP(OFF) side effect 304
METHOD-ID paragraph 406
methods
constructor 431
factory 431
hiding factory 432
instance 405, 428
invoking 418, 432
invoking superclass 422
Java access control 447
obtaining passed arguments 409
overloading 410
overriding 409, 432
PROCEDURE DIVISION RETURNING 490
returning a value from 409
signature 406
middleware subsystems, 32-bit mode versus 64-bit mode 571
migration considerations
32-bit mode to 64-bit mode 570
overview 569
millennium language extensions
assumed century window 530
compatible dates 527
concepts 520
date windowing 519
DATEPROC compiler option 256
nondates 531
objectives 521
principles 520
YEARWINDOW compiler option 292
MIN intrinsic function
example 102
using 108
MLE 520
mnemonic-name
SPECIAL-NAMES paragraph 7
modules, exit
loading and invoking 262
MOVE statement
assigning arithmetic results 34
converting to national data 182
CORRESPONDING 33
effect of ODO on lengths of sending and receiving items 73
group move contrasted with elementary move 33, 180
with elementary receiving items 32
with group receiving items 33
with national items 32
multiple currency signs
example 59
using 59
multiple inheritance, not permitted 400, 425
multiple thread environment, running in 287
multitasking, definition 506
multithreading
choosing data section
in an OO client 416
control transfer 509
effect on return code 478
ending programs 509
example 510
limitations 510
overview 505
preinitializing 509
preparing COBOL programs for 505
recursion 508
scope of language elements
program invocation instance 507
run unit 507
summary table 508
synchronizing access to resources 510
terminology 505
THREAD compiler option
restrictions with 287
when to choose 508

N

N delimiter for national or DBCS literals 26
name declaration
searching for 470
naming
programs 5
NATIONAL (USAGE IS)
external decimal 43
floating point 44
national comparison 86
national data
cannot use with DATE FORMAT clause 520
communicating with Java 448
comparing
effect of collating sequence 202
effect of NCOLLSEQ 187
overview 186
to alphabetic, alphanumeric, or DBCS 189
to alphanumeric groups 189
to numeric 188
two operands 187
concatenating (STRING) 93
converting
from alphanumeric or DBCS with NATIONAL-OF 183
from alphanumeric, DBCS, or integer with MOVE 182
overview 182
to alphanumeric with DISPLAY-OF 184
to numbers with NUMVAL, NUMVAL-C 105
to or from Chinese GB 18030 186
to or from Greek alphanumeric, example 184
national data *(continued)*
converting *(continued)*
to or from UTF-8 185
to uppercase or lowercase 105
with INSPECT 103
defining 174
encoding in XML documents 371
evaluating with intrinsic functions 107
external decimal 43
external floating-point 44
figurative constants 176
finding the smallest or largest item 108
in conditional expressions 186, 187
in generated XML documents 380
in keys
in MERGE statement 149
in OCCURS clause 62
in SORT statement 149
initializing, example of 29
input with ACCEPT 35
inspecting (INSPECT) 103
LENGTH intrinsic function and 110
LENGTH OF special register 110
literals
using 175
MOVE statement with 32, 182
NSYMBOL compiler option if no USAGE clause 175
output with DISPLAY 36
reference modification of 100
reversing characters 105
specifying 174
splitting (UNSTRING) 96
VALUE clause with alphanumeric literal, example 109
national decimal data (USAGE NATIONAL)
defining 177
example 39
format 43
initializing, example of 30
national floating-point data (USAGE NATIONAL)
defining 177
definition 44
national group item
advantages over alphanumeric groups 178
can contain only national data 24, 179
communicating with Java 448
contrasted with USAGE NATIONAL group 25
defining 179
example 24
for defining tables 62
in generated XML documents 380
initializing
using a VALUE clause 70
using INITIALIZE 31, 68
LENGTH intrinsic function and 110
MOVE statement with 33
overview 177
passing as an argument 485

national group item *(continued)*
treated as a group item
example with INITIALIZE 180
in INITIALIZE 31
in MOVE CORRESPONDING 33
summary 180
treated as an elementary item
example with MOVE 33
in most cases 24, 177
using
as an elementary item 180
overview 179
VALUE clause with alphanumeric literal, example 70
national language support
messages 197
national language support (NLS)
accessing locale and code-page values 203
collating sequence 199
DBCS 189
locale 193
locale-based collating 199
processing data 169
setting the locale 193
specifying locale and code page 212
national literals
description 26
using 175
national-edited data
defining 175
editing symbols 175
initializing
example 29
using INITIALIZE 68
MOVE statement with 32
PICTURE clause 175
NATIONAL-OF intrinsic function
example with Chinese data 186
example with Greek data 184
example with UTF-8 data 185
using 183
with XML documents 372
native format
-host option effect on command-line arguments 495
CHAR option 251
FLOAT option 267
for JNI service arguments 573
for method arguments 573
NCOLLSEQ compiler option
description 273
effect on national collating sequence 199, 202
effect on national comparisons 187
effect on sort and merge keys 149
nested COPY statement 263, 557
nested delimited scope statements 20
nested IF statement
coding 82
CONTINUE statement 81
EVALUATE statement preferred 82
with null branches 81
nested intrinsic functions 54
nested program integration 551
nested program map
description 316
nested program map *(continued)*
example 323
nested programs
calling 466
description 467
guidelines 466
map 316, 323
scope of names 469
transfer of control 466
nesting level
program 320, 323
statement 320
NLSPATH environment variable 212
NOCOMPILE compiler option
use to find syntax errors 312
NODESC suboption of CALLINT compiler option 250
NODESCRIPTOR suboption of CALLINT compiler option 250
nondates with MLE 531
NOSSRANGE compiler option
effect on checking errors 301
Notices 771
NSYMBOL compiler option
description 273
effect on N literals 26
for DBCS literals 175
for national data items 175
for national literals 175
null branch 81
null-terminated strings
example 99
handling 486
manipulating 98
NUMBER compiler option
description 274
for debugging 318
numeric class test
checking for valid data 51
numeric comparison 86
numeric data
binary
USAGE BINARY 44
USAGE COMPUTATIONAL (COMP) 44
USAGE COMPUTATIONAL-4 (COMP-4) 44
USAGE COMPUTATIONAL-5 (COMP-5) 45
can compare algebraic values regardless of USAGE 188
comparing to national 188
converting
between fixed- and floating-point 48
precision 49
to national with MOVE 182
defining 39
display floating-point (USAGE DISPLAY) 44
editing symbols 41
external decimal
USAGE DISPLAY 43
USAGE NATIONAL 43
external floating-point
USAGE DISPLAY 44
USAGE NATIONAL 44
numeric data *(continued)*
internal floating-point
USAGE COMPUTATIONAL-1 (COMP-1) 46
USAGE COMPUTATIONAL-2 (COMP-2) 46
national decimal (USAGE NATIONAL) 43
national floating-point (USAGE NATIONAL) 44
packed-decimal
sign representation 50
USAGE COMPUTATIONAL-3 (COMP-3) 45
USAGE PACKED-DECIMAL 45
PICTURE clause 39, 41
storage formats 42
USAGE DISPLAY 39
USAGE NATIONAL 39
zoned decimal (USAGE DISPLAY)
format 43
sign representation 50
numeric intrinsic functions
example of
ANNUITY 55
CURRENT-DATE 55
INTEGER 102
INTEGER-OF-DATE 55
LENGTH 54, 109
LOG 56
MAX 54, 79, 108, 109
MEAN 56
MEDIAN 56, 79
MIN 102
NUMVAL 105
NUMVAL-C 54, 105
ORD 107
ORD-MAX 79
PRESENT-VALUE 55
RANGE 56, 79
REM 56
SQRT 56
SUM 79
integer, floating-point, mixed 53
nested 54
special registers as arguments 54
table elements as arguments 54
uses for 53
numeric literals, description 26
numeric-edited data
BLANK WHEN ZERO clause
coding with numeric data 175
example 41
defining 175
editing symbols 41
initializing
examples 30
using INITIALIZE 68
PICTURE clause 41
USAGE DISPLAY
displaying 41
initializing, example of 30
USAGE NATIONAL
displaying 41
initializing, example of 30
NUMVAL intrinsic function
description 105

NUMVAL-C intrinsic function
  description 105
  example 54
NX delimiter for national literals 26

O

object
  creating 422
  definition of 397
  deleting 424
object code
  compatibility 569
  generating 255
object instances, definition of 397
OBJECT paragraph
  instance data 404, 427
  instance methods 405
object references
  32-bit mode versus 64-bit mode 571
  comparing 417
  converting from local to global 422
  example of passing 420
  setting 417
  size depends on ADDR 247
  typed 416
  universal 416
OBJECT-COMPUTER paragraph 7
object-oriented COBOL
  communicating with Java 448
  compatibility 569
  compiling 237
  linking
    example 239
    overview 238
  preparing applications 238
  restrictions
    CICS 397
    EXEC CICS statements 397
    EXEC SQL statements 397
    SQL compiler option 397
  running
    Java SDK support 241
    overview 239
  writing OO programs 397
objectives of millennium language extensions 521
OCCURS clause
  ASCENDING|DESCENDING KEY phrase
    example 78
    needed for binary search 77
    specify order of table elements 62
  cannot use in a level-01 item 61
  for defining table elements 61
  for defining tables 61
  INDEXED BY phrase for creating indexes 66
  nested for creating multidimensional tables 62
OCCURS DEPENDING ON (ODO) clause
  complex 593
  for creating variable-length tables 73
  initializing ODO elements 75
  ODO object 73
  ODO subject 73
OCCURS DEPENDING ON (ODO) clause *(continued)*
  optimization 548
  simple 73
OCCURS INDEXED BY clause, creating indexes with 66
ODBC
  accessing return values 349
  advantages of 345
  background 346
  call interface convention 357
  comparison with embedded SQL 345
  copybooks
    example 351
    odbc3d 355
    odbc3p 352
  driver manager 346
  embedded SQL 345
  installing and configuring the drivers 346
  mapping of C data types 346
  overview 345
  passing a COBOL pointer to 347
  supplied copybooks 350
  using APIs from COBOL 346
ODO object 73
ODO subject 73
OMITTED parameters 559
OMITTED phrase for omitting arguments 484
ON EXCEPTION phrase
  INVOKE statement 419, 433
ON SIZE ERROR
  with windowed date fields 534
OPEN operation code 262
OPEN statement
  file availability 129
  file status key 161
opening files
  effect of STL and VSAM format 121
  overview 129
  using environment variables 212
operational force
  description 140
  environment variable for 212
  suppressing 140
optimization
  avoid ALTER statement 544
  avoid backward branches 544
  BINARY data items 546
  consistent data 546
  constant computations 545
  constant data items 545
  contained program integration 551
  duplicate computations 545
  effect of compiler options on 552
  effect on parameter passing 483
  effect on performance 544
  factor expressions 544
  index computations 549
  indexing 547
  MAXMEM 271
  nested program integration 551
  OCCURS DEPENDING ON 548
  out-of-line PERFORM 544
  packed-decimal data items 546
  performance implications 548
optimization *(continued)*
  structured programming 544
  subscript computations 549
  subscripting 547
  table elements 547
  top-down programming 544
  unused data items 275
OPTIMIZE compiler option
  description 275
  effect on parameter passing 483
  effect on performance 550
  performance considerations 552
  using 550
optimizer
  overview 551
options
  specifying for linker 230
ORD intrinsic function, example 107
ORD-MAX intrinsic function
  example table calculation 79
  using 108
ORD-MIN intrinsic function 108
order of evaluation
  arithmetic operators 52, 585
  compiler options 245
out-of-line PERFORM 90
output
  overview 122
  to files 113
output procedure
  coding 147
  example 150
  requires RETURN or RETURN INTO 147
  restrictions 148
overflow condition
  CALL 165
  joining and splitting strings 157
  UNSTRING 95
overloading instance methods 410
overriding
  factory methods 432
  instance methods 409

P

packed-decimal data item
  date fields, potential problems 539
  description 45
  sign representation 50
  synonym 42
  using efficiently 45, 546
padding records with blanks 120
paragraph
  grouping 92
  introduction 17
parameter list
  address of with INEXIT 262
  for ADEXIT 264
  for PRTEXIT 263
parameters
  describing in called program 483
  in main program 495
parse data item, definition 363
parsing XML documents
  description 363
  overview 361

parsing XML documents *(continued)*
  UTF-8 374
passing data between programs
  addresses 486
  arguments in calling program 483
  BY CONTENT 481
  BY REFERENCE 481
  BY VALUE
    overview 481
    restrictions 483
  EXTERNAL data 491
  in the RETURN-CODE special register 490
  JNI services 444
  OMITTED arguments 484
  parameters in called program 483
  with Java 448
PATH environment variable
  description 212
path name
  for copybook search 227, 295
  library text 210, 295
  multiple, specifying 210, 295
  precedence 209
  specifying for catalogs and help files 212
  specifying for executable programs 212
  specifying for locale information database 212
  specifying for runtime library 212
  specifying with LIB compiler option 268
PERFORM statement
  coding loops 89
  for a table
    example using indexing 72
    example using subscripting 71
  for changing an index 67
  inline 90
  out-of-line 90
  performed an explicit number of times 91
  TEST AFTER 91
  TEST BEFORE 91
  THRU 92
  TIMES 91
  UNTIL 91
  VARYING 92
  VARYING WITH TEST AFTER 92
  WITH TEST AFTER . . . UNTIL 91
  WITH TEST BEFORE . . . UNTIL 91
performance
  arithmetic evaluations 546
  arithmetic expressions 547
  CICS environment 543
  coding 543
  coding tables 547
  compiler option
    DYNAM 552
    FLOAT 460
    OPTIMIZE 550, 552
    SSRANGE 552
    TEST 552
    THREAD 287, 552
    TRUNC 288, 552
  consistent data types 546
performance *(continued)*
  creating a trace file 228
  data usage 546
  effect of compiler options on 552
  environment variable for SFS 212
  exponentiations 547
  improving for SFS files
    enabling client-side caching 139
    reducing frequency of saving changes 140
    suppressing operational force 140
  OCCURS DEPENDING ON 548
  optimizer
    overview 551
  order of WHEN phrases in EVALUATE 84
  out-of-line PERFORM compared with inline 90
  programming style 544
  runtime considerations 543
  table handling 549
  table searching
    binary compared with serial 76
    improving serial search 76
  variable subscript data format 65
  worksheet 554
performing calculations
  date and time services 560
period as scope terminator 19
PGMNAME compiler option 276
physical record 13
PICTURE clause
  cannot use for internal floating point 40
  determining symbol used 255
  incompatible data 51
  N for national data 175
  national-edited data 175
  numeric data 39
  numeric-edited data 175
  Z for zero suppression 41
picture strings
  date and time services and 562
platform differences 460
pointer data item
  32-bit mode versus 64-bit mode 571
  description 37
  incrementing addresses with 487
  NULL value 486
  passing addresses 486
  processing chained lists 486
  size depends on ADDR 247
  used to process chained list 487
porting applications
  CICS 338
  differences between platforms 457
  effect of separate sign 40
  environment differences 460
  file-status keys 461
  language differences 457
  mainframe to workstation
    choosing compiler options 457
    running mainframe applications on the workstation 458
  multibyte 460
  multitasking 462
  overview 457
porting applications *(continued)*
  SBCS 458
  using COPY to isolate platform-specific code 457
  Windows to AIX 462
  workstation to mainframe
    workstation-only compiler options 462
    workstation-only language features 462
    workstation-only names 462
precedence
  arithmetic operators 52, 585
preinitializing the COBOL environment
  example 515
  for C/C++ program 472
  initialization 513
  overview 513
  restriction under CICS 513
  termination 514
  with multithreading 509
PRESENT-VALUE intrinsic function 55
printer files 124
procedure and data-name cross-reference, description 315
PROCEDURE DIVISION
  client 414
  description 16
  in subprograms 485
  instance method 408
  RETURNING
    methods, use of 490
    to return a value 16
  statements
    compiler-directing 19
    conditional 18
    delimited scope 18
    imperative 18
  terminology 16
  USING
    BY VALUE 485
    to receive parameters 16, 483
procedure-pointer data item
  32-bit mode versus 64-bit mode 571
  definition 489
  passing parameters to callable services 489
  SET statement and 489
  size depends on ADDR 247
  using 489
process
  definition 505
  terminating 473
PROCESS (CBL) statement
  conflicting options in 245
  description 295
  specifying compiler options 221
processing
  chained lists
    example 487
    overview 486
  tables
    example using indexing 72
    example using subscripting 71
producing XML output 379
product support 807
PROFILE compiler option 277

profile file, setting environment variables in 209
profiling support, PROFILE option 277
program
  attribute codes 323
  decisions
    EVALUATE statement 81
    IF statement 81
    loops 91
    PERFORM statement 91
    switches and flags 87
  diagnostics 319
  limitations 543
  main 465
  nesting level 320
  statistics 319
  structure 5
  subprogram 465
PROGRAM COLLATING SEQUENCE clause
  COLLSEQ interaction 254
  does not affect national or DBCS operands 9
  effect on alphanumeric comparisons 200
  establishing collating sequence 8
  no effect on DBCS comparisons 201
  no effect on national comparisons 202
  overridden by COLLATING SEQUENCE phrase 8
  overrides default collating sequence 150
PROGRAM-ID paragraph
  coding 5
  COMMON attribute 6
  INITIAL attribute 6
program-names
  cross-reference 325
  handling of case 276
  specifying 5
programs, running 234
putenv() to set environment variables 209

Q

QUOTE compiler option 278

R

railroad track diagrams, how to read xiv
RANGE intrinsic function
  example statistics calculation 56
  example table calculation 79
read cache 139
reading records from files
  dynamically 132
  randomly 132
  sequentially 132
record
  description 12
  format 122
  TRAP runtime option, effect of 304
RECORD CONTAINS clause
  FILE SECTION entry 13
RECORDING MODE clause
  QSAM files 13
recursive calls
  and the LINKAGE SECTION 16
  coding 478
  identifying 6
REDEFINES clause, making a record into a table using 69
redistributing library components 235
reentrant code 573
reference modification
  example 101
  expression checking with SSRANGE 285
  generated XML documents 380
  intrinsic functions 99
  national data 100
  out-of-range values 101
  tables 65, 100
  UTF-8 documents 186
reference modifier
  arithmetic expression as 102
  intrinsic function as, example 102
  variables as 100
relate items to system-names 7
relation condition 86
relative files
  file access mode 124
  organization 124
RELEASE FROM statement
  compared to RELEASE 146
  example 146
RELEASE statement
  compared to RELEASE FROM 146
  with SORT 146
REM intrinsic function 56
replacing
  data items (INSPECT) 103
  records in files 134
REPLACING phrase (INSPECT), example 103
REPOSITORY paragraph
  class 402
  client 415
  coding 7
  subclass 427
representation
  data 51
  sign 50
restrictions
  CICS
    overview 337
    separate translator 342
  input/output procedures 148
  OO programs 397
  subscripting 65
return code
  compiler 225
  feedback code from date and time services 559
  files
    example 163
    overview 162
  from DB2 SQL statements 333
  RETURN-CODE special register 478, 490, 559
return code *(continued)*
  when control returns to operating system 478
RETURN statement
  required in output procedure 147
  with INTO phrase 147
RETURN-CODE special register
  not set by INVOKE 419
  passing data between programs 490
  sharing return codes between programs 490
  value after call to date and time service 559
  when control returns to operating system 478
  with nonthreaded support 478
  with threaded support 478
RETURNING phrase
  CALL statement 490
  INVOKE statement 421
  methods, use of 490
  PROCEDURE DIVISION header 409
REVERSE intrinsic function 105
reversing characters 105
ROUNDED phrase 584
rows in tables 63
RSD file system 120
RSD files
  identifying 113
  processing 115
run time
  arguments 495
  changing file-name 11
  differences between platforms 458
  messages 725
  performance considerations 543
run unit
  role in multithreading 505
  terminating 473
running OO applications
  Java SDK support 241
  overview 239
running programs 234
runtime environment, preinitializing
  example 515
  overview 513
runtime library, specifying path name 212
runtime loading, restrictions on 502
runtime messages
  format 725
  incomplete or abbreviated 234
  list of 725
  listing with dspcat command 725
  setting file for language 212
  setting national language 212
runtime options
  CHECK 301
  CHECK(OFF)
    performance considerations 552
  DEBUG 302, 310
  ERRCOUNT 302
  FILEMODE
    compatibility 121
    description 303
  FILESYS 303
  for CICS 341

runtime options *(continued)*
  overview 301
  specifying 212
  TRAP
    description 304
    ON SIZE ERROR 158
  UPSI 305

S

S-level error message 225, 313
scope of names
  effect of multithreading 506
  global 469
  local 469
  program invocation instance 507
  run unit 507
  summary table 508
scope terminator
  aids in debugging 308
  explicit 18, 19
  implicit 19
SD (sort description) entry, example 145
SEARCH ALL statement
  binary search 77
  example 78
  for changing an index 67
  table must be ordered 77
search rules for linker 233
SEARCH statement
  example 76
  for changing an index 67
  nesting to search more than one level of a table 76
  serial search 76
searching
  for name declarations 470
  tables
    binary search 77
    overview 75
    performance 76
    serial search 76
section
  declarative 21
  description of 17
  grouping 92
SELECT clause
  vary input-output file 11
SELECT OPTIONAL 129
SELF 417
sentence, definition 17
separate CICS translator
  restrictions 342
separate sign
  portability 40
  printing 40
  required for signed national decimal 40
SEPOBJ compiler option 278
SEQUENCE compiler option 280
sequential files
  file access mode 123
  organization 123
sequential search
  description 76
  example 76
serial search
  description 76
  example 76
SET condition-name TO TRUE statement
  example 90, 92
  switches and flags 88
SET statement
  for changing an index 66
  for changing index data items 66
  for object references 417
  for procedure-pointer data items 489
  for setting a condition, example 88
  handling of program-name in 276
  using for debugging 309
setting
  index data items 66
  indexes 66
  linker options 229
  switches and flags 88
SFS (Encina) file system
  accessing SFS files
    example 137
    overview 136
  description 117
  fully qualified file names 117
  improving performance 139
  nonhierarchical 117
  restriction 117
  system administration of 117
SFS (Encina) files
  accessing 340
    example 137
    overview 136
  adding alternate indexes 138
  alternate index file name 114
  base file name 114
  COBOL coding example 137
  compatibility 570
  compiling for 220
  creating alternate index files 136
  creating SFS files
    environment variables for 136
    sfsadmin command for 138
  determining available data volumes 136
  file names 117
  fully qualified name 137
  identifying 113
  improving performance 139
  nontransactional access 117
  organization 117
  primary and secondary indexes 117
  processing 115
  specifying data volume for 136
SFS (Encina) server
  fully qualified name 114
  specifying server name 136
sfsadmin command
  adding alternate indexes 138
  creating indexed files 138
  description 117
  determining available data volumes 136
shared libraries
  advantages and disadvantages 498
  building 497
    example 498
shared libraries *(continued)*
  CICS considerations 340
  creating for OO 238
  definition 497
  for Java interoperability 238
  overview 498
  purpose 498
  resolution of references 498
  setting directory path 212
  subprograms and outermost programs 497
  using with Java interoperability 239
  using with OO 239
sharing
  data
    between separately compiled programs 491
    coding the LINKAGE SECTION 484
    from another program 15
    in recursive or multithreaded programs 16
    in separately compiled programs 15
    overview 481
    parameter-passing mechanisms 481
    passing arguments to a method 419
    PROCEDURE DIVISION header 485
    RETURN-CODE special register 490
    returning a value from a method 421
    scope of names 469
    with Java 448
  files
    scope of names 469
    using EXTERNAL clause 12, 491
    using GLOBAL clause 12
shell script, compiling using 220
short listing, example 318
sign condition
  testing sign of numeric operand 86
  using in date processing 532
SIGN IS SEPARATE clause
  portability 40
  printing 40
  required for signed national decimal data 40
sign representation 50
signature
  definition of 406
  must be unique 406
SIZE compiler option 280
slack bytes, definition 571
sliding century window 523
sort
  alternate collating sequence 150
  completion code 151
  criteria 149
  description 143
  determining success 151
  diagnostic message 151
  error number
    list of possible values 152

sort *(continued)*
  error number *(continued)*
    obtaining with iwzGetSortErrno 151
  files, describing 145
  input procedures
    coding 146
    example 150
  keys
    default 201
    defining 149
    overview 143
  output procedures
    coding 147
    example 150
  process 143
  restrictions on input/output procedures 148
  terminating 154
SORT statement
  ASCENDING|DESCENDING KEY phrase 149
  COLLATING SEQUENCE phrase 8, 150
  description 148
  GIVING phrase 148
  overview 143
  USING phrase 148
SORT work files 212
SORT-RETURN special register
  determining sort or merge success 151
  terminating sort or merge 154
SOSI compiler option
  description 281
  multibyte portability 460
SOURCE and NUMBER output, example 320
source code
  line number 320, 321, 323
  listing, description 317
SOURCE compiler option
  description 282
  getting output 317
SOURCE-COMPUTER paragraph 7
SPACE compiler option 283
special feature specification 7
special register
  ADDRESS OF
    32-bit mode versus 64-bit mode 571
    size depends on ADDR 247
    use in CALL statement 482
  arguments in intrinsic functions 54
  JNIEnvPtr
    32-bit mode versus 64-bit mode 571
    size depends on ADDR 247
    use for JNI callable services 444
  LENGTH OF 110, 482
  RETURN-CODE 490
  SORT-RETURN
    determining sort or merge success 151
    terminating sort or merge 154
  WHEN-COMPILED 111
  XML-CODE 365, 366
special register *(continued)*
  XML-EVENT 365, 366
  XML-NTEXT 365, 367
  XML-TEXT 365, 367
SPECIAL-NAMES paragraph
  coding 7
SPILL compiler option 284
splitting data items (UNSTRING) 95
SQL compiler option
  coding 333
  description 284
  multioption interaction 245
  restriction in OO programs 397
SQL statements
  coding 332
  overview 331
  return codes 333
  SQL INCLUDE 332
  use for DB2 services 331
  using binary data in 333
SQLCA
  declare for programs that use SQL statements 332
  return codes from DB2 333
SQRT intrinsic function 56
SSRANGE compiler option
  description 285
  performance considerations 552
  reference modification 101
  turn off by using CHECK(OFF) runtime option 552
  using 312
stack frames, collapsing 473
stack space, allocating with cob2 command 231
Standard COBOL 85
  definition xiii
stanza
  adding 223
  attributes in configuration file 223
  cob2 222
  cob2_j 222
  cob2_r 222
  description 222
statement
  compiler-directing 19
  conditional 18
  definition 17
  delimited scope 18
  explicit scope terminator 19
  imperative 18
  implicit scope terminator 19
statement nesting level 320
static data, definition of 397
static linking
  advantages 497
  definition 470
  disadvantages 497
  overview 497
static methods
  definition of 397
  invoking 432
statistics intrinsic functions 56
status code, files
  example 163
  overview 162
STDCALL interface convention
  specified with CALLINT 250
STL file system
  description 118
  return codes 118
STL files
  compatibility 121
  identifying 113
  large format 121
  processing 115
  size affected by FILEMODE 303
  small format
    compatibility 121
    definition 121
STOP RUN statement
  in main program 466
  in subprogram 466
storage
  allocation depends on ADDR 247
  character data 181
  for arguments 483
  mapping 316
stride, table 549
STRING statement
  example 94
  overflow condition 157
  using 93
strings
  handling 93
  Java
    declaring 449
    manipulating 453
  null-terminated 486
structure, initializing using INITIALIZE 30
structured programming 544
structuring OO applications 439
subclass
  coding
    example 428
    overview 425
  instance data 427
subprogram
  and main program 465
  definition 481
  description 465
  in shared library 497
  linkage
    common data items 483
  PROCEDURE DIVISION in 485
subprograms
  using 465
subscript
  computations 549
  definition 64
  literal, example 64
  range checking 312
  variable, example 64
subscripting
  definition 64
  example 71
  literal, example 64
  reference modification 65
  relative 65
  restrictions 65
  use data-name or literal 65
  variable, example 64

substitution character 176
substrings
  of table elements 100
  reference modification of 99
SUM intrinsic function, example table calculation 79
SUPER 422
support 807
switch-status condition 86
switches and flags
  defining 87
  description 87
  resetting 88
  setting switches off, example 89
  setting switches on, example 88
  testing multiple values, example 88
  testing single values, example 87
SYMBOLIC CHARACTERS clause 9
symbolic constant 545
symbols used in MAP output 322
SYNCHRONIZED clause
  32-bit mode versus 64-bit mode 571
  alignment depends on ADDR 247
syntax diagrams, how to read xiv
syntax errors
  finding with NOCOMPILE compiler option 312
SYSADATA
  output 246
SYSADATA file
  example 659
  file contents 657
  record descriptions 660
  record types 658
SYSADATA records
  exit module called 264
  supplying modules 259
SYSIN
  supplying alternative modules 259
SYSIN, SYSIPT, SYSOUT, SYSLIST, SYSLST, CONSOLE, SYSPUNCH, SYSPCH environment variables 212
SYSLIB
  supplying alternative modules 259
  when not used 262
SYSLIB environment variable 210
  specifying location of JNI.cpy 237
SYSPRINT
  supplying alternative modules 259
  when not used 263
system date
  under CICS 338
SYSTEM interface convention
  specified with CALLINT 250
SYSTEM suboption of CALLINT compiler option 250
system-name 7
SYSTERM data set
  sending messages to 286

T

table
  assigning values to 69
  columns 61
  compare to array 37
  defining with OCCURS clause 61
table *(continued)*
  definition 61
  depth 63
  description 37
  dynamically loading 67
  efficient coding 547, 549
  elements 61
  identical element specifications 547
  index, definition 64
  initializing
    all occurrences of an element 70
    at the group level 70
    each item individually 69
    using INITIALIZE 68
    using PERFORM VARYING 92
  loading values in 67
  looping through 92
  multidimensional 62
  one-dimensional 61
  processing with intrinsic functions 78
  redefining a record as 69
  reference modification 65
  referencing substrings of elements 100
  referencing with indexes, example 64
  referencing with subscripts, example 64
  referring to elements 64
  rows 63
  searching
    binary 77
    overview 75
    performance 76
    sequential 76
    serial 76
  stride computation 549
  subscript, definition 64
  three-dimensional 63
  two-dimensional 63
  variable-length
    creating 73
    example of loading 74
    initializing 75
    preventing overlay in 595
TALLYING phrase (INSPECT), example 103
temporary work-file location
  specifying with TMP 212
TERMINAL compiler option 286
terminal, sending messages to 286
terminating XML parsing 377
terms used in MAP output 322
test
  conditions 91
  data 86
  numeric operand 86
  UPSI switch 86
TEST AFTER 91
TEST BEFORE 91
TEST compiler option
  description 286
  multioption interaction 245
  performance considerations 552
  use for debugging 316
text-name cross-reference, description 315
THREAD compiler option
  and the LINKAGE SECTION 16
  default for cob2_j 287
  description 287
  for Java interoperability 237
  for OO COBOL 237
  performance considerations 552
threading
  and preinitialization 509
  control transfer 509
  ending programs 509
three-digit years 562
time information, formatting 212
time-zone information
  specifying with TZ 212
time, getting local (CEELOCT) 623
timestamp 608, 632
TITLE statement
  controlling header on listing 6
TMP environment variable 212
top-down programming
  constructs to avoid 544
Trademarks 773
transferring control
  between COBOL programs 466
  called program 465
  calling program 465
  main and subprograms 465
  nested programs 467
transforming COBOL data to XML
  example 385
  overview 379
translating CICS into COBOL 337
TRAP runtime option
  description 304
  ON SIZE ERROR 158
TRUNC compiler option
  description 288
  performance considerations 552
tuning considerations, performance 552
two-digit years
  querying within 100-year range (CEEQCEN) 625
    example 626
  setting within 100-year range (CEESCEN) 627
    example 628
  valid values for 562
typed object references 416
TZ environment variable 212

U

U-level error message 225, 313
UNDATE intrinsic function
  example 536
  using 536
Unicode
  description 173
  encoding and storage 181
  JNI services 453
  processing data 169
universal object references 416
UNSTRING statement
  example 96
  overflow condition 157
  using 95

UPPER-CASE intrinsic function 105
uppercase, converting to 105
UPSI runtime options 305
UPSI switches, setting 305
USAGE clause
  at the group level 25
  incompatible data 51
  INDEX phrase, creating index data items with 66
  NATIONAL phrase at the group level 178
  OBJECT REFERENCE 416
USE FOR DEBUGGING declaratives
  DEBUG runtime option 302
  overview 309
user-defined condition 86
user-exit work area 261
USING phrase
  INVOKE statement 419
  PROCEDURE DIVISION header 409, 485
UTF-16
  definition 173
  encoding for national data 173
UTF-8
  avoid reference modification with XML documents 186
  converting to or from national 185
  definition 173
  encoding and storage 181
  encoding for ASCII invariant characters 173
  example of generating an XML document 381
  JNI services 453
  parsing XML documents 374
  processing data items 185
  XML document encoding 371

V

VALUE clause
  alphanumeric literal with national data, example 109
  alphanumeric literal with national group, example 70
  assigning table values
    at the group level 70
    to each item individually 69
    to each occurrence of an element 70
  assigning to a variable-length group 75
  cannot use for external floating point 44
  initializing internal floating-point literals 40
  large literals with COMP-5 45
  large, with TRUNC(BIN) 288
VALUE IS NULL 486
VALUE OF clause 13
variable
  as reference modifier 100
  definition 23
variable-length records
  OCCURS DEPENDING ON (ODO) clause 548
variable-length table
  assigning values to 75
  creating 73
  example 74
  example of loading 74
  preventing overlay in 595
variables, environment
  accessing 209
  assignment-name 212
  CLASSPATH 212
  COBCPYEXT 210
  COBJVMINITOPTIONS 212
  COBLSTDIR 210
  COBMSGS 212
  COBOPT 210
  COBPATH
    CICS dynamic calls 339
    description 212
  COBRTOPT 212
  compiler 210
  definition 209
  EBCDIC_CODEPAGE 212
  ENCINA_CDS_ROOT 212
  ENCINA_SFS_DATA_VOLUME 212
  ENCINA_SFS_INDEX_VOLUME 212
  ENCINA_VSAM_AUTO_FLUSH 212
  ENCINA_VSAM_CACHE 212
  example of setting and accessing 217
  LANG 212
  LC_ALL 212
  LC_COLLATE 212
  LC_CTYPE 212
  LC_MESSAGES 212
  LC_TIME 212
  LIBPATH 212
  library-name 210, 295
  LOCPATH 212
  NLSPATH 212
  PATH 212
  precedence 209
  runtime 211
  setting
    in .profile 209
    in command shell 209
    in program 209
    locale 196
    overview 209
  SYSIN, SYSIPT, SYSOUT, SYSLIST, SYSLST, CONSOLE, SYSPUNCH, SYSPCH 212
  SYSLIB 210
  text-name 210, 295
  TMP 212
  TZ 212
variably located data item 593
variably located group 593
VBREF compiler option
  description 290
  output example 326
  using 317
verb cross-reference listing
  description 317
verbs used in program 317
VSAM file system 116
VSAM files
  compatibility 121
  error processing 159
VSAM files *(continued)*
  identifying 113
  in a multithreaded environment 510
  large format 121
  processing 115
  size affected by FILEMODE 303
  small format
    compatibility 121
    definition 121
  status codes
    example 163
    overview 162

W

W-level message 225, 313
WHEN phrase
  EVALUATE statement 83
  SEARCH ALL statement 77
  SEARCH statement 76
WHEN-COMPILED intrinsic function 111
WHEN-COMPILED special register 111
windowed date fields
  contracting 539
WITH DEBUGGING MODE clause
  for debugging lines 310
  for debugging statements 310
WITH POINTER phrase
  STRING 93
  UNSTRING 95
wlist file 269
WORKING-STORAGE SECTION
  client 416
  comparison with LOCAL-STORAGE
    example 14
    OO client 416
    overview 13
  factory data 430
  initializing 291
  instance data 404, 427
  instance method 407
  multithreading considerations 416
workstation and workstation COBOL
  differences from host 573
wrapper, definition of 438
wrapping procedure-oriented programs 438
writing compatible code 462
WSCLEAR compiler option 291

X

X delimiter for control characters in alphanumeric literals 26
XML declaration
  generating 381
  specifying encoding declaration 373
XML document
  accessing 363
  code pages supported 371
  controlling the encoding of 383
  document encoding declaration 372
  EBCDIC special characters 373
  encoding 371

XML document *(continued)*
enhancing
example of converting hyphens to underscores 393
example of modifying data definitions 390
rationale and techniques 389
external code page 372
generating
example 385
overview 379
handling parsing exceptions 374
national language 371
parser 361
parsing
description 363
example 368
UTF-8 374
processing 361
specifying the code page 372
UTF-8 encoding 371
XML event
EXCEPTION 376
XML events
description 361
processing 364
processing procedure 363
XML exception codes
for generating 649
for parsing
handleable 639
not handleable 643
XML GENERATE statement
COUNT IN 384
NAMESPACE 381
NAMESPACE-PREFIX 382
NOT ON EXCEPTION 383
ON EXCEPTION 384
WITH ATTRIBUTES 380
WITH ENCODING 383
XML-DECLARATION 381
XML generation
compatibility 570
counting generated characters 380
description 379
effect of CHAR(EBCDIC) 371
enhancing output
example of converting hyphens to underscores 393
example of modifying data definitions 390
rationale and techniques 389
example 385
generating attributes 380
generating elements 380
handling errors 384
ignored data items 380
no byte order mark 384
overview 379
using namespace prefixes 382
using namespaces 381
XML output
controlling the encoding of 383
enhancing
example of converting hyphens to underscores 393

XML output *(continued)*
enhancing *(continued)*
example of modifying data definitions 390
rationale and techniques 389
generating
example 385
overview 379
XML PARSE statement
NOT ON EXCEPTION 376
ON EXCEPTION 376
overview 361
using 363
XML parser
conformance 646
error handling 376
overview 361
XML parsing
control flow with processing procedure 366
description 363
effect of CHAR(EBCDIC) 371
handling code-page conflicts 376
handling exceptions 374
overview 361
special registers 365
terminating 377
XML processing procedure
control flow with parser 366
error with EXIT PROGRAM or GOBACK 365
example 368
handling parsing exceptions 374
restriction on XML PARSE 365
specifying 363
using special registers 365
with code-page conflicts 377
writing 365
XML-CODE special register
content 366
control flow between parser and processing procedure 366
description 365
exception codes for generating 649
exception codes for parsing
encoding conflicts 375
handleable 639
not handleable 643
terminating parsing 377
using in generating 382
using in parsing 361
with code-page conflicts 376
with generating exceptions 384
with parsing exceptions 376
XML-EVENT special register
content 366
description 365
using 361, 364
with parsing exceptions 376
XML-NTEXT special register
content 367
description 365
using 361
with parsing exceptions 376
XML-TEXT special register
content 367
description 365

XML-TEXT special register *(continued)*
encoding 365
using 361
with parsing exceptions 376
XREF compiler option
description 291
finding copybook files 315
finding data- and procedure-names 315
getting output 317
XREF output
COPY/BASIS cross-references 325
data-name cross-references 324
program-name cross-references 325

Y

year field expansion 525
year windowing
advantages 522
how to control 535
MLE approach 522
when not supported 528
year-first date fields 527
year-last date fields 527
year-only date fields 527
YEARWINDOW compiler option
description 292

Z

zero comparison (See sign condition) 532
zero suppression
example of BLANK WHEN ZERO clause 41
PICTURE symbol Z 41
zoned decimal data (USAGE DISPLAY)
effect of ZWB on comparison to alphanumeric 293
example 39
format 43
sign representation 50
zSeries host data format considerations 581
ZWB compiler option 293

Readers' Comments — We'd Like to Hear from You

COBOL for AIX
Programming Guide
Version 3.1

Publication No. SC23-8566-00

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.

Send your comments to the address on the reverse side of this form.

If you would like a response from IBM, please fill in the following information:

Name	Address
Company or Organization	
Phone No.	E-mail address

Readers' Comments — We'd Like to Hear from You
SC23-8566-00

Cut or Fold
Along Line

Fold and Tape　　**Please do not staple**　　Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL　PERMIT NO. 40　ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Reader Comments
DTX/E269
555 Bailey Avenue
San Jose, CA
U.S.A. 95141-9989

Fold and Tape　　**Please do not staple**　　Fold and Tape

SC23-8566-00

Cut or Fold
Along Line

Program Number: 5724-V62

Printed in USA

SC23-8566-00