Enterprise COBOL for z/OS
Version 6.3

Language Reference
**Note**
Before using this information and the product it supports, be sure to read the general information under “Notices” on page 627.

---

**Third edition (31 July 2020 update)**

This edition applies to Version 6 Release 3 of IBM® Enterprise COBOL for z/OS® (program number 5655-EC6) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure that you are using the correct edition for the level of the product.

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

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


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

<table>
<thead>
<tr>
<th>Tables</th>
<th>xv</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>xix</td>
</tr>
<tr>
<td>About this information</td>
<td>xix</td>
</tr>
<tr>
<td>How to read the syntax diagrams</td>
<td>xix</td>
</tr>
<tr>
<td>IBM extensions</td>
<td>xxi</td>
</tr>
<tr>
<td>Obsolete language elements</td>
<td>xxi</td>
</tr>
<tr>
<td>DBCS notation</td>
<td>xxi</td>
</tr>
<tr>
<td>Acknowledgment</td>
<td>xxi</td>
</tr>
<tr>
<td>Additional documentation and support</td>
<td>xxiii</td>
</tr>
<tr>
<td>Summary of changes</td>
<td>xxiii</td>
</tr>
<tr>
<td>Version 6 Release 3 with PTFs installed</td>
<td>xxiii</td>
</tr>
<tr>
<td>Version 6 Release 3</td>
<td>xxiii</td>
</tr>
<tr>
<td>How to send your comments</td>
<td>xxiv</td>
</tr>
</tbody>
</table>

#### Part 1. COBOL language structure

1. Chapter 1. Characters
   - User-defined COBOL words with single-byte characters
   - User-defined COBOL words with DBCS characters
   - System-names
   - Function-names
   - Reserved words
   - Figurative constants
   - Special registers

2. Chapter 2. Character sets and code pages
   - Character encoding units

3. Chapter 3. Character-strings: COBOL words and literals
   - COBOL words with single-byte characters
   - User-defined words with DBCS characters
   - User-defined words
   - System-names
   - Function-names
   - Reserved words
   - Figurative constants
   - Special registers
   - ADDRESS OF
   - DEBUG-ITEM
   - JNIENVPTR
   - JSON-CODE
   - JSON-STATUS
   - LENGTH OF
   - LINAGE-COUNTER
   - RETURN-CODE
   - SHIFT-OUT and SHIFT-IN
   - SORT-CONTROL
   - SORT-CORE-SIZE
   - SORT-FILE-SIZE
   - SORT-MESSAGE
   - SORT-MODE-SIZE
   - SORT-RETURN
   - TALLY
   - WHEN-COMPILED
   - XML-CODE
   - XML-EVENT
Chapter 6. Reference format

- Sequence number area
- Indicator area
- Area A
  - Division headers
  - Section headers
  - Paragraph headers or paragraph names
  - Level indicators (FD and SD) or level-numbers (01 and 77)
  - DECLARATIVES and END DECLARATIVES
  - End program, end class, and end method markers
- Area B
  - Entries, sentences, statements, clauses
  - Continuation lines
- Area A or Area B
  - Level-numbers
  - Comment lines
  - Floating comment indicators (*
  - Compiler-directing statements
  - Compiler directives
  - Debugging lines
  - Pseudo-text
  - Blank lines

Chapter 7. Scope of names

- Types of names
- External and internal resources
- Resolution of names

Chapter 8. Referencing data names, copy libraries, and PROCEDURE DIVISION names

- Uniqueness of reference
- Qualification
### Part 5. Data division

<table>
<thead>
<tr>
<th>Chapter 16. DATA DIVISION overview</th>
<th>145</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILE SECTION</td>
<td>147</td>
</tr>
<tr>
<td>WORKING-STORAGE SECTION</td>
<td>147</td>
</tr>
<tr>
<td>LOCAL-STOREAGE SECTION</td>
<td>148</td>
</tr>
<tr>
<td>LINKAGE SECTION</td>
<td>149</td>
</tr>
<tr>
<td>Data units</td>
<td>149</td>
</tr>
<tr>
<td>File data</td>
<td>149</td>
</tr>
<tr>
<td>Program data</td>
<td>150</td>
</tr>
<tr>
<td>Method data</td>
<td>150</td>
</tr>
<tr>
<td>Factory data</td>
<td>150</td>
</tr>
<tr>
<td>Instance data</td>
<td>150</td>
</tr>
<tr>
<td>Data relationships</td>
<td>150</td>
</tr>
<tr>
<td>Levels of data</td>
<td>150</td>
</tr>
<tr>
<td>Levels of data in a record description entry</td>
<td>151</td>
</tr>
<tr>
<td>Special level-numbers</td>
<td>152</td>
</tr>
<tr>
<td>Indentation</td>
<td>152</td>
</tr>
<tr>
<td>Classes and categories of group items</td>
<td>152</td>
</tr>
<tr>
<td>Classes and categories of data</td>
<td>153</td>
</tr>
<tr>
<td>Category descriptions</td>
<td>155</td>
</tr>
<tr>
<td>Alignment rules</td>
<td>158</td>
</tr>
<tr>
<td>Character-string and item size</td>
<td>158</td>
</tr>
<tr>
<td>Signed data</td>
<td>159</td>
</tr>
<tr>
<td>Dynamic-length items</td>
<td>159</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 17. DATA DIVISION--file description entries</th>
<th>161</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILE SECTION</td>
<td>166</td>
</tr>
<tr>
<td>EXTERNAL clause</td>
<td>166</td>
</tr>
<tr>
<td>GLOBAL clause</td>
<td>167</td>
</tr>
<tr>
<td>BLOCK CONTAINS clause</td>
<td>167</td>
</tr>
<tr>
<td>RECORD clause</td>
<td>168</td>
</tr>
<tr>
<td>Format 1</td>
<td>168</td>
</tr>
<tr>
<td>Format 2</td>
<td>169</td>
</tr>
<tr>
<td>Format 3</td>
<td>169</td>
</tr>
<tr>
<td>Statement</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>ALLOCATE statement</td>
<td>289</td>
</tr>
<tr>
<td>Example: ALLOCATE and FREE storage for UNBOUNDED tables</td>
<td>291</td>
</tr>
<tr>
<td>ALTER statement</td>
<td>292</td>
</tr>
<tr>
<td>Segmentation considerations</td>
<td>293</td>
</tr>
<tr>
<td>CALL statement</td>
<td>293</td>
</tr>
<tr>
<td>CANCEL statement</td>
<td>300</td>
</tr>
<tr>
<td>CLOSE statement</td>
<td>301</td>
</tr>
<tr>
<td>Effect of CLOSE statement on file types</td>
<td>302</td>
</tr>
<tr>
<td>COMPUTE statement</td>
<td>304</td>
</tr>
<tr>
<td>CONTINUE statement</td>
<td>305</td>
</tr>
<tr>
<td>DELETE statement</td>
<td>305</td>
</tr>
<tr>
<td>DISPLAY statement</td>
<td>307</td>
</tr>
<tr>
<td>DIVIDE statement</td>
<td>309</td>
</tr>
<tr>
<td>ENTRY statement</td>
<td>312</td>
</tr>
<tr>
<td>EVALUATE statement</td>
<td>313</td>
</tr>
<tr>
<td>Determining values</td>
<td>315</td>
</tr>
<tr>
<td>Comparing selection subjects and objects</td>
<td>316</td>
</tr>
<tr>
<td>Executing the EVALUATE statement</td>
<td>316</td>
</tr>
<tr>
<td>EXIT statement</td>
<td>316</td>
</tr>
<tr>
<td>Format 1 (simple)</td>
<td>317</td>
</tr>
<tr>
<td>Format 2 (program)</td>
<td>317</td>
</tr>
<tr>
<td>Format 3 (method)</td>
<td>317</td>
</tr>
<tr>
<td>Format 5 (inline-perform)</td>
<td>318</td>
</tr>
<tr>
<td>Format 6 (procedure)</td>
<td>318</td>
</tr>
<tr>
<td>FREE statement</td>
<td>319</td>
</tr>
<tr>
<td>GOBACK statement</td>
<td>319</td>
</tr>
<tr>
<td>GO TO statement</td>
<td>320</td>
</tr>
<tr>
<td>Unconditional GO TO</td>
<td>320</td>
</tr>
<tr>
<td>Conditional GO TO</td>
<td>321</td>
</tr>
<tr>
<td>Altered GO TO</td>
<td>321</td>
</tr>
<tr>
<td>IF statement</td>
<td>322</td>
</tr>
<tr>
<td>Transferring control</td>
<td>322</td>
</tr>
<tr>
<td>Nested IF statements</td>
<td>323</td>
</tr>
<tr>
<td>INITIALIZE statement</td>
<td>324</td>
</tr>
<tr>
<td>INITIALIZE statement rules</td>
<td>326</td>
</tr>
<tr>
<td>INSPECT statement</td>
<td>327</td>
</tr>
<tr>
<td>Data flow</td>
<td>333</td>
</tr>
<tr>
<td>Comparison cycle</td>
<td>333</td>
</tr>
<tr>
<td>Example of the INSPECT statement</td>
<td>334</td>
</tr>
<tr>
<td>INVOKE statement</td>
<td>335</td>
</tr>
<tr>
<td>Interoperable data types for COBOL and Java</td>
<td>339</td>
</tr>
<tr>
<td>Miscellaneous argument types for COBOL and Java</td>
<td>341</td>
</tr>
<tr>
<td>JSON GENERATE statement</td>
<td>342</td>
</tr>
<tr>
<td>Nested JSON GENERATE or JSON PARSE statements</td>
<td>346</td>
</tr>
<tr>
<td>Operation of JSON GENERATE</td>
<td>347</td>
</tr>
<tr>
<td>Format conversion of elementary data</td>
<td>347</td>
</tr>
<tr>
<td>Trimming of generated JSON data</td>
<td>349</td>
</tr>
<tr>
<td>JSON name formation</td>
<td>349</td>
</tr>
<tr>
<td>JSON PARSE statement</td>
<td>350</td>
</tr>
<tr>
<td>Nested JSON GENERATE or JSON PARSE statements</td>
<td>355</td>
</tr>
<tr>
<td>Operation of JSON PARSE</td>
<td>355</td>
</tr>
<tr>
<td>Examples of matched and mismatched data definitions and JSON text</td>
<td>356</td>
</tr>
<tr>
<td>Count of table elements set by JSON PARSE</td>
<td>357</td>
</tr>
<tr>
<td>Valid and invalid elementary moves</td>
<td>357</td>
</tr>
<tr>
<td>MERGE statement</td>
<td>358</td>
</tr>
<tr>
<td>MERGE special registers</td>
<td>362</td>
</tr>
<tr>
<td>Segmentation considerations</td>
<td>362</td>
</tr>
<tr>
<td>MOVE statement</td>
<td>363</td>
</tr>
</tbody>
</table>
Part 7. Intrinsic functions .......................................................... 457

Chapter 21. Intrinsic functions .......................................................... 459

Function definition and evaluation ................................................. 460
Types of functions ............................................................................ 460
Rules for usage ................................................................................ 461
Arguments ...................................................................................... 462
Examples ....................................................................................... 463

Function definitions ....................................................................... 465

ABS ............................................................................................ 471
ACOS ....................................................................................... 471
ANNUITY ................................................................................. 472
ASIN ....................................................................................... 472
ATAN ....................................................................................... 473
BIT-OF .................................................................................... 473
BIT-TO-CHAR ........................................................................ 474
BYTE-LENGTH ........................................................................ 474
CHAR ..................................................................................... 475
COS ....................................................................................... 475
CURRENT-DATE ...................................................................... 476
DATE-OF-INTEGER ................................................................... 477
DATE-TO-YYYYMMDD .............................................................. 477
DAY-OF-INTEGER .................................................................... 478
DAY-TO-YYYYDDD ................................................................. 478
DISPLAY-OF ........................................................................... 479
E ............................................................................................ 480
EXP ....................................................................................... 480
EXP10 ................................................................................... 481
FACTORIAL ............................................................................. 481
HEX-OF .................................................................................. 481
HEX-TO-CHAR ........................................................................ 482
INTEGER ............................................................................... 483
INTEGER-OF-DATE ................................................................. 483
INTEGER-OF-DAY ................................................................. 483
INTEGER-PART ....................................................................... 484
LENGTH .................................................................................. 484
LOG ....................................................................................... 486
LOG10 ................................................................................... 486
LOWER-CASE ....................................................................... 486
MAX ..................................................................................... 487
MEAN .................................................................................... 488
MEDIAN ................................................................................ 488
MIDRANGE ............................................................................. 488
MIN ..................................................................................... 489
MOD ..................................................................................... 490
NATIONAL-OF ................................................................. 490
Part 8. Compiler-directing statements and compiler directives................................. 519

Chapter 22. Compiler-directing statements.................................................................... 521
  BASIS statement............................................................................................ 521
  PROCESS(CBL) statement............................................................................ 522
  *CONTROL (*CBL) statement.................................................................... 522
  Source code listing...................................................................................... 523
  Object code listing...................................................................................... 523
  Storage map listing..................................................................................... 523
  COPY statement.......................................................................................... 524
    Comparison and replacement rules.......................................................... 527
    Comparison and replacement examples................................................... 529
    Copy member search order...................................................................... 533
  DELETE statement....................................................................................... 533
  EJECT statement......................................................................................... 534
  ENTER statement......................................................................................... 534
  INSERT statement....................................................................................... 535
  READY or RESET TRACE statement.......................................................... 536
  REPLACE statement.................................................................................... 536
    Comparison rules..................................................................................... 537
    Replacement rules.................................................................................... 538
  SERVICE LABEL statement........................................................................ 539
  SERVICE RELOAD statement...................................................................... 540
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>Permissible statements for line-sequential files</td>
</tr>
<tr>
<td>50</td>
<td>Sending and receiving fields for format-1 SET statement</td>
</tr>
<tr>
<td>51</td>
<td>Sending and receiving fields for format-5 SET statement</td>
</tr>
<tr>
<td>52</td>
<td>Character positions examined when DELIMITED BY is not specified</td>
</tr>
<tr>
<td>53</td>
<td>Meanings of environment-names in SPECIAL NAMES paragraph</td>
</tr>
<tr>
<td>54</td>
<td>Table of functions</td>
</tr>
<tr>
<td>55</td>
<td>REVERSE function of character Kä</td>
</tr>
<tr>
<td>56</td>
<td>TRIM function types depending on the argument types</td>
</tr>
<tr>
<td>57</td>
<td>ULENGTH function of character ä</td>
</tr>
<tr>
<td>58</td>
<td>Returned values of the UPOS function</td>
</tr>
<tr>
<td>59</td>
<td>Returned values of the USUBSTR function</td>
</tr>
<tr>
<td>60</td>
<td>Returned values of the USUPPLEMENTARY function</td>
</tr>
<tr>
<td>61</td>
<td>Byte validity for UTF-8 data</td>
</tr>
<tr>
<td>62</td>
<td>Encoding unit validity for UTF-16 data</td>
</tr>
<tr>
<td>63</td>
<td>Returned values of the UWIDTH function</td>
</tr>
<tr>
<td>64</td>
<td>Execution of debugging declaratives</td>
</tr>
<tr>
<td>65</td>
<td>Predefined compilation variables</td>
</tr>
<tr>
<td>66</td>
<td>IBM extension language elements</td>
</tr>
<tr>
<td>67</td>
<td>Compiler limits</td>
</tr>
<tr>
<td>68</td>
<td>EBCDIC collating sequence</td>
</tr>
<tr>
<td>69</td>
<td>ASCII collating sequence</td>
</tr>
<tr>
<td>70</td>
<td>Reserved words</td>
</tr>
<tr>
<td>71</td>
<td>Context-sensitive words</td>
</tr>
<tr>
<td>72</td>
<td>2002/2014 COBOL Standard features implemented in V3 or later versions that will potentially affect existing programs</td>
</tr>
<tr>
<td>73</td>
<td>2002/2014 COBOL Standard features implemented in V3 or later versions that will not affect existing programs</td>
</tr>
</tbody>
</table>
About this information

This information describes the COBOL language supported by IBM Enterprise COBOL for z/OS, referred to in this information as Enterprise COBOL.

See the IBM Enterprise COBOL for z/OS Programming Guide for information and examples that will help you write, compile, and debug programs and classes.

How to read the syntax diagrams

Throughout the document, diagrams illustrate Enterprise COBOL syntax.

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

• 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).

  Format

  ►► STATEMENT —► required item —►

• Optional items appear below the main path.

  Format

  ►► STATEMENT —► optional item —►

• When 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.

  Format

  ►► STATEMENT —► required choice 1 —► required choice 2 —►

  If choosing one of the items is optional, the entire stack appears below the main path.
An arrow returning to the left above the main line indicates an item that can be repeated.

A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

Variables appear in italic lowercase letters (for example, `parmX`). They represent user-supplied names or values.

If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, they must be entered as part of the syntax.

The following example shows how the syntax is used.

Notes:
1. The `STATEMENT` keyword must be specified and coded as shown.
2. This operand is required. Either `identifier-1` or `literal-1` must be coded.
The item 1 fragment is optional; it can be coded or not, as required by the application. If item 1 is coded, it can be repeated with each entry separated by one or more COBOL separators. Entry selections allowed for this fragment are described at the bottom of the diagram.

The operand identifier-3 and associated TO keyword are required and can be repeated with one or more COBOL separators separating each entry. Each entry can be assigned the keyword ROUNDED.

The ON SIZE ERROR phrase with associated imperative-statement-1 is optional. If the ON SIZE ERROR phrase is coded, the keyword ON is optional.

The END-STATEMENT keyword can be coded to end the statement. It is not a required delimiter.

IBM extensions

IBM extensions generally add features, syntax, or rules that are not specified in the ANSI and ISO COBOL standards that are listed in Appendix H, “Industry specifications,” on page 613. In this document, the term 85 COBOL Standard refers to those standards.

Extensions range from minor relaxation of rules to major capabilities, such as XML support, Unicode support, object-oriented COBOL for Java™ interoperability, and DBCS character handling.

The rest of this document describes the complete language without identifying extensions. You will need to review Appendix A, “IBM extensions,” on page 557 and the Compiler options in the Enterprise COBOL Programming Guide if you want to use only standard language elements.

Obsolete language elements

Obsolete language elements are elements that are categorized as obsolete in the 85 COBOL Standard. Those elements are not part of the 2002 COBOL Standard.

This does not imply that IBM will remove the 85 COBOL Standard obsolete elements from a future release of Enterprise COBOL.

The following language elements are categorized as obsolete by the 85 COBOL Standard:

- ALTER statement
- AUTHOR paragraph
- Comment entry
- DATA RECORDS clause
- DATE-COMPILED paragraph
- DATE-WRITTEN paragraph
- DEBUG-ITEM special register
- Debugging sections
- ENTER statement
- GO TO without a specified procedure-name
- INSTALLATION paragraph
- LABEL RECORDS clause
- MEMORY SIZE clause
- MULTIPLE FILE TAPE clause
- RERUN clause
- REVERSED phrase
- SECURITY paragraph
- Segmentation module
• STOP literal format of the STOP statement
• USE FOR DEBUGGING declarative
• VALUE OF clause
• The figurative constant ALL literal with a length greater than one, when the figurative constant is associated with a numeric or numeric-edited item

**DBCS notation**

Double-Byte Character Set (DBCS) strings in literals, comments, and user-defined words are delimited by shift-out and shift-in characters.

In this document, the shift-out delimiter is represented pictorially by the < character, and the shift-in character is represented pictorially by the > character. The single-byte EBCDIC codes for the shift-out and shift-in delimiters are X'0E' and X'0F', respectively.

The <> symbol denotes contiguous shift-out and shift-in characters. The > symbol denotes contiguous shift-in and shift-out characters.

DBCS characters are shown in this form: D1D2D3. Latin alphabet characters in DBCS representation are shown in this form: .A.B.C. The dots that precede the letters represent the hexadecimal value X'42'.

**Notes:**

• In EBCDIC DBCS data containing mixed single-byte and double-byte characters, double-byte character strings are delimited by shift-out and shift-in characters.

• In ASCII DBCS data containing mixed single-byte and double-byte characters, double-byte character strings are not delimited by shift-out and shift-in characters.

**Acknowledgment**

The following extract from Government Printing Office Form Number 1965-0795689 is presented for the information and guidance of the user:

Any organization interested in reproducing the COBOL report and specifications in whole or in part, using ideas taken from this report as the basis for an instruction manual or for any other purpose is free to do so. However, all such organizations are requested to reproduce this section as part of the introduction to the document. Those using a short passage, as in a book review, are requested to mention COBOL in acknowledgment of the source, but need not quote this entire section.

COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

No warranty, expressed or implied, is made by any contributor or by the COBOL Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection there with.

Procedures have been established for the maintenance of COBOL. Inquiries concerning the procedures for proposing changes should be directed to the Executive Committee of the Conference on Data Systems Languages.

The authors and copyright holders of copyrighted material:

• FLOW-MATIC (Trademark of Sperry Rand Corporation), Programming for the UNIVAC(R) I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation
• IBM Commercial Translator, Form No. F28-8013, copyrighted 1959 by IBM
• FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell

have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.
Note: The Conference on Data Systems Languages (CODASYL), mentioned above, is no longer in existence.

Additional documentation and support

Enterprise COBOL provides Portable Document Format (PDF) versions of the entire library for this version and for previous versions on the library page at http://www.ibm.com/support/docview.wss?uid=swg27036733. These documents are also available in Japanese.

Support information is also available at https://www.ibm.com/support/home/product/B984385H82239E03/Enterprise_COBOL_for_z/OS.

Summary of changes

This section lists the key changes that have been made to this document since Enterprise COBOL for z/OS Version 6 Release 3. The changes that are described in this information have an associated cross-reference for your convenience. The latest technical changes are marked within >| and |< in the HTML version, or marked by vertical bars (|) in the left margin in the PDF version.

Version 6 Release 3 with PTFs installed

- PH18638: You can compile programs with the LP(64) option when the program contains UTF-8 data items. (“Symbols used in the PICTURE clause” on page 189 and “UTF-8 literals” on page 39)
- PH18640: You can compile programs with the LP(64) option when the program contains dynamic-length elementary items. (“Dynamic-length items” on page 159)
- PH18641: A new "NAME is OMITTED" phrase is added to the JSON GENERATE statement to allow generation of an anonymous JSON object, whose top-level parent name is not generated. (“JSON GENERATE statement” on page 342)
- PH20724: The use of passing a file-name to a subprogram with the USING phrase of the CALL statement is restored. (“CALL statement” on page 293)
- PH20997: The UUID4 intrinsic function is introduced. (“UUID4” on page 511)
  Note that COBOL Runtime LE PTF UI66560(V2R2)/UI66555(V2R3)/UI66557(V2R4) must also be applied on all systems where programs that make use of this new feature are linked or run.
- PH26789: A new "CONVERTING" phrase is added to the JSON GENERATE and JSON PARSE statements so that you can generate and parse JSON boolean values. (“JSON GENERATE statement” on page 342 and “JSON PARSE statement” on page 350)
  Note that COBOL Runtime LE APAR PH26698 must also be applied on all systems where programs that make use of this new feature are linked or run.

Version 6 Release 3

Language element changes

- A DYNAMIC LENGTH clause is supported to specify a dynamic-length elementary item. A dynamic-length elementary item is a data item whose length might change at run time. This is part of the 2014 COBOL Standard. (“DYNAMIC LENGTH clause” on page 178)
- A UTF-8 phrase is added to the USAGE clause, which can be used to define UTF-8 character data is added. The new USAGE clause plus the ‘U’ PICTURE symbol creates a new class of data (UTF-8) and a new category of data (UTF-8). (“Symbols used in the PICTURE clause” on page 189 and “UTF-8 literals” on page 39)
- A POINTER-32 phrase is added to the USAGE clause, which can be used to define pointer data items or data-pointers. (“POINTER-32 phrase” on page 224)
• The REPOSITORY paragraph FUNCTION specifier INTRINSIC allows declaration of intrinsic function names that may be used without specifying the word FUNCTION. This is part of the 2002 COBOL Standard. ("REPOSITORY paragraph" on page 116)

• Programs can no longer pass a file-name to a subprogram with the USING phrase of the CALL statement. ("CALL statement" on page 293)

AMODE 64 support
You can now develop AMODE 64 (64-bit) applications with Enterprise COBOL. The following language elements are enhanced to provide the AMODE 64 support:

Using POINTER and POINTER-32 phrases under LP(64)
When LP(32) is in effect, the USAGE POINTER and USAGE POINTER-32 data types are synonyms. However, they behave differently when LP(64) is in effect.

• A POINTER data item can be SET to a value from a POINTER-32 data item. The high-order word of the POINTER data item is cleared to zero. ("POINTER phrase" on page 223)

• A POINTER-32 data item can be SET to a value from a POINTER data item. Only the low-order word of the POINTER data item is used in this case. It is a programming error if the high-order word of the AMODE 64 POINTER data item is not zero. ("POINTER-32 phrase" on page 224)

Using SYNCHRONIZED clause under LP(64)
If the SYNCHRONIZED clause is specified for data items that have the following usages, the items will be aligned on a fullword boundary if LP(32) is in effect, or on a doubleword boundary if LP(64) is in effect. ("SYNCHRONIZED clause" on page 212)

Note the following usages when you compile code under LP(64):

• USAGE FUNCTION-POINTER ("FUNCTION-POINTER phrase" on page 221)
• USAGE INDEX ("INDEX phrase" on page 222)
• USAGE OBJECT REFERENCE ("OBJECT REFERENCE phrase" on page 222)
• USAGE POINTER ("POINTER phrase" on page 223)
• USAGE POINTER-32 ("POINTER-32 phrase" on page 224)
• USAGE PROCEDURE-POINTER ("PROCEDURE-POINTER phrase" on page 225)

Using intrinsic functions under LP(64)
When you compile with LP(64), you must be aware that the returned value is an 18-digit integer. On the other hand, when you compile with LP(32), the returned value of the following intrinsic functions is a 9-digit integer.

• LENGTH ("LENGTH" on page 484)
• ULENGTH ("ULENGTH" on page 506)
• UPOS ("UPOS" on page 507)
• USUPPLEMENTARY ("USUPPLEMENTARY" on page 510)
• UVALID ("UVALID" on page 512)

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 documentation for this product, send your comments to: compinfo@cn.ibm.com.

Be sure to include the name of the document, the publication number of the document, the version of the product, and, if applicable, the specific location (for example, 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.
The most basic and indivisible unit of the COBOL language is the character. The basic character set includes the letters of the Latin alphabet, digits, and special characters.

In the COBOL language, individual characters are joined to form character-strings and separators. Character-strings and separators, then, are used to form the words, literals, phrases, clauses, statements, and sentences that form the language.

The basic characters used in forming character-strings and separators in source code are shown in Table 1 on page 3.

For certain language elements, the basic character set is extended with the EBCDIC Double-Byte Character Set (DBCS).

DBCS characters can be used in forming user-defined words.

The content of alphanumeric literals, comment lines, and comment entries can include any of the characters in the computer's compile-time character set, and can include both single-byte and DBCS characters.

Runtime data can include any characters from the runtime character set of the computer. The runtime character set of the computer can include alphanumeric characters, DBCS characters, national characters, and UTF-8 characters. National characters are represented in UTF-16, a 16-bit encoding form of Unicode. UTF-8 characters are represented in UTF-8, a variable length encoding form of Unicode (1 to 4 bytes for each character).

When the NSYMBOL (NATIONAL) compiler option is in effect, literals identified by the opening delimiter N" or N' are national literals and can contain any single-byte or double-byte characters, or both, that are valid for the compile-time code page in effect (either the default code page or the code page specified for the CODEPAGE compiler option). Characters contained in national literals are represented as national characters at run time.

For details, see “User-defined words with DBCS characters” on page 10, “DBCS literals” on page 38, and “National literals” on page 42.

Literals identified by the opening delimiter U" or U' are UTF-8 literals. They can contain any single-byte or double-byte characters, or both, that are valid for the compile-time code page in effect (either the default code page or the code page specified for the CODEPAGE compiler option). Characters contained in UTF-8 literals are represented as UTF-8 characters at run time.

Table 1. Basic COBOL character set. This table lists basic COBOL character set.

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Space</td>
</tr>
<tr>
<td>+</td>
<td>Plus sign</td>
</tr>
<tr>
<td>-</td>
<td>Minus sign or hyphen</td>
</tr>
<tr>
<td>*</td>
<td>Asterisk</td>
</tr>
<tr>
<td>/</td>
<td>Forward slash or solidus</td>
</tr>
<tr>
<td>=</td>
<td>Equal sign</td>
</tr>
<tr>
<td>$</td>
<td>Currency sign¹</td>
</tr>
<tr>
<td>,</td>
<td>Comma</td>
</tr>
<tr>
<td>;</td>
<td>Semicolon</td>
</tr>
</tbody>
</table>
Table 1. Basic COBOL character set. This table lists basic COBOL character set. (continued)

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Decimal point or period</td>
</tr>
<tr>
<td>&quot;</td>
<td>Quotation mark(^2)</td>
</tr>
<tr>
<td>'</td>
<td>Apostrophe</td>
</tr>
<tr>
<td>(</td>
<td>Left parenthesis</td>
</tr>
<tr>
<td>)</td>
<td>Right parenthesis</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>:</td>
<td>Colon</td>
</tr>
<tr>
<td>_</td>
<td>Underscore</td>
</tr>
<tr>
<td>A - Z</td>
<td>Alphabet (uppercase)</td>
</tr>
<tr>
<td>a - z</td>
<td>Alphabet (lowercase)</td>
</tr>
<tr>
<td>0 - 9</td>
<td>Numeric characters</td>
</tr>
</tbody>
</table>

1. The currency sign is the character with the value X'5B', regardless of the code page in effect. The assigned graphic character can be the dollar sign or a local currency sign.
2. The quotation mark is the character with the value X'7F'.
Chapter 2. Character sets and code pages

A character set is a set of letters, numbers, special characters, and other elements used to represent information. The term code page refers to a coded character set.

A character set is independent of a coded representation. A coded character set is the coded representation of a set of characters, where each character is assigned a numerical position, called a code point, in the encoding scheme. ASCII and EBCDIC are examples of types of coded character sets. Each variation of ASCII or EBCDIC is a specific coded character set.

Each code page that IBM defines is identified by a code page name, for example IBM-1252, and a coded character set identifier (CCSID), for example 1252.

Enterprise COBOL provides the CODEPAGE compiler option for specifying a coded character set for use at compile time and run time for code-page-sensitive elements, such as:
- The encoding of literals in the source program
- The default encoding for data items described with USAGE DISPLAY or DISPLAY-1
- The default encoding for XML parsing and XML generation

Some COBOL operations can override the encoding established by the CODEPAGE compiler option, for example:
- The DISPLAY-OF and NATIONAL-OF intrinsic functions can specify a CCSID as argument-2.
- The XML PARSE and XML GENERATE statements can specify a code page in the ENCODING phrase.

For further details about the CODEPAGE compiler option, see CODEPAGE in the Enterprise COBOL Programming Guide.

If you do not specify a code page, the default is code page IBM-1140, CCSID 1140.

The encoding of national and UTF-8 data is not affected by the CODEPAGE compiler option. The encoding for national literals and data items described with usage NATIONAL is UTF-16BE (big endian), CCSID 1200. A reference to UTF-16 in this document is a reference to UTF-16BE. The encoding for UTF-8 literals and data items described with usage UTF-8 is UTF-8, CCSID 1208.

Character encoding units

A character encoding unit (or encoding unit) is the unit of data that COBOL treats as a single character at run time. In this information, the terms character and character position refer to a single encoding unit.

The size of an encoding unit for data items and literals depends on the USAGE clause of the data item or the category of the literal as follows:
- For data items described with USAGE DISPLAY and for alphanumeric literals, an encoding unit is 1 byte, regardless of the code page used and regardless of the number of bytes used to represent a given graphic character.
- For data items described with USAGE DISPLAY-1 (DBCS data items) and for DBCS literals, an encoding unit is 2 bytes.
- For data items described with USAGE NATIONAL and for national literals, an encoding unit is 2 bytes.
- For data items described with USAGE UTF-8 and for UTF-8 literals, an encoding unit is 1 byte.

The relationship between a graphic character and an encoding unit depends on the type of code page used for the data item or literal. See the following types of runtime code pages:
- Single-byte EBCDIC
- EBCDIC DBCS
- Unicode UTF-16

© Copyright IBM Corp. 1991, 2020
• Unicode UTF-8

See the following sections for the details of each type of code page.

Also see the section *Specifying the encoding* in the *Enterprise COBOL Programming Guide*.

**Single-byte code pages**

You can use a single-byte EBCDIC code page in data items described with USAGE DISPLAY and in literals of category alphanumeric. An encoding unit is 1 byte and each graphic character is represented in 1 byte. For these data items and literals, you need not be concerned with encoding units.

**EBCDIC DBCS code pages**

**USAGE DISPLAY**

You can use a mixture of single-byte and double-byte EBCDIC characters in data items described with USAGE DISPLAY and in literals of category alphanumeric. Double-byte characters must be delimited by shift-out and shift-in characters. An encoding unit is 1 byte and the size of a graphic character is 1 byte or 2 bytes.

When alphanumeric data items or literals contain DBCS data, programmers are responsible for ensuring that operations do not unintentionally separate the multiple encoding units that form a graphic character. Care should be taken with reference modification, and truncation during moves should be avoided. The COBOL runtime system does not check for a split between the encoding units that form a graphic character or for the loss of shift-out or shift-in codes.

To avoid problems, you can convert alphanumeric literals and data items described with usage DISPLAY to national data (UTF-16) by moving the data items or literals to data items described with usage NATIONAL or by using the NATIONAL-OF intrinsic function. You can then perform operations on the national data with less concern for splitting graphic characters. You can convert the data back to USAGE DISPLAY by using the DISPLAY-OF intrinsic function.

**USAGE DISPLAY-1**

You can use double-byte characters of an EBCDIC DBCS code page in data items described with USAGE DISPLAY-1 and in literals of category DBCS. An encoding unit is 2 bytes and each graphic character is represented in a single 2-byte encoding unit. For these data items and literals, you need not be concerned with encoding units.

**Unicode UTF-16**

You can use UTF-16 in data items described with USAGE NATIONAL. National literals are stored as UTF-16 characters regardless of the code page used for the source program. An encoding unit for data items of usage NATIONAL and national literals is 2 bytes.

For most of the characters in UTF-16, a graphic character is one encoding unit. Characters converted to UTF-16 from an EBCDIC, ASCII, or EUC code page are represented in one UTF-16 encoding unit. Some of the other graphic characters in UTF-16 are represented by a surrogate pair or a combining character sequence. A surrogate pair consists of two encoding units (4 bytes). A combining character sequence consists of a base character and one or more combining marks or a sequence of one or more combining marks (4 bytes or more, in 2-byte increments). In data items of usage NATIONAL, each 2-byte encoding unit is treated as a character.

When national data contains surrogate pairs or combining character sequences, programmers are responsible for ensuring that operations on national characters do not unintentionally separate the multiple encoding units that form a graphic character. Care should be taken with reference modification, and truncation during moves should be avoided. The COBOL runtime system does not check for a split between the encoding units that form a graphic character.
Unicode UTF-8

You can use UTF-8 in data items described with the “U” symbol in their picture clause. These items are usage UTF-8. UTF-8 literals are stored as UTF-8 characters regardless of the code page used for the source program. An encoding unit for data items of usage UTF-8 and UTF-8 literals is 1 byte. In data items of usage UTF-8, each Unicode code point represented in the data item is encoded using between 1 and 4 encoding units.

For most of the characters in UTF-8, a graphic character is 1 to 4 encoding units (i.e., a character typically corresponds to a single Unicode code point). Characters converted to UTF-8 from an EBCDIC, ASCII, or EUC code page are represented in 1 to 4 UTF-8 encoding units. However, some of the other graphic characters in UTF-8 are represented by a combining character sequence, which corresponds to a sequence of Unicode code points. In particular, a combining character sequence consists of a base character and one or more combining marks or a sequence of one or more combining marks, with each combining mark being represented by 1 to 4 encoding units.

When UTF-8 data contains combining character sequences, programmers are responsible for ensure that operations on UTF-8 characters do not unintentionally separate the multiple groupings of encoding units that form graphic characters. You must pay attention to the reference modification and avoid truncations during moves. The COBOL runtime system does not check for a split between the groups of encoding units that form a graphic character.
Chapter 3. Character-strings: COBOL words and literals

A character-string is a character or a sequence of contiguous characters that forms a COBOL word, a literal, a PICTURE character-string, or a comment-entry. A character-string is delimited by separators.

A separator is a string of contiguous characters used to delimit character strings. Separators are described in detail under Chapter 4, “Separators,” on page 45.

Character strings and certain separators form text words. A text word is a character or a sequence of contiguous characters (possibly continued across lines) between character positions 8 and 72 inclusive in source text, library text, or pseudo-text. For more information about pseudo-text, see “Pseudo-text” on page 57.

Source text, library text, and pseudo-text can be written in single-byte EBCDIC and, for some character-strings, DBCS. (The compiler cannot process source code written in ASCII or Unicode.)

You can use single-byte and double-byte character-strings to form the following items:

• COBOL words
• Literals
• Comment text

You can use only single-byte characters to form PICTURE character-strings.

COBOL words with single-byte characters

A COBOL word is a character-string that forms a user-defined word, a system-name, or a reserved word. The maximum size of a COBOL user-defined word is 30 bytes. The number of characters that can be specified depends on the code page indicated by the compile-time locale.

Except for arithmetic operators and relation characters, each character of a COBOL word is selected from the following set:

• Latin uppercase letters A through Z
• Latin lowercase letters a through z
• digits 0 through 9
• - (hyphen)
• _ (underscore)

The hyphen cannot appear as the first or last character in such words. The underscore cannot appear as the first character in such words. Most user-defined words (all except section-names, paragraph-names, priority-numbers, and level-numbers) must contain at least one alphabetic character. Priority numbers and level numbers need not be unique; a given specification of a priority-number or level-number can be identical to any other priority-number or level-number.

In COBOL words (but not in the content of alphanumeric, DBCS, national, and UTF-8 literals), each lowercase single-byte alphabetic letter is considered to be equivalent to its corresponding single-byte uppercase alphabetic letter.

The following rules apply for all COBOL words:

• A reserved word cannot be used as a user-defined word or as a system-name.
• The same COBOL word, however, can be used as both a user-defined word and as a system-name. The classification of a specific occurrence of a COBOL word is determined by the context of the clause or phrase in which it occurs.

© Copyright IBM Corp. 1991, 2020
User-defined words with DBCS characters

There are the rules for forming user-defined words with DBCS characters.

The rules are:

**Contained characters**

DBCS user-defined words can contain only double-byte characters, and must contain at least one DBCS character that is not in the set A through Z, a through z, 0 through 9, hyphen, and underscore (DBCS representation of these characters has X'42' in the first byte).

DBCS user-defined words can contain characters that correspond to single-byte EBCDIC characters and those that do not correspond to single-byte EBCDIC characters. DBCS characters that correspond to single-byte EBCDIC characters follow the normal rules for COBOL user-defined words; that is, the characters A - Z, a - z, 0 - 9, the hyphen (-), and the underscore (_) are allowed. The hyphen cannot appear as the first or last character. The underscore cannot appear as the first character. Any of the DBCS characters that have no corresponding single-byte EBCDIC character can be used in DBCS user-defined words.

**Uppercase and lowercase letters**

In COBOL words, each lowercase single-byte encoded character "a" through "z" is considered to be equivalent to its corresponding single-byte encoded uppercase character. DBCS-encoded uppercase and lowercase letters are not equivalent.

**Value range**

DBCS user-defined words can contain characters whose values range from X'41' to X'FE' for both bytes.

**Maximum length**

14 characters

**Continuation**

Words formed with DBCS characters cannot be continued across lines.

**Use of shift-out and shift-in characters**

DBCS user-defined words begin with a shift-out character and end with a shift-in character.

User-defined words

A user-defined word is a COBOL word that must be supplied by the user to satisfy the format of a clause or statement.

The following sets of user-defined words are supported. The second column indicates whether DBCS characters are allowed in words of a given set.

<table>
<thead>
<tr>
<th>User-defined word</th>
<th>DBCS characters allowed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alphabet-name</td>
<td>Yes</td>
</tr>
<tr>
<td>Class-name (of data)</td>
<td>Yes</td>
</tr>
<tr>
<td>Condition-name</td>
<td>Yes</td>
</tr>
<tr>
<td>Data-name</td>
<td>Yes</td>
</tr>
<tr>
<td>File-name</td>
<td>Yes</td>
</tr>
<tr>
<td>Index-name</td>
<td>Yes</td>
</tr>
<tr>
<td>Level-numbers: 01–49, 66, 77, 88</td>
<td>No</td>
</tr>
<tr>
<td>User-defined word</td>
<td>DBCS characters allowed?</td>
</tr>
<tr>
<td>------------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>Library-name</td>
<td>No</td>
</tr>
<tr>
<td>Mnemonic-name</td>
<td>Yes</td>
</tr>
<tr>
<td>Object-oriented class-name</td>
<td>No</td>
</tr>
<tr>
<td>Paragraph-name</td>
<td>Yes</td>
</tr>
<tr>
<td>Priority-numbers: 00–99</td>
<td>No</td>
</tr>
<tr>
<td>Program-name</td>
<td>No</td>
</tr>
<tr>
<td>Record-name</td>
<td>Yes</td>
</tr>
<tr>
<td>Section-name</td>
<td>Yes</td>
</tr>
<tr>
<td>Symbolic-character</td>
<td>Yes</td>
</tr>
<tr>
<td>Text-name</td>
<td>No</td>
</tr>
<tr>
<td>XML-schema-name</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The maximum length of a user-defined word is 30 bytes, except for level-numbers and priority-numbers. Level-numbers and priority numbers must each be a one-digit or two-digit integer.

A given user-defined word can belong to only one of these sets, except that a given number can be both a priority-number and a level-number. Each user-defined word within a set must be unique, except for priority-numbers and level-numbers and except as specified in Chapter 8, “Referencing data names, copy libraries, and PROCEDURE DIVISION names,” on page 63.

The following types of user-defined words can be referenced by statements and entries in the program in which the user-defined word is declared:

- Paragraph-name
- Section-name

The following types of user-defined words can be referenced by any COBOL program, provided that the compiling system supports the associated library or other system and that the entities referenced are known to that system:

- Library-name
- Text-name

The following types of names, when they are declared within a configuration section, can be referenced by statements and entries in the program that contains the configuration section or in any program contained within that program:

- Alphabet-name
- Class-name
- Condition-name
- Mnemonic-name
- Symbolic-character
- XML-schema-name

The function of each user-defined word is described in the clause or statement in which it appears.

**Related references**

Appendix F, “Context-sensitive words,” on page 607
System-names

A system-name is a character string that has a specific meaning to the system.

There are three types of system-names:

- Computer-name
- Language-name
- Implementor-name

There are four types of implementer-names:

- Environment-name
- External-class-name
- External-fileid
- Assignment-name

The meaning of each system-name is described with the format in which it appears.

Computer-name can be written in DBCS characters, but the other system-names cannot.

Function-names

A function-name specifies the mechanism provided to determine the value of an intrinsic function.

The same word, in a different context, can appear in a program as a user-defined word or a system-name. For a list of function-names and their definitions, see Table 54 on page 466.

Reserved words

A reserved word is a character-string with a predefined meaning in a COBOL source unit.

Reserved words are listed in Appendix E, “Reserved words,” on page 589. There are six types of reserved words:

- Keywords
- Optional words
- Figurative constants
- Special character words
- Special object identifiers
- Special registers

Keywords

Keywords are reserved words that are required within a given clause, entry, or statement. Within each format, such words appear in uppercase on the main path.

Optional words

Optional words are reserved words that can be included in the format of a clause, entry, or statement in order to improve readability. They have no effect on the execution of the program.

Figurative constants

See “Figurative constants” on page 13.

Special character words

There are five types of special character words, which are recognized as special characters only when represented in single-byte characters:
• **Arithmetic operators:** + - / * **
  See “Arithmetic expressions” on page 244.
• **Relational operators:** < > = <= >=
  See “Conditional expressions” on page 246.
• **Floating comment indicators:** *
  See “Floating comment indicators (*>)” on page 56.
• **Pseudo-text delimiters in COPY and REPLACE statements:** ==
  See “COPY statement” on page 524 and “REPLACE statement” on page 536.
• **Compiler directive indicators:** >
  See Chapter 23, “Compiler directives,” on page 545.

**Special object identifiers**
COBOL provides two special object identifiers, SELF and SUPER:

**SELF**
A special object identifier that you can use in the PROCEDURE DIVISION of a method. SELF refers to the object instance used to invoke the currently executing method. You can specify SELF only in places that are explicitly listed in the syntax diagrams.

**SUPER**
A special object identifier that you can use in the PROCEDURE DIVISION of a method only as the object identifier in an INVOKE statement. When used in this way, SUPER refers to the object instance used to invoke the currently executing method. The resolution of the method to be invoked ignores any methods declared in the class definition of the currently executing method and methods defined in any class derived from that class. Thus, the method invoked is inherited from an ancestor class.

**Special registers**
See “Special registers” on page 15.

**Figurative constants**

*Figurative constants* are reserved words that name and refer to specific constant values. The reserved words for figurative constants and their meanings are listed in this section.

**ZERO, ZEROS, ZEROES**
Represents the numeric value zero (0) or one or more occurrences of the character zero, depending on context.

When the figurative constant ZERO, ZEROS, or ZEROES is used in a context that requires an alphanumeric character, an alphanumeric character zero is used. When the context requires a national character zero, a national character zero is used (value NX'0030'). When the context cannot be determined, an alphanumeric character zero is used.

**SPACE, SPACES**
Represents one or more blanks or spaces. SPACE is treated as an alphanumeric literal when used in a context that requires an alphanumeric character, as a DBCS literal when used in a context that requires a DBCS character, as a national literal when used in a context that requires a national character.

The EBCDIC DBCS space character has the value X'4040', the national space character has the value NX'0020', and the UTF-8 space character has the value UX'20'.

**HIGH-VALUE, HIGH-VALUES**
Represents one or more occurrences of the character that has the highest ordinal position in the collating sequence used.
HIGH-VALUE is treated as an alphanumeric literal in a context that requires an alphanumeric character. For alphanumeric data with the EBCDIC collating sequence, the value is X'FF'. For other alphanumeric data, the value depends on the collating sequence in effect.

HIGH-VALUE is treated as a national literal when used in a context that requires a national literal. The value is national character NX'FFFF'.

HIGH-VALUE is treated as a UTF-8 literal when used in a context that requires a UTF-8 literal. The value is UTF-8 character UX'F7BFBFBF' corresponding to Unicode code point U+10FFFF.

When the context cannot be determined, an alphanumeric context is assumed and the value X'FF' is used.

Usage note: You should not use HIGH-VALUE (or a value assigned from HIGH-VALUE) in a way that results in conversion between one data representation and another. X'FF' does not represent a valid EBCDIC character, and NX'FFFF' does not represent a valid national character. Conversion of either the alphanumeric or the national HIGH-VALUE representation to another representation results in a substitution character. For example, conversion of X'FF' to UTF-16 would give a substitution character, not NX'FFFF'.

LOW-VALUE, LOW-VALUES
Represents one or more occurrences of the character that has the lowest ordinal position in the collating sequence used.

LOW-VALUE is treated as an alphanumeric literal in a context that requires an alphanumeric character. For alphanumeric data with the EBCDIC collating sequence, the value is X'00'. For other alphanumeric data, the value depends on the collating sequence in effect.

LOW-VALUE is treated as a national literal when used in a context that requires a national literal. The value is national character NX'0000'.

LOW-VALUE is treated as a UTF-8 literal when used in a context that requires a UTF-8 literal. The value is UTF-8 character UX'00' corresponding to Unicode code point U+0000.

When the context cannot be determined, an alphanumeric context is assumed and the value X'00' is used.

QUOTE, QUOTES
Represents one or more occurrences of:
• The quotation mark character ("), if the QUOTE compiler option is in effect
• The apostrophe character ('), if the APOST compiler option is in effect

QUOTE or QUOTES represents an alphanumeric character when used in a context that requires an alphanumeric character, represents a national character when used in a context that requires a national character, and represents a UTF-8 character when used in a context that requires a UTF-8 character. The national character value of quotation mark is NX'0022'. The national character value of apostrophe is NX'0027'. The UTF-8 character value of quotation mark is UX'22'. The UTF-8 character value of apostrophe is UX'27'.

QUOTE and QUOTES cannot be used in place of a quotation mark or an apostrophe to enclose an alphanumeric literal, a DBCS literal, a national literal, or a UTF-8 literal.

ALL literal
literal can be an alphanumeric literal, a DBCS literal, a national literal, a UTF-8 literal, or a figurative constant other than the ALL literal.

When literal is not a figurative constant, ALL literal represents one or more occurrences of the string of characters that compose the literal.

When literal is a figurative constant, the word ALL has no meaning and is used only for readability.

The figurative constant ALL literal must not be used with the CALL, INSPECT, INVOKE, STOP, or STRING statements.
symbolic-character
   Represents one or more of the characters specified as a value of the symbolic-character in the
   SYMBOLIC CHARACTERS clause of the SPECIAL-NAMES paragraph.

   symbolic-character always represents an alphanumeric character; it can be used in a context that
   requires a national or UTF-8 character only when implicit conversion of alphanumeric to national or
   UTF-8 characters is defined. (It can be used, for example, in a MOVE statement where the receiving
   item is of class national or UTF-8 because implicit conversion is defined when the sending item is
   alphanumeric and the receiving item is national or UTF-8.)

NULL, NULLS
   Represents a value used to indicate that data items defined with USAGE POINTER, USAGE
   PROCEDURE-POINTER, USAGE FUNCTION-POINTER, USAGE OBJECT REFERENCE, or the ADDRESS
   OF special register do not contain a valid address. NULL can be used only where explicitly allowed in
   the syntax formats. NULL has the value zero.

The singular and plural forms of NULL, ZERO, SPACE, HIGH-VALUE, LOW-VALUE, and QUOTE can be used
interchangeably. For example, if DATA-NAME-1 is a five-character data item, each of the following
statements moves five spaces to DATA-NAME-1:

| MOVE SPACE   TO DATA-NAME-1 |
| MOVE SPACES  TO DATA-NAME-1 |
| MOVE ALL SPACES TO DATA-NAME-1 |

When the rules of COBOL permit any one spelling of a figurative constant name, any alternative spelling of
that figurative constant name can be specified.

You can use a figurative constant wherever literal appears in a syntax diagram, except where explicitly
prohibited. When a numeric literal appears in a syntax diagram, only the figurative constant ZERO (or
ZEROS or ZEROES) can be used. Figurative constants are not allowed as function arguments except in an
arithmetic expression, where the expression is an argument to a function.

The length of a figurative constant depends on the context of its use. The following rules apply:

• When a figurative constant is specified in a VALUE clause or associated with a data item (for example,
  when it is moved to or compared with another item), the length of the figurative constant character-
  string is equal to 1 or the number of character positions in the associated data item, whichever is
  greater.

• When a figurative constant, other than the ALL literal, is not associated with another data item (for
  example, in a CALL, INVOKE, STOP, STRING, or UNSTRING statement), the length of the character-
  string is one character.

Special registers

Special registers are reserved words that name storage areas generated by the compiler. Their primary
use is to store information produced through specific COBOL features. Each such storage area has a fixed
name, and must not be defined within the program.

For programs with the RECURSIVE attribute, for programs compiled with the THREAD option, and for
methods, storage for the following special registers is allocated on a per-invocation basis:

• ADDRESS OF
• JSON-CODE
• JSON-STATUS
• RETURN-CODE
• SORT-CONTROL
• SORT-CORE-SIZE
• SORT-FILE-SIZE
For the first call to a program after a cancel of that program, or for a method invocation, the compiler initializes the special register fields to their initial values.

For the following four cases:

- Programs that have the INITIAL clause specified
- Programs that have the RECURSIVE clause specified
- Programs compiled with the THREAD option
- Methods

the following special registers are reset to their initial value on each program or method entry:

- JSON-CODE
- JSON-STATUS
- RETURN-CODE
- SORT-CONTROL
- SORT-CORE-SIZE
- SORT-FILE-SIZE
- SORT-MESSAGE
- SORT-MODE-SIZE
- SORT-RETURN
- TALLY
- XML-CODE
- XML-EVENT
- XML-TEXT

Further, in the above four cases, values set in ADDRESS OF special registers persist only for the span of the particular program or method invocation.

In all other cases, the special registers will not be reset; they will be unchanged from the value contained on the previous CALL or INVOKE.

Unless otherwise explicitly restricted, a special register can be used wherever a data-name or identifier that has the same definition as the implicit definition of the special register can be used. Implicit definitions, if applicable, are given in the specification of each special register.

You can specify an alphanumeric special register in a function wherever an alphanumeric argument to a function is allowed, unless specifically prohibited.

If qualification is allowed, special registers can be qualified as necessary to provide uniqueness. (For more information, see “Qualification” on page 63.)
ADDRESS OF

The ADDRESS OF special register references the address of a data item in the LINKAGE SECTION, the LOCAL-STORAGE SECTION, or the WORKING-STORAGE SECTION.

For 01 and 77 level items in the LINKAGE SECTION, the ADDRESS OF special register can be used as either a sending item or a receiving item. For all other operands, the ADDRESS OF special register can be used only as a sending item.

The ADDRESS OF special register is implicitly defined as USAGE POINTER.

If LP(32) is in effect, 4 bytes are allocated for the special register; if LP(64) is in effect, 8 bytes are allocated for the special register.

A function-identifier is not allowed as the operand of the ADDRESS OF special register.

DEBUG-ITEM

The DEBUG-ITEM special register provides information for a debugging declarative procedure about the conditions that cause debugging section execution.

DEBUG-ITEM has the following implicit description:

<table>
<thead>
<tr>
<th>01  DEBUG-ITEM.</th>
</tr>
</thead>
<tbody>
<tr>
<td>02 DEBUG-LINE PICTURE IS X(6).</td>
</tr>
<tr>
<td>02 FILLER PICTURE IS X VALUE SPACE.</td>
</tr>
<tr>
<td>02 DEBUG-NAME PICTURE IS X(30).</td>
</tr>
<tr>
<td>02 FILLER PICTURE IS X VALUE SPACE.</td>
</tr>
<tr>
<td>02 DEBUG-SUB-1 PICTURE IS S9999 SIGN IS LEADING SEPARATE CHARACTER.</td>
</tr>
<tr>
<td>02 FILLER PICTURE IS X VALUE SPACE.</td>
</tr>
<tr>
<td>02 DEBUG-SUB-2 PICTURE IS S9999 SIGN IS LEADING SEPARATE CHARACTER.</td>
</tr>
<tr>
<td>02 FILLER PICTURE IS X VALUE SPACE.</td>
</tr>
<tr>
<td>02 DEBUG-SUB-3 PICTURE IS S9999 SIGN IS LEADING SEPARATE CHARACTER.</td>
</tr>
<tr>
<td>02 FILLER PICTURE IS X VALUE SPACE.</td>
</tr>
<tr>
<td>02 DEBUG-CONTENTS PICTURE IS X(n).</td>
</tr>
</tbody>
</table>

Before each debugging section is executed, DEBUG-ITEM is filled with spaces. The contents of the DEBUG-ITEM subfields are updated according to the rules for the MOVE statement, with one exception: DEBUG-CONTENTS is updated as if the move were an alphanumeric-to-alphanumeric elementary move without conversion of data from one form of internal representation to another.

After updating, the contents of the DEBUG-ITEM subfields are:

DEBUG-LINE
The source-statement sequence number (or the compiler-generated sequence number, depending on the compiler option chosen) that caused execution of the debugging section.

DEBUG-NAME
The first 30 characters of the name that caused execution of the debugging section. Any qualifiers are separated by the word 'OF'.

DEBUG-SUB-1, DEBUG-SUB-2, DEBUG-SUB-3
Always set to spaces. These subfields are documented for compatibility with previous COBOL products.

DEBUG-CONTENTS
Data is moved into DEBUG-CONTENTS, as shown in the following table.
### Table 2. DEBUG-ITEM subfield contents

<table>
<thead>
<tr>
<th>Cause of debugging section execution</th>
<th>Statement referred to in DEBUG-LINE</th>
<th>Contents of DEBUG-NAME</th>
<th>Contents of DEBUG-CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>procedure-name-1</code> ALTER reference</td>
<td>ALTER statement</td>
<td><code>procedure-name-1</code></td>
<td><code>procedure-name-n</code> in TO PROCEED TO phrase</td>
</tr>
<tr>
<td>GO TO <code>procedure-name-n</code></td>
<td>GO TO statement</td>
<td><code>procedure-name-n</code></td>
<td>Spaces</td>
</tr>
<tr>
<td><code>procedure-name-n</code> in SORT or MERGE input/output procedure</td>
<td>SORT or MERGE statement</td>
<td><code>procedure-name-n</code></td>
<td>&quot;SORT INPUT&quot;, &quot;SORT OUTPUT&quot;, or &quot;MERGE OUTPUT&quot; (as applicable)</td>
</tr>
<tr>
<td>PERFORM statement transfer of control</td>
<td>This PERFORM statement</td>
<td><code>procedure-name-n</code></td>
<td>&quot;PERFORM LOOP&quot;</td>
</tr>
<tr>
<td><code>procedure-name-n</code> in a USE procedure</td>
<td>Statement causing USE procedure execution</td>
<td><code>procedure-name-n</code></td>
<td>&quot;USE PROCEDURE&quot;</td>
</tr>
<tr>
<td>Implicit transfer from a previous sequential procedure</td>
<td>Previous statement executed in previous sequential procedure</td>
<td><code>procedure-name-n</code></td>
<td>&quot;FALL THROUGH&quot;</td>
</tr>
<tr>
<td>First execution of first nondeclarative procedure</td>
<td>Line number of first nondeclarative procedure-name</td>
<td>Name of first nondeclarative procedure</td>
<td>&quot;START PROGRAM&quot;</td>
</tr>
</tbody>
</table>

1. If this procedure is preceded by a section header, and control is passed through the section header, the statement number refers to the section header.

---

**JNIENVPTR**

The JNIENVPTR special register references the Java Native Interface (JNI) environment pointer. The JNI environment pointer is used in calling Java callable services.

JNIENVPTR is implicitly defined as USAGE POINTER, and cannot be specified as a receiving data item.

For information about using JNIENVPTR and JNI callable services, see *Accessing JNI services* in the *Enterprise COBOL Programming Guide*.

**JSON-CODE**

The JSON-CODE special register is used to indicate either that a JSON GENERATE or JSON PARSE statement executed successfully or that an exception occurred during JSON generation or parsing.

The JSON-CODE special register has the implicit definition:

```cobol
01 JSON-CODE PICTURE S9(9) USAGE BINARY VALUE 0.
```

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

At termination of a JSON GENERATE or JSON PARSE statement, JSON-CODE contains either zero, indicating successful completion of JSON generation or parsing, or a nonzero error code, indicating that an exception occurred during JSON generation or parsing. JSON GENERATE and JSON PARSE exception codes are detailed in *JSON GENERATE exceptions* and *JSON PARSE conditions and associated codes and runtime messages* in the *Enterprise COBOL Programming Guide*.

**Related references**

“JSON-STATUS” on page 19
**JSON-STATUS**

The JSON-STATUS special register is used to indicate either that a JSON PARSE statement executed successfully or that a nonexception condition occurred during the JSON parse operation.

The JSON-STATUS special register has the implicit definition:

```cobol
01 JSON-STATUS PICTURE S9(9) USAGE BINARY VALUE 0.
```

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

During execution of a JSON PARSE statement, nonexception conditions result in reason codes in the JSON-STATUS special register, but do not terminate execution of the statement. At termination of a JSON PARSE statement, JSON-STATUS contains either zero, indicating successful completion of the JSON parse operation, or a nonzero status value, representing one or more nonexception conditions that occurred prior to the exception condition. JSON-STATUS reason codes are detailed in *Nonexception conditions and corresponding values of JSON-STATUS* in the *Enterprise COBOL Programming Guide*.

**Related references**

“JSON-CODE” on page 18

---

**LENGTH OF**

The LENGTH OF special register contains the number of bytes used by a data item.

LENGTH OF creates an implicit special register that contains the current byte length of the data item referenced by the identifier.

For data items described with usage DISPLAY-1 (DBCS data items) and data items described with usage NATIONAL, each character occupies 2 bytes of storage.

LENGTH OF can be used in the PROCEDURE DIVISION anywhere a numeric data item that has the same definition as the implied definition of the LENGTH OF special register can be used.

If LP(32) is in effect, the LENGTH OF special register has the implicit definition:

```cobol
PICTURE 9(9) USAGE IS BINARY.
```

If LP(64) is in effect, the LENGTH OF special register has the implicit definition:

```cobol
PICTURE 9(18) USAGE IS BINARY.
```

If the data item referenced by the identifier contains the GLOBAL clause, the LENGTH OF special register is a global data item.

The LENGTH OF special register can appear within either the starting character position or the length expressions of a reference-modification specification. However, the LENGTH OF special register cannot be applied to any operand that is reference-modified.

The LENGTH OF operand cannot be a function, but the LENGTH OF special register is allowed in a function where an integer argument is allowed.

If the LENGTH OF special register is used as the argument to the LENGTH function, the result is 4 when LP(32) is in effect, and the result is 8 when LP(64) is in effect.

If the ADDRESS OF special register is used as the argument to the LENGTH function, the result is 4 when LP(32) is in effect, and the result is 8 when LP(64) is in effect.

LENGTH OF cannot be either of the following items:

- A receiving data item
- A subscript

When the LENGTH OF special register is used as a parameter on a CALL statement, it must be passed BY CONTENT or BY VALUE.
When a table element is specified, the LENGTH OF special register contains the length in bytes of one occurrence. When referring to a table element, the element name need not be subscripted.

A value is returned for any identifier whose length can be determined, even if the area referenced by the identifier is currently not available to the program.

A separate LENGTH OF special register exists for each identifier referenced with the LENGTH OF phrase. For example:

```
MOVE LENGTH OF A TO B
DISPLAY LENGTH OF A, A
ADD LENGTH OF A TO B
CALL "PROGX" USING BY REFERENCE A BY CONTENT LENGTH OF A
```

The intrinsic function LENGTH can also be used to obtain the length of a data item. For data items of usage NATIONAL, the length returned by the LENGTH function is the number of national character positions, rather than bytes; thus the LENGTH OF special register and the LENGTH intrinsic function have different results for data items of usage NATIONAL. Also, for table elements, the intrinsic function LENGTH requires a subscript, while the LENGTH OF special register does not. For all other data items, the result is the same.

The LENGTH intrinsic function, when applied to a null-terminated alphanumeric literal, returns the number of bytes in the literal prior to but not including the terminating null. (The LENGTH special register does not support literal operands.) For details about null-terminated alphanumeric literals, see “Null-terminated alphanumeric literals” on page 37.

**LINAGE-COUNTER**

A separate LINAGE-COUNTER special register is generated for each FD entry that contains a LINAGE clause. When more than one is generated, you must qualify each reference to a LINAGE-COUNTER with its related file-name.

The implicit description of the LINAGE-COUNTER special register is in one of the following cases:

- If the LINAGE clause specifies a data-name, LINAGE-COUNTER has the same PICTURE and USAGE as that data-name.
- If the LINAGE clause specifies an integer, LINAGE-COUNTER is a binary item with the same number of digits as that integer.

For more information, see “LINAGE clause” on page 171.

The value in LINAGE-COUNTER at any given time is the line number at which the device is positioned within the current page. LINAGE-COUNTER can be referred to in PROCEDURE DIVISION statements; it must not be modified by them.

LINAGE-COUNTER is initialized to 1 when an OPEN statement for its associated file is executed.

LINAGE-COUNTER is automatically modified by any WRITE statement for this file. (See “WRITE statement” on page 432.)

If the file description entry for a sequential file contains the LINAGE clause and the EXTERNAL clause, the LINAGE-COUNTER data item is an external data item. If the file description entry for a sequential file contains the LINAGE clause and the GLOBAL clause, the LINAGE-COUNTER data item is a global data item.

You can specify the LINAGE-COUNTER special register wherever an integer argument to a function is allowed.

**RETURN-CODE**

The RETURN-CODE special register can be used to pass a return code to the calling program or operating system when the current COBOL program ends.

When a COBOL program ends:
• If control returns to the operating system, the value of the RETURN-CODE special register is passed to the operating system as a user return code. The supported user return code values are determined by the operating system, and might not include the full range of RETURN-CODE special register values.

• If control returns to a calling program, the value of the RETURN-CODE special register is passed to the calling program. If the calling program is a COBOL program, the RETURN-CODE special register in the calling program is set to the value of the RETURN-CODE special register in the called program.

The RETURN-CODE special register has the implicit definition:

```
01  RETURN-CODE GLOBAL PICTURE S9(4) USAGE BINARY VALUE ZERO.
```

When used in nested programs, this special register is implicitly defined with the GLOBAL clause in the outermost program.

The following examples show how to set the RETURN-CODE special register:

• COMPUTE RETURN-CODE = 8.
• MOVE 8 to RETURN-CODE.

The RETURN-CODE special register does not return a value from an invoked method or from a program that uses CALL ... RETURNING. For more information, see “INVOKE statement” on page 335 or “CALL statement” on page 293.

You can specify the RETURN-CODE special register in a function wherever an integer argument is allowed.

The RETURN-CODE special register does not return information from a service call for a Language Environment® callable service. For more information, see Using Language Environment callable services in the Enterprise COBOL Programming Guide and the Language Environment Programming Guide.

**SHIFT-OUT and SHIFT-IN**

You can specify the SHIFT-OUT and SHIFT-IN special registers in a function wherever an alphanumeric argument is allowed.

The SHIFT-OUT and SHIFT-IN special registers are implicitly defined as alphanumeric data items of the format:

```
01  SHIFT-OUT GLOBAL PICTURE X(1) USAGE DISPLAY VALUE X"0E".
01  SHIFT-IN  GLOBAL PICTURE X(1) USAGE DISPLAY VALUE X"0F".
```

When used in nested programs, these special registers are implicitly defined with the global attribute in the outermost program.

These special registers represent EBCDIC shift-out and shift-in control characters, which are unprintable characters.

These special registers cannot be receiving items. SHIFT-OUT and SHIFT-IN cannot be used in place of the keyboard control characters when you are defining DBCS user-defined words or specifying EBCDIC DBCS literals.

The following example shows how SHIFT-OUT and SHIFT-IN might be used:

```
DATA DIVISION.
WORKING-STORAGE.
01  DBCSGRP.
   05  SO       PIC X.
   05  DBCSITEM PIC G(3) USAGE DISPLAY-1.
   05  SI       PIC X.
... 
PROCEDURE DIVISION.
MOVE SHIFT-OUT TO SO
MOVE G"<D1D2D3>" TO DBCSITEM
MOVE SHIFT-IN TO SI
DISPLAY DBCSGRP
```
SORT-CONTROL
The SORT-CONTROL special register is the name of an alphanumeric data item.

**Restriction:** The SORT-CONTROL special register is not applicable to sorting a table with the format 2 SORT statement.

The SORT-CONTROL special register has the implicit definition:

```
01 SORT-CONTROL GLOBAL PICTURE X(8) USAGE DISPLAY VALUE "IGZSRTCD".
```

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

This register contains the ddname of the data set that holds the control statements used to improve the performance of a sorting or merging operation.

You can provide a DD statement for the data set identified by the SORT-CONTROL special register. Enterprise COBOL will attempt to open the data set at execution time. Any error will be diagnosed with an informational message.

You can specify the SORT-CONTROL special register in a function wherever an alphanumeric argument is allowed.

The SORT-CONTROL special register is not necessary for a successful sorting or merging operation.

The sort control file takes precedence over the SORT special registers.

SORT-CORE-SIZE
The SORT-CORE-SIZE special register is the name of a binary data item that you can use to specify the number of bytes of storage available to the sort utility.

**Restriction:** The SORT-CORE-SIZE special register is not applicable to sorting a table with the format 2 SORT statement.

The SORT-CORE-SIZE special register has the implicit definition:

```
01 SORT-CORE-SIZE GLOBAL PICTURE S9(8) USAGE BINARY VALUE ZERO.
```

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

SORT-CORE-SIZE can be used in place of the MAINSIZE or RESINV control statements in the sort control file:

- The 'MAINSIZE=' option control statement keyword is equivalent to SORT-CORE-SIZE with a positive value.
- The 'RESINV=' option control statement keyword is equivalent to SORT-CORE-SIZE with a negative value.
- The 'MAINSIZE=MAX' option control statement keyword is equivalent to SORT-CORE-SIZE with a value of +9999999 or +99999999.

You can specify the SORT-CORE-SIZE special register in a function wherever an integer argument is allowed.

SORT-FILE-SIZE
The SORT-FILE-SIZE special register is the name of a binary data item that you can use to specify the estimated number of records in the sort input file, `**file-name-1**`.

**Restriction:** The SORT-FILE-SIZE special register is not applicable to sorting a table with the format 2 SORT statement.
The SORT-FILE-SIZE special register has the implicit definition:

```cobol
01 SORT-FILE-SIZE GLOBAL PICTURE S9(8) USAGE BINARY VALUE ZERO.
```

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

SORT-FILE-SIZE is equivalent to the 'FILSZ=Ennn' control statement in the sort control file.

You can specify the SORT-FILE-SIZE special register in a function wherever an integer argument is allowed.

**SORT-MESSAGE**

The SORT-MESSAGE special register is the name of an alphanumeric data item that is available to both sort and merge programs.

**Restriction:** The SORT-MESSAGE special register is not applicable to sorting a table with the format 2 SORT statement.

The SORT-MESSAGE special register has the implicit definition:

```cobol
01 SORT-MESSAGE GLOBAL PICTURE X(8) USAGE DISPLAY VALUE "SYSOUT".
```

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

You can use the SORT-MESSAGE special register to specify the ddname of a data set that the sort utility should use in place of the SYSOUT data set.

The ddname specified in SORT-MESSAGE is equivalent to the name specified on the 'MSGDDN=' control statement in the sort control file.

You can specify the SORT-MESSAGE special register in a function wherever an alphanumeric argument is allowed.

**SORT-MODE-SIZE**

The SORT-MODE-SIZE special register is the name of a binary data item that you can use to specify the length of variable-length records that occur most frequently.

**Restriction:** The SORT-MODE-SIZE special register is not applicable to sorting a table with the format 2 SORT statement.

The SORT-MODE-SIZE special register has the implicit definition:

```cobol
01 SORT-MODE-SIZE GLOBAL PICTURE S9(5) USAGE BINARY VALUE ZERO.
```

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

SORT-MODE-SIZE is equivalent to the 'SMS=' control statement in the sort control file.

You can specify the SORT-MODE-SIZE special register in a function wherever an integer argument is allowed.

**SORT-RETURN**

The SORT-RETURN special register is the name of a binary data item and is available to both sort and merge programs.

**Restriction:** The SORT-RETURN special register is not applicable to sorting a table with the format 2 SORT statement.
The SORT-RETURN special register has the implicit definition:

```
01 SORT-RETURN GLOBAL PICTURE S9(4) USAGE BINARY VALUE ZERO.
```

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

The SORT-RETURN special register contains a return code of 0 (successful) or 16 (unsuccessful) at the completion of a sort or merge operation. If the sort or merge is unsuccessful and there is no reference to this special register anywhere in the program, a message is displayed on the terminal.

You can set the SORT-RETURN special register to 16 in an error declarative or input/output procedure to terminate a sort or merge operation before all records are processed. The operation is terminated on the next input or output function for the sort or merge operation.

You can specify the SORT-RETURN special register in a function wherever an integer argument is allowed.

**TALLY**

The TALLY special register is the name of a binary data item.

See the following definition of a binary data item:

```
01 TALLY GLOBAL PICTURE 9(5) USAGE BINARY VALUE ZERO.
```

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

You can refer to or modify the contents of TALLY.

You can specify the TALLY special register in a function wherever an integer argument is allowed.

**WHEN-COMPILED**

The WHEN-COMPILED special register contains the date at the start of the compilation.

WHEN-COMPILED is an alphanumeric data item that has the implicit definition:

```
01 WHEN-COMPILED GLOBAL PICTURE X(16) USAGE DISPLAY.
```

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

The WHEN-COMPILED special register has the format:

```
MM/DD/YYhh.mm.ss (MONTH/DAY/YEARhour.minute.second)
```

For example, if compilation began at 2:04 PM on 15 October 2007, WHEN-COMPILED would contain the value 10/15/0714.04.00.

WHEN-COMPILED can be used only as the sending field in a MOVE statement.

WHEN-COMPILED special register data cannot be reference-modified.

The compilation date and time can also be accessed with the intrinsic function WHEN-COMPILED (see “WHEN-COMPILED” on page 516). That function supports four-digit year values and provides additional information.
**XML-CODE**

The XML-CODE special register is used to communicate status between the XML parser and the processing procedure that was identified in an XML PARSE statement, and to indicate either that an XML GENERATE statement executed successfully or that an exception occurred during XML generation.

The XML-CODE special register has the implicit definition:

```cobol
01 XML-CODE PICTURE S9(9) USAGE BINARY VALUE 0.
```

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

When the XML parser encounters an XML event, it sets XML-CODE and then passes control to the processing procedure. For all events except an EXCEPTION event, XML-CODE contains zero when the processing procedure receives control.

For an EXCEPTION event, the parser sets XML-CODE to an exception code that indicates the nature of the exception. XML PARSE exception codes are discussed in *Handling XML PARSE exceptions* in the Enterprise COBOL Programming Guide.

For some XML events, you can set XML-CODE before returning to the parser to control subsequent processing of the document. For details, see *XML-CODE* in the Enterprise COBOL Programming Guide.

When the parser returns control to the XML PARSE statement, XML-CODE contains the most recent value set by the processing procedure or the parser. In some cases, the parser overrides the value set by the processing procedure.

At termination of an XML GENERATE statement, XML-CODE contains either zero, indicating successful completion of XML generation, or a nonzero error code, indicating that an exception occurred during XML generation. XML GENERATE exception codes are detailed in *XML GENERATE exceptions* in the Enterprise COBOL Programming Guide.

**Related concepts**

XML-CODE (Enterprise COBOL Programming Guide)

**Related tasks**

Handling XML PARSE exceptions (Enterprise COBOL Programming Guide)

**Related references**

XML GENERATE exceptions (Enterprise COBOL Programming Guide)

**XML-EVENT**

The XML-EVENT special register communicates event information from the XML parser to the processing procedure identified in the XML PARSE statement.

Before passing control to the processing procedure, the XML parser sets the XML-EVENT special register to the name of the XML event. The specific events and the associated special registers that are set depend on the setting of the XMLPARSE compiler option, XMLPARSE(XMLSS) or XMLPARSE(COMPAT).

The parser uses the following special registers when XMLPARSE(XMLSS) is in effect:

- XML-CODE
- XML-EVENT
- XML-TEXT or XML-NTEXT
- XMLNAMESPACE or XML-NNAMESPACE
- XMLNAMESPACE-PREFIX or XML-NNAMESPACE-PREFIX

The parser uses the following special registers when XMLPARSE(COMPAT) is in effect:

- XML-CODE
- XML-EVENT
The parser sets XML-NTEXT to associated XML text when the XML document is in a national data item, and sets XML-TEXT when the XML document is in an alphanumeric data item. When the XMLPARSE(COMPAT) compiler option is in effect, the parser sets XML-NTEXT to the text of any numeric character reference (for events ATTRIBUTE-NATIONAL-CHARACTER and CONTENT-NATIONAL-CHARACTER) regardless of the type of the XML document data item.

When the XMLPARSE(XMLSS) compiler option is in effect, the parser sets XML-NNAMESPACE and XML-NNAMESPACE-PREFIX when the XML document is in a national data item and when the RETURNING NATIONAL phrase is specified in the XML PARSE statement; otherwise, the parser sets XML-NAMESPACE and XML-NAMESPACE-PREFIX.

Table 3 on page 26 shows XML events and special register contents for parsing with the XMLPARSE(XMLSS) and XMLPARSE(COMPAT) options.

XML-EVENT has the implicit definition:

```
01 XML-EVENT USAGE DISPLAY PICTURE X(30) VALUE SPACE.
```

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

XML-EVENT cannot be used as a receiving data item.

<table>
<thead>
<tr>
<th>XML-EVENT</th>
<th>XMLPARSE(XMLSS)</th>
<th>XMLPARSE(COMPAT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTRIBUTE-CHARACTER</td>
<td>n/a</td>
<td>XML-TEXT or XML-NTEXT contains the single character that corresponds with the predefined entity reference in the attribute value.</td>
</tr>
<tr>
<td>ATTRIBUTE-CHARACTERS</td>
<td>XML-TEXT or XML-NTEXT contains the value within quotation marks or apostrophes. This can be a substring of the attribute value.</td>
<td>XML-TEXT or XML-NTEXT contains the value within quotation marks or apostrophes. This can be a substring of the attribute value if the value includes a character reference or an entity reference.</td>
</tr>
<tr>
<td>ATTRIBUTE-NAME</td>
<td>For attribute names that are not in a namespace, XML-TEXT or XML-NTEXT contains the attribute name.</td>
<td>XML-TEXT or XML-TEXT contains the attribute name (the string to the left of the equal sign).</td>
</tr>
<tr>
<td></td>
<td>For attributes with names in a nondefault namespace, attribute names are always prefixed and have the form: prefix:local-part = &quot;AttValue&quot;.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XML-TEXT or XML-NTEXT contains the local-part, XML-NAMESPACE or XML-NNAMESPACE contains the namespace identifier, and XML-NAMESPACE-PREFIX or XML-NNAMESPACE-PREFIX contains the prefix.</td>
<td></td>
</tr>
<tr>
<td>ATTRIBUTE-NATIONAL-CHARACTER</td>
<td>Regardless of the type of the XML document, XML-TEXT is empty with length zero and XML-NTEXT contains the single national character that corresponds with the numeric character reference.</td>
<td>XML-TEXT or XML-NTEXT content is the same as for XMLPARSE(XMLSS).</td>
</tr>
</tbody>
</table>
### Table 3. XML events and associated special register contents (continued)

<table>
<thead>
<tr>
<th>XML-EVENT</th>
<th>XMLPARSE(XMLSS)¹</th>
<th>XMLPARSE(COMPAT)¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMENT</td>
<td>XML-TEXT or XML-NTEXT contains the text of the comment between the opening character sequence &quot;&lt;!-- &quot; and the closing character sequence &quot;--&gt;&quot;. This can be a substring of the text.</td>
<td>XML-TEXT or XML-NTEXT always contains the complete text of the comment.</td>
</tr>
<tr>
<td>CONTENT-CHARACTER</td>
<td>n/a²</td>
<td>XML-TEXT or XML-NTEXT contains the single character that corresponds with the predefined entity reference in the element content.</td>
</tr>
<tr>
<td>CONTENT-CHARACTERS</td>
<td>XML-TEXT or XML-NTEXT contains the character content of the element between start and end tags. This can be a substring of the content.</td>
<td>XML-TEXT or XML-NTEXT contains the character content of the element between start and end tags. This can be a substring of the character content if the content includes a character reference or an entity reference.</td>
</tr>
<tr>
<td>CONTENT-NATIONAL-CHARACTER</td>
<td>Regardless of the type of the XML document, XML-TEXT is empty with length zero and XML-NTEXT contains the single national character that corresponds with the numeric character reference.</td>
<td>XML-TEXT or XML-NTEXT content is the same as for XMLPARSE(XMLSS).</td>
</tr>
<tr>
<td>DOCUMENT-TYPE-DECLARATION</td>
<td>XML-TEXT or XML-NTEXT contains the name of the root element, as specified in the document type declaration.</td>
<td>XML-TEXT or XML-NTEXT contains the entire document type declaration, including the opening and closing character sequences &quot;&lt;!DOCTYPE&quot; and &quot;&gt;&quot;.</td>
</tr>
<tr>
<td>ENCODING-DECLARATION</td>
<td>XML-TEXT or XML-NTEXT contains the value, between quotation marks or apostrophes, of the encoding declaration in the XML declaration.</td>
<td>XML-TEXT or XML-NTEXT content is the same as for XMLPARSE(XMLSS).</td>
</tr>
<tr>
<td>END-OF-CDATA-SECTION</td>
<td>All XML special registers except XML-CODE and XML-EVENT are empty with length zero.</td>
<td>XML-TEXT or XML-NTEXT contains the string &quot;]]&gt;&quot;.</td>
</tr>
<tr>
<td>END-OF-DOCUMENT</td>
<td>All XML special registers except XML-CODE and XML-EVENT are empty with length zero.</td>
<td>XML-TEXT or XML-NTEXT content is the same as for XMLPARSE(XMLSS).</td>
</tr>
<tr>
<td>END-OF-ELEMENT</td>
<td>XML-TEXT or XML-NTEXT contains the local part of the end element tag or empty element tag name. If the element name is in a nondefault namespace, XML-NAMESPACE or XML-NNAMESPACE contains the namespace identifier. If the element name is in a namespace and is prefixed (of the form prefix:local-part), XML-NAMESPACE-PREFIX or XML-NNAMESPACE-PREFIX contains the prefix.</td>
<td>XML-TEXT or XML-NTEXT contains the name of the end element tag or empty element tag.</td>
</tr>
<tr>
<td>XML-EVENT</td>
<td>XMLPARSE(XMLSS)¹</td>
<td>XMLPARSE(COMPAT)¹</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>END-OF-INPUT</td>
<td>All XML special registers except XML-CODE and XML-EVENT are empty with length zero. To parse an additional segment of an XML document, move the next segment to identifier-1 and set XML-CODE to 1.</td>
<td>n/a⁶</td>
</tr>
<tr>
<td>EXCEPTION</td>
<td>XML-CODE contains the unique return code and reason code that identifies the exception. XML-TEXT or XML-NTEXT contains the document fragment up to the point of the error or anomaly that caused the exception. All other XML special registers are empty with length zero.</td>
<td>XML-CODE contains the unique error code that identifies the exception. XML-TEXT or XML-NTEXT contains the part of the document that was successfully scanned, up to and including the point at which the exception was detected.</td>
</tr>
<tr>
<td>NAMESPACE-DECLARATION</td>
<td>XML-TEXT and XML-NTEXT are both empty with length zero. XML-NAMESPACE or XML-NNAMESPACE contains the declared namespace identifier. If the namespace is &quot;undeclared&quot; by specifying the empty string, XML-NAMESPACE and XML-NNAMESPACE are empty with length zero. XML-NAMESPACE-PREFIX or XML-NNAMESPACE-PREFIX contains the prefix if the namespace declaration is of the form xmlns:prefix = &quot;namespace-identifier&quot;; otherwise, if the declaration is for the default namespace and thus the attribute name is xmlns, XML-NAMESPACE-PREFIX and XML-NNAMESPACE-PREFIX are both empty with length zero.</td>
<td>n/a⁶                      (ATTRIBUTE-NAME and ATTRIBUTE-CHARACTERS events are signaled instead.)</td>
</tr>
<tr>
<td>PROCESSING-INSTRUCTION-DATA</td>
<td>XML-TEXT or XML-NTEXT contains the rest of the processing instruction (after the target name), not including the closing sequence &quot;?&gt;&quot;, but including trailing, and not leading, white space characters. This can be a substring of the processing instruction data.</td>
<td>XML-TEXT or XML-NTEXT always contains the complete processing instruction data.</td>
</tr>
<tr>
<td>PROCESSING-INSTRUCTION-TARGET</td>
<td>XML-TEXT or XML-NTEXT contains the processing instruction target name, which occurs immediately after the processing instruction opening sequence, &quot;&lt;&quot;. This event can occur multiple times for a given processing instruction: one occurrence preceding each substring of the data.</td>
<td>XML-TEXT or XML-NTEXT content is the same as for XMLPARSE(XMLSS). This event occurs only once for a given processing instruction.</td>
</tr>
<tr>
<td>STANDALONE-DECLARATION</td>
<td>XML-TEXT or XML-NTEXT contains the value, between quotation marks or apostrophes (&quot;yes&quot; or &quot;no&quot;), of the stand-alone declaration in the XML declaration.</td>
<td>XML-TEXT or XML-NTEXT content is the same as for XMLPARSE(XMLSS).</td>
</tr>
</tbody>
</table>
Table 3. XML events and associated special register contents (continued)

<table>
<thead>
<tr>
<th>XML-EVENT</th>
<th>XMLPARSE(XMLSS)$^1$</th>
<th>XMLPARSE(COMPAT)$^1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>START-OF-CDATA-SECTION</td>
<td>All XML special registers except XML-CODE and XML-EVENT are empty with length zero.</td>
<td>XML-TEXT or XML-NTEXT contains the string &quot;&lt;![CDATA[&quot;.</td>
</tr>
<tr>
<td>START-OF-DOCUMENT</td>
<td>All XML special registers except XML-CODE and XML-EVENT are empty with length zero.</td>
<td>XML-TEXT or XML-NTEXT contains the entire document.</td>
</tr>
<tr>
<td>START-OF-ELEMENT</td>
<td>XML-TEXT or XML-NTEXT contains the local part of the start element tag name or the local part of the empty element tag name. If the element name is in a namespace, XML-NAMESPACE or XML-NNAMESPACE contains the namespace identifier. If the element name is in a namespace and is prefixed (of the form prefix:local-part, XML-NAMESPACE-PREFIX or XML-NNAMESPACE-PREFIX contains the prefix.</td>
<td>XML-TEXT or XML-NTEXT contains the name of the start element tag or empty element tag, also known as the element type.</td>
</tr>
<tr>
<td>UNKNOWN-REFERENCE-IN-ATTRIBUTE</td>
<td>n/a$^5$</td>
<td>XML-TEXT or XML-NTEXT contains the entity reference name, not including the &quot;&amp;&quot; and &quot;;&quot; delimiters.</td>
</tr>
<tr>
<td></td>
<td>For XMLPARSE(XMLSS), the parser always signals EXCEPTION.</td>
<td></td>
</tr>
<tr>
<td>UNKNOWN-REFERENCE-IN-CONTENT</td>
<td>n/a$^5$</td>
<td>XML-TEXT or XML-NTEXT contains the entity reference name, not including the &quot;&amp;&quot; and &quot;;&quot; delimiters.</td>
</tr>
<tr>
<td></td>
<td>For XMLPARSE(XMLSS), the parser signals UNRESOLVED-REFERENCE or EXCEPTION instead.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See &quot;Unresolved references&quot; below for additional details.</td>
<td></td>
</tr>
<tr>
<td>UNRESOLVED-REFERENCE</td>
<td>XML-TEXT or XML-NTEXT contains the entity name from XML content, not including the &quot;&amp;&quot; and &quot;;&quot; delimiters. See &quot;Unresolved references&quot; below for additional details.</td>
<td>n/a$^6$</td>
</tr>
<tr>
<td></td>
<td>(The parser signals UNKNOWN-REFERENCE-IN-CONTENT instead.)</td>
<td></td>
</tr>
<tr>
<td>VERSION-INFORMATION</td>
<td>XML-TEXT or XML-NTEXT contains the value, between quotation marks or apostrophes, of the version information in the XML declaration.</td>
<td>XML-TEXT or XML-NTEXT content is the same as for XMLPARSE(XMLSS).</td>
</tr>
</tbody>
</table>
Table 3. XML events and associated special register contents (continued)

<table>
<thead>
<tr>
<th>XML-EVENT</th>
<th>XMLPARSE(XMLSS)¹</th>
<th>XMLPARSE(COMPAT)¹</th>
</tr>
</thead>
</table>

1. For all events except EXCEPTION, XML-CODE contains zero. Unless stated otherwise, the namespace XML registers (XML-NAMESPACE, XML-NNAMESPACE, XML-NAMESPACE-PREFIX, and XML-NNAMESPACE-PREFIX) are empty and have length zero.

2. National characters with scalar values greater than 65,535 (NX"FFFF") are represented using two encoding units (a "surrogate pair"). Programmers are responsible for ensuring that operations on the content of XML-NTEXT do not split the pair of encoding units that together form a graphic character, thereby forming invalid data.

3. For XMLPARSE(COMPAT), exceptions for encoding conflicts are signaled before parsing begins. For these exceptions, XML-TEXT or XML-NTEXT is either zero length or contains only the encoding declaration value from the document. See XML PARSE exceptions with XMLPARSE(COMPAT) in effect in the Enterprise COBOL Programming Guide for information about XML exception codes.

4. If an END-OF-INPUT XML event previously occurred and the processing procedure provided a new document segment, XML-TEXT or XML-NTEXT contains only the new segment.

   If the anomaly occurs before parsing begins (for example, the encoding specification is invalid), XML-TEXT or XML-NTEXT are empty with length zero.

   The fragment might or might not include the anomaly. For a duplicate attribute name, for example, the fragment includes the incorrect attribute. For an invalid character, the fragment includes document text up to, but not including, the invalid character.

5. n/a. Not applicable; occurs only with XMLPARSE(COMPAT).

6. n/a. Not applicable; occurs only with XMLPARSE(XMLSS).

**Unresolved References:**

An unresolved entity reference is a reference to the name of an entity that has no declaration in the document type definition (DTD).

The parser signals an UNRESOLVED-REFERENCE event only if all of the following conditions are true:

- The unresolved reference is within element content, not an attribute value.
- The XML document starts with an XML declaration that specifies standalone="no".
- The XML document contains a document type declaration, for example,

```xml
<!DOCTYPE rootElementName>
```

- If the VALIDATING phrase is specified on the XML PARSE statement, the document type declaration must also specify an external DTD subset, for example:

```xml
<!DOCTYPE rootElementName SYSTEM "someOther.dtd">
```

Otherwise the parser signals an EXCEPTION event instead of UNRESOLVED-REFERENCE.

**XML-INFORMATION**

The XML-INFORMATION special register is used to provide additional information to an XML PARSE processing procedure about the status of the parse.

To use XML-INFORMATION, you must compile with the XMLPARSE(XMLSS) compiler option.

The XML-INFORMATION special register has the implicit definition:

```cobol
01 XML-INFORMATION PICTURE S9(9) USAGE BINARY VALUE 0.
```

This register provides a mechanism to easily determine whether an XML EVENT is complete. Sometimes XML content might be split across multiple events and the application must concatenate the pieces of content together. The XML-INFORMATION register is used to indicate whether or not content of the XML event is complete.

The value of the XML-INFORMATION register is set as follows for the various XML events:
• ATTRIBUTE-CHARACTERS
  – 1 indicates that the attribute value in XML-TEXT or XML-NTEXT special register is complete
  – 2 indicates that the attribute value in XML-TEXT or XML-NTEXT special register is not complete
  – 4, 8, 16, ... are reserved for future use

• CONTENT-CHARACTERS
  – 1 indicates that the content value in XML-TEXT or XML-NTEXT special register is complete
  – 2 indicates that the content value in XML-TEXT or XML-NTEXT special register is not complete
  – 4, 8, 16, ... are reserved for future use

• All other events
  – 0 indicates that no additional information is currently available
  – 2, 4, 8, 16, ... are reserved for future use

XML-NAMESPACE
The XML-NAMESPACE special register is defined during XML parsing to contain the identifier of the namespace, if any, associated with the name in XML-TEXT for XML events START-OF-ELEMENT, END-OF-ELEMENT, and ATTRIBUTE-NAME, and to contain the declared namespace identifier for XML event NAMESPACE-DECLARATION.

The parser sets XML-NAMESPACE to the identifier of the namespace associated with a name before transferring control to the processing procedure when the operand of the XML PARSE statement is an alphanumeric data item and the RETURNING NATIONAL phrase is not specified in the XML PARSE statement.

To use XML-NAMESPACE, you must compile with the XMLPARSE(XMLSS) compiler option.

XML-NAMESPACE is an elementary data item of category alphanumeric. The length of XML-NAMESPACE can vary from 0 through 32,768 bytes. The length at run time is the length of the contained namespace identifier.

There is no equivalent COBOL data description entry.

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

XML-NAMESPACE has a length of zero for:
• The START-OF-ELEMENT, END-OF-ELEMENT, and ATTRIBUTE-NAME XML events if there is no namespace associated with a name
• The NAMESPACE-DECLARATION XML event if the namespace is undeclared by specifying the empty string
• All other XML events

When XML-NAMESPACE is set, the XML-NNAMESPACE special register has a length of zero. At any given time, only one of the two special registers XML-NAMESPACE and XML-NNAMESPACE has a nonzero length.

Use the LENGTH function or the LENGTH OF special register to determine the number of bytes that XML-NAMESPACE contains.

XML-NAMESPACE cannot be used as a receiving item.

XML-NNAMESPACE
The XML-NNAMESPACE special register is defined during XML parsing to contain the identifier of the namespace, if any, associated with the name in XML-NTEXT for XML events START-OF-ELEMENT, END-
OF-ELEMENT, and ATTRIBUTE-NAME, and to contain the declared namespace identifier for XML event NAMESPACE-DECLARATION.

The parser sets XML-NNAMESPACE to the identifier of the namespace associated with a name before transferring control to the processing procedure when the RETURNING NATIONAL phrase is specified in the XML PARSE statement or the operand of the XML PARSE statement is a national data item.

To use XML-NNAMESPACE, you must compile with the XMLPARSE(XMLSS) compiler option.
XML-NNAMESPACE is an elementary data item of category national. The length of XML-NNAMESPACE can vary from 0 through 16,384 national characters (0 through 32,768 bytes). The length at run time is the length of the contained namespace identifier.

There is no equivalent COBOL data description entry.

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

XML-NNAMESPACE has a length of zero for:

- The START-OF-ELEMENT, END-OF-ELEMENT, and ATTRIBUTE-NAME XML events, if there is no namespace associated with a name
- The NAMESPACE-DECLARATION XML event if the namespace is undeclared by specifying the empty string
- All other XML events

When XML-NNAMESPACE is set, the XML-NAMESPACE special register has a length of zero. At any given time, only one of the two special registers XML-NNAMESPACE and XML-NAMESPACE has a nonzero length.

Use the LENGTH function to determine the number of national character positions that XML-NNAMESPACE contains; use the LENGTH OF special register to determine the number of bytes.

XML-NNAMESPACE cannot be used as a receiving item.

**XML-NAMESPACE-PREFIX**

The XML-NAMESPACE-PREFIX special register is defined during XML parsing to contain the prefix, if any, of the name in XML-TEXT for XML events START-OF-ELEMENT, END-OF-ELEMENT, and ATTRIBUTE-NAME, and to contain the local attribute name for XML event NAMESPACE-DECLARATION.

The namespace prefix is used as an alias for the complete namespace identifier.

The parser sets XML-NAMESPACE-PREFIX before transferring control to the processing procedure when the operand of the XML PARSE statement is an alphanumeric data item and the RETURNING NATIONAL phrase is not specified.

To use XML-NAMESPACE-PREFIX, you must compile with the XMLPARSE(XMLSS) compiler option.
XML-NAMESPACE-PREFIX is an elementary data item of category national. The length of XML-NAMESPACE-PREFIX can vary from 0 through 4,096 bytes. The length at run time is the length of the contained namespace prefix.

There is no equivalent COBOL data description entry.

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

XML-NAMESPACE-PREFIX has a length of zero for:

- The START-OF-ELEMENT, END-OF-ELEMENT, and ATTRIBUTE-NAME XML events if the name does not have a prefix
- The NAMESPACE-DECLARATION XML event if the declaration is for the default namespace, in which case the namespace declaration attribute name is not prefixed.
- All other XML events
When XML-NAMESPACE-PREFIX is set, the XML-NNAMESPACE-PREFIX special register has a length of zero. At any given time, only one of the two special registers XML-NAMESPACE-PREFIX and XML-NNAMESPACE-PREFIX has a nonzero length.

Use the LENGTH function or the LENGTH OF special register to determine the number of bytes that XML-NAMESPACE-PREFIX contains.

XML-NAMESPACE-PREFIX cannot be used as a receiving item.

**XML-NNAMESPACE-PREFIX**

The XML-NNAMESPACE-PREFIX special register is defined during XML parsing to contain the prefix, if any, of the name in XML-NTEXT for XML events START-OF-ELEMENT, END-OF-ELEMENT, and ATTRIBUTE-NAME, and to contain the local attribute name for XML event NAMESPACE-DECLARATION.

The namespace prefix is used as an alias for the complete namespace identifier.

The parser sets XML-NNAMESPACE-PREFIX before transferring control to the processing procedure when the operand of the XML PARSE statement is a national data item or the RETURNING NATIONAL phrase is specified in the XML PARSE statement.

To use XML-NNAMESPACE-PREFIX, you must compile with the XMLPARSE(XMLSS) compiler option.

XML-NNAMESPACE-PREFIX is an elementary data item of category national. The length of XML-NNAMESPACE-PREFIX can vary from 0 through 2048 national character positions (0 through 4096 bytes). The length at run time is the length of the contained namespace prefix.

There is no equivalent COBOL data description entry.

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

XML-NNAMESPACE-PREFIX has a length of zero for:

- The START-OF-ELEMENT, END-OF-ELEMENT, and ATTRIBUTE-NAME XML events if the name does not have a prefix
- NAMESPACE-DECLARATION XML event if the declaration is for the default namespace, in which case the namespace declaration attribute name is not prefixed.
- All other XML events

When XML-NNAMESPACE-PREFIX is set, the XML-NAMESPACE-PREFIX special register has a length of zero. At any given time, only one of the two special registers XML-NNAMESPACE-PREFIX and XML-NAMESPACE-PREFIX has a nonzero length.

Use the LENGTH function to determine the number of national character positions that XML-NNAMESPACE contains; use the LENGTH OF special register to determine the number of bytes.

XML-NNAMESPACE-PREFIX cannot be used as a receiving item.

**XML-NTEXT**

The XML-NTEXT special register is defined during XML parsing to contain document fragments that are represented in usage NATIONAL.

XML-NTEXT is an elementary data item of category national of the length of the contained XML document fragment. The length of XML-NTEXT can vary from 0 through 67,090,431 *national character positions*. The maximum byte length is 134,180,862.

There is no equivalent COBOL data description entry.

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

The parser sets XML-NTEXT to the document fragment associated with an event before transferring control to the processing procedure in these cases:
• When the operand of the XML PARSE statement is a data item of category national or the RETURNING NATIONAL phrase is specified in the XML PARSE statement
• For the ATTRIBUTE-NATIONAL-CHARACTER event
• For the CONTENT-NATIONAL-CHARACTER event

When XML-NTEXT is set, the XML-TEXT special register has a length of zero. At any given time, only one of the two special registers XML-NTEXT and XML-TEXT has a nonzero length.

Use the LENGTH function to determine the number of national characters that XML-NTEXT contains. Use the LENGTH OF special register to determine the number of bytes, rather than the number of national characters, that XML-NTEXT contains.

XML-NTEXT cannot be used as a receiving item.

**XML-TEXT**

The XML-TEXT special register is defined during XML parsing to contain document fragments that are represented in usage DISPLAY.

XML-TEXT is an elementary data item of category alphanumeric of the length of the contained XML document fragment. The length of XML-TEXT can vary from 0 through 134,180,862 bytes.

There is no equivalent COBOL data description entry.

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

The parser sets XML-TEXT to the document fragment associated with an event before transferring control to the processing procedure when the operand of the XML PARSE statement is an alphanumeric data item and the RETURNING NATIONAL phrase is not specified in the XML PARSE statement, except for the ATTRIBUTE-NATIONAL-CHARACTER event and the CONTENT-NATIONAL-CHARACTER event.

When XML-TEXT is set, the XML-NTEXT special register has a length of zero. At any given time, only one of the two special registers XML-NTEXT and XML-TEXT has a nonzero length.

Use the LENGTH function or the LENGTH OF special register for XML-TEXT to determine the number of bytes that XML-TEXT contains.

XML-TEXT cannot be used as a receiving item.

**Literals**

A literal is a character-string whose value is specified either by the characters of which it is composed or by the use of a figurative constant.

For more information about figurative constants, see “Figurative constants” on page 13.

For descriptions of the different types of literals, see the following topics:

• “Alphanumeric literals” on page 34
• “DBCS literals” on page 38
• “National literals” on page 42
• “Numeric literals” on page 41

**Alphanumeric literals**

Enterprise COBOL provides several formats of alphanumeric literals.

The formats of alphanumeric literals are:

• Format 1: “Basic alphanumeric literals” on page 35
• Format 2: “Alphanumeric literals with DBCS characters” on page 35
• Format 3: “Hexadecimal notation for alphanumeric literals” on page 36
Basic alphanumeric literals

Basic alphanumeric literals can contain any character in a single-byte EBCDIC character set.

The following format is for a basic alphanumeric literal:

<table>
<thead>
<tr>
<th>Format 1: Basic alphanumeric literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;single-byte-characters&quot;</td>
</tr>
<tr>
<td>'single-byte-characters'</td>
</tr>
</tbody>
</table>

The enclosing quotation marks or apostrophes are excluded from the literal when the program is compiled.

An embedded quotation mark or apostrophe must be represented by a pair of quotation marks (" ") or a pair of apostrophes ( ' '), respectively, when it is the character used as the opening delimiter. For example:

| "THIS ISN""T WRONG" |
| "THIS ISN"'T WRONG" |

The delimiter character used as the opening delimiter for a literal must be used as the closing delimiter for that literal. For example:

| 'THIS IS RIGHT' |
| "THIS IS RIGHT" |
| 'THIS IS WRONG' |

You can use apostrophes or quotation marks as the literal delimiters independent of the APOST/QUOTE compiler option.

Any punctuation characters included within an alphanumeric literal are part of the value of the literal.

The maximum length of an alphanumeric literal is 160 bytes. The minimum length is 1 byte.

Alphanumeric literals are in the alphanumeric data class and category. (Data classes and categories are described in “Classes and categories of data” on page 153.)

Alphanumeric literals with DBCS characters

When the DBCS compiler option is in effect, the characters X'0E' and X'0F' in an alphanumeric literal will be recognized as shift codes for DBCS characters. That is, the characters between paired shift codes will be recognized as DBCS characters. Unlike an alphanumeric literal compiled under the NODBCS option, additional syntax rules apply to DBCS characters in an alphanumeric literal.

Alphanumeric literals with DBCS characters have the following format:

<table>
<thead>
<tr>
<th>Format 2: Alphanumeric literals with DBCS characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;mixed-SBCS-and-DBCS-characters&quot;</td>
</tr>
<tr>
<td>'mixed-SBCS-and-DBCS-characters'</td>
</tr>
</tbody>
</table>

" or '

The opening and closing delimiter. The closing delimiter must match the opening delimiter.

mixed-SBCS-and-DBCS-characters

Any mix of single-byte and DBCS characters.
Shift-out and shift-in control characters are part of the literal and must be paired. They must contain zero or an even number of intervening bytes.

Nested shift codes are not allowed in the DBCS portion of the literal.

The syntax rules for single-byte characters in the literal follow the rules for basic alphanumeric literals. The syntax rules for DBCS characters in the literal follow the rules for DBCS literals.

The move and comparison rules for alphanumeric literals with DBCS characters are the same as those for any alphanumeric literal.

The length of an alphanumeric literal with DBCS characters is its byte length, including the shift control characters. The maximum length is limited by the available space on one line in Area B. An alphanumeric literal with DBCS characters cannot be continued.

An alphanumeric literal with DBCS characters is of the alphanumeric category.

Alphanumeric literals with DBCS characters cannot be used:

- As a literal in the following cases:
  - ALPHABET clause
  - ASSIGN clause
  - CALL statement `program-ID`
  - CANCEL statement
  - CLASS clause
  - CURRENCY SIGN clause
  - END PROGRAM marker
  - ENTRY statement
  - PADDING CHARACTER clause
  - PROGRAM-ID paragraph
  - RERUN clause
  - STOP statement
  - XML-SCHEMA clause
- As the external class-name for an object-oriented class
- As the basis-name in a BASIS statement
- As the text-name in a COPY statement
- As the library-name in a COPY statement

Enterprise COBOL statements process alphanumeric literals with DBCS characters without sensitivity to the shift codes and character codes. The use of statements that operate on a byte-to-byte basis (for example, STRING and UNSTRING) can result in strings that are not valid mixtures of single-byte EBCDIC and DBCS characters. See Processing alphanumeric data items that contain DBCS data in the Enterprise COBOL Programming Guide for more information about using alphanumeric literals and data items with DBCS characters in statements that operate on a byte-by-byte basis.

**Hexadecimal notation for alphanumeric literals**

Hexadecimal notation can be used for alphanumeric literals.

Hexadecimal notation has the following format:

### Format 3: Hexadecimal notation for alphanumeric literals

```
X"hexadecimal-digits"
X'hexadecimal-digits'
```
X" or X'
    The opening delimiter for the hexadecimal notation of an alphanumeric literal.

" or '
    The closing delimiter for the hexadecimal notation of an alphanumeric literal. If a quotation mark is
    used in the opening delimiter, a quotation mark must be used as the closing delimiter. Similarly, if an
    apostrophe is used in the opening delimiter, an apostrophe must be used as the closing delimiter.

Hexadecimal digits are characters in the range '0' to '9', 'a' to 'f', and 'A' to 'F', inclusive. Two hexadecimal
digits represent one character in a single-byte character set (EBCDIC or ASCII). Four hexadecimal digits
represent one character in a DBCS character set. A string of EBCDIC DBCS characters represented in
hexadecimal notation must be preceded by the hexadecimal representation of a shift-out control
character (X'0E') and followed by the hexadecimal representation of a shift-in control character (X'0F').
An even number of hexadecimal digits must be specified. The maximum length of a hexadecimal literal is
320 hexadecimal digits.

The continuation rules are the same as those for any alphanumeric literal. The opening delimiter (X" or
X’) cannot be split across lines.

The DBCS compiler option has no effect on the processing of hexadecimal notation of alphanumeric
literals.

An alphanumeric literal in hexadecimal notation has data class and category alphanumeric. Hexadecimal
notation for alphanumeric literals can be used anywhere alphanumeric literals can be used.

See also “Hexadecimal notation for national literals” on page 43.

Null-terminated alphanumeric literals
Alphanumeric literals can be null-terminated.

The format for null-terminated alphanumeric literals is:

```
Z"mixed-characters"
Z'mixed-characters'
```

Z" or Z'
    The opening delimiter for a null-terminated alphanumeric literal. Both characters of the opening
delimiter (Z" or Z’) must be on the same source line.

" or '
    The closing delimiter for a null-terminated alphanumeric literal.

    If a quotation mark is used in the opening delimiter, a quotation mark must be used as the closing
delimiter. Similarly, if an apostrophe is used in the opening delimiter, an apostrophe must be used as
the closing delimiter.

mixed-characters
    Can be any of the following characters:
    • Solely single-byte characters
    • Mixed single-byte and DBCS characters
    • Solely DBCS characters

    However, you cannot specify the single-byte character with the value X'00'. X'00' is the null character
automatically appended to the end of the literal. The content of the literal is otherwise subject to the
same rules and restrictions as an alphanumeric literal with DBCS characters (format 2).

The length of the string of characters in the literal content can be 0 to 159 bytes. The actual length of the
literal includes the terminating null character, and is a maximum of 160 bytes.

Chapter 3. Character-strings: COBOL words and literals 37
A null-terminated alphanumeric literal has data class and category alphanumeric. It can be used anywhere an alphanumeric literal can be used except that null-terminated literals are not supported in ALL literal figurative constants.

The LENGTH intrinsic function, when applied to a null-terminated alphanumeric literal, returns the number of bytes in the literal prior to but not including the terminating null. (The LENGTH special register does not support literal operands.)

**DBCS literals**

The formats and rules for DBCS literals are listed in this section.

### Format for DBCS literals

<table>
<thead>
<tr>
<th>Format for DBCS literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>G&quot;&lt;DBCS-characters&gt;&quot;</td>
</tr>
<tr>
<td>G'&lt;DBCS-characters&gt;'</td>
</tr>
<tr>
<td>N&quot;&lt;DBCS-characters&gt;&quot;</td>
</tr>
<tr>
<td>N'&lt;DBCS-characters&gt;'</td>
</tr>
</tbody>
</table>

**G", G', N", or N'**

Opening delimiters.

N" and N' identify a DBCS literal when the NSYMBOL(DBCS) compiler option is in effect. They identify a national literal when the NSYMBOL(NATIONAL) compiler option is in effect, and the rules specified in “National literals” on page 42 apply.

The opening delimiter must be followed immediately by a shift-out control character.

For literals with opening delimiter N" or N', when embedded quotes or apostrophes are specified as part of DBCS characters in a DBCS literal, a single embedded DBCS quote or apostrophe is represented by two DBCS quotes or apostrophes. If a single embedded DBCS quote or apostrophe is found, an E-level compiler message will be issued and a second embedded DBCS quote or apostrophe will be assumed.

<

Represents the shift-out control character (X'0E')

>

Represents the shift-in control character (X'0F')

" or '

The closing delimiter. If a quotation mark is used in the opening delimiter, a quotation mark must be used as the closing delimiter. Similarly, if an apostrophe is used in the opening delimiter, an apostrophe must be used as the closing delimiter.

The closing delimiter must appear immediately after the shift-in control character.

**DBCS-characters**

*DBCS-characters* can be one or more characters in the range of X'00' through X'FF' for either byte. Any value will be accepted in the content of the literal, although whether it is a valid value at run time depends on the CCSID in effect for the CODEPAGE compiler option.

**Maximum length**

28 characters

**Continuation rules**

Cannot be continued across lines

**Where DBCS literals can be used**

DBCS literals can be used in the following places:

- DATA DIVISION
  - In the VALUE clause of data description entries that define a data item of class DBCS.
– In the VALUE OF clause of file description entries.

• PROCEDURE DIVISION
– In a relation condition when the comparand is a DBCS data item, an elementary data item of class
  national, a national group item, or an alphanumeric group item
– As an argument passed BY CONTENT in a CALL statement
– In the DISPLAY and EVALUATE statements
– In the following statements:
  - INITIALIZE; for details, see “INITIALIZE statement” on page 324.
  - INSPECT; for details, see “INSPECT statement” on page 327.
  - MOVE; for details, see “MOVE statement” on page 363.
  - STRING; for details, see “STRING statement” on page 418.
  - UNSTRING, for details, see “UNSTRING statement” on page 426.
– In figurative constant ALL
– As an argument to the NATIONAL-OF intrinsic function

• Compiler-directing statements COPY, REPLACE, and TITLE

UTF-8 literals
The UTF-8 literal formats that Enterprise COBOL provides are basic UTF-8 literals and hexadecimal
notation for UTF-8 literals.

Basic UTF-8 literals
The format and rules for basic UTF-8 literals are listed in this section.

<table>
<thead>
<tr>
<th>Format 1: Basic UTF-8 literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>U&quot;character-data&quot;</td>
</tr>
<tr>
<td>U'character-data'</td>
</tr>
</tbody>
</table>

U" or U'
Opening delimiters. The opening delimiter must be coded as single-byte characters. It cannot be split
across lines.

" or '
The closing delimiter. The closing delimiter must be coded as a single-byte character. If a quotation
mark is used in the opening delimiter, it must be used as the closing delimiter. Similarly, if an
apostrophe is used in the opening delimiter, it must be used as the closing delimiter.

To include the quotation mark or apostrophe used in the opening delimiter in the content of the literal,
specify a pair of quotation marks or apostrophes, respectively. For example:

U'This literal''s content includes an apostrophe ';
U'This literal includes ", which is not used in the opening delimiter '\';
U"This literal includes "", which is used in the opening delimiter ".

character-data
The source text representation of the content of the UTF-8 literal. character-data can include any
combination of EBCDIC single-byte characters and double-byte characters encoded in the Coded
Character Set ID (CCSID) specified by the CODEPAGE compiler option.

DBCS characters in the content of the literal must be delimited by shift-out and shift-in control
characters.

Character-data can contain the following Unicode escape sequences:
• \uhhhh, where each \( h \) represents a hexadecimal digit in the range '0' to '9', 'a' to 'f', and 'A' to 'F', inclusive. This Unicode escape sequence represents a Unicode code point from the Basic Multilingual Plane (i.e., Unicode code points in the range U+0000 through U+FFFF).

• \U00hhhhhh, where each \( h \) represents a hexadecimal digit in the range '0' to '9', 'a' to 'f', and 'A' to 'F'. This Unicode escape sequence can represent any legal Unicode code point, including code points from the Supplementary Planes, specifically, Unicode code points in the range U+10000 through U+10FFFF (e.g., an emoji symbol).

Note:

1. Code points U+D800 through U+DFFF are reserved for the high and low halves of surrogate pairs used by UTF-16. There is no legal encoding of these Unicode code points in UTF-8 and hence \( \textbf{\textit{\textbackslash u}}\textbf{D}8\textbf{0}0 \) through \( \textbf{\textit{\textbackslash u}}\textbf{D}FFFF \) and \( \textbf{\textit{\textbackslash U0000D}8\textbf{0}0} \) through \( \textbf{\textit{\textbackslash U0000D}FFFF} \) cannot be specified as Unicode escape sequences in UTF-8 literals.

2. To avoid having a string of characters of the form \( \textbf{\textit{\textbackslash u}}\textbf{h}hhhh \) or \( \textbf{\textit{\textbackslash U000hh}hhhh} \) in a UTF-8 literal be interpreted as a Unicode escape sequence, the escape character \( \textbf{\textit{\textbackslash \textbackslash}} \) (\( \textbf{\textbackslash \textbackslash} \)) can itself be escaped with \( \textbf{\textit{\textbackslash \textbackslash \textbackslash}} \) to cause it to be interpreted literally. Thus, the sequence \( \textbf{\textit{\textbackslash \textbackslash uO0E9}} \) will not be treated as a Unicode escape sequence.

Wherever a Unicode escape sequence appears in a basic UTF-8 literal, it is replaced by the compiler with the corresponding UTF-8 encoding of the Unicode code point, which makes it convenient to represent general Unicode code points in the literal using only EBCDIC characters. For example, \( \textbf{u \textit{\textbackslash \textbackslash u00E9}} \) represents the string 'café'.

Maximum length
The maximum number of UTF-8 characters that can be represented in a basic UTF-8 literal varies depending on the size (1 to 4 bytes) of each UTF-8 character being represented. However, a maximum of 160 bytes after conversion of the literal characters form the EBCDIC codepage to UTF-8 is allowed before truncation occurs. Truncation will be performed on a character boundary.

If the source content of the literal contains one or more DBCS characters, the maximum length is limited by the available space in Area B of a single source line.

The literal must contain at least one character. Each single-byte character in the literal counts as one character position and each DBCS character in the literal counts as one character position. Shift-in and shift-out delimiters for DBCS characters are not counted.

Continuation rules
When the content of the literal includes DBCS characters, the literal cannot be continued.

When the content of the literal does not include DBCS characters, normal continuation rules apply.

The source text representation of \( \text{character-data} \) is automatically converted to UTF-8 for use at run time. For example, when the literal is moved to or compared with a data item of category UTF-8, it is automatically converted to UTF-8.

Hexadecimal notation for UTF-8 literals
The format and rules for the hexadecimal notation format of UTF-8 literals are listed in this section.

<table>
<thead>
<tr>
<th>Format 2: Hexadecimal notation for UTF-8 literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>UX&quot;hexadecimal-digits&quot;</td>
</tr>
<tr>
<td>UX'hexadecimal-digits'</td>
</tr>
</tbody>
</table>

UX" or UX'
Opening delimiters. The opening delimiter must be represented in single-byte characters. It must not be split across lines.

" or '
The closing delimiter. The closing delimiter must be coded as a single-byte character.
If a quotation mark is used in the opening delimiter, it must be used as the closing delimiter. Similarly, if an apostrophe is used in the opening delimiter, it must be used as the closing delimiter.

**Hexadecimal-digits**

Hexadecimal digits in the range '0' to '9', 'a' - 'f', and 'A' to 'F', inclusive. Each group of two hexadecimal digits represents a single encoding unit of a UTF-8 character.

**Maximum length**

The length of a UTF-8 literal in hexadecimal notation must be from two to 320 hexadecimal digits, excluding the opening and closing delimiters.

**Continuation rules**

Normal continuation rules apply.

The sequence of bytes represented by hexadecimal-digits is validated to ensure that it contains a legal sequence of UTF-8 bytes.

The content of a UTF-8 literal in hexadecimal notation is stored as UTF-8 characters. The resulting content has the same meaning as a basic UTF-8 literal that specifies the same UTF-8 characters.

A UTF-8 literal in hexadecimal notation has data class and category UTF-8 and can be used anywhere that a basic UTF-8 literal can be used.

**Numeric literals**

A numeric literal is a character-string whose characters are selected from the digits 0 through 9, a sign character (+ or -), and the decimal point.

If the literal contains no decimal point, it is an integer. (In this documentation, the word integer appearing in a format represents a numeric literal of nonzero value that contains no sign and no decimal point, except when other rules are included with the description of the format.) The following rules apply:

- If the ARITH(COMPAT) compiler option is in effect, one through 18 digits are allowed. If the ARITH(EXTEND) compiler option is in effect, one through 31 digits are allowed.
- Only one sign character is allowed. If included, it must be the leftmost character of the literal. If the literal is unsigned, it is a positive value.
- Only one decimal point is allowed. If a decimal point is included, it is treated as an assumed decimal point (that is, as not taking up a character position in the literal). The decimal point can appear anywhere within the literal except as the rightmost character.

The value of a numeric literal is the algebraic quantity expressed by the characters in the literal. The size of a numeric literal is equal to the number of digits specified by the user.

Numeric literals can be fixed-point or floating-point numbers.

Numeric literals are in the numeric data class and category. (Data classes and categories are described under “Classes and categories of data” on page 153.)

**Rules for floating-point literal values**

The format and rules for floating-point literals are listed below.

**Format**

```
+  mantissa E  exponent +
```

- The sign is optional before the mantissa and the exponent; if you omit the sign, the compiler assumes a positive number.
- The mantissa can contain between one and 16 digits. A decimal point must be included in the mantissa.
- The exponent is represented by an E followed by an optional sign and one or two digits.
The magnitude of a floating-point literal value must fall between 0.54E-78 and 0.72E+76. For values outside of this range, an E-level diagnostic message is produced and the value is replaced by either 0 or 0.72E+76, respectively.

National literals
The national literal formats that Enterprise COBOL provides are basic national literals and hexadecimal notation for national literals.

For more information about the formats, see “Basic national literals” on page 42 and “Hexadecimal notation for national literals” on page 43.

Basic national literals
The format and rules for basic national literals are listed in this section.

### Format 1: Basic national literals

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N&quot;character-data&quot;</td>
<td>Opening delimiter. The opening delimiter must be coded as single-byte characters. It cannot be split across lines.</td>
</tr>
<tr>
<td>N'character-data'</td>
<td>The closing delimiter. The closing delimiter must be coded as a single-byte character. If a quotation mark is used in the opening delimiter, it must be used as the closing delimiter. Similarly, if an apostrophe is used in the opening delimiter, it must be used as the closing delimiter. To include the quotation mark or apostrophe used in the opening delimiter in the content of the literal, specify a pair of quotation marks or apostrophes, respectively. Examples:</td>
</tr>
</tbody>
</table>

- N"This literal's content includes an apostrophe" |
- N'This literal includes '', which is not used in the opening delimiter" |
- N"This literal includes '', which is used in the opening delimiter" |

**character-data**
The source text representation of the content of the national literal. character-data can include any combination of EBCDIC single-byte characters and double-byte characters encoded in the Coded Character Set ID (CCSID) specified by the CODEPAGE compiler option.

DBCS characters in the content of the literal must be delimited by shift-out and shift-in control characters.

**Maximum length**
The maximum length of a national literal is 80 character positions, excluding the opening and closing delimiters. If the source content of the literal contains one or more DBCS characters, the maximum length is limited by the available space in Area B of a single source line.

The literal must contain at least one character. Each single-byte character in the literal counts as one character position and each DBCS character in the literal counts as one character position. Shift-in and shift-out delimiters for DBCS characters are not counted.

**Continuation rules**
When the content of the literal includes DBCS characters, the literal cannot be continued. When the content of the literal does not include DBCS characters, normal continuation rules apply.
The source text representation of character-data is automatically converted to UTF-16 for use at run time (for example, when the literal is moved to or compared with a data item of category national).

**Hexadecimal notation for national literals**
The format and rules for the hexadecimal notation format of national literals are listed in this section.

<table>
<thead>
<tr>
<th>Format 2: Hexadecimal notation for national literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{NX&quot;hexadecimal-digits&quot;}</td>
</tr>
<tr>
<td>\texttt{NX'hexadecimal-digits'}</td>
</tr>
</tbody>
</table>

The hexadecimal notation format of national literals is not affected by the NSYMBOL compiler option.

\texttt{NX"} or \texttt{NX'}

Opening delimiters. The opening delimiter must be represented in single-byte characters. It must not be split across lines.

\texttt{"} or \texttt{'}

The closing delimiter. The closing delimiter must be represented as a single-byte character.

If a quotation mark is used in the opening delimiter, a quotation mark must be used as the closing delimiter. Similarly, if an apostrophe is used in the opening delimiter, an apostrophe must be used as the closing delimiter.

**hexadecimal-digits**

Hexadecimal digits in the range '0' to '9', 'a' - 'f', and 'A' to 'F', inclusive. Each group of four hexadecimal digits represents a single national character and must represent a valid code point in UTF-16. The number of hexadecimal digits must be a multiple of four.

**Maximum length**

The length of a national literal in hexadecimal notation must be from four to 320 hexadecimal digits, excluding the opening and closing delimiters. The length must be a multiple of four.

**Continuation rules**

Normal continuation rules apply.

The content of a national literal in hexadecimal notation is stored as national characters. The resulting content has the same meaning as a basic national literal that specifies the same national characters.

A national literal in hexadecimal notation has data class and category national and can be used anywhere that a basic national literal can be used.

**Where national literals can be used**

National literals can be used in multiple ways.

National literals can be used:

- In a VALUE clause associated with a data item of class national or a VALUE clause associated with a condition-name for a conditional variable that is defined with usage NATIONAL
- In figurative constant ALL
- In a relation condition
- In the WHEN phrase of a format-2 SEARCH statement (binary search)
- In the ALL, LEADING, or FIRST phrase of an INSPECT statement
- In the BEFORE or AFTER phrase of an INSPECT statement
- In the DELIMITED BY phrase of a STRING statement
- In the DELIMITED BY phrase of an UNSTRING statement
- As the method-name in a METHOD-ID paragraph, an END METHOD marker, and an INVOKE statement
- As an argument passed BY CONTENT in the CALL statement
- As an argument passed BY VALUE in an INVOKE or CALL statement

Chapter 3. Character-strings: COBOL words and literals 43
• In the DISPLAY and EVALUATE statements
• As a sending item in the following procedural statements:
  – INITIALIZE
  – INSPECT
  – MOVE
  – STRING
  – UNSTRING
• In the argument list to the following intrinsic functions:
  DISPLAY-OF, LENGTH, LOWER-CASE, MAX, MIN, ORD-MAX, ORD-MIN, REVERSE, UPPER-CASE, USUPPLEMENTARY and UVALID
  **Note:** DBCS literals can't be used in the USUPPLEMENTARY and UVALID functions.
• In the compiler-directing statements COPY, REPLACE, and TITLE

A national literal can be used only as specified in the detailed rules in this document.

**PICTURE character-strings**

A **PICTURE character-string** is composed of the currency symbol and certain combinations of characters in the COBOL character set. PICTURE character-strings are delimited only by the separator space, separator comma, separator semicolon, or separator period.

A chart of PICTURE clause symbols appears in Table 12 on page 190.

**Comments**

A **comment** is a character-string that can contain any combination of characters from the character set of the computer.

It has no effect on the execution of the program. There are three forms of comments:

**Comment entry (IDENTIFICATION DIVISION)**

This form is described under “Optional paragraphs” on page 100.

**Comment line (any division)**

This form is described under “Comment lines” on page 56.

**Inline comments (any division)**

An inline comment is identified by a floating comment indicator (*>) preceded by one or more character-strings in the program-text area, and can be written on any line of a compilation group. All characters that follow the floating comment indicator up to the end of area B are comment text.

Character-strings that form comments can contain DBCS characters or a combination of DBCS and single-byte EBCDIC characters.

Multiple comment lines that contain DBCS strings are allowed. The embedding of DBCS characters in a comment line must be done on a line-by-line basis. Words containing those characters cannot be continued to a following line. No syntax checking for valid strings is provided in comment lines.
Chapter 4. Separators

A separator is a character or a string of two or more contiguous characters that delimits character-strings. The separators are shown in the following table.

<table>
<thead>
<tr>
<th>Separator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>b</code></td>
<td>Space</td>
</tr>
<tr>
<td><code>,b</code></td>
<td>Comma</td>
</tr>
<tr>
<td><code>.b</code></td>
<td>Period</td>
</tr>
<tr>
<td><code>:b</code></td>
<td>Semicolon</td>
</tr>
<tr>
<td><code>(</code></td>
<td>Left parenthesis</td>
</tr>
<tr>
<td><code>)</code></td>
<td>Right parenthesis</td>
</tr>
<tr>
<td><code>:</code></td>
<td>Colon</td>
</tr>
<tr>
<td><code>&quot;b</code></td>
<td>Quotation mark</td>
</tr>
<tr>
<td><code>'b</code></td>
<td>Apostrophe</td>
</tr>
<tr>
<td><code>U&quot;</code></td>
<td>Opening delimiter for UTF-8 literal</td>
</tr>
<tr>
<td><code>U'</code></td>
<td>Opening delimiter for UTF-8 literal</td>
</tr>
<tr>
<td><code>UX&quot;</code></td>
<td>Opening delimiter for a hexadecimal format UTF-8 literal</td>
</tr>
<tr>
<td><code>UX'</code></td>
<td>Opening delimiter for a hexadecimal format UTF-8 literal</td>
</tr>
<tr>
<td><code>X&quot;</code></td>
<td>Opening delimiter for a hexadecimal format alphanumeric literal</td>
</tr>
<tr>
<td><code>X'</code></td>
<td>Opening delimiter for a hexadecimal format alphanumeric literal</td>
</tr>
<tr>
<td><code>Z&quot;</code></td>
<td>Opening delimiter for a null-terminated alphanumeric literal</td>
</tr>
<tr>
<td><code>Z'</code></td>
<td>Opening delimiter for a null-terminated alphanumeric literal</td>
</tr>
<tr>
<td><code>N&quot;</code></td>
<td>Opening delimiter for a national literal</td>
</tr>
<tr>
<td><code>N'</code></td>
<td>Opening delimiter for a national literal</td>
</tr>
<tr>
<td><code>NX&quot;</code></td>
<td>Opening delimiter for a hexadecimal format national literal</td>
</tr>
<tr>
<td><code>NX'</code></td>
<td>Opening delimiter for a hexadecimal format national literal</td>
</tr>
<tr>
<td><code>G&quot;</code></td>
<td>Opening delimiter for a DBCS literal</td>
</tr>
<tr>
<td><code>G'</code></td>
<td>Opening delimiter for a DBCS literal</td>
</tr>
<tr>
<td><code>==</code></td>
<td>Pseudo-text delimiter</td>
</tr>
</tbody>
</table>

1. `b` represents a blank.
2. `N"` and `N'` are the opening delimiter for a DBCS literal when the NSYMBOL(DBCS) compiler option is in effect.
Rules for separators

A separator is a string of one or more punctuation characters.

In the following description, {} (curly braces) enclose each separator, and \( b \) represents a space. Anywhere a space is used as a separator or as part of a separator, more than one space can be used.

**Space \{b\}**

A space can immediately precede or follow any separator except:

- The opening pseudo-text delimiter, where the preceding space is required.
- Within quotation marks. Spaces between quotation marks are considered part of the alphanumeric literal; they are not considered separators.

**Period \{.\}, Comma \{,\}, Semicolon \{;\}**

A separator comma is composed of a comma followed by a space. A separator period is composed of a period followed by a space. A separator semicolon is composed of a semicolon followed by a space.

The separator period must be used only to indicate the end of a sentence, or as shown in formats. The separator comma and separator semicolon can be used anywhere the separator space is used.

- In the IDENTIFICATION DIVISION, each paragraph must end with a separator period.
- In the ENVIRONMENT DIVISION, the SOURCE-COMPUTER, OBJECT-COMPUTER, SPECIAL-NAMES, and I-O-CONTROL paragraphs must each end with a separator period. In the FILE-CONTROL paragraph, each file-control entry must end with a separator period.
- In the DATA DIVISION, file (FD), sort/merge file (SD), and data description entries must each end with a separator period.
- In the PROCEDURE DIVISION, separator commas or separator semicolons can separate statements within a sentence and operands within a statement. Each sentence and each procedure must end with a separator period.

**Parentheses \{()\}**

Except in pseudo-text, parentheses can appear only in balanced pairs of left and right parentheses. They delimit subscripts, a list of function arguments, reference-modifiers, arithmetic expressions, or conditions.

**Colon \{:\}**

The colon is a separator and is required when shown in general formats.

**Quotation marks \{"\}, \{'\}**

An opening quotation mark must be immediately preceded by a space or a left parenthesis. A closing quotation mark must be immediately followed by a separator space, comma, semicolon, period, right parenthesis, or pseudo-text delimiter. Quotation marks must appear as balanced pairs. They delimit alphanumeric literals, except when the literal is continued (see “Continuation lines” on page 54).

**Apostrophes \{‘\}, \{‘\}**

An opening apostrophe must be immediately preceded by a space or a left parenthesis. A closing apostrophe must be immediately followed by a separator space, comma, semicolon, period, right parenthesis, or pseudo-text delimiter. Apostrophes must appear as balanced pairs. They delimit alphanumeric literals, except when the literal is continued (see “Continuation lines” on page 54).

**Null-terminated literal delimiters \{Z"\}, \{Z’\}**

The opening delimiter must be immediately preceded by a space or a left parenthesis. The closing delimiter must be immediately followed by a separator space, comma, semicolon, period, right parenthesis, or pseudo-text delimiter.

**DBCS literal delimiters \{G"\}, \{G’\}, \{N"\}, \{N’\}**

The opening delimiter must be immediately preceded by a space or a left parenthesis. The closing delimiter must be immediately followed by a separator space, comma, semicolon, period, right parenthesis, or pseudo-text delimiter. \( N'' \) and \( N' \) are DBCS literal delimiters when the NSYMBOL(DBCS) compiler option is in effect.
UTF-8 literal delimiters \{U\} ... \{
\}, \{U\'} ... \{'\}, \{UX\} ... \{"\}, \{UX\'} ... \{'\}

The opening delimiter must be immediately preceded by a space or a left parenthesis. The closing delimiter must be immediately followed by a separator space, comma, semicolon, period, right parenthesis, or pseudo-text delimiter.

National literal delimiters \{N\} ... \{
\}, \{N\'} ... \{'\}, \{NX\} ... \{"\}, \{NX\'} ... \{'\}

The opening delimiter must be immediately preceded by a space or a left parenthesis. The closing delimiter must be immediately followed by a separator space, comma, semicolon, period, right parenthesis, or pseudo-text delimiter. N" and N' are DBCS literal delimiters when the NSYMBOL(DBCS) compiler option is in effect.

Pseudo-text delimiters \{b==} ... \{==b\}

An opening pseudo-text delimiter must be immediately preceded by a space. A closing pseudo-text delimiter must be immediately followed by a separator space, comma, semicolon, or period. Pseudo-text delimiters must appear as balanced pairs. They delimit pseudo-text. (See “COPY statement” on page 524.)

Any punctuation character included in a PICTURE character-string, a comment character-string, or an alphanumeric literal is not considered a punctuation character, but is part of the character-string or literal.
Chapter 5. Sections and paragraphs

Sections and paragraphs define a program. Sections and paragraphs are subdivided into sentences, statements, and entries.

Sentences are subdivided into statements, and statements are subdivided into phrases. Entries are subdivided into clauses.

For details, see:
• “Sentences, statements, and entries” on page 49
• “Statements” on page 50
• “Phrases” on page 50
• “Clauses” on page 50

For more information about sections, paragraphs, and statements, see “Procedures” on page 242.

Sentences, statements, and entries

Unless the associated rules explicitly state otherwise, each required clause or statement must be written in the sequence shown in its format. If optional clauses or statements are used, they must be written in the sequence shown in their formats. These rules are true even for clauses and statements treated as comments.

The syntactical hierarchy follows this form:

• IDENTIFICATION DIVISION
  – Paragraphs
    - Entries
      • Clauses
  • ENVIRONMENT DIVISION
    – Sections
      - Paragraphs
        • Entries
          – Clauses
          - Phrases
  • DATA DIVISION
    – Sections
      - Entries
        • Clauses
        – Phrases
  • PROCEDURE DIVISION
    – Sections
      - Paragraphs
        • Sentences
        – Statements
        - Phrases
Entries

An entry is a series of clauses that ends with a separator period. Entries are constructed in the identification, environment, and data divisions.

Clauses

A clause is an ordered set of consecutive COBOL character-strings that specifies an attribute of an entry. Clauses are constructed in the identification, environment, and data divisions.

Sentences

A sentence is a sequence of one or more statements that ends with a separator period. Sentences are constructed in the PROCEDURE DIVISION.

Statements

A statement specifies an action to be taken by the program. Statements are constructed in the PROCEDURE DIVISION.

For descriptions of the different types of statements, see:

- “Imperative statements” on page 267
- “Conditional statements” on page 269
- Chapter 7, “Scope of names,” on page 59
- Chapter 22, “Compiler-directing statements,” on page 521

Phrases

Each clause or statement in a program can be subdivided into smaller units called phrases.
Chapter 6. Reference format

COBOL source text must be written in COBOL reference format.
Reference format consists of the following areas in a 72-character line.

Sequence number area
Columns 1 through 6

Indicator area
Column 7

Area A
Columns 8 through 11

Area B
Columns 12 through 72

This figure illustrates reference format for a COBOL source line.

If you want to continue a literal, code a hyphen (-) in the indicator area (column 7) of each continuation line. For details about line continuation rules, see “Continuation lines” on page 54.

The following topics provide details about these areas:

• “Sequence number area” on page 51
• “Indicator area” on page 51
• “Area A” on page 52
• “Area B” on page 53
• “Area A or Area B” on page 56

Sequence number area

The sequence number area can be used to label a source statement line. The content of this area can consist of any character in the character set of the computer.

Indicator area

Use the indicator area to specify the continuation of words or alphanumeric literals from the previous line onto the current line, the treatment of text as documentation, and debugging lines.

See “Continuation lines” on page 54, “Comment lines” on page 56, and “Debugging lines” on page 57.

The indicator area can be used for source listing formatting. A slash (/) placed in the indicator column causes the compiler to start a new page for the source listing, and the corresponding source record to be treated as a comment. The effect can be dependent on the LINECOUNT compiler option. For information about the LINECOUNT compiler option, see LINECOUNT in the Enterprise COBOL Programming Guide.
Certain items must begin in Area A.

These items are:

- Division headers
- “Section headers” on page 52
- Paragraph headers or paragraph names
- Level indicators or level-numbers (01 and 77)
- DECLARATIVES and END DECLARATIVES
- End program, end class, and end method markers

**Division headers**

A division header is a combination of words, followed by a separator period to indicate the beginning of a division.

See the following division headers:

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

A division header (except when a USING phrase is specified with a PROCEDURE DIVISION header) must be immediately followed by a separator period. Except for the USING phrase, no text can appear on the same line.

**Section headers**

In the environment and procedure divisions, a section header indicates the beginning of a series of paragraphs.

For example:

```
INPUT-OUTPUT SECTION.
```

In the DATA DIVISION, a section header indicates the beginning of an entry; for example:

```
FILE SECTION.
LINKAGE SECTION.
LOCAL-STORAGE SECTION.
WORKING-STORAGE SECTION.
```

A section header must be immediately followed by a separator period.

**Paragraph headers or paragraph names**

A paragraph header or paragraph name indicates the beginning of a paragraph.

In the ENVIRONMENT DIVISION, a paragraph consists of a paragraph header followed by one or more entries. For example:

```
OBJECT-COMPUTER. computer-name.
```
In the PROCEDURE DIVISION, a paragraph consists of a paragraph-name followed by one or more sentences.

**Level indicators (FD and SD) or level-numbers (01 and 77)**

A level indicator can be either FD or SD.

A level indicator must begin in Area A and be followed by a space. (See “FILE SECTION” on page 166.) A level-number that must begin in Area A is a one- or two-digit integer with a value of 01 or 77. It must be followed by a space or separator period.

**DECLARATIVES and END DECLARATIVES**

DECLARATIVES and END DECLARATIVES are keywords that begin and end the declaratives part of the source unit.

In the PROCEDURE DIVISION, each of the keywords DECLARATIVES and END DECLARATIVES must begin in Area A and be followed immediately by a separator period; no other text can appear on the same line. After the keywords END DECLARATIVES, no text can appear before the following section header. (See “Declaratives” on page 241.)

**End program, end class, and end method markers**

The end markers are a combination of words followed by a separator period that indicates the end of a COBOL program, method, class, factory, or object definition.

For example:

```plaintext
END PROGRAM program-name.
END CLASS class-name.
END METHOD "method-name".
END OBJECT.
END FACTORY.
```

**For programs**

`program-name` must be identical to the `program-name` of the corresponding PROGRAM-ID paragraph. Every COBOL program, except an outermost program that contains no nested programs and is not followed by another batch program, must end with an END PROGRAM marker.

**For classes**

`class-name` must be identical to the `class-name` in the corresponding CLASS-ID paragraph.

**For methods**

`method-name` must be identical to the `method-name` in the corresponding METHOD-ID paragraph.

**For object paragraphs**

There is no name in an object paragraph header or in its end marker. The syntax is simply END OBJECT.

**For factory paragraphs**

There is no name in a factory paragraph header or in its end marker. The syntax is simply END FACTORY.

**Area B**

Certain items must begin in Area B.

These items are:

- Entries, sentences, statements, and clauses
- Continuation lines
Entries, sentences, statements, clauses

The first entry, sentence, statement, or clause begins on either the same line as the header or paragraph-name that it follows, or in Area B of the next nonblank line that is not a comment line. Successive sentences or entries either begin in Area B of the same line as the preceding sentence or entry, or in Area B of the next nonblank line that is not a comment line.

Within an entry or sentence, successive lines in Area B can have the same format or can be indented to clarify program logic. The output listing is indented only if the input statements are indented. Indentation does not affect the meaning of the program. The programmer can choose the amount of indentation, subject only to the restrictions on the width of Area B. See also Chapter 5, “Sections and paragraphs,” on page 49.

Continuation lines

Any sentence, entry, clause, or phrase that requires more than one line can be continued in Area B of the next line that is neither a comment line nor a blank line.

The line being continued is a continued line; the succeeding lines are continuation lines. Area A of a continuation line must be blank.

If there is no hyphen (-) in the indicator area (column 7) of a line, the last character of the preceding line is assumed to be followed by a space.

The following items cannot be continued:

• DBCS user-defined words
• DBCS literals
• Alphanumeric literals containing DBCS characters
• National literals containing DBCS characters

However, alphanumeric literals and national literals in hexadecimal notation can be continued regardless of the kind of characters expressed in hexadecimal notation.

All characters that make up an opening literal delimiter must be on the same line. For example, Z", G", N", NX", or X".

Both characters that make up the pseudo-text delimiter separator, ==, the floating comment indicator, *>>, or the compiler directive indicator, >>, must be on the same line.

A compiler directive or compiler directive phrase, which begins with >>, must be specified on the same line.

If there is a hyphen in the indicator area of a line, the first nonblank character of the continuation line immediately follows the last nonblank character of the continued line without an intervening space.

Continuation of alphanumeric and national literals

Alphanumeric and national literals can be continued only when there are no DBCS characters in the content of the literal.

The following rules apply to alphanumeric and national literals that do not contain DBCS characters:

• If the continued line contains an alphanumeric or national literal without a closing quotation mark, all spaces at the end of the continued line (through column 72) are considered to be part of the literal. The continuation line must contain a hyphen in the indicator area, and the first nonblank character must be a quotation mark. The continuation of the literal begins with the character immediately following the quotation mark.
• If an alphanumeric or national literal that is to be continued on the next line has as its last character a quotation mark in column 72, the continuation line must start with two consecutive quotation marks. This will result in a quotation mark as part of the value of the literal.
If the last character on the continued line of an alphanumeric or national literal is a quotation mark in Area B, the continuation line can start with a quotation mark. This will result in two consecutive literals instead of one continued literal.

The rules are the same when an apostrophe is used instead of a quotation mark in delimiters.

If you want to continue a literal such that the continued lines and the continuation lines are part of one literal:

- Code a hyphen in the indicator area of each continuation line.
- Code the literal value using all columns of each continued line, up to and including column 72. (Do not terminate the continued lines with a quotation mark followed by a space.)
- Code a quotation mark before the first character of the literal on each continuation line.
- Terminate the last continuation line with a quotation mark followed by a space.

In the following examples, the number and size of literals created are indicated below the example:

```
|...+.*..1....+....2....+....3....+....4....+....5....+....6....+....7...
000001  "AAAAAAAAABBBBBBBBCCCCCCCCDDDDDDDDDDDDDEEEEEE
         "GGGGGGGGGHHHHHHHHIIIIIIIIIIIJJJJJJJJJKKKKKKKKK
         "LLLLLLLLLLMMMMMM

Literal 000001 is interpreted as one alphanumeric literal that is 120 bytes long. Each character between the starting quotation mark and up to and including column 72 of continued lines is counted as part of the literal.

|...+.*..1....+....2....+....3....+....4....+....5....+....6....+....7...
000003  "GGGGGGGGG"

Literal 000003 is interpreted as one national literal that is 60 national character positions in length (120 bytes). Each character between the starting quotation mark and the ending quotation mark on the continued line is counted as part of the literal. Although single-byte characters are entered, the value of the literals is stored as national characters.

|...+.*..1....+....2....+....3....+....4....+....5....+....6....+....7...
000005  "AAAAAAAAABBBBBBBBCCCCCCCCDDDDDDDDDDDDDEEEEEE
         "GGGGGGGGGHHHHHHHHIIIIIIIIIIIJJJJJJJJJKKKKKKKKK
         "LLLLLLLLLLMMMMMM"

Literal 000005 is interpreted as one literal that is 140 bytes long. The blanks at the end of each continued line are counted as part of the literal because the continued lines do not end with a quotation mark.

|...+.*..1....+....2....+....3....+....4....+....5....+....6....+....7...
000010  "AAAAAAAAABBBBBBBBCCCCCCCCDDDDDDDDDDDDDEEEEEE
         "GGGGGGGGGHHHHHHHHIIIIIIIIIIIJJJJJJJJJKKKKKKKKK
         "LLLLLLLLLLMMMMMM"

Literal 000010 is interpreted as three separate literals that have lengths of 50, 50, and 20, respectively. The quotation mark with the following space terminates the continued line. Only the characters within the quotation marks are counted as part of the literals. Literal 000010 is not valid as a VALUE clause literal for non-level-88 data items.

To code a continued literal where the length of each continued part of the literal is less than the length of Area B, adjust the starting column such that the last character of the continued part is in column 72.
Area A or Area B

Certain items can begin in either Area A or Area B.

These items are:

• Level-numbers
• Comment lines
• Floating comment indicators (*>)
• Compiler-directing statements
• Debugging lines
• Pseudo-text
• Blank lines

Level-numbers

A level-number that can begin in Area A or B is a one- or two-digit integer with a value of 02 through 49, 66, or 88.

A level-number that must begin in Area A is a one- or two-digit integer with a value of 01 or 77. A level-number must be followed by a space or a separator period. For more information, see “Level-numbers” on page 176.

Comment lines

A comment line is any line with an asterisk (*) or slash (/) in the indicator area (column 7) of the line, or with a floating comment indicator (*>) as the first character-string in the program text area (Area A plus Area B).

The comment can be written anywhere in the program text area of that line, and can consist of any combination of characters from the character set of the computer.

Comment lines can be placed anywhere in a program, method, or class definition. Comment lines placed before the IDENTIFICATION DIVISION header must follow any control cards (for example, PROCESS or CBL).

Important: Comments intermixed with control cards could nullify some of the control cards and cause them to be diagnosed as errors.

Multiple comment lines are allowed. Each must begin with an asterisk (*) or a slash (/) in the indicator area, or with a floating comment indicator (*>).

For more information about floating comment indicators, see “Floating comment indicators (*>)” on page 56.

An asterisk (*) comment line is printed on the next available line in the output listing. The effect can be dependent on the LINECOUNT compiler option. For information about the LINECOUNT compiler option, see LINECOUNT in the Enterprise COBOL Programming Guide. A slash (/) comment line is printed on the first line of the next page, and the current page of the output listing is ejected.

The compiler treats a comment line as documentation, and does not check it syntactically.

Floating comment indicators (*>)

In addition to the fixed indicators that can only be specified in the indicator area of the source reference format, a floating comment indicator (*>) can be specified anywhere in the program-text area to indicate a comment line or an inline comment.

A floating comment indicator indicates a comment line if it is the first character string in the program-text area (Area A plus Area B), or indicates an inline comment if it is after one or more character strings in the program-text area.

These are the rules for floating comment indicators:
• Both characters (* and >) that form the multiple-character floating indicator must be contiguous and on the same line.
• The floating comment indicator for an inline comment must be preceded by a separator space, and can be specified wherever a separator space can be specified.
• All characters following the floating comment indicator up to the end of Area B are comment text.

Compiler-directing statements
Most compiler-directing statements, including COPY and REPLACE, can start in either Area A or Area B. BASIS, CBL (PROCESS), *CBL (*CONTROL), DELETE, EJECT, INSERT, SKIP1, SKIP2, SKIP3, and TITLE statements can also start in Area A or Area B.

Compiler directives
Compiler directives must start in Area B.
For more information, see Chapter 23, “Compiler directives,” on page 545.

Debugging lines
A debugging line is any line with a D (or d) in the indicator area of the line.
Debugging lines can be written in the ENVIRONMENT DIVISION (after the OBJECT-COMPUTER paragraph), the DATA DIVISION, and the PROCEDURE DIVISION. If a debugging line contains only spaces in Area A and Area B, it is considered a blank line.
See “WITH DEBUGGING MODE” in “SOURCE-COMPUTER paragraph” on page 106.

Pseudo-text
The character-strings and separators that comprise pseudo-text can start in either Area A or Area B.
If, however, there is a hyphen in the indicator area (column 7) of a line that follows the opening pseudo-text delimiter, Area A of the line must be blank, and the rules for continuation lines apply to the formation of text words. See “Continuation lines” on page 54 for details.

Blank lines
A blank line contains nothing but spaces in column 7 through column 72. A blank line can be anywhere in a program.
Chapter 7. Scope of names

A user-defined word names a data resource or a COBOL programming element. Examples of named data resources are a file, a data item, or a record. Examples of named programming elements are a program, a paragraph, a method, or a class definition.

The sections below define the types of names in COBOL and explain where the names can be referenced:

- “Types of names” on page 59
- “External and internal resources” on page 61
- “Resolution of names” on page 62

Types of names

In addition to identifying a resource, a name can have global or local attributes. Some names are always global, some names are always local, and some names are either local or global depending on specifications in the program in which the names are defined.

For programs
A global name can be used to refer to the resource with which it is associated both:
- From within the program in which the global name is defined
- From within any other program that is contained in the program that defines the global name

Use the GLOBAL clause in the data description entry to indicate that a name is global. For more information about using the GLOBAL clause, see “GLOBAL clause” on page 167.

A local name can be used only to refer to the resource with which it is associated from within the program in which the local name is defined.

By default, if a data-name, a file-name, a record-name, or a condition-name definition in a data description entry does not include the GLOBAL clause, the name is local.

For methods
All names defined in methods are implicitly local.

For classes
Names defined in a class definition are global to all the methods contained in that class definition.

For object paragraphs
Names defined in the DATA DIVISION of an object paragraph are global to the methods contained in that object paragraph.

For factory paragraphs
Names defined in the DATA DIVISION of a factory paragraph are global to the methods contained in that factory paragraph.

Restriction: Specific rules sometimes prohibit specifying the GLOBAL clause for certain data description, file description, or record description entries.

The following list indicates the names that you can use and whether the name can be local or global:

data-name
data-name assigns a name to a data item.

A data-name is global if the GLOBAL clause is specified either in the data description entry that defines the data-name or in another entry to which that data description entry is subordinate.

file-name
file-name assigns a name to a file connector.

A file-name is global if the GLOBAL clause is specified in the file description entry for that file-name.
record-name
record-name assigns a name to a record.

A record-name is global if the GLOBAL clause is specified in the record description that defines the record-name, or in the case of record description entries in the FILE SECTION, if the GLOBAL clause is specified in the file description entry for the file name associated with the record description entry.

condition-name
condition-name associates a value with a conditional variable.

A condition-name that is defined in a data description entry is global if that entry is subordinate to another entry that specifies the GLOBAL clause.

A condition-name that is defined within the configuration section is always global.

program-name
program-name assigns a name to an external or internal (nested) program. For more information, see “Conventions for program-names” on page 82.

A program-name is neither local nor global. For more information, see “Conventions for program-names” on page 82.

method-name
method-name assigns a name to a method. method-name must be specified as the content of an alphanumeric literal or a national literal.

section-name
section-name assigns a name to a section in the PROCEDURE DIVISION.

A section-name is always local.

paragraph-name
paragraph-name assigns a name to a paragraph in the PROCEDURE DIVISION.

A paragraph-name is always local.

basis-name
basis-name specifies the name of source text that is be included by the compiler into the source unit. For details, see “BASIS statement” on page 521.

library-name
library-name specifies the COBOL library that the compiler uses for including COPY text. For details, see “COPY statement” on page 524.

text-name
text-name specifies the name of COPY text to be included by the compiler into the source unit. For details, see “COPY statement” on page 524.

alphabet-name
alphabet-name assigns a name to a specific character set or collating sequence, or both, in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION.

An alphabet-name is always global.

class-name (of data)
class-name assigns a name to the proposition in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION for which a truth value can be defined.

A class-name is always global.

class-name (object-oriented)
class-name assigns a name to an object-oriented class or subclass.

mnemonic-name
mnemonic-name assigns a user-defined word to an implementer-name.

A mnemonic-name is always global.
symbolic-character

A symbolic-character is always global.

index-name

index-name assigns a name to an index associated with a specific table.

If a data item that possesses the global attribute includes a table accessed with an index, that index also possesses the global attribute. In addition, the scope of that index-name is identical to the scope of the data-name that includes the table.

xml-schema-name

xml-schema-name assigns a name to the system identifier of a file containing an XML schema.

An xml-schema-name is always global.

External and internal resources

The storage associated with a data item or a file connector can be external or internal to the program or method in which the resource is declared.

A data item or file connector is external if the storage associated with that resource is associated with the run unit rather than with any particular program or method within the run unit. An external resource can be referenced by any program or method in the run unit that describes the resource. References to an external resource from different programs or methods using separate descriptions of the resource are always to the same resource. In a run unit, there is only one representation of an external resource.

A resource is internal if the storage associated with that resource is associated only with the program or method that describes the resource.

External and internal resources can have either global or local names.

A data record described in the WORKING-STORAGE SECTION is given the external attribute by the presence of the EXTERNAL clause in its data description entry. Any data item described by a data description entry subordinate to an entry that describes an external record also attains the external attribute. If a record or data item does not have the external attribute, it is part of the internal data of the program or method in which it is described.

Two programs or methods in a run unit can reference the same file connector in the following circumstances:

• An external file connector can be referenced from any program or method that describes that file connector.

• If a program is contained within another program, both programs can refer to a global file connector by referring to an associated global file-name either in the containing program or in any program that directly or indirectly contains the containing program.

Two programs or methods in a run unit can reference common data in the following circumstances:

• The data content of an external data record can be referenced from any program or method provided that program or method has described that data record.

• If a program is contained within another program, both programs can refer to data that possesses the global attribute either in the program or in any program that directly or indirectly contains the containing program.

The data records described as subordinate to a file description entry that does not contain the EXTERNAL clause or to a sort-merge file description entry, as well as any data items described subordinate to the data description entries for such records, are always internal to the program or method that describes the file-name. If the EXTERNAL clause is included in the file description entry, the data records and the data items attain the external attribute.
Resolution of names

The rules for resolution of names depend on whether the names are specified in a program or in a class definition.

Names within programs

When a program, program B, is directly contained within another program, program A, both programs can define a condition-name, a data-name, a file-name, or a record-name using the same user-defined word. When such a duplicated name is referenced in program B, the following steps determine the referenced resource (these rules also apply to classes and contained methods):

1. The referenced resource is identified from the set of all names that are defined in program B and all global names defined in program A and in any programs that directly or indirectly contain program A. The normal rules for qualification and any other rules for uniqueness of reference are applied to this set of names until one or more resources is identified.

2. If only one resource is identified, it is the referenced resource.

3. If more than one resource is identified, no more than one resource can have a name local to program B. If zero or one of the resources has a name local to program B, the following rules apply:
   - If the name is declared in program B, the resource in program B is the referenced resource.
   - If the name is not declared in program B, the referenced resource is:
     - The resource in program A if the name is declared in program A
     - The resource in the containing program if the name is declared in the program that contains program A

This rule is applied to further containing programs until a valid resource is found.

Names within a class definition

Within a class definition, resources can be defined within the following units:

- The factory data division
- The object data division
- A method data division

If a resource is defined with a given name in the DATA DIVISION of an object definition, and there is no resource defined with the same name in an instance method of that object definition, a reference to that name from an instance method is a reference to the resource in the object DATA DIVISION.

If a resource is defined with a given name in the DATA DIVISION of a factory definition, and there is no resource defined with the same name in a factory method of that factory definition, a reference to that name from a factory method is a reference to the resource in the factory data division.

If a resource is defined within a method, any reference within the method to that resource name is always a reference to the resource in the method.

The normal rules for qualification and uniqueness of reference apply when the same name is associated with more than one resource within a given method data division, object data division, or factory data division.
Chapter 8. Referencing data names, copy libraries, and PROCEDURE DIVISION names

References can be made to external and internal resources. References to data and procedures can be either explicit or implicit.

For more information about rules for qualification, and for explicit and implicit data references, see the following topics:

- “Uniqueness of reference” on page 63
- “Data attribute specification” on page 74

Uniqueness of reference

Every user-defined name in a COBOL program is assigned by the user to name a resource for solving a data processing problem. To use a resource, a statement in a COBOL program must contain a reference that uniquely identifies that resource.

To ensure uniqueness of reference, a user-defined name can be qualified. A subscript is required for unique reference to a table element, except as specified in “Subscripting” on page 68. A data-name or function-name, any subscripts, and the specified reference-modifier uniquely reference a data item defined by reference modification.

When the same name has been assigned in separate programs to two or more occurrences of a resource of a given type, and when qualification by itself does not allow the references in one of those programs to differentiate between the identically named resources, then certain conventions that limit the scope of names apply. The conventions ensure that the resource identified is that described in the program containing the reference. For more information about resolving program-names, see “Resolution of names” on page 62.

Unless otherwise specified by the rules for a statement, any subscripts and reference modification are evaluated only once as the first step in executing that statement.

Qualification

A name that exists within a hierarchy of names can be made unique by specifying one or more higher-level names in the hierarchy. The higher-level names are called qualifiers, and the process by which such names are made unique is called qualification.

Qualification is specified by placing one or more phrases after a user-specified name, with each phrase made up of the word IN or OF followed by a qualifier. (IN and OF are logically equivalent.)

If there is only one 01 level with a given name, that name can be referenced even if it is not unique when the QUALIFY(EXTEND) option is in effect.

You must specify enough qualification to make the name unique; however, it is not always necessary to specify all the levels of the hierarchy. For example, if there is more than one file whose records contain the field EMPLOYEE-NO, but only one of the files has a record named MAIN-RECORD:

- EMPLOYEE-NO OF MAIN-RECORD sufficiently qualifies EMPLOYEE-NO.
- EMPLOYEE-NO OF MAIN-RECORD OF MAIN-FILE is valid but unnecessary.

Qualification rules

The rules for qualifying a name are:

- A name can be qualified even though it does not need qualification except in a REDEFINES clause, in which case it must not be qualified.
• Each qualifier must be of a higher level than the name it qualifies and must be within the same hierarchy.
• If there is more than one combination of qualifiers that ensures uniqueness, any of those combinations can be used.
• If compiler option QUALIFY (EXTEND) is in effect, and if there is only one fully qualified name that matches your combination of qualifiers, that reference will be considered unique, even if the set of qualifiers also matches a partial qualification for a different data item. Fully qualified means every qualifier is specified.

Related references
QUALIFY (Enterprise COBOL Programming Guide)

Identical names
When programs are directly or indirectly contained within other programs, each program can use identical user-defined words to name resources.
A program references the resources that program describes rather than the same-named resources described in another program, even if the names are different types of user-defined words.
These same rules apply to classes and their contained methods.

References to COPY libraries
If library-name-1 is not specified, SYSLIB is assumed as the library name.

Format
text-name-1
IN
OF
library-name-1

For rules on referencing COPY libraries, see “COPY statement” on page 524.

References to PROCEDURE DIVISION names
PROCEDURE DIVISION names that are explicitly referenced in a program must be unique within a section.

Format 1
paragraph-name-1
IN
OF
section-name-1

Format 2
section-name-1

A section-name is the highest and only qualifier available for a paragraph-name and must be unique if referenced. (Section-names are described under “Procedures” on page 242.)
If explicitly referenced, a paragraph-name must not be duplicated within a section. When a paragraph-name is qualified by a section-name, the word SECTION must not appear. A paragraph-name need not be qualified when referred to within the section in which it appears. A paragraph-name or section-name that appears in a program cannot be referenced from any other program.
References to DATA DIVISION names

This section discusses the following types of references.

- “Simple data reference” on page 65
- “Identifiers” on page 65

Simple data reference

The most basic method of referencing data items in a COBOL program is *simple data reference*, which is *data-name-1* without qualification, subscripting, or reference modification. Simple data reference is used to reference a single elementary or group item.

```
Format

  data-name-1
```

*Data-name-1*

Can be any data description entry.

*Data-name-1* must be unique in a program.

Identifiers

When used in a syntax diagram in this information, the term *identifier* refers to a valid combination of a data-name or function-identifier with its qualifiers, subscripts, and reference-modifiers as required for uniqueness of reference.

Rules for identifiers associated with a format can however specifically prohibit qualification, subscripting, or reference modification.

The term *data-name* refers to a name that must not be qualified, subscripted, or reference modified unless specifically permitted by the rules for the format.

- For a description of qualification, see “Qualification” on page 63.
- For a description of subscripting, see “Subscripting” on page 68.
- For a description of reference modification, see “Reference modification” on page 71.

```
Format 1

  data-name-1
      IN
      OF
      data-name-2

      IN
      OF
      file-name-1

      (subscript)

      (leftmost-character-position :
      length)
```

*Data-name-1, data-name-2*

Can be a record-name.
**file-name-1**
Must be identified by an FD or SD entry in the DATA DIVISION.

(file-name-1) must be unique within this program.

---

**Format 2**

![Diagram of Format 2](image)

---

**Format 3**

![Diagram of Format 3](image)

---

**data-name-1, data-name-2**
Can be a record-name.

**condition-name-1**
Can be referenced by statements and entries either in the program that contains the configuration section or in a program contained within that program.

**file-name-1**
Must be identified by an FD or SD entry in the DATA DIVISION.

(file-name-1) must be unique within this program.

**LINAGE-COUNTER**
Must be qualified each time it is referenced if more than one file description entry that contains a LINAGE clause has been specified in the source unit.

**file-name-2**
Must be identified by the FD or SD entry in the DATA DIVISION. \(file-name-2\) must be unique within this program.

Duplication of data-names must not occur in those places where the data-names cannot be made unique by qualification.

In the same program, the data-name specified as the subject of the entry whose level-number is 01 that includes the EXTERNAL clause must not be the same data-name specified for any other data description entry that includes the EXTERNAL clause.

In the same DATA DIVISION, the data description entries for any two data items for which the same data-name is specified must not include the GLOBAL clause.

DATA DIVISION names that are explicitly referenced must either be uniquely defined or made unique through qualification. Unreferenced data items need not be uniquely defined. The highest level in a data hierarchy (a data item associated with a level indicator (FD or SD in the FILE SECTION) or with level-number 01) must be uniquely named if referenced. Data items associated with level-numbers 02 through 49 are successively lower levels of the hierarchy.
**Condition-name**

See the syntax and description for details.

---

### Format 1: condition-name in data division

```
condition-name-1
  IN
  OF data-name-1

condition-name-1
  IN
  OF file-name-1
```

```
(subscript)
```

### Format 2: condition-name in SPECIAL-NAMES paragraph

```
condition-name-1
  IN
  OF mnemonic-name-1
```

**condition-name-1**

Can be referenced by statements and entries either in the program that contains the definition of `condition-name-1`, or in a program contained within that program.

If explicitly referenced, a condition-name must be unique or be made unique through qualification or subscripting (or both) except when the scope of names by itself ensures uniqueness of reference.

If qualification is used to make a condition-name unique, the associated conditional variable can be used as the first qualifier. If qualification is used, the hierarchy of names associated with the conditional variable itself must be used to make the condition-name unique.

If references to a conditional variable require subscripting, reference to any of its condition-names also requires the same combination of subscripting.

In this information, `condition-name` refers to a condition-name qualified or subscripted, as necessary.

**data-name-1**

Can be a record-name.

**file-name-1**

Must be identified by an FD or SD entry in the DATA DIVISION.

`file-name-1` must be unique within this program.

**mnemonic-name-1**

For information about acceptable values for `mnemonic-name-1`, see “SPECIAL-NAMES paragraph” on page 108.

---

**Index-name**

An index-name identifies an index. An index can be regarded as a private special register that the compiler generates for working with a table. You name an index by specifying the INDEXED BY phrase in the OCCURS clause that defines a table.

You can use an index-name in only the following language elements:
Index data item
An index data item is a data item that can hold the value of an index.

You define an index data item by specifying the USAGE IS INDEX clause in a data description entry. The name of an index data item is a data-name. An index data item can be used anywhere a data-name or identifier can be used, unless stated otherwise in the rules of a particular statement. You can use the SET statement to save the value of an index (referenced by index-name) in an index data item.

Subscripting
Subscripting is a method of providing table references through the use of subscripts. A subscript is a positive integer whose value specifies the occurrence number of a table element.

Format

```
condition-name-1
    data-name-1
        IN
        OF
        data-name-2
          IN
        OF
          file-name-1
            (arrow)
        integer-1
            ALL
            data-name-3
                +
                integer-2
                -
                index-name-1
                    +
                    integer-3
```

**condition-name-1**
The conditional variable for condition-name-1 must contain an OCCURS clause or must be subordinate to a data description entry that contains an OCCURS clause.

**data-name-1**
Must contain an OCCURS clause or must be subordinate to a data description entry that contains an OCCURS clause.

**data-name-2, file-name-1**
Must name data items or records that contain data-name-1.
integer-1
Can be signed. If signed, it must be positive.

ALL
Shall not be specified if condition-name-1 is specified.
Must be used only when the subscripted identifier is used as an intrinsic function argument or to identify a table in a format 2 SORT statement (table SORT statement).
If ALL is specified, the subscript is all of the possible values of a subscript for the associated table as specified in the rules for the functions for which the subscript ALL is allowed.

data-name-3
Must be a numeric elementary item representing an integer.
data-name-3 can be qualified.

index-name-1
Corresponds to a data description entry in the hierarchy of the table being referenced that contains an INDEXED BY phrase that specifies that name.

integer-2, integer-3
Cannot be signed.

The subscripts, enclosed in parentheses, are written immediately following any qualification for the name of the table element. The number of subscripts in such a reference must equal the number of dimensions in the table whose element is being referenced. That is, there must be a subscript for each OCCURS clause in the hierarchy that contains the data-name including the data-name itself.

When more than one subscript is required, they are written in the order of successively less inclusive dimensions of the data organization. If a multidimensional table is thought of as a series of nested tables and the most inclusive or outermost table in the nest is considered to be the major table with the innermost or least inclusive table being the minor table, the subscripts are written from left to right in the order major, intermediate, and minor.

For example, if TABLE-THREE is defined as:

```
01  TABLE-THREE.
   05  ELEMENT-ONE OCCURS 3 TIMES.
      10  ELEMENT-TWO OCCURS 3 TIMES.
      15  ELEMENT-THREE OCCURS 2 TIMES PIC X(8).
```

a valid subscripted reference to TABLE-THREE is:

```
ELEMENT-THREE (2 2 1)
```

Subscripted references can also be reference modified. See the third example under “Reference modification examples” on page 72. A reference to an item must not be subscripted unless the item is a table element or an item or condition-name associated with a table element.

Each table element reference must be subscripted except when such reference appears:

- In a USE FOR DEBUGGING statement
- As the subject of a SEARCH statement
- In a REDEFINES clause
- In the KEY IS phrase of an OCCURS clause
- In a format 2 SORT statement (table SORT statement)

In a format 2 SORT statement, subscripting may be specified with the rightmost subscript being the word ALL.

The lowest permissible occurrence number represented by a subscript is 1. The highest permissible occurrence number in any particular case is the maximum number of occurrences of the item as specified in the OCCURS clause.
Subscripting using data-names

When a data-name is used to represent a subscript, it can be used to reference items within different tables. These tables need not have elements of the same size. The same data-name can appear as the only subscript with one item and as one of two or more subscripts with another item. A data-name subscript can be qualified; it cannot be subscripted or indexed. For example, valid subscripted references to TABLE-THREE, assuming that SUB1, SUB2, and SUB3 are all items subordinate to SUBSCRIPT-ITEM, include:

- ELEMENT-THREE (SUB1 SUB2 SUB3)
- ELEMENT-THREE IN TABLE-THREE (SUB1 OF SUBSCRIPT-ITEM, SUB2 OF SUBSCRIPT-ITEM, SUB3 OF SUBSCRIPT-ITEM)

Subscripting using index-names (indexing)

Indexing allows such operations as table searching and manipulating specific items. To use indexing, you associate one or more index-names with an item whose data description entry contains an OCCURS clause.

An index associated with an index-name acts as a subscript, and its value corresponds to an occurrence number for the item to which the index-name is associated.

The INDEXED BY phrase, by which the index-name is identified and associated with its table, is an optional part of the OCCURS clause. There is no separate entry to describe the index associated with index-name. At run time, the contents of the index corresponds to an occurrence number for that specific dimension of the table with which the index is associated.

The initial value of an index at run time is undefined, and the index must be initialized before it is used as a subscript. An initial value is assigned to an index with one of the following statements:

- The PERFORM statement with the VARYING phrase
- The SEARCH statement with the ALL phrase
- The SET statement

The use of an integer or data-name as a subscript that references a table element or an item within a table element does not cause the alteration of any index associated with that table.

An index-name can be used to reference any table. However, the element length of the table being referenced and of the table that the index-name is associated with should match. Otherwise, the reference will not be to the same table element in each table, and you might get runtime errors.

Data that is arranged in the form of a table is often searched. The SEARCH statement provides facilities for producing serial and nonserial searches. It is used to search for a table element that satisfies a specific condition and to adjust the value of the associated index to indicate that table element.

To be valid during execution, an index value must correspond to a table element occurrence of neither less than one, nor greater than the highest permissible occurrence number.

For more information about index-names, see “Index-name” on page 67 and “INDEXED BY phrase” on page 184.

Relative subscripting

In relative subscripting, the name of a table element is followed by a subscript of the form data-name or index-name followed by the operator + or -, and a positive or unsigned integer literal.

The operators + and - must be preceded and followed by a space. The value of the subscript used is the same as if the index-name or data-name had been set up or down by the value of the integer. The use of relative indexing does not cause the program to alter the value of the index.
Reference modification

Reference modification defines a data item by specifying a leftmost character and optional length for the data item.

**Format: reference modification**

```plaintext
FUNCTION function-name-1
  (argument-1
    (leftmost-character-position : length))
```

**data-name-1**

Must reference a data item described explicitly or implicitly with usage DISPLAY, DISPLAY-1, NATIONAL, or UTF-8. A national group item is processed as an elementary data item of category national.

data-name-1 can be qualified or subscripted.

data-name-1 cannot be a dynamic-length group item. If data-name-1 is a dynamic-length elementary item, it is treated as though it were a fixed-length item whose length is the same as the current length of the dynamic-length elementary item. When used as a reference modified receiver in a PROCEDURE DIVISION statement, the current length of the dynamic-length elementary item is not modified.

**leftmost-character-position**

Must be an arithmetic expression. The evaluation of leftmost-character-position must result in a positive nonzero integer that is less than or equal to the number of characters in the data item referenced by data-name-1.

**length**

Must be an arithmetic expression.

The evaluation of length must result in a positive nonzero integer.

The sum of leftmost-character-position and length minus the value 1 must be less than or equal to the number of character positions in data-name-1. If length is omitted, the length used will be equal to the number of character positions in data-name-1 plus 1, minus leftmost-character-position.

**function-name-1**

Must reference an alphanumeric or national function.

For usages DISPLAY-1 and NATIONAL, each character position occupies 2 bytes. Reference modification operates on whole character positions and not on the individual bytes of the characters in usages DISPLAY-1 and NATIONAL. For usage DISPLAY, reference modification operates as though each character were a single-byte character.

Unless otherwise specified, reference modification is allowed anywhere an identifier or function-identifier that references a data item or function with the same usage as the reference-modified data item is permitted.

Each character position referenced by data-name-1 or function-name-1 is assigned an ordinal number incrementing by one from the leftmost position to the rightmost position. The leftmost position is assigned the ordinal number one. If the data description entry for data-name-1 contains a SIGN IS SEPARATE clause, the sign position is assigned an ordinal number within that data item.

If data-name-1 is described with usage DISPLAY and category numeric, numeric-edited, alphabetic, alphanumeric-edited, or external floating-point, data-name-1 is operated upon for purposes of reference.
modification as if it were redefined as a data item of category alphanumeric with the same size as the data item referenced by data-name-1.

If data-name-1 is described with usage NATIONAL and category numeric, numeric-edited, national-edited, or external floating-point, data-name-1 is operated upon for purposes of reference modification as if it were redefined as a data item of category national with the same size as the data item referenced by data-name-1.

If data-name-1 is a national group item, data-name-1 is processed as an elementary data item of category national.

Reference modification creates a unique data item that is a subset of data-name-1 or a subset of the value referenced by function-name-1 and its arguments, if any. This unique data item is considered an elementary data item without the JUSTIFIED clause.

When a function is reference-modified, the unique data item has class, category, and usage national if the type of the function is national; otherwise, it has class and category alphanumeric and usage display.

When data-name-1 is reference-modified, the unique data item has the same class, category, and usage as that defined for the data item referenced by data-name-1 except that:

• If data-name-1 has category national-edited, the unique data item has category national.
• If data-name-1 has usage NATIONAL and category numeric-edited, numeric, or external floating-point, the unique data item has category national.
• If data-name-1 has usage DISPLAY, and category numeric-edited, alphanumeric-edited, numeric, or external floating-point, the unique data item has category alphanumeric.
• If data-name-1 references an alphanumeric group item, the unique data item is considered to have usage DISPLAY and category alphanumeric.
• If data-name-1 references a national group item, the unique data item has usage NATIONAL and category national.

If length is not specified, the unique data item created extends from and includes the character position identified by leftmost-character-position up to and including the rightmost character position of the data item referenced by data-name-1.

Evaluation of operands

Reference modification for an operand is evaluated as follows:

• If subscripting is specified for the operand, the reference modification is evaluated immediately after evaluation of the subscript.

• If subscripting is not specified for the operand, the reference modification is evaluated at the time subscripting would be evaluated if subscripts had been specified.

Reference modification examples

The statements in the examples transfer the first 10 characters of the data-item referenced by WHOLE-NAMES to the data-item referenced by FIRST-NAMES.

```
77 WHOLE-NAME PIC X(25).
77 FIRST-NAME PIC X(10).
77 START-P PIC 9(4) BINARY VALUE 1.
77 STR-LENGTH PIC 9(4) BINARY VALUE 10.
...
MOVE WHOLE-NAME(1:10) TO FIRST-NAME.
MOVE WHOLE-NAME(START-P:STR-LENGTH) TO FIRST-NAME.
```
The following statement transfers the last 15 characters of the data-item referenced by WHOLE-NAME to the data-item referenced by LAST-NAME.

```
77 WHOLE-NAME   PIC X(25).
77 LAST-NAME    PIC X(15).
...
MOVE WHOLE-NAME(11:) TO LAST-NAME.
```

The following statement transfers the fourth and fifth characters of the third occurrence of TAB to the variable SUFFIX.

```
01 TABLE-1.
  02 TAB OCCURS 10 TIMES PICTURE X(5).
  77 SUFFIX                    PICTURE X(2).
...
MOVE TAB OF TABLE-1 (3) (4:2) TO SUFFIX.
```

**Function-identifier**

A *function-identifier* is a sequence of character strings and separators that uniquely references the data item that results from the evaluation of a function.

**Format**

```
FUNCTION — function-name-1
```

- **argument-1**
  - Must be an identifier, literal (other than a figurative constant), or arithmetic expression.
  - For more information, see Chapter 21, “Intrinsic functions,” on page 459.

- **function-name-1**
  - *function-name-1* must be one of the intrinsic function names.

- **reference-modifier**
  - Can be specified only for functions of the type alphanumeric or national.

  A function-identifier that makes reference to an alphanumeric or national function can be specified anywhere that a data item of category alphanumeric or category national, respectively, can be referenced and where references to functions are not specifically prohibited, except as follows:

  - As a receiving operand of any statement
  - Where a data item is required to have particular characteristics (such as class and category, size, sign, and permissible values) and the evaluation of the function according to its definition and the particular arguments specified would not have these characteristics

A function-identifier that makes reference to an integer or numeric function can be used wherever an arithmetic expression can be used.
Explicit data attributes are data attributes that you specify in COBOL coding. Implicit data attributes are default values. If you do not explicitly code a data attribute, the compiler assumes a default value.

For example, you need not specify the USAGE of a data item. If USAGE is omitted and the symbol N or symbol U is not specified in the PICTURE clause, the default is USAGE DISPLAY, which is the implicit data attribute. When PICTURE symbol N is used, USAGE DISPLAY-1 is the default when the NSYMBOL(DBCS) compiler option is in effect; USAGE NATIONAL is the default when the NSYMBOL(NATIONAL) compiler option is in effect. Similarly, when PICTURE symbol U is used, USAGE UTF-8 is assumed. These are implicit data attributes.
Chapter 9. Transfer of control

In the PROCEDURE DIVISION, unless there is an explicit control transfer or there is no next executable statement, program flow transfers control from statement to statement in the order in which the statements are written. This normal program flow is an implicit transfer of control.

In addition to the implicit transfers of control between consecutive statements, implicit transfer of control also occurs when the normal flow is altered without the execution of a procedure branching statement. The following examples show implicit transfers of control, overriding statement-to-statement transfer of control:

• After execution of the last statement of a procedure that is executed under control of another COBOL statement, control implicitly transfers. (COBOL statements that control procedure execution are, for example, MERGE, PERFORM, SORT, and USE.) Further, if a paragraph is being executed under the control of a PERFORM statement that causes iterative execution, and that paragraph is the first paragraph in the range of that PERFORM statement, an implicit transfer of control occurs between the control mechanism associated with that PERFORM statement and the first statement in that paragraph for each iterative execution of the paragraph.

• During SORT or MERGE statement execution, control is implicitly transferred to an input or output procedure.

• During XML PARSE statement execution, control is implicitly transferred to a processing procedure.

• During execution of any COBOL statement that causes execution of a declarative procedure, control is implicitly transferred to that procedure.

• At the end of execution of any declarative procedure, control is implicitly transferred back to the control mechanism associated with the statement that caused its execution.

COBOL also provides explicit control transfers through the execution of any procedure branching, program call, or conditional statement. (Lists of procedure branching and conditional statements are contained in “Statement categories” on page 267.)

Definition: The term next executable statement refers to the next COBOL statement to which control is transferred, according to the rules given above. There is no next executable statement under the following circumstances:

• When the program contains no PROCEDURE DIVISION

• Following the last statement in a declarative section when the paragraph in which it appears is not being executed under the control of some other COBOL statement

• Following the last statement in a program or method when the paragraph in which it appears is not being executed under the control of some other COBOL statement in that program

• Following the last statement in a declarative section when the statement is in the range of an active PERFORM statement executed in a different section and this last statement of the declarative section is not also the last statement of the procedure that is the exit of the active PERFORM statement

• Following a STOP RUN statement or EXIT PROGRAM statement that transfers control outside the COBOL program

• Following a GOBACK statement that transfers control outside the COBOL program

• Following an EXIT METHOD statement that transfers control outside the COBOL method

• The end program or end method marker

When there is no next executable statement and control is not transferred outside the COBOL program, the program flow of control is undefined unless the program execution is in the nondeclarative procedures portion of a program under control of a CALL statement, in which case an implicit EXIT PROGRAM statement is executed.

Similarly, if control reaches the end of the PROCEDURE DIVISION of a method and there is no next executable statement, an implicit EXIT METHOD statement is executed.
Part 2. COBOL source unit structure
Chapter 10. COBOL program structure

A COBOL source program is a syntactically correct set of COBOL statements.

Nested programs

A nested program is a program that is contained in another program. Contained programs can reference some of the resources of the programs that contain them. If program B is contained in program A, it is directly contained if there is no program contained in program A that also contains program B. Program B is indirectly contained in program A if there exists a program contained in program A that also contains program B. For more information about nested programs, see Nested programs in the Enterprise COBOL Programming Guide.

Object program

An object program is a set or group of executable machine language instructions and other material designed to interact with data to provide problem solutions. An object program is generally the machine language result of the operation of a COBOL compiler on a source program. The term object program also refers to the methods that result from compiling a class definition.

Run unit

A run unit is one or more object programs that interact with one another and that function at run time as an entity to provide problem solutions.

Sibling program

Sibling programs are programs that are directly contained in the same program.

With the exception of the COPY and REPLACE statements and the end program marker, the statements, entries, paragraphs, and sections of a COBOL source program are grouped into the following four divisions:

• IDENTIFICATION DIVISION
• ENVIRONMENT DIVISION
• DATA DIVISION
• PROCEDURE DIVISION

The end of a COBOL source program is indicated by the END PROGRAM marker. If there are no nested programs, the absence of additional source program lines also indicates the end of a COBOL program.

The following format is for the entries and statements that constitute a separately compiled COBOL source program.
A sequence of separate COBOL programs can also be input to the compiler. The following format is for the entries and statements that constitute a sequence of source programs (batch compile).

**Format: sequence of COBOL source programs**

```
COBOL-source-program

END PROGRAM program-name
```

**END PROGRAM program-name**

An end program marker separates each program in the sequence of programs. `program-name` must be identical to a program-name declared in a preceding program-ID paragraph.

`program-name` can be specified either as a user-defined word or in an alphanumeric literal. Either way, `program-name` must follow the rules for forming program-names. `program-name` cannot be a figurative constant. Any lowercase letters in the literal are folded to uppercase.

An end program marker is optional for the last program in the sequence only if that program does not contain any nested source programs.

### Nested programs

A COBOL program can contain other COBOL programs, which in turn can contain still other COBOL programs. These contained programs are called *nested programs*. Nested programs can be *directly* or *indirectly* contained in the containing program.

Nested programs are not supported for programs compiled with the THREAD option.

In the following code fragment, program `Outer-program` *directly* contains program `Inner-1`. Program `Inner-1` *directly* contains program `Inner-1a`, and `Outer-program` *indirectly* contains `Inner-1a`:

```
Id division.
Program-id. Outer-program.
  Procedure division.
    Call "Inner-1".
    Stop run.
Id division.
  Program-id. Inner-1
    ...
    Call Inner-1a.
    Stop run.
Id division.
  Program-id. Inner-1a.
    ...
  End Inner-1a.
  End Inner-1.
End Outer-program.
```

The following figure describes a more complex nested program structure with directly and indirectly contained programs.
Conventions for program-names

The program-name of a program is specified in the PROGRAM-ID paragraph of the program’s IDENTIFICATION DIVISION. A program-name can be referenced only by the CALL statement, the CANCEL statement, the SET statement, or the END PROGRAM marker.

Names of programs that constitute a run unit are not necessarily unique, but when two programs in a run unit are identically named, at least one of the programs must be directly or indirectly contained within another separately compiled program that does not contain the other of those two programs.

A separately compiled program and all of its directly and indirectly contained programs must have unique program-names within that separately compiled program.

Rules for program-names

The following rules define the scope of a program-name:

- If the program-name is that of a program that does not possess the COMMON attribute and that program is directly contained within another program, that program-name can be referenced only by statements included in that containing program.

- If the program-name is that of a program that does possess the COMMON attribute and that program is directly contained within another program, that program-name can be referenced only by statements included in the containing program and any programs directly or indirectly contained within that containing program, except that program possessing the COMMON attribute and any programs contained within it.

- If the program-name is that of a program that is separately compiled, that program-name can be referenced by statements included in any other program in the run unit, except programs it directly or indirectly contains.

The mechanism used to determine which program to call is as follows:
• If one of two programs that have the same name as that specified in the CALL statement is directly contained within the program that includes the CALL statement, that program is called.

• If one of two programs that have the same name as that specified in the CALL statement possesses the COMMON attribute and is directly contained within another program that directly or indirectly contains the program that includes the CALL statement, that common program is called unless the calling program is contained within that common program.

• Otherwise, the separately compiled program is called.

The following rules apply to referencing a program-name of a program that is contained within another program. For this discussion, Program-A contains Program-B and Program-C; Program-C contains Program-D and Program-F; and Program-D contains Program-E.

If Program-D does not possess the COMMON attribute, then Program-D can be referenced only by the program that directly contains Program-D, that is, Program-C.

If Program-D does possess the COMMON attribute, then Program-D can be referenced by Program-C (because Program-C contains Program-D) and by any programs contained in Program-C except for programs contained in Program-D. In other words, if Program-D possesses the COMMON attribute, Program-D can be referenced in Program-C and Program-F but not by statements in Program-E, Program-A, or Program-B.
Chapter 11. COBOL class definition structure

Enterprise COBOL provides object-oriented syntax to facilitate interoperability of COBOL and Java programs.

You can use object-oriented syntax to:

- Define classes, with methods and data implemented in COBOL
- Create instances of Java or COBOL classes
- Invoke methods on Java or COBOL objects
- Write classes that inherit from Java classes or from other COBOL classes
- Define and invoke overloaded methods

Basic Java-oriented object capabilities are accessed directly through COBOL language features. Additional capabilities are available to the COBOL programmer by calling services through the Java Native Interface (JNI), as described in Accessing JNI services in the Enterprise COBOL Programming Guide.

Java programs can be multithreaded, and Java interoperability requires tolerance of asynchronous signals. Therefore, to mix COBOL with these Java programs, you must use the thread enablement provided by the THREAD compiler option, as described in THREAD in the Enterprise COBOL Programming Guide.

Java String data is represented at run time in Unicode. The Unicode support provided in Enterprise COBOL with the national data type enables COBOL programs to exchange String data with Java.

The following entities and concepts are used in object-oriented COBOL for Java interoperability:

**Class**

The entity that defines operations and state for zero, one, or more object instances and defines operations and state for a common object (a factory object) that is shared by multiple object instances.

You create object instances using the NEW operand of the COBOL INVOKE statement or using a Java class instance creation expression.

Object instances are automatically freed by the Java runtime system's garbage collection when they are no longer in use. You cannot explicitly free individual objects.

**Instance method**

Procedural code that defines one of the operations supported for the object instances of a class. Instance methods introduced by a COBOL class are defined within the object paragraph of the class definition.

COBOL instance methods are equivalent to public nonstatic methods in Java.

You execute instance methods on a particular object instance by using a COBOL INVOKE statement or a Java method invocation expression.

**Instance data**

Data that defines the state of an individual object instance. Instance data in a COBOL class is defined in the WORKING-STORAGE SECTION of the DATA DIVISION within the object paragraph of a class definition.

COBOL instance data is equivalent to private nonstatic member data in a Java class.

The state of an object also includes the state of the instance data introduced by inherited classes. Each instance object has its own copy of the instance data defined within its class definition and its own copy of the instance data defined in inherited classes.

You can access COBOL object instance data only from within COBOL instance methods defined in the class definition that defines the data.
You can initialize object instance data with VALUE clauses or you can write an instance method to perform custom initialization.

**Factory method, static method**
Procedural code that defines one of the operations supported for the common factory object of the class. COBOL factory methods are defined within the factory paragraph of a class definition. Factory methods are associated with a class, not with any individual instance object of the class.

COBOL factory methods are equivalent to public static methods in Java.

You execute COBOL factory methods from COBOL using an INVOKE statement that specifies the class-name as the first operand. You execute them from a Java program using a static method invocation expression.

A factory method cannot operate directly on instance data of its class, even though the data is described in the same class definition; a factory method must invoke instance methods to act on instance data.

COBOL factory methods are typically used to define customized methods that create object instances. For example, you can code a customized factory method that accepts initial values as parameters, creates an instance object using the NEW operand of the INVOKE statement, and then invokes a customized instance method passing those initial values as arguments for use in initializing the instance object.

**Factory data, static data**
Data associated with a class, rather than with an individual object instance. COBOL factory data is defined in the WORKING-STORAGE SECTION of the DATA DIVISION within the factory paragraph of a class definition.

COBOL factory data is equivalent to private static data in Java.

There is a single copy of factory data for a class. Factory data is associated only with the class and is shared by all object instances of the class. It is not associated with any particular instance object. A factory data item might be used, for example, to keep a count of the number of instance objects that have been created.

You can access COBOL factory data only within COBOL factory methods defined in the same class definition.

**Inheritance**
Inheritance is a mechanism whereby a class definition (the inheriting class) acquires the methods, data descriptions, and file descriptions written in another class definition (the inherited class). When two classes in an inheritance relationship are considered together, the inheriting class is the subclass (or derived class or child class); the inherited class is the superclass (or parent class). The inheriting class also indirectly acquires the methods, data descriptions, and file descriptions that the parent class inherited from its parent class.

A COBOL class must inherit from exactly one parent class, which can be implemented in COBOL or Java.

Every COBOL class must inherit directly or indirectly from the java.lang.Object class.

**Instance variable**
An individual data item defined in the DATA DIVISION of an object paragraph.

**Java Native Interface (JNI)**
A facility of Java designed for interoperation with non-Java programs.

**Java Native Interface (JNI) environment pointer**
A pointer used to obtain the address of the JNI environment structure used for calling JNI services. The COBOL special register JNIENVPTR is provided for referencing the JNI environment pointer.

**Object reference**
A data item that contains information used to identify and reference an individual object. An object reference can refer to an object that is an instance of a Java or COBOL class.
**Subclass**
A class that inherits from another class; also called a *derived class* or *child class* of the inherited class.

**Superclass**
A class that is inherited by another class; also called a *parent class* of the inheriting class.

With the exception of the COPY and REPLACE statements and the END CLASS marker, the statements, entries, paragraphs, and sections of a COBOL class definition are grouped into the following structure:

- IDENTIFICATION DIVISION
- ENVIRONMENT DIVISION (configuration section only)
- Factory definition
  - IDENTIFICATION DIVISION
  - DATA DIVISION
  - PROCEDURE DIVISION (containing one or more method definitions)
- Object definition
  - IDENTIFICATION DIVISION
  - DATA DIVISION
  - PROCEDURE DIVISION (containing one or more method definitions)

The end of a COBOL class definition is indicated by the END CLASS marker.

The following format is for a COBOL class definition.

---

**Format: COBOL class definition**

- IDENTIFICATION DIVISION.
  - CLASS-ID . class-name-1 INHERITS .
  - other-identification-division-content

- ENVIRONMENT DIVISION.
  - class-environment-division-content

- Factory-definition
  - IDENTIFICATION DIVISION.
    - FACTORY.

  - DATA DIVISION.
    - factory-data-division-content

  - PROCEDURE DIVISION.
    - method-definition

  END FACTORY.

- Object-definition

END CLASS class-name-1.

---

Chapter 11. COBOL class definition structure 87
END CLASS
   Specifies the end of a class definition.

END FACTORY
   Specifies the end of a factory definition.

END OBJECT
   Specifies the end of an object definition.
Chapter 12. COBOL method definition structure

A COBOL method definition describes a method. You can specify method definitions only within the factory paragraph and the object paragraph of a class definition.

With the exception of COPY and REPLACE statements and the END METHOD marker, the statements, entries, paragraphs, and sections of a COBOL method definition are grouped into the following four divisions:

• IDENTIFICATION DIVISION
• ENVIRONMENT DIVISION (input-output section only)
• DATA DIVISION
• PROCEDURE DIVISION

The end of a COBOL method definition is indicated by the END METHOD marker.

The following format is for a COBOL method definition.

Format: method definition

```
IDENTIFICATION DIVISION.
  METHOD-ID . method-name-1 .
  other-identification-division-content

ENVIRONMENT DIVISION.
  method-environment-division-content

DATA DIVISION.
  method-data-division-content

method-procedure-division-header.
  method-procedure-division-content

END METHOD . method-name-1 .
```

**METHOD-ID**
Identifies a method definition. See “METHOD-ID paragraph” on page 99 for details.

**method-procedure-division-header**
Indicates the start of the PROCEDURE DIVISION and identifies method parameters and the returning item, if any. See “The PROCEDURE DIVISION header” on page 238 for details.

**END METHOD**
Specifies the end of a method definition.

Methods defined in an object definition are *instance methods*. An instance method in a given class can access:

• Data defined in the DATA DIVISION of the object paragraph of that class (*instance data*)
• Data defined in the DATA DIVISION of that instance method (*method data*)

An instance method cannot directly access instance data defined in a parent class, factory data defined in its own class, or method data defined in another method of its class. It must invoke a method to access such data.
Methods defined in a factory definition are factory methods. A factory method in a given class can access:

- Data defined in the DATA DIVISION of the factory paragraph of that class (factory data)
- Data defined in the DATA DIVISION of that factory method (method data)

A factory method cannot directly access factory data defined in a parent class, instance data defined in its own class, or method data defined in another method of its class. It must invoke a method to access such data.

Methods can be invoked from COBOL programs and methods, and they can be invoked from Java programs. A method can execute an INVOKE statement that directly or indirectly invokes itself. Therefore, COBOL methods are implicitly recursive (unlike COBOL programs, which support recursion only if the RECURSIVE attribute is specified in the program-ID paragraph.)
Part 3. Identification division
Chapter 13. IDENTIFICATION DIVISION

The IDENTIFICATION DIVISION must be the first division in each COBOL source program, factory definition, object definition, and method definition. The identification division names the program, class, or method and identifies the factory definition and object definition. The IDENTIFICATION DIVISION can include the date a program, class, or method was written, the date of compilation, and other such documentary information.

Program IDENTIFICATION DIVISION
For a program, the first paragraph of the IDENTIFICATION DIVISION must be the PROGRAM-ID paragraph. The other paragraphs are optional and can appear in any order.

Class IDENTIFICATION DIVISION
For a class, the first paragraph of the IDENTIFICATION DIVISION must be the CLASS-ID paragraph. The other paragraphs are optional and can appear in any order.

Factory IDENTIFICATION DIVISION
A factory IDENTIFICATION DIVISION contains only a factory paragraph header.

Object IDENTIFICATION DIVISION
An object IDENTIFICATION DIVISION contains only an object paragraph header.

Method IDENTIFICATION DIVISION
For a method, the first paragraph of the IDENTIFICATION DIVISION must be the METHOD-ID paragraph. The other paragraphs are optional and can appear in any order.

The following format is for a program IDENTIFICATION DIVISION.
The following format is for a class IDENTIFICATION DIVISION.
The following format is for a factory IDENTIFICATION DIVISION.

**Format: factory identification division**

```
IDENTIFICATION DIVISION. FACTORY. 
```

The following format is for an object IDENTIFICATION DIVISION.

**Format: object identification division**

```
IDENTIFICATION DIVISION. OBJECT. 
```

The following format is for a method IDENTIFICATION DIVISION.
**Format: method identification division**

```
IDENTIFICATION DIVISION. METHOD-ID method-name-1

AUTHOR

INSTALLATION

DATE-WRITTEN

DATE-compiled

SECURITY

comment-entry
```

**PROGRAM-ID paragraph**

The PROGRAM-ID paragraph specifies the name by which the program is known and assigns selected program attributes to that program. It is required and must be the first paragraph in the IDENTIFICATION DIVISION.

**program-name**

A user-defined word or alphanumeric literal, but not a figurative constant, that identifies your program. It must follow the following rules of formation, depending on the setting of the PGMNAME compiler option:

**PGMNAME(COMPAT)**

- The name can be up to 30 characters in length.
- Only the hyphen, underscore, digits 0-9, and alphabetic characters are allowed in the name when it is specified as a user-defined word.
- At least one character must be alphabetic.
- The hyphen cannot be the first or last character.
If *program-name* is an alphanumeric literal, the rules for the name are the same except that the extension characters $, #, and @ can be included in the name of the outermost program and the underscore can be the first character.

**PGMNAME (LONGUPPER)**

If *program-name* is a user-defined word, it can be up to 30 characters in length.

If *program-name* is an alphanumeric literal, the literal can be up to 160 characters in length. The literal cannot be a figurative constant.

Only the hyphen, underscore, digits 0-9, and alphabetic characters are allowed in the name when the name is specified as a user-defined word.

At least one character must be alphabetic.

The hyphen cannot be the first or last character.

If *program-name* is an alphanumeric literal, the underscore character can be the first character.

External program-names are processed with alphabetic characters folded to uppercase.

**PGMNAME (LONGMIXED)**

*program-name* must be specified as an alphanumeric literal, which can be up to 160 characters in length. The literal cannot be a figurative constant.

*program-name* can consist of any character in the range X'41' to X'FE'.

For information about the PGMNAME compiler option and how the compiler processes the names, see PGMNAME in the Enterprise COBOL Programming Guide.

**RECURSIVE**

An optional clause that allows COBOL programs to be recursively reentered.

You can specify the RECURSIVE clause only on the outermost program of a compilation unit. Recursive programs cannot contain nested subprograms.

If the RECURSIVE clause is specified, *program-name* can be recursively reentered while a previous invocation is still active. If the RECURSIVE clause is not specified, an active program cannot be recursively reentered.

The WORKING-STORAGE SECTION of a recursive program defines storage that is statically allocated and initialized on the first entry to a program and is available in a last-used state to any of the recursive invocations.

The LOCAL-STORAGE SECTION of a recursive program (as well as a nonrecursive program) defines storage that is automatically allocated, initialized, and deallocated on a per-invocation basis.

Internal file connectors that correspond to an FD in the FILE SECTION of a recursive program are statically allocated. The status of internal file connectors is part of the last-used state of a program that persists across invocations.

The following language elements are not supported in a recursive program:

- ALTER
- GO TO without a specified procedure-name
- RERUN
- SEGMENT-LIMIT
- USE FOR DEBUGGING

The RECURSIVE clause is required for programs compiled with the THREAD option.

**COMMON**

Specifies that the program named by *program-name* is contained (that is, nested) within another program and can be called from siblings of the common program and programs contained within
them. The COMMON clause can be used only in nested programs. For more information about
conventions for program names, see “Conventions for program-names” on page 82.

INITIAL
Specifies that when program-name is called, program-name and any programs contained (nested)
within it are placed in their initial state. The initial attribute is not supported for programs compiled
with the THREAD option.

A program is in the initial state:
• The first time the program is called in a run unit
• Every time the program is called, if it possesses the initial attribute
• The first time the program is called after the execution of a CANCEL statement that references the
  program or a CANCEL statement that references a program that directly or indirectly contains the
  program
• The first time the program is called after the execution of a CALL statement that references a
  program that possesses the initial attribute and that directly or indirectly contains the program

When a program is in the initial state:
• The program’s internal data contained in the WORKING-STORAGE SECTION is initialized. If a VALUE
  clause is used in the description of the data item, the data item is initialized to the defined value. If a
  VALUE clause is not associated with a data item, the initial value of the data item is undefined.
• Files with internal file connectors associated with the program are not in the open mode.
• The control mechanisms for all PERFORM statements contained in the program are set to their
  initial states.
• An altered GO TO statement contained in the program is set to its initial state.

For the rules governing nonunique program names, see “Rules for program-names” on page 82.

CLASS-ID paragraph
The CLASS-ID paragraph specifies the name by which the class is known and assigns selected attributes
to that class. The CLASS-ID paragraph is required and must be the first paragraph in a class
IDENTIFICATION DIVISION.

class-name-1
A user-defined word that identifies the class. class-name-1 can optionally have an entry in the
REPOSITORY paragraph of the configuration section of the class definition.

INHERITS
A clause that defines class-name-1 to be a subclass (or derived class) of class-name-2 (the parent
class). class-name-1 cannot directly or indirectly inherit from class-name-1.

class-name-2
The name of a class inherited by class-name-1. You must specify class-name-2 in the REPOSITORY
paragraph of the configuration section of the class definition.

General rules
class-name-1 and class-name-2 must conform to the normal rules of formation for a COBOL user-defined
word, using single-byte characters.

See “REPOSITORY paragraph” on page 116 for details on specifying a class-name that is part of a Java
package or for using non-COBOL naming conventions for class-names.

You cannot include a class definition in a sequence of programs or other class definitions in a single
compilation group. Each class must be specified as a separate source file; that is, a class definition cannot
be included in a batch compile.
Inheritance

Every method available on instances of a class is also available on instances of any subclass directly or indirectly derived from that class.

A subclass can introduce new methods that do not exist in the parent or ancestor class and can override a method from the parent or ancestor class. When a subclass overrides an existing method, it defines a new implementation for that method, which replaces the inherited implementation.

The instance data of class-name-1 is the instance data declared in class-name-2 together with the data declared in the WORKING-STORAGE SECTION of class-name-1. Note, however, that instance data is always private to the class that introduces it.

The semantics of inheritance are as defined by Java. All classes must be derived directly or directly from the java.lang.Object class.

Java supports single inheritance; that is, no class can inherit directly from more than one parent. Only one class-name can be specified in the INHERITS phrase of a class definition.

FACTORY paragraph

The factory IDENTIFICATION DIVISION introduces the factory definition, which is the portion of a class definition that defines the factory object of the class.

A factory object is the single common object that is shared by all object instances of the class. The factory definition contains factory data and factory methods.

OBJECT paragraph

The object IDENTIFICATION DIVISION introduces the object definition, which is the portion of a class definition that defines the instance objects of the class.

The object definition contains object data and object methods.

METHOD-ID paragraph

The METHOD-ID paragraph specifies the name by which a method is known and assigns selected attributes to that method. The METHOD-ID paragraph is required and must be the first paragraph in a method identification division.

method-name-1

An alphanumeric literal or national literal that contains the name of the method. The name must conform to the rules of formation for a Java method name. Method names are used directly, without translation. The method name is processed in a case-sensitive manner.

Method signature

The signature of a method consists of the name of the method and the number and types of the formal parameters to the method as specified in the PROCEDURE DIVISION USING phrase.

Method overloading, overriding, and hiding

COBOL methods can be overloaded, overridden, or hidden, based on the rules of the Java language.

Method overloading

Method names that are defined for a class are not required to be unique. (The set of methods defined for a class includes the methods introduced by the class definition and the methods inherited from parent classes.)
Method names defined for a class must have unique signatures. Two methods defined for a class and that have the same name but different signatures are said to be overloaded.

The type of the method return value, if any, is not included in the method signature.

A class must not define two methods with the same signature but different return value types, or with the same signature but where one method specifies a return value and the other does not.

The rules for overloaded method definitions and resolution of overloaded method invocations are based on the corresponding rules for Java.

**Method overriding (for instance methods)**

An instance method in a subclass overrides an instance method with the same name that is inherited from a parent class if the two methods have the same signature.

When a method overrides an instance method defined in a parent class, the presence or absence of a method return value (the PROCEDURE DIVISION RETURNING data-name) must be consistent in the two methods. Further, when method return values are specified, the return values in the overridden method and the overriding method must have identical data types.

An instance method must not override a factory method in a COBOL parent class, or a static method in a Java parent class.

**Method hiding (for factory methods)**

A factory method is said to hide any and all methods with the same signature in the superclasses of the method definition that would otherwise be accessible. A factory method must not hide an instance method.

---

**Optional paragraphs**

Some optional paragraphs in the IDENTIFICATION DIVISION can be omitted.

The optional paragraphs are:

**AUTHOR**

Name of the author of the program.

**INSTALLATION**

Name of the company or location.

**DATE-WRITTEN**

Date the program was written.

**DATE-COMPILED**

The DATE-COMPILED paragraph provides the compilation date in the source listing. If a comment-entry is specified, the entire entry is replaced with the current date, even if the entry spans lines. If the comment entry is omitted, the compiler adds the current date to the line on which DATE-COMPILED is printed. For example:

```
DATE-COMPILED. 06/30/10.
```

**SECURITY**

Level of confidentiality of the program.

The comment-entry in any of the optional paragraphs can be any combination of characters from the character set of the computer. The comment-entry is written in Area B on one or more lines.

Comment-entries serve only as documentation; they do not affect the meaning of the program. A hyphen in the indicator area (column 7) is not permitted in comment-entries.

You can include DBCS character strings as comment-entries in the IDENTIFICATION DIVISION of your program. Multiple lines are allowed in a comment-entry that contains DBCS character strings.
A DBCS character string must be preceded by a shift-out control character and followed by a shift-in control character. For example:

| AUTHOR | .A.U.T.H.O.R.-.N.A.M.E>, XYZ CORPORATION |
| DATE-WRITTEN | .D.A.T.E |

When a comment-entry that is contained on multiple lines uses DBCS characters, shift-out and shift-in characters must be paired on a line.
Chapter 14. Configuration section

The configuration section is an optional section for programs and classes, and can describe the computer environment on which the program or class is compiled and executed.

**Program configuration section**
The configuration section can be specified only in the ENVIRONMENT DIVISION of the outermost program of a COBOL source program.

You should not specify the configuration section in a program that is contained within another program. The entries specified in the configuration section of a program apply to any program contained within that program.

**Class configuration section**
Specify the configuration section in the ENVIRONMENT DIVISION of a class definition. The repository paragraph can be specified in the ENVIRONMENT DIVISION of a class definition.

Entries in a class configuration section apply to the entire class definition, including all methods introduced by that class.

**Method configuration section**
The input-output section can be specified in a method configuration section. The entries apply only to the method in which the configuration section is specified.

Format:

```
CONFIGURATION SECTION.
  source-computer-paragraph
  object-computer-paragraph
  special-names-paragraph
  repository-paragraph
```

The configuration section can:
- Relate IBM-defined environment-names to user-defined mnemonic names
- Specify the collating sequence
- Specify a currency sign value, and the currency symbol used in the PICTURE clause to represent the currency sign value
- Exchange the functions of the comma and the period in PICTURE clauses and numeric literals
- Relate alphabet-names to character sets or collating sequences
- Specify symbolic characters
- Relate class-names to sets of characters
- Relate object-oriented class names to external class-names and identify class-names that can be used in a class definition or program
- Relate xml-schema-names to ddnames or environment variable names identifying files containing XML schemas

© Copyright IBM Corp. 1991, 2020
The SOURCE-COMPUTER paragraph describes the computer on which the source text is to be compiled.

**Format**

```plaintext
SOURCE-COMPUTER.

computer-name WITH DEBUGGING MODE.
```

**computer-name**

A system-name. For example:

```
IBM-system
```

**WITH DEBUGGING MODE**

Activates a compile-time switch for debugging lines written in the source text.

A *debugging line* is a statement that is compiled only when the compile-time switch is activated. Debugging lines allow you, for example, to check the value of a data-name at certain points in a procedure.

To specify a debugging line in your program, code a D in column 7 (indicator area). You can include successive debugging lines, but each must have a D in column 7, and you cannot break character strings across lines.

All your debugging lines must be written so that the program is syntactically correct, whether the debugging lines are compiled or treated as comments.

The presence or absence of the DEBUGGING MODE clause is logically determined after all COPY and REPLACE statements have been processed.

You can code debugging lines in the ENVIRONMENT DIVISION (after the OBJECT-COMPUTER paragraph), and in the data and procedure divisions.

If a debugging line contains only spaces in Area A and in Area B, the debugging line is treated the same as a blank line.

All of the SOURCE-COMPUTER paragraph is syntax checked, but only the WITH DEBUGGING MODE clause has an effect on the execution of the program.
The **OBJECT-COMPUTER** paragraph specifies the system for which the object program is designated.

**Format**

```
OBJECT-COMPUTER,   .
```

**entry 1**

```
MEMORY SIZE integer.
WORDS CHARACTERS MODULES.
PROGRAM COLLATING SEQUENCE IS alphabet-name.
SEGMENT-LIMIT IS priority-number.
```

**computer-name**

A system-name. For example:

```
IBM-system
```

**MEMORY SIZE integer**

integer specifies the amount of main storage needed to run the object program, in words, characters or modules. The MEMORY SIZE clause is syntax checked but has no effect on the execution of the program.

**PROGRAM COLLATING SEQUENCE IS alphabet-name**

The collating sequence used in this program is the collating sequence associated with the specified alphabet-name.

The collating sequence pertains to this program and to any programs that this program might contain.

PROGRAM COLLATING SEQUENCE determines the truth value of the following alphanumeric comparisons:

- Those explicitly specified in relation conditions
- Those explicitly specified in condition-name conditions

The PROGRAM COLLATING SEQUENCE clause also applies to any merge or sort keys described with usage DISPLAY, unless the COLLATING SEQUENCE phrase is specified in the MERGE or SORT statement.

The PROGRAM COLLATING SEQUENCE clause does not apply to DBCS data items or data items of usage NATIONAL.

If the PROGRAM COLLATING SEQUENCE clause is omitted, the EBCDIC collating sequence is used. (See Appendix C, “EBCDIC and ASCII collating sequences,” on page 579.)
**SEGMENT-LIMIT IS**

The SEGMENT-LIMIT clause is syntax checked but has no effect on the execution of the program.

**priority-number**

An integer ranging from 1 through 49. All sections with priority-numbers 0 through 49 are fixed permanent segments. See “Procedures” on page 242 for a description of priority-numbers and segmentation support.

Segmentation is not supported for programs compiled with the THREAD option.

All of the OBJECT-COMPUTER paragraph is syntax checked, but only the PROGRAM COLLATING SEQUENCE clause has an effect on the execution of the program.

---

**SPECIAL-NAMES paragraph**

The SPECIAL-NAMES paragraph is the name of an ENVIRONMENT DIVISION paragraph in which environment-names are related to user-specified mnemonic-names.

The SPECIAL-NAMES paragraph:

- Relates IBM-specified environment-names to user-defined mnemonic-names
- Relates alphabet-names to character sets or collating sequences
- Specifies symbolic characters
- Relates class names to sets of characters
- Specifies one or more currency sign values and defines a picture symbol to represent each currency sign value in PICTURE clauses
- Specifies that the functions of the comma and decimal point are to be interchanged in PICTURE clauses and numeric literals
- Relates xml-schema-names to ddnames or environment variable names identifying files containing XML schemas

The clauses in the SPECIAL-NAMES paragraph can appear in any order.
Notes:

1 This separator period is optional when no clauses are selected. If you use any clauses, you must code the period after the last clause.
environment-name-1
System devices or standard system actions taken by the compiler.
Valid specifications for environment-name-1 are shown in the following table.

<table>
<thead>
<tr>
<th>environment-name-1</th>
<th>Meaning</th>
<th>Allowed in</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSIN SYSIPT</td>
<td>System logical input unit</td>
<td>ACCEPT</td>
</tr>
<tr>
<td>SYSOUT SYSLIST SYSLST</td>
<td>System logical output unit</td>
<td>DISPLAY</td>
</tr>
<tr>
<td>SYSPUNCH SYSPCH</td>
<td>System punch device</td>
<td>DISPLAY</td>
</tr>
<tr>
<td>CONSOLE</td>
<td>Console</td>
<td>ACCEPT and DISPLAY</td>
</tr>
<tr>
<td>C01 through C12</td>
<td>Skip to channel 1 through channel 12, respectively</td>
<td>WRITE ADVANCING</td>
</tr>
<tr>
<td>CSP</td>
<td>Suppress spacing</td>
<td>WRITE ADVANCING</td>
</tr>
</tbody>
</table>
Table 5. **Meanings of environment names** (continued)

<table>
<thead>
<tr>
<th>environment-name</th>
<th>Meaning</th>
<th>Allowed in</th>
</tr>
</thead>
<tbody>
<tr>
<td>S01 through S05</td>
<td>Pocket select 1 through 5 on punch devices</td>
<td>WRITE ADVANCING</td>
</tr>
<tr>
<td>AFP-5A</td>
<td>Advanced Function Printing</td>
<td>WRITE ADVANCING</td>
</tr>
</tbody>
</table>

**environment-name-2**

A 1-byte user-programmable status indicator (UPSI) switch. Valid specifications for environment-name-2 are UPSI-0 through UPSI-7.

**mnemonic-name-1, mnemonic-name-2**

mnemonic-name-1 and mnemonic-name-2 follow the rules of formation for user-defined names. mnemonic-name-1 can be used in ACCEPT, DISPLAY, and WRITE statements. mnemonic-name-2 can be referenced only in the SET statement. mnemonic-name-2 can qualify condition-1 or condition-2 names.

Mnemonic-names and environment-names need not be unique. If you choose a mnemonic-name that is also an environment-name, its definition as a mnemonic-name will take precedence over its definition as an environment-name.

**ON STATUS IS, OFF STATUS IS**

UPSI switches process special conditions within a program, such as year-beginning or year-ending processing. For example, at the beginning of the PROCEDURE DIVISION, an UPSI switch can be tested; if it is ON, the special branch is taken. (See “Switch-status condition” on page 261.)

**condition-1, condition-2**

Condition-names follow the rules for user-defined names. At least one character must be alphabetic. The value associated with the condition-name is considered to be alphanumeric. A condition-name can be associated with the on status or off status of each UPSI switch specified.

In the PROCEDURE DIVISION, the UPSI switch status is tested through the associated condition-name. Each condition-name is the equivalent of a level-88 item; the associated mnemonic-name, if specified, is considered the conditional variable and can be used for qualification.

Condition-names specified in the SPECIAL-NAMES paragraph of a containing program can be referenced in any contained program.

**ALPHABET clause**

The ALPHABET clause provides a means of relating an alphabet-name to a specified character code set or collating sequence.

The related character code set or collating sequence can be used for alphanumeric data, but not for DBCS or national data.

**ALPHABET alphabet-name-1 IS**

alphabet-name-1 specifies a collating sequence when used in:

- The PROGRAM COLLATING SEQUENCE clause of the object-computer paragraph
- The COLLATING SEQUENCE phrase of the SORT or MERGE statement

alphabet-name-1 specifies a character code set when used in:

- The FD entry CODE-SET clause
- The SYMBOLIC CHARACTERS clause
STANDARD-1
 Specifies the ASCII character set.

STANDARD-2
 Specifies the International Reference Version of ISO/IEC 646, 7-bit coded character set for information interchange.

NATIVE
 Specifies the native character code set. If the ALPHABET clause is omitted, EBCDIC is assumed.

EBCDIC
 Specifies the EBCDIC character set.

literal-1, literal-2, literal-3
 Specifies that the collating sequence for alphanumeric data is determined by the program, according to the following rules:

- The order in which literals appear specifies the ordinal number, in ascending sequence, of the characters in this collating sequence.
- Each numeric literal specified must be an unsigned integer.
- Each numeric literal must have a value that corresponds to a valid ordinal position within the collating sequence in effect.

See Appendix C, “EBCDIC and ASCII collating sequences,” on page 579 for the ordinal numbers for characters in the single-byte EBCDIC and ASCII collating sequences.

- Each character in an alphanumeric literal represents that actual character in the character set. (If the alphanumeric literal contains more than one character, each character, starting with the leftmost, is assigned a successively ascending position within this collating sequence.)
- Any characters that are not explicitly specified assume positions in this collating sequence higher than any of the explicitly specified characters.
- Within one alphabet-name clause, a given character must not be specified more than once.
- Each alphanumeric literal associated with a THROUGH or ALSO phrase must be one character in length.
- When the THROUGH phrase is specified, the contiguous characters in the native character set beginning with the character specified by literal-1 and ending with the character specified by literal-2 are assigned successively ascending positions in this collating sequence.

This sequence can be either ascending or descending within the original native character set. That is, if "Z" THROUGH "A" is specified, the ascending values, left-to-right, for the uppercase letters are:

ZYXWVUTSRQOPONMLKJIHGFEDCBA

- When the ALSO phrase is specified, the characters specified as literal-1, literal-3, ... are assigned to the same position in this collating sequence. For example, if you specify:

"D" ALSO "N" ALSO "%"

the characters D, N, and % are all considered to be in the same position in the collating sequence.

- When the ALSO phrase is specified and alphabet-name-1 is referenced in a SYMBOLIC CHARACTERS clause, only literal-1 is used to represent the character in the character set.
- The character that has the highest ordinal position in this collating sequence is associated with the figurative constant HIGH-VALUE. If more than one character has the highest position because of specification of the ALSO phrase, the last character specified (or defaulted to when any characters are not explicitly specified) is considered to be the HIGH-VALUE character for procedural statements such as DISPLAY and as the sending field in a MOVE statement. (If the
ALSO phrase example given above were specified as the high-order characters of this collating sequence, the HIGH-VALUE character would be %.

• The character that has the lowest ordinal position in this collating sequence is associated with the figurative constant LOW-VALUE. If more than one character has the lowest position because of specification of the ALSO phrase, the first character specified is the LOW-VALUE character. (If the ALSO phrase example given above were specified as the low-order characters of the collating sequence, the LOW-VALUE character would be D.)

When literal-1, literal-2, or literal-3 is specified, the alphabet-name must not be referred to in a CODE-SET clause (see “CODE-SET clause” on page 174).

literal-1, literal-2, and literal-3 must be alphanumeric or numeric literals. All must have the same category. A floating-point literal, a national literal, a DBCS literal, or a symbolic-character figurative constant must not be specified.

**CLASS clause**

The CLASS clause provides a means for relating a name to the specified set of characters listed in that clause.

**CLASS class-name-1 IS**

Provides a means for relating a name to the specified set of characters listed in that clause. class-name-1 can be referenced only in a class condition. The characters specified by the values of the literals in this clause define the exclusive set of characters of which this class consists.

The class-name in the CLASS clause can be a DBCS user-defined word.

**literal-4, literal-5**

Must be category numeric or alphanumeric, and both must be of the same category.

If numeric, literal-4 and literal-5 must be unsigned integers and must have a value that is greater than or equal to 1 and less than or equal to the number of characters in the alphabet specified. Each number corresponds to the ordinal position of each character in the single-byte EBCDIC or ASCII collating sequence.

If alphanumeric, literal-4 and literal-5 are an actual single-byte EBCDIC character.

literal-4 and literal-5 must not specify a symbolic-character figurative constant. If the value of the alphanumeric literal contains multiple characters, each character in the literal is included in the set of characters identified by class-name.

Floating-point literals cannot be used in the CLASS clause.

If the alphanumeric literal is associated with a THROUGH phrase, the literal must be one character in length.

**THROUGH, THRU**

THROUGH and THRU are equivalent. If THROUGH is specified, class-name includes those characters that begin with the value of literal-4 and that end with the value of literal-5. In addition, the characters specified by a THROUGH phrase can be in either ascending or descending order.

**CURRENCY SIGN clause**

The CURRENCY SIGN clause affects numeric-edited data items whose PICTURE character-strings contain a currency symbol.

A currency symbol represents a currency sign value that is:

• Inserted in such data items when they are used as receiving items
• Removed from such data items when they are used as sending items for a numeric or numeric-edited receiver
Typically, currency sign values identify the monetary units stored in a data item. For example: ‘$', 'EUR', 'CHF', 'JPY', 'HK$', 'HKD', or X'9F' (hexadecimal code point in some EBCDIC code pages for €, the Euro currency sign). For details on programming techniques for handling the Euro, see *Using currency signs* in the *Enterprise COBOL Programming Guide*.

The CURRENCY SIGN clause specifies a currency sign value and the currency symbol used to represent that currency sign value in a PICTURE clause.

The SPECIAL-NAMES paragraph can contain multiple CURRENCY SIGN clauses. Each CURRENCY SIGN clause must specify a different currency symbol. Unlike all other PICTURE clause symbols, currency symbols are case sensitive. For example, 'D' and 'd' specify different currency symbols.

**CURRENCY SIGN IS literal-6**

*literal-6* must be an alphanumeric literal. *literal-6* must not be a figurative constant or a null-terminated literal. *literal-6* must not contain a DBCS character.

If the PICTURE SYMBOL phrase is not specified, *literal-6*:

- Specifies both a currency sign value and the currency symbol for this currency sign value
- Must be a single character
- Must not contain any of the following digits or characters:
  - Digits 0 through 9
  - Alphabetic characters A, B, C, D, E, G, N, P, R, S, V, X, Z, their lowercase equivalents, or the space
  - Special characters + - . * / ; ( ) “ = ’ (plus sign, minus sign, comma, period, asterisk, slash, semicolon, left parenthesis, right parenthesis, quotation mark, equal sign, apostrophe)
- Can be one of the following lowercase alphabetic characters: f, h, i, j, k, l, m, o, q, t, u, w, y

If the PICTURE SYMBOL phrase is specified, *literal-6*:

- Specifies a currency sign value. *literal-7* in the PICTURE SYMBOL phrase specifies the currency symbol for this currency sign value.
- Can consist of one or more characters.
- Must not contain any of the following digits or characters:
  - Digits 0 through 9
  - Special characters + - . ,

**PICTURE SYMBOL literal-7**

Specifies a currency symbol that can be used in a PICTURE clause to represent the currency sign value specified by *literal-6*.

*literal-7* must be an alphanumeric literal consisting of one single-byte character. *literal-7* must not contain any of the following digits or characters:

- A figurative constant
- Digits 0 through 9
- Alphabetic characters A, B, C, D, E, G, N, P, R, S, V, X, Z, their lowercase equivalents, or the space
- Special characters + - . * / ; ( ) “ = ’

If the CURRENCY SIGN clause is specified, the CURRENCY and NOCURRENCY compiler options are ignored. 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. For more information about the CURRENCY and NOCURRENCY compiler options, see *CURRENCY* in the *Enterprise COBOL Programming Guide*.
DECIMAL-POINT IS COMMA clause

The DECIMAL-POINT IS COMMA clause exchanges the functions of the period and the comma in PICTURE character-strings and in numeric literals.

SYMBOLIC CHARACTERS clause

The SYMBOLIC CHARACTERS clause is applicable only to single-byte character sets. Each character represented is an alphanumeric character.

SYMBOLIC CHARACTERS symbolic-character-1

Provides a means of specifying one or more symbolic characters. symbolic-character-1 is a user-defined word and must contain at least one alphabetic character. The same symbolic-character can appear only once in a SYMBOLIC CHARACTERS clause. The symbolic character can be a DBCS user-defined word.

The internal representation of symbolic-character-1 is the internal representation of the character that is represented in the specified character set. The following rules apply:

• The relationship between each symbolic-character-1 and the corresponding integer-1 is by their position in the SYMBOLIC CHARACTERS clause. The first symbolic-character-1 is paired with the first integer-1; the second symbolic-character-1 is paired with the second integer-1; and so forth.

• There must be a one-to-one correspondence between occurrences of symbolic-character-1 and occurrences of integer-1 in a SYMBOLIC CHARACTERS clause.

• If the IN phrase is specified, integer-1 specifies the ordinal position of the character that is represented in the character set named by alphabet-name-2. This ordinal position must exist.

• If the IN phrase is not specified, symbolic-character-1 represents the character whose ordinal position in the native character set is specified by integer-1.

Ordinal positions are numbered starting from 1.

XML-SCHEMA clause

The XML-SCHEMA clause provides the means of relating xml-schema-name-1 to an external file identifier: a ddbname or environment variable that identifies the actual external file that contains the optimized XML schema.

The external file identifier can be specified as a user-defined word external-fileid-1 or as an alphanumeric literal literal-8, and identifies an existing external z/OS UNIX file or MVS™ data set that contains the optimized XML schema.

The external file identifier must be either the name specified in the DD statement for the file or the name of an environment variable that contains the file identification information.

For details on specifying an environment variable, see “Environment variable contents for an XML schema file” on page 116.

XML-SCHEMA xml-schema-name-1 IS

xml-schema-name-1 can be referenced only in an XML PARSE statement.

The xml-schema-name in the XML SCHEMA clause can be a DBCS user-defined word.

external-fileid-1

Specifies a user-defined word that must conform to the following rules:

• The user-defined word can contain one to eight characters.

• The user-defined word can contain the characters, A-Z, a-z, 0-9.
The leading character must be alphabetic.

**literal-8**

Specifies an alphanumeric literal that must conform to the following rules:

- The literal can contain one to eight characters.
- The literal can contain the characters, A-Z, a-z, 0-9, @, #, and $.
- The leading character must be alphabetic, @, #, and $.

The compiler folds external-fileid-1 or literal-8 to uppercase to form the ddname or environment variable name for the file.

**Environment variable contents for an XML schema file**

The environment variable name must be defined using only uppercase because the COBOL compiler automatically folds the external file identifier to uppercase.

For an XML schema in an MVS data set, the environment variable must contain a DSN option in the format shown below.

**Format: environment variable for XML schema in an MVS data set, DSN option**

```
DSN(data-set-name)(member-name)
```

- `data-set-name` must be fully qualified. You must not code blanks within the parentheses.
- For an XML schema in a z/OS UNIX file, the environment variable must contain a PATH option in the format shown below.

**Format: environment variable for XML schema in a z/OS UNIX file, PATH option**

```
PATH(path-name)
```

- `path-name` must be an absolute path name; that is, it must begin with a slash. Special characters in the path name must be "escaped" by preceding them with a backslash. For example, to include a backslash in the path name, code two backslashes in sequence.
- For more information about specifying `path-name`, see the description of the PATH parameter in the z/OS MVS JCL Reference.
- For both formats, blanks at the beginning and end of the environment variable contents are ignored. You must not code blanks between a keyword and the left parenthesis that immediately follows the keyword.

**REPOSITORY paragraph**

The REPOSITORY paragraph allows specification of class-names that may be used within the scope of the environment division. It also allows declaration of intrinsic function names that may be used without specifying the keyword FUNCTION.
class-name-1
A user-defined word that identifies the class.

external-class-name-1
An alphanumeric literal containing a name that enables a COBOL program to define or access classes with class-names that are defined using Java rules of formation.

The name must conform to the rules of formation for a fully qualified Java class-name. If the class is part of a Java package, external-class-name-1 must specify the fully qualified name of the package, followed by a period, followed by the simple name of the Java class.


java-array-class-reference
A reference that enables a COBOL program to access a class that represents an array object, where the elements of the array are themselves objects. java-array-class-reference must be an alphanumeric literal with content in the following format:

Format
jobobjectArray : external-class-name-2

jobobjectArray
Specifies a Java object array class.

: A required separator when external-class-name-2 is specified. The colon must not be preceded or followed by space characters.

external-class-name-2
The external class-name of the type of the elements of the array. external-class-name-2 must follow the same rules of formation as external-class-name-1.

When the repository entry specifies jobobjectArray without the colon separator and external-class-name-2, the elements of the object array are of type java.lang.Object.

ALL
If ALL is specified, it is as if all supported Enterprise COBOL intrinsic function names listed in the Table 54 on page 466 were specified.

If ALL is specified, you shall not specify any intrinsic function name as a user-defined word, within the scope of this REPOSITORY paragraph.
**intrinsic-function-name-1**
The name of a supported Enterprise COBOL intrinsic function.

If any intrinsic-function-name-1 is specified more than once in the REPOSITORY paragraph, all specifications for that name shall be identical.

The intrinsic-function-name-1 shall not be specified as a user-defined word within the scope of this REPOSITORY paragraph.

Within the scope of the containing environment division, the intrinsic-function-name-1 may be specified as a function-identifier without being preceded by the keyword FUNCTION, unless specific rules require the use of the keyword FUNCTION.

**Note:** Since WHEN-COMPILED is both a special register and an intrinsic function name, it may not be specified in the FUNCTION clause of the REPOSITORY paragraph.

### General rules
This topic lists the general rules of the REPOSITORY paragraph.

1. All referenced class-names must have an entry in the repository paragraph of the COBOL program or class definition that contains the reference. You can specify a given class-name only once in a given repository paragraph.

2. In program definitions, the repository paragraph can be specified only in the outermost program.

3. The repository paragraph of a COBOL class definition can optionally contain an entry for the name of the class itself, but this entry is not required. Such an entry can be used to specify an external class-name that uses non-COBOL characters or that specifies a fully package-qualified class-name when a COBOL class is to be part of a Java package.

4. Entries in a class repository paragraph apply to the entire class definition, including all methods introduced by that class. Entries in a program repository paragraph apply to the entire program, including its contained programs.

### Identifying and referencing a class
An external-class-name is used to identify and reference a given class from outside the class definition that defines the class.

The external class-name is determined by using the contents of external-class-name-1, external-class-name-2, or class-name-1 (as specified in the repository paragraph of a class), as described below:

1. external-class-name-1 and external-class-name-2 are used directly, without translation. They are processed in a case-sensitive manner.

2. class-name-1 is used if external-class-name-1 or java-array-class-reference is not specified. To create an external name that identifies the class and conforms to Java rules of formation, class-name-1 is processed as follows:
   - The name is converted to uppercase.
   - Hyphens are translated to zero.
   - Underscores are not translated.
   - If the first character of the name is a digit, it is converted as follows:
     - Digits 1 though 9 are changed to A through I.
     - 0 is changed to J.

The class can be implemented in Java or COBOL.

When referencing a class that is part of a Java package, external-class-name-1 must be specified and must give the fully qualified Java class-name.
For example, the repository entry

```
Repository.
Class JavaException is "java.lang.Exception"
```

defines local class-name JavaException for referring to the fully qualified external-class-name "java.lang.Exception."

When defining a COBOL class that is to be part of a Java package, specify an entry in the repository paragraph of that class itself, giving the full Java package-qualified name as the external class-name.
Chapter 15. Input-Output section

The input-output section of the ENVIRONMENT DIVISION contains FILE-CONTROL paragraph and I-O-CONTROL paragraph.

The exact contents of the input-output section depend on the file organization and access methods used. See “ORGANIZATION clause” on page 130 and “ACCESS MODE clause” on page 132.

Program input-output section
The same rules apply to program and method I-O sections.

Class input-output section
The input-output section is not valid for class definitions.

Method input-output section
The same rules apply to program and method I-O sections.

Format: input-output section

INPUT-OUTPUT SECTION. ---- FILE-CONTROL. file-control-paragraph

I-O-CONTROL. file-control-paragraph

FILE-CONTROL
The keyword FILE-CONTROL identifies the file-control paragraph. This keyword can appear only once, at the beginning of the FILE-CONTROL paragraph. It must begin in Area A and be followed by a separator period.

The keyword FILE-CONTROL and the period can be omitted if no file-control-paragraph is specified and there are no files defined in the program.

file-control-paragraph
Names the files and associates them with the external data sets.

Must begin in Area B with a SELECT clause. It must end with a separator period. See “FILE-CONTROL paragraph” on page 122.

file-control-paragraph can be omitted if there are no files defined in the program, even if the FILE-CONTROL keyword is specified.

I-O-CONTROL
The keyword I-O-CONTROL identifies the I-O-CONTROL paragraph.

i-o-control-paragraph
Specifies information needed for efficient transmission of data between the external data set and the COBOL program. The series of entries must end with a separator period. See “I-O-CONTROL paragraph” on page 138.
FILE-CONTROL paragraph

The FILE-CONTROL paragraph associates each file in the COBOL program with an external data set, and specifies file organization, access mode, and other information.

The following formats are for the FILE-CONTROL paragraph:

• Sequential file entries
• Indexed file entries
• Relative file entries
• Line-sequential file entries

The table below lists the different type of files available to programs and methods.

<table>
<thead>
<tr>
<th>File organization</th>
<th>Access method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>QSAM, VSAM&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
<tr>
<td>Relative</td>
<td>VSAM&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
<tr>
<td>Indexed</td>
<td>VSAM&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
<tr>
<td>Line sequential&lt;sup&gt;2&lt;/sup&gt;</td>
<td>Text stream I-O</td>
</tr>
</tbody>
</table>

1. VSAM does not support z/OS UNIX files.
2. Line-sequential support is limited to z/OS UNIX files.

The FILE-CONTROL paragraph begins with the word FILE-CONTROL followed by a separator period. It must contain one and only one entry for each file described in an FD or SD entry in the DATA DIVISION.

Within each entry, the SELECT clause must appear first. The other clauses can appear in any order, except that the PASSWORD clause for indexed files, if specified, must immediately follow the RECORD KEY or ALTERNATE RECORD KEY data-name with which it is associated.

The name component of assignment-name-1 cannot contain an underscore.
Format 1: sequential-file-control-entry

- SELECT
  - OPTIONAL
  - file-name-1
  - ASSIGN
  - TO
  - assignment-name-1

- RESERVE
  - integer
  - AREA
  - AREAS

- ORGANIZATION
  - SEQUENTIAL

- PADDING
  - CHARACTER
  - IS
  - data-name-5
  - literal-2

- RECORD DELIMITER
  - STANDARD-1
  - IS
  - assignment-name-2

- ACCESS
  - MODE
  - IS
  - SEQUENTIAL

- PASSWORD
  - IS
  - data-name-6

- STATUS
  - IS
  - data-name-1
  - data-name-8
Format 2: indexed-file-control-entry

```
SELECT OPTIONAL file-name-1 ASSIGN TO assignment-name-1

RESERVE integer AREA AREAS ORGANIZATION IS INDEXED ACCESS MODE IS SEQUENTIAL RANDOM DYNAMIC

RECORD KEY IS data-name-2

PASSWORD IS data-name-6 ENTRY 1

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

entry 1

ALTERNATE RECORD KEY IS data-name-3

WITH DUPLICATES PASSWORD IS data-name-7
```
Chapter 15. Input-Output section 125
**SELECT clause**

The SELECT clause identifies a file in the COBOL program to be associated with an external data set.

**SELECT OPTIONAL**

Can be specified only for files opened in the input, I-O, or extend mode. You must specify SELECT OPTIONAL for those input files that are not necessarily available each time the object program is executed. For more information, see “OPEN statement notes” on page 373.

**file-name-1**

Must be identified by an FD or SD entry in the DATA DIVISION. A file-name must conform to the rules for a COBOL user-defined name, must contain at least one alphabetic character, and must be unique within this program.

When file-name-1 specifies a sort or a merge file, only the ASSIGN clause can follow the SELECT clause.

If the file connector referenced by file-name-1 is an external file connector, all file-control entries in the run unit that reference this file connector must have the same specification for the OPTIONAL phrase.

**ASSIGN clause**

The ASSIGN clause associates the name of a file in a program with the actual external name of the data file.

**assignment-name-1**

Identifies the external data file. It can be specified as a name or as an alphanumeric literal.

assignment-name-1 is not the name of a data item, and assignment-name-1 cannot be contained in a data item. It is just a character string. It cannot contain an underscore character.

Any assignment-name after the first is syntax checked, but has no effect on the execution of the program.

assignment-name-1 has the following formats:

**Format: assignment-name for QSAM files**

```
label- S- name
```

**Format: assignment-name for VSAM sequential file**

```
label- AS- name
```

**Format: assignment-name for line-sequential, VSAM indexed, or VSAM relative file**

```
label- name
```

**label-**

Documents (for the programmer) the device and device class to which a file is assigned. It must end in a hyphen; the specified value is not otherwise checked. It has no effect on the execution of the program. If specified, it must end with a hyphen.
For QSAM files, the S- (organization) field can be omitted.

For VSAM sequential files, the AS- (organization) field must be specified.

For VSAM indexed and relative files, the organization field must be omitted.

name

A required field that specifies the external name for this file.

It must be either the name specified in the DD statement for this file or the name of an environment variable that contains file allocation information. For details on specifying an environment variable, see “Assignment name for environment variable” on page 127.

name must conform to the following rules of formation:

- If assignment-name-1 is a user-defined word:
  - The name can contain from one to eight characters.
  - The name can contain the characters A-Z, a-z, and 0-9.
  - The leading character must be alphabetic.
  - The name cannot contain an underscore.

- If assignment-name-1 is a literal:
  - The name can contain from one to eight characters.
  - The name can contain the characters A-Z, a-z, 0-9, @, #, and $.
  - The leading character must be alphabetic.
  - The name cannot contain an underscore.

For both user-defined words and literals, the compiler folds name to uppercase to form the ddname for the file.

In a sort or merge file, name is treated as a comment.

If the file connector referenced by file-name-1 in the SELECT clause is an external file connector, all file-control entries in the run unit that reference this file connector must have a consistent specification for assignment-name-1 in the ASSIGN clause. For QSAM files and VSAM indexed and relative files, the name specified on the first assignment-name-1 must be identical. For VSAM sequential files, it must be specified as AS-name.

Assignment name for environment variable

The name component of assignment-name-1 is initially treated as a ddname. If no file has been allocated using this ddname, then name is treated as an environment variable.

The environment variable name must be defined using only uppercase because the COBOL compiler automatically folds the external file-name to uppercase.

If this environment variable exists and contains a valid PATH or DSN option (described below), then the file is dynamically allocated using the information supplied by that option.

If the environment variable does not contain a valid PATH or DSN option or if the dynamic allocation fails, then attempting to open the file results in file status 98.

The contents of the environment variable are checked at each OPEN statement. If a file was dynamically allocated by a previous OPEN statement and the contents of the environment variable have changed since the previous OPEN, then the previous allocation is dynamically deallocated prior to dynamically reallocating the file using the options currently set in the environment variable.

When the run unit terminates, the COBOL runtime system automatically deallocates all automatically generated dynamic allocations.
Environment variable contents for a QSAM file

For a QSAM file, the environment variable must contain either a DSN or a PATH option in the format shown below.

**Format: environment variable for QSAM files, DSN option**

```
DSN( data-set-name )
    (member-name )
    NEW
    OLD
    SHR
    MOD
    TRACKS
    CYL
    SPACE( nnn, mmmmm )
    VOL( volume-serial )
    UNIT( type )
    KEEP
    DELETE
    CATALOG
    UNCATALOG
    STORCLAS( storage-class )
    MGMTCLAS( management-class )
    DATACLAS( data-class )
```

*data-set-name* must be fully qualified. The data set must not be a temporary data set; that is, it must not start with an ampersand.

*After data-set-name or member-name*, the data set attributes can follow in any order.

The options that follow DSN (such as NEW or TRACKS) must be separated by a comma or by one or more blanks.

Blanks at the beginning and end of the environment variable contents are ignored. You must not code blanks within the parentheses or between a keyword and the left parenthesis that immediately follows the keyword.

COBOL does not provide a default for data set disposition (NEW, OLD, SHR, or MOD); however, your operating system might provide one. To avoid unexpected results when opening the file, you should always specify NEW, OLD, SHR, or MOD with the DSN option when you use environment variables for dynamic allocation of QSAM files.

For information about specifying the values of the data set attributes, see the description of the DD statement in the *z/OS MVS JCL Reference*.

**Format: environment variable for QSAM files, PATH option**

```
PATH( path-name )
```

*path-name* must be an absolute path name; that is, it must begin with a slash. For more information about specifying *path-name*, see the description of the PATH parameter in *z/OS MVS JCL Reference*.

Blanks at the beginning and end of the environment variable contents are ignored. You must not code blanks within the parentheses or between a keyword and the left parenthesis that immediately follows the keyword.
Environment variable contents for a line-sequential file

For a line-sequential file, the environment variable must contain a PATH option in the following format:

```
Format: environment variable for line-sequential files
PATH(path-name)
```

`path-name` must be an absolute path name; that is, it must begin with a slash. For more information about specifying `path-name`, see the description of the PATH parameter in z/OS MVS JCL Reference.

Blanks at the beginning and end of the environment variable contents are ignored. You must not code blanks within the parentheses or between a keyword and the left parenthesis that immediately follows the keyword.

Environment variable contents for a VSAM file

For an indexed, relative, or sequential VSAM file, the environment variable must contain a DSN option in the following format:

```
Format: environment variable for VSAM files, DSN option
DSN(data-set-name) OLD SHR
```

`data-set-name` specifies the data set name for the base cluster. `data-set-name` must be fully qualified and must reference an existing predefined and cataloged VSAM data set.

If an indexed file has alternate indexes, then additional environment variables must be defined that contain DSN options (as above) for each of the alternate index paths. The names of these environment variables must follow the same naming convention as used for alternate index ddnames. That is:

- The environment variable name for each alternate index path is formed by concatenating the base cluster environment variable name with an integer, beginning with 1 for the path associated with the first alternate index and incrementing by 1 for the path associated with each successive alternate index. (For example, if the environment variable name for the base cluster is CUST, then the environment variable names for the alternate indexes would be CUST1, CUST2, ..., .)

- If the length of the base cluster environment variable name is already eight characters, then the environment variable names for the alternate indexes are formed by truncating the base cluster portion of the environment variable name on the right to reduce the concatenated result to eight characters. (For example, if the environment variable name for the base cluster is DATAFILE, then the environment variable names for the alternate clusters would be DATAFIL1, DATAFIL2, ..., .)

The options that follow DSN (such as SHR) must be separated by a comma or by one or more blanks. Blanks at the beginning and end of the environment variable contents are ignored. You must not code blanks within the parentheses or between a keyword and the left parenthesis that immediately follows the keyword.

COBOL does not provide a default for data set disposition (OLD or SHR); however, your operating system might provide one. To avoid unexpected results when opening the file, you should always specify OLD or SHR with the DSN option when you use environment variables for dynamic allocation of VSAM files.
**RESERVE clause**

The RESERVE clause allows the user to specify the number of input/output buffers to be allocated at run time for the files.

The RESERVE clause is not supported for line-sequential files.

If the RESERVE clause is omitted, the number of buffers at run time is taken from the DD statement. If none is specified, the system default is taken.

If the file connector referenced by `file-name-1` in the SELECT clause is an external file connector, all file-control entries in the run unit that reference this file connector must have the same value for the integer specified in the RESERVE clause.

**ORGANIZATION clause**

The ORGANIZATION clause identifies the logical structure of the file. The logical structure is established at the time the file is created and cannot subsequently be changed.

You can find a discussion of the different ways in which data can be organized and of the different access methods that you can use to retrieve the data under “File organization and access modes” on page 133.

**ORGANIZATION IS SEQUENTIAL (format 1)**

A predecessor-successor relationship among the records in the file is established by the order in which records are placed in the file when it is created or extended.

**ORGANIZATION IS INDEXED (format 2)**

The position of each logical record in the file is determined by indexes created with the file and maintained by the system. The indexes are based on embedded keys within the file’s records.

**ORGANIZATION IS RELATIVE (format 3)**

The position of each logical record in the file is determined by its relative record number.

**ORGANIZATION IS LINE SEQUENTIAL (format 4)**

A predecessor-successor relationship among the records in the file is established by the order in which records are placed in the file when it is created or extended. A record in a LINE SEQUENTIAL file can consist only of printable characters.

If you omit the ORGANIZATION clause, the compiler assumes ORGANIZATION IS SEQUENTIAL.

If the file connector referenced by `file-name-1` in the SELECT clause is an external file connector, the same organization must be specified for all file-control entries in the run unit that reference this file connector.

**File organization**

You establish the organization of the data when you create a file. Once the file has been created, you can expand the file, but you cannot change the organization.

**Sequential organization**

The physical order in which the records are placed in the file determines the sequence of records. The relationships among records in the file do not change, except that the file can be extended. Records can be fixed length or variable length; there are no keys.

Each record in the file except the first has a unique predecessor record; and each record except the last has a unique successor record.
Indexed organization

Each record in the file has one or more embedded keys (referred to as key data items); each key is associated with an index. An index provides a logical path to the data records according to the contents of the associated embedded record key data items. Indexed files must be direct-access storage files. Records can be fixed length or variable length.

Each record in an indexed file must have an embedded prime key data item. When records are inserted, updated, or deleted, they are identified solely by the values of their prime keys. Thus, the value in each prime key data item must be unique and must not be changed when the record is updated. You tell COBOL the name of the prime key data item in the RECORD KEY clause of the file-control paragraph.

In addition, each record in an indexed file can contain one or more embedded alternate key data items. Each alternate key provides another means of identifying which record to retrieve. You tell COBOL the name of any alternate key data items on the ALTERNATE RECORD KEY clause of the file-control paragraph.

The key used for any specific input-output request is known as the key of reference.

Relative organization

Think of the file as a string of record areas, each of which contains a single record. Each record area is identified by a relative record number; the access method stores and retrieves a record based on its relative record number. For example, the first record area is addressed by relative record number 1 and the 10th is addressed by relative record number 10. The physical sequence in which the records were placed in the file has no bearing on the record area in which they are stored, and thus no effect on each record’s relative record number. Relative files must be direct-access files. Records can be fixed length or variable length.

Line-sequential organization

In a line-sequential file, each record contains a sequence of characters that ends with a record delimiter. The delimiter is not counted in the length of the record.

When a record is written, any trailing blanks are removed prior to adding the record delimiter. The characters in the record area from the first character up to and including the added record delimiter constitute one record and are written to the file.

When a record is read, characters are read one at a time into the record area until:

- The first record delimiter is encountered. The record delimiter is discarded and the remainder of the record is filled with spaces.
- The entire record area is filled with characters. If the first unread character is the record delimiter, it is discarded. Otherwise, the first unread character becomes the first character read by the next READ statement.
- End-of-file is encountered. The remainder of the record area is filled with spaces.

Records written to line-sequential files must consist of data items described as USAGE DISPLAY or DISPLAY-1 or a combination of DISPLAY and DISPLAY-1 items. A zoned decimal data item either must be unsigned or, if signed, must be declared with the SEPARATE CHARACTER phrase.

A line-sequential file can contain printable characters and control characters. Be aware though that if your file contains a newline character (X’15’), the newline character will function as a record delimiter.

The following clauses are not supported for line-sequential files:

- APPLY WRITE-ONLY clause
- CODE-SET clause
- DATA RECORDS clause
- LABEL RECORDS clause
- LINAGE clause
• I-O phrase of the OPEN statement
• PADDING CHARACTER clause
• RECORD CONTAINS 0 clause
• RECORD CONTAINS clause format 2 (for example: RECORD CONTAINS 100 to 200 CHARACTERS)
• RECORD DELIMITER clause
• RECORDING MODE clause
• RERUN clause
• RESERVE clause
• REVERSED phrase of the OPEN statement
• REWRITE statement
• VALUE OF clause of file description entry
• WRITE ... AFTER ADVANCING mnemonic-name
• WRITE ... AT END-OF-PAGE
• WRITE ... BEFORE ADVANCING

### PADDING CHARACTER clause

The PADDING CHARACTER clause specifies a character to be used for block padding on sequential files.

**data-name-5**

Must be defined in the DATA DIVISION as a one-character data item of category alphabetic, alphanumeric, or national, and must not be defined in the FILE SECTION. `data-name-5` can be qualified.

**literal-2**

Must be a one-character alphanumeric literal or national literal.

For external files, `data-name-5`, if specified, must reference an external data item.

The PADDING CHARACTER clause is syntax checked, but has no effect on the execution of the program.

### RECORD DELIMITER clause

The RECORD DELIMITER clause indicates the method of determining the length of a variable-length record on an external medium. It can be specified only for variable-length records.

**STANDARD-1**

If STANDARD-1 is specified, the external medium must be a magnetic tape file.

**assignment-name-2**

Can be any COBOL word.

The RECORD DELIMITER clause is syntax checked, but has no effect on the execution of the program.

### ACCESS MODE clause

The ACCESS MODE clause defines the manner in which the records of the file are made available for processing. If the ACCESS MODE clause is not specified, sequential access is assumed.

For sequentially accessed relative files, the ACCESS MODE clause does not have to precede the RELATIVE KEY clause.

**ACCESS MODE IS SEQUENTIAL**

Can be specified in all formats.
Format 1: sequential
Records in the file are accessed in the sequence established when the file is created or extended. Format 1 supports only sequential access.

Format 2: indexed
Records in the file are accessed in the sequence of ascending record key values according to the collating sequence of the file.

Format 3: relative
Records in the file are accessed in the ascending sequence of relative record numbers of existing records in the file.

Format 4: line-sequential
Records in the file are accessed in the sequence established when the file is created or extended. Format 4 supports only sequential access.

ACCESS MODE IS RANDOM
Can be specified in formats 2 and 3 only.

Format 2: indexed
The value placed in a record key data item specifies the record to be accessed.

Format 3: relative
The value placed in a relative key data item specifies the record to be accessed.

ACCESS MODE IS DYNAMIC
Can be specified in formats 2 and 3 only.

Format 2: indexed
Records in the file can be accessed sequentially or randomly, depending on the form of the specific input-output statement used.

Format 3: relative
Records in the file can be accessed sequentially or randomly, depending on the form of the specific input-output request.

File organization and access modes
File organization is the permanent logical structure of the file. You tell the computer how to retrieve records from the file by specifying the access mode (sequential, random, or dynamic).

For details on the access methods and data organization, see Table 6 on page 122.
Sequently organized data can be accessed only sequentially; however, data that has indexed or relative organization can be accessed in any of the three access modes.

Access modes
See the descriptions of the following types of access modes.

Sequential-access mode
Allows reading and writing records of a file in a serial manner; the order of reference is implicitly determined by the position of a record in the file.

Random-access mode
Allows reading and writing records in a programmer-specified manner; the control of successive references to the file is expressed by specifically defined keys supplied by the user.

Dynamic-access mode
Allows the specific input-output statement to determine the access mode. Therefore, records can be processed sequentially or randomly or both.

For external files, every file-control entry in the run unit that is associated with that external file must specify the same access mode. In addition, for relative file entries, data-name-4 must reference an external data item, and the RELATIVE KEY phrase in each associated file-control entry must reference that same external data item.
### Relationship between data organizations and access modes

This section discusses which access modes are valid for each type of data organization.

**Sequential files**
Files with sequential organization can be accessed only sequentially. The sequence in which records are accessed is the order in which the records were originally written.

**Line-sequential files**
Same as for sequential files (described above).

**Indexed files**
All three access modes are allowed.

In the sequential access mode, the sequence in which records are accessed is the ascending order of the record key value. The order of retrieval within a set of records that have duplicate alternate record key values is the order in which records were written into the set.

In the random access mode, you control the sequence in which records are accessed. A specific record is accessed by placing the value of its key or keys in the RECORD KEY data item (and the ALTERNATE RECORD KEY data item). If a set of records has duplicate alternate record key values, only the first record written is available.

In the dynamic access mode, you can change as needed from sequential access to random access by using appropriate forms of input-output statements.

**Relative files**
All three access modes are allowed.

In the sequential access mode, the sequence in which records are accessed is the ascending order of the relative record numbers of all records that exist within the file.

In the random access mode, you control the sequence in which records are accessed. A specific record is accessed by placing its relative record number in the RELATIVE KEY data item; the RELATIVE KEY must not be defined within the record description entry for the file.

In the dynamic access mode, you can change as needed from sequential access to random access by using appropriate forms of input-output statements.

### RECORD KEY clause

The RECORD KEY clause (format 2) specifies the data item within the record that is the prime RECORD KEY for an indexed file. The values contained in the prime RECORD KEY data item must be unique among records in the file.

**data-name-2**
The prime RECORD KEY data item.

*data-name-2* must be described within a record description entry associated with the file. The key can have any of the following data categories:

- Alphanumeric
- Numeric
- Numeric-edited (with usage DISPLAY or NATIONAL)
- Alphanumeric-edited
- Alphabetic
- External floating-point (with usage DISPLAY or NATIONAL)
- Internal floating-point
- DBCS
- National
- National-edited
Regardless of the category of the key data item, the key is treated as an alphanumeric item. The collation order of the key is determined by the item's binary value order when the key is used for locating a record or for setting the file position indicator associated with the file.

`data-name-2` must not reference a variable-length data item. `data-name-2` can be qualified.

If the indexed file contains variable-length records, `data-name-2` need not be contained within the minimum record size specified for the file. That is, `data-name-2` can exceed the minimum record size, but this is not recommended.

The data description of `data-name-2` and its relative location within the record must be the same as those used when the file was defined.

If the file has more than one record description entry, `data-name-2` need be described in only one of those record description entries. The identical character positions referenced by `data-name-2` in any one record description entry are implicitly referenced as keys for all other record description entries for that file.

For files defined with the EXTERNAL clause, all file description entries in the run unit that are associated with the file must have data description entries for `data-name-2` that specify the same relative location in the record and the same length.

ALTERNATE RECORD KEY clause

The ALTERNATE RECORD KEY clause (format 2) specifies a data item within the record that provides an alternative path to the data in an indexed file.

`data-name-3`

An ALTERNATE RECORD KEY data item.

`data-name-3` must be described within a record description entry associated with the file. The key can have any of the following data categories:

- Alphanumeric
- Numeric
- Numeric-edited (with usage DISPLAY or NATIONAL)
- Alphanumeric-edited
- Alphabetic
- External floating-point (with usage DISPLAY or NATIONAL)
- Internal floating-point
- DBCS
- National
- National-edited
- UTF-8

Regardless of the category of the key data item, the key is treated as an alphanumeric item. The collation order of the key is determined by the item's binary value order when the key is used for locating a record or for setting the file position indicator associated with the file.

`data-name-3` must not reference a group item that contains a variable-occurrence data item. `data-name-3` can be qualified.

If the indexed file contains variable-length records, `data-name-3` need not be contained within the minimum record size specified for the file. That is, `data-name-3` can exceed the minimum record size, but this is not recommended.
If the indexed file contains variable-length records, \textit{data-name-3} need not be contained within the minimum record size specified for the file. That is, \textit{data-name-3} can exceed the minimum record size, but this is not recommended.

The data description of \textit{data-name-3} and its relative location within the record must be the same as those used when the file was defined. The number of alternate record keys for the file must also be the same as that used when the file was created.

The leftmost character position of \textit{data-name-3} must not be the same as the leftmost character position of the prime record key, or of another alternate record key.

If the \textit{DUPLICATES} phrase is not specified, the values contained in the \textit{ALTERNATE RECORD KEY} data item must be unique among records in the file.

If the \textit{DUPLICATES} phrase is specified, the values contained in the \textit{ALTERNATE RECORD KEY} data item can be duplicated within any records in the file. In sequential access, the records with duplicate keys are retrieved in the order in which they were placed in the file. In random access, only the first record written in a series of records with duplicate keys can be retrieved.

For files defined with the \textit{EXTERNAL} clause, all file description entries in the run unit that are associated with the file must have data description entries for \textit{data-name-3} that specify the same relative location in the record and the same length. The file description entries must specify the same number of alternate record keys and the same \textit{DUPLICATES} phrase.

**RELATIVE KEY clause**

The RELATIVE KEY clause (format 3) identifies a data-name that specifies the relative record number for a specific logical record within a relative file.

\textit{data-name-4}

Must be defined as an unsigned integer data item whose description does not contain the PICTURE symbol P. \textit{data-name-4} must not be defined in a record description entry associated with this relative file. That is, the RELATIVE KEY is not part of the record. \textit{data-name-4} can be qualified.

\textit{data-name-4} is required for ACCESS IS SEQUENTIAL only when the START statement is to be used. It is always required for ACCESS IS RANDOM and ACCESS IS DYNAMIC. When the START statement is executed, the system uses the contents of the RELATIVE KEY data item to determine the record at which sequential processing is to begin.

If a value is placed in \textit{data-name-4}, and a START statement is not executed, the value is ignored and processing begins with the first record in the file.

If a relative file is to be referenced by a START statement, you must specify the RELATIVE KEY clause for that file.

For external files, \textit{data-name-4} must reference an external data item, and the RELATIVE KEY phrase in each associated file-control entry must reference that same external data item in each case.

The ACCESS MODE IS RANDOM clause must not be specified for file-names specified in the \textit{USING} or \textit{GIVING} phrase of a \textit{SORT} or \textit{MERGE} statement.

**PASSWORD clause**

The PASSWORD clause controls access to files.

\textit{data-name-6}, \textit{data-name-7}

Password data items. Each must be defined in the \textit{WORKING-STORAGE SECTION} of the \textit{DATA DIVISION} as a data item of category alphabetic, alphanumeric, or alphanumeric-edited. The first eight characters are used as the password; a shorter field is padded with blanks to eight characters. Each password data item must be equivalent to one that is externally defined.
When the PASSWORD clause is specified, at object time the PASSWORD data item must contain a valid password for this file before the file can be successfully opened.

**Format 1 considerations:**
The PASSWORD clause is not valid for QSAM sequential files.

**Format 2 and 3 considerations:**
The PASSWORD clause, if specified, must immediately follow the RECORD KEY or ALTERNATE RECORD KEY data-name with which it is associated.

For indexed files that have been completely predefined to VSAM, only the PASSWORD data item for the RECORD KEY need contain the valid password before the file can be successfully opened at file creation time.

For any other type of file processing (including the processing of dynamic calls at file creation time through a COBOL runtime subroutine), every PASSWORD data item for the file must contain a valid password before the file can be successfully opened, regardless of whether all paths to the data are used in this object program.

For external files, data-name-6 and data-name-7 must reference external data items. The PASSWORD clauses in each associated file-control entry must reference the same external data items.

**FILE STATUS clause**

The FILE STATUS clause monitors the execution of each input-output operation for the file.

When the FILE STATUS clause is specified, the system moves a value into the file status key data item after each input-output operation that explicitly or implicitly refers to this file. The value indicates the status of execution of the statement. (See the file status key description under “Common processing facilities” on page 276.)

**data-name-1**
The file status key data item can be defined in the WORKING-STORAGE, LOCAL-STORAGE, or LINKAGE SECTION as one of the following items:

- A two-character data item of category alphanumeric
- A two-character data item of category national
- A two-digit data item of category numeric with usage DISPLAY or NATIONAL (an external decimal data item)

**data-name-1** must not contain the PICTURE symbol 'P'.

**data-name-1** can be qualified.

The file status key data item must not be variably located; that is, the data item cannot follow a data item that contains an OCCURS DEPENDING ON clause.

**data-name-8**
Must be defined as an alphanumeric group item of 6 bytes in the WORKING-STORAGE SECTION or LINKAGE SECTION of the DATA DIVISION.

Specify **data-name-8** only if the file is a VSAM file (that is, ESDS, KSDS, RRDS).

**data-name-8** holds the 6-byte VSAM return code, which is composed as follows:

- The first 2 bytes of **data-name-8** contain the VSAM return code in binary format. The value for this code is defined (by VSAM) as 0, 8, or 12.
- The next 2 bytes of **data-name-8** contain the VSAM function code in binary format. The value for this code is defined (by VSAM) as 0, 1, 2, 3, 4, or 5.
- The last 2 bytes of **data-name-8** contain the VSAM feedback code in binary format. The code value is 0 through 255.
If VSAM returns a nonzero return code, \textit{data-name-8} is set.

If FILE STATUS is returned without having called VSAM, \textit{data-name-8} is zero.

If \textit{data-name-1} is set to zero, the content of \textit{data-name-8} is undefined. VSAM status return code information is available without transformation in the currently defined COBOL FILE STATUS code. User identification and handling of exception conditions are allowed at the same level as that defined by VSAM.

\textit{Function code} and \textit{feedback code} are set if and only if the \textit{return code} is set to a nonzero value. If they are referenced when the return code is set to zero, the contents of the fields are not dependable.

Values in the \textit{return code}, \textit{function code}, and \textit{feedback code} fields are defined by VSAM. There are no COBOL additions, deletions, or modifications to the VSAM definitions.

For more information, see \textit{DFSMS Macro Instructions for Data Sets}.

**I-O-CONTROL paragraph**

The I-O-CONTROL paragraph of the input-output section specifies when checkpoints are to be taken and the storage areas to be shared by different files. This paragraph is optional in a COBOL program.

The keyword I-O-CONTROL can appear only once, at the beginning of the paragraph. The word I-O-CONTROL must begin in Area A and must be followed by a separator period.

The order in which I-O-CONTROL paragraph clauses are written is not significant. The I-O-CONTROL paragraph ends with a separator period.

**Format: QSAM-i-o-control-entry**

\begin{verbatim}
RERUN ON assignment-name-1 file-name-1 EVERY phrase 1
SAME RECORD AREA FOR file-name-3 file-name-4
MULTIPLE FILE 1 TAPE CONTAINS file-name-5
POSITION integer-2
APPLY WRITE-ONLY 1 ON file-name-2 phrase 1
integer-1 RECORDS file-name-1
END OF REEL UNIT OF file-name-1
\end{verbatim}

**Notes:**

1. The MULTIPLE FILE clause and APPLY WRITE-ONLY clause are not supported for VSAM files.
**RERUN clause**

The RERUN clause specifies that checkpoint records are to be taken. Subject to the restrictions given with each phrase, more than one RERUN clause can be specified.

For information regarding the checkpoint data set definition and the checkpoint method required for complete compliance to the 85 COBOL Standard, see *DD statements for defining checkpoint data sets* in the *Enterprise COBOL Programming Guide*.

Do not use the RERUN clause:

- For files described with the EXTERNAL clause
• In programs with the RECURSIVE clause specified
• In programs compiled with the THREAD option
• In methods

`file-name-1`
Must be a sequentially organized file.

**VSAM and QSAM considerations:**
The file named in the RERUN clause must be a file defined in the same program as the I-O-CONTROL paragraph, even if the file is defined as GLOBAL.

`assignment-name-1`
The external data set for the checkpoint file. It must not be the same assignment-name as that specified in any ASSIGN clause throughout the entire program, including contained and containing programs.

For QSAM files, `assignment-name-1` has the format:

<table>
<thead>
<tr>
<th>Format: assignment-name for QSAM files</th>
</tr>
</thead>
<tbody>
<tr>
<td>label-S-name</td>
</tr>
</tbody>
</table>

The QSAM file must reside on a tape or direct access device. See also Appendix G, “ASCII considerations,” on page 609.

**SORT/MERGE considerations:**
When the RERUN clause is specified in the I-O-CONTROL paragraph, checkpoint records are written at logical intervals determined by the sort/merge program during execution of each SORT or MERGE statement in the program. When the RERUN clause is omitted, checkpoint records are not written.

There can be only one SORT/MERGE I-O-CONTROL paragraph in a program, and it cannot be specified in contained programs. It will have a global effect on all SORT and MERGE statements in the program unit.

**EVERY integer-1 RECORDS**
A checkpoint record is to be written for every `integer-1` records in `file-name-1` that are processed.

When multiple `integer-1 RECORDS` phrases are specified, no two of them can specify the same value for `file-name-1`.

If you specify the `integer-1 RECORDS` phrase, you must specify `assignment-name-1`.

**EVERY END OF REEL/UNIT**
A checkpoint record is to be written whenever end-of-volume for `file-name-1` occurs. The terms REEL and UNIT are interchangeable.

When multiple END OF REEL/UNIT phrases are specified, no two of them can specify the same value for `file-name-1`.

The END OF REEL/UNIT phrase can be specified only if `file-name-1` is a sequentially organized file.

**SAME AREA clause**
The SAME AREA clause is syntax checked, but has no effect on the execution of the program. The SAME AREA clause specifies that two or more files that do not represent sort or merge files are to use the same main storage area during processing.

The files named in a SAME AREA clause need not have the same organization or access.
file-name-3, file-name-4
Must be specified in the file-control paragraph of the same program. file-name-3 and file-name-4 must not reference a file that is defined with the EXTERNAL clause.

- For QSAM files, the SAME clause is treated as documentation.
- For VSAM files, the SAME clause is treated as if equivalent to the SAME RECORD AREA clause.

More than one SAME AREA clause can be included in a program. However:

- A specific file-name must not appear in more than one SAME AREA clause.
- If one or more file-names of a SAME AREA clause appear in a SAME RECORD AREA clause, all the file-names in that SAME AREA clause must appear in that SAME RECORD AREA clause. However, the SAME RECORD AREA clause can contain additional file-names that do not appear in the SAME AREA clause.
- The rule that in the SAME AREA clause only one file can be open at one time takes precedence over the SAME RECORD AREA rule that all the files can be open at the same time.

**SAME RECORD AREA clause**

The SAME RECORD AREA clause specifies that two or more files are to use the same main storage area for processing the current logical record.

The files named in a SAME RECORD AREA clause need not have the same organization or access.

file-name-3, file-name-4
Must be specified in the file-control paragraph of the same program. file-name-3 and file-name-4 must not reference a file that is defined with the EXTERNAL clause.

All of the files can be opened at the same time. A logical record in the shared storage area is considered to be both of the following ones:

- A logical record of each opened output file in the SAME RECORD AREA clause
- A logical record of the most recently read input file in the SAME RECORD AREA clause

More than one SAME RECORD AREA clause can be included in a program. However:

- A specific file-name must not appear in more than one SAME RECORD AREA clause.
- If one or more file-names of a SAME AREA clause appear in a SAME RECORD AREA clause, all the file-names in that SAME AREA clause must appear in that SAME RECORD AREA clause. However, the SAME RECORD AREA clause can contain additional file-names that do not appear in the SAME AREA clause.
- The rule that in the SAME AREA clause only one file can be open at one time takes precedence over the SAME RECORD AREA rule that all the files can be open at the same time.
- If the SAME RECORD AREA clause is specified for several files, the record description entries or the file description entries for these files must not include the GLOBAL clause.
- The SAME RECORD AREA clause must not be specified when the RECORD CONTAINS 0 CHARACTERS clause is specified.

The files named in the SAME RECORD AREA clause need not have the same organization or access.

**SAME SORT AREA clause**

The SAME SORT AREA clause is syntax checked but has no effect on the execution of the program.

file-name-3, file-name-4
Must be specified in the file-control paragraph of the same program. file-name-3 and file-name-4 must not reference a file that is defined with the EXTERNAL clause.

When the SAME SORT AREA clause is specified, at least one file-name specified must name a sort file. Files that are not sort files can also be specified. The following rules apply:
• More than one SAME SORT AREA clause can be specified. However, a given sort file must not be named in more than one such clause.
• If a file that is not a sort file is named in both a SAME AREA clause and in one or more SAME SORT AREA clauses, all the files in the SAME AREA clause must also appear in that SAME SORT AREA clause.
• Files named in a SAME SORT AREA clause need not have the same organization or access.
• Files named in a SAME SORT AREA clause that are not sort files do not share storage with each other unless they are named in a SAME AREA or SAME RECORD AREA clause.
• During the execution of a SORT or MERGE statement that refers to a sort or merge file named in this clause, any nonsort or nonmerge files associated with file-names named in this clause must not be in the open mode.

SAME SORT-MERGE AREA clause

The SAME SORT-MERGE AREA clause is equivalent to the SAME SORT AREA clause.

For more details, see “SAME SORT AREA clause” on page 141.

MULTIPLE FILE TAPE clause

The MULTIPLE FILE TAPE clause (format 1) specifies that two or more files share the same physical reel of tape.

This clause is syntax checked, but has no effect on the execution of the program. The function is performed by the system through the LABEL parameter of the DD statement.

APPLY WRITE-ONLY clause

The APPLY WRITE-ONLY clause optimizes buffer and device space allocation for files that have standard sequential organization, have variable-length records, and are blocked.

If you specify this phrase, the buffer is truncated only when the space available in the buffer is smaller than the size of the next record. Otherwise, the buffer is truncated when the space remaining in the buffer is smaller than the maximum record size for the file.

APPLY WRITE-ONLY is effective only for QSAM files.

file-name-2

Each file must have standard sequential organization.

APPLY WRITE-ONLY clauses must agree among corresponding external file description entries. For an alternate method of achieving the APPLY WRITE-ONLY results, see the description of the compiler option, AWO in the Enterprise COBOL Programming Guide.
Part 5. Data division
Chapter 16. DATA DIVISION overview

This overview describes the structure of the DATA DIVISION for programs, object definitions, factory definitions, and methods.

Each section in the DATA DIVISION has a specific logical function within a COBOL program, object definition, factory definition, or method and can be omitted when that logical function is not needed. If included, the sections must be written in the order shown. The DATA DIVISION is optional.

Program data division
The DATA DIVISION of a COBOL source program describes, in a structured manner, all the data to be processed by the program.

Object data division
The object data division contains data description entries for instance object data (instance data). Instance data is defined in the WORKING-STORAGE SECTION of the object paragraph of a class definition.

Factory data division
The factory data division contains data description entries for factory object data (factory data). Factory data is defined in the WORKING-STORAGE SECTION of the factory paragraph of a class definition.

Method data division
A method data division contains data description entries for data accessible within the method. A method data division can contain a LOCAL-STORAGE SECTION or a WORKING-STORAGE SECTION, or both. The term method data applies to both. Method data in LOCAL-STORAGE is dynamically allocated and initialized on each invocation of the method; method data in WORKING-STORAGE is static and persists across invocations of the method.
FILE SECTION

The FILE SECTION defines the structure of data files. The FILE SECTION must begin with the header FILE SECTION, followed by a separator period.

file-description-entry

Represents the highest level of organization in the FILE SECTION. It provides information about the physical structure and identification of a file, and gives the record-names associated with that file. For the format and the clauses required in a file description entry, see Chapter 17, “DATA DIVISION--file description entries,” on page 161.

record-description-entry

A set of data description entries (described in Chapter 18, “DATA DIVISION--data description entry,” on page 175) that describe the particular records contained within a particular file.

A record in the FILE SECTION must be described as an alphanumeric group item, a national group item, or an elementary data item of class alphabetic, alphanumeric, DBCS, national, or numeric.

More than one record description entry can be specified; each is an alternative description of the same record storage area.

Data areas described in the FILE SECTION are not available for processing unless the file that contains the data area is open.

A method FILE SECTION can define external files only. A single run-unit-level file connector is shared by all programs and methods that contain a definition of a given external file.

WORKING-STORAGE SECTION

The WORKING-STORAGE SECTION describes data records that are not part of data files but are developed and processed by a program or method. The WORKING-STORAGE SECTION also describes data items whose values are assigned in the source program or method and do not change during execution of the object program.

The WORKING-STORAGE SECTION must begin with the section header WORKING-STORAGE SECTION, followed by a separator period.

Program WORKING-STORAGE

The WORKING-STORAGE SECTION for programs (and methods) can also describe external data records, which are shared by programs and methods throughout the run unit. All clauses that are used in record descriptions in the FILE SECTION and also the VALUE and EXTERNAL clauses (which might not be specified in record description entries in the FILE SECTION) can be used in record descriptions in the WORKING-STORAGE SECTION.

Method WORKING-STORAGE

A single copy of the WORKING-STORAGE for a method is statically allocated on the first invocation of the method and persists in a last-used state for the duration of the run unit. The same copy is used whenever the method is invoked regardless of which object instance the method is invoked upon.

If a VALUE clause is specified on a method WORKING-STORAGE data item, the data item is initialized to the VALUE clause value on the first invocation.

If the EXTERNAL clause is specified on a data description entry in a method WORKING-STORAGE SECTION, a single copy of the storage for that data item is allocated once for the duration of the run unit. That storage is shared by all programs and methods in the run unit that contain a definition for the external data item.

Object WORKING-STORAGE

The data described in the WORKING-STORAGE SECTION of an object paragraph is object instance data, usually called instance data. A separate copy of instance data is statically allocated for each
object instance when the object is instantiated. Instance data persists in a last-used state until the object instance is freed by the Java runtime system.

Instance data can be initialized by VALUE clauses specified in data declarations or by logic specified in an instance method.

**Factory WORKING-STORAGE**
The data described in the WORKING-STORAGE SECTION of a factory paragraph is factory data. A single copy of factory data is statically allocated when the factory object for the class is created. Factory data persists in a last-used state for the duration of the run unit.

Factory data can be initialized by VALUE clauses specified in data declarations or by logic specified in a factory method.

**Data storage location**
By default, data items in WORKING-STORAGE are allocated above the 2 GB bar if the LP(64) compiler option is in effect; data items in WORKING-STORAGE are allocated below the 2 GB bar if the LP(32) compiler option is in effect.

For 31-bit programs, you can use the DATA compiler option to control whether WORKING-STORAGE data items are allocated below the 16 MB line (DATA(24)) or try to get storage above the 16 MB line (DATA(31)).

**Note:** All subordinate items belonging to the same group item will be allocated in the same data location.

The WORKING-STORAGE SECTION contains record description entries and data description entries for independent data items, called *data item description entries*.

**record-description-entry**
Data entries in the WORKING-STORAGE SECTION that bear a definite hierarchic relationship to one another must be grouped into records structured by level number. See Chapter 18, “DATA DIVISION--data description entry,” on page 175 for more information.

**data-item-description-entry**
Independent items in the WORKING-STORAGE SECTION that bear no hierarchic relationship to one another need not be grouped into records provided that they do not need to be further subdivided. Instead, they are classified and defined as independent elementary items. Each is defined in a separate data-item description entry that begins with either the level number 77 or 01. See Chapter 18, “DATA DIVISION--data description entry,” on page 175 for more information.

**LOCAL-STORAGE SECTION**
The LOCAL-STORAGE SECTION defines storage that is allocated and freed on a per-invocation basis.

On each invocation, data items defined in the LOCAL-STORAGE SECTION are reallocated. Each data item that has a VALUE clause is initialized to the value specified in that clause.

For nested programs, data items defined in the LOCAL-STORAGE SECTION are allocated upon each invocation of the containing outermost program. However, each data item is reinitialized to the value specified in its VALUE clause each time the nested program is invoked.

For methods, a separate copy of the data defined in LOCAL-STORAGE is allocated and initialized on each invocation of the method. The storage allocated for the data is freed when the method returns.

Data items defined in the LOCAL-STORAGE SECTION cannot specify the EXTERNAL clause.

The LOCAL-STORAGE SECTION must begin with the header LOCAL-STORAGE SECTION, followed by a separator period.

You can specify the LOCAL-STORAGE SECTION in recursive programs, in nonrecursive programs, and in methods.
Method LOCAL-STORAGE content is the same as program LOCAL-STORAGE content except that the GLOBAL clause has no effect (because methods cannot be nested).

**LINKAGE SECTION**

The LINKAGE SECTION describes data made available from another program or method.

- **record-description-entry**
  See “WORKING-STORAGE SECTION” on page 147 for a description.

- **data-item-description-entry**
  See “WORKING-STORAGE SECTION” on page 147 for a description.

Record description entries and data item description entries in the LINKAGE SECTION provide names and descriptions, but storage within the program or method is not reserved because the data area exists elsewhere.

Any data description clause, except for the EXTERNAL clause, can be used to describe items in the LINKAGE SECTION.

You can specify the GLOBAL clause in the LINKAGE SECTION. The GLOBAL clause has no effect for methods, however.

**Data units**

Data is grouped into the conceptual units as listed in the topic.

- File data
- Program data
- Method data
- Factory data
- Instance data

**File data**

File data is contained in files. A file is a collection of data records that exist on some input-output device. A file can be considered as a group of physical records; it can also be considered as a group of logical records. The DATA DIVISION describes the relationship between physical and logical records.

For more information, see “FILE SECTION” on page 166.

A physical record is a unit of data that is treated as an entity when moved into or out of storage. The size of a physical record is determined by the particular input-output device on which it is stored. The size does not necessarily have a direct relationship to the size or content of the logical information contained in the file.

A logical record is a unit of data whose subdivisions have a logical relationship. A logical record can itself be a physical record (that is, be contained completely within one physical unit of data); several logical records can be contained within one physical record, or one logical record can extend across several physical records.

File description entries specify the physical aspects of the data (such as the size relationship between physical and logical records, the size and names of the logical records, labeling information, and so forth).

Record description entries describe the logical records in the file (including the category and format of data within each field of the logical record), different values the data might be assigned, and so forth.

After the relationship between physical and logical records has been established, only logical records are made available to you. For this reason, a reference in this information to "records" means logical records, unless the term "physical records" is used.
Program data
Program data is created by a program instead of being read from a file.

The concept of logical records applies to program data as well as to file data. Program data can thus be grouped into logical records, and be defined by a series of record description entries. Items that need not be so grouped can be defined in independent data description entries (called data item description entries).

Method data
Method data is defined in the DATA DIVISION of a method and is processed by the procedural code in that method. Method data is organized into logical records and independent data description entries in the same manner as program data.

Factory data
Factory data is defined in the DATA DIVISION in the factory paragraph of a class definition and is processed by procedural code in the factory methods of that class. Factory data is organized into logical records and independent data description entries in the same manner as program data.

There is one factory object for a given class in a run unit, and therefore only one instance of factory data in a run unit for that class.

Instance data
Instance data is defined in the DATA DIVISION in the object paragraph of a class definition and is processed by procedural code in the instance methods of that class. Instance data is organized into logical records and independent data description entries in the same manner as program data.

There is one copy of instance data in each object instance of a given class. There can be many object instances for a given class. Each has its own separate copy of instance data.

Data relationships
The relationships among all data to be used in a program are defined in the DATA DIVISION through a system of level indicators and level-numbers.

A level indicator, with its descriptive entry, identifies each file in a program. Level indicators represent the highest level of any data hierarchy with which they are associated. FD is the file description level indicator and SD is the sort-merge file description level indicator.

A level-number, with its descriptive entry, indicates the properties of specific data. Level-numbers can be used to describe a data hierarchy; they can indicate that this data has a special purpose. Although they can be associated with (and subordinate to) level indicators, they can also be used independently to describe internal data or data common to two or more programs. (See “Level-numbers” on page 176 for level-number rules.)

Levels of data
After a record has been defined, it can be subdivided to provide more detailed data references.

For example, in a customer file for a department store, one complete record could contain all data that pertains to one customer. Subdivisions within that record could be, for example, customer name, customer address, account number, department number of sale, unit amount of sale, dollar amount of sale, previous balance, and other pertinent information.

The basic subdivisions of a record (that is, those fields not further subdivided) are called elementary items. Thus a record can be made up of a series of elementary items or can itself be an elementary item.

It might be necessary to refer to a set of elementary items; thus, elementary items can be combined into group items. Groups can also be combined into a more inclusive group that contains one or more subgroups. Thus within one hierarchy of data items, an elementary item can belong to more than one group item.
A system of level-numbers specifies the organization of elementary and group items into records. Special level-numbers are also used to identify data items used for special purposes.

**Levels of data in a record description entry**

Each group and elementary item in a record requires a separate entry, and each must be assigned a level-number.

A level-number is a one-digit or two-digit integer between 01 and 49, or one of three special level-numbers: 66, 77, or 88. The following level-numbers are used to structure records:

**01**

This level-number specifies the record itself, and is the most inclusive level-number possible. A level-01 entry can be an alphanumeric group item, a national group item, or an elementary item. The level number must begin in Area A.

**02 through 49**

These level-numbers specify group and elementary items within a record. They can begin in Area A or Area B. Less inclusive data items are assigned higher (not necessarily consecutive) level-numbers in this series.

The relationship between level-numbers within a group item defines the hierarchy of data within that group.

A group item includes all group and elementary items that follow it until a level-number less than or equal to the level-number of that group is encountered.

The following figure illustrates a group wherein all groups immediately subordinate to the level-01 entry have the same level-number.

The COBOL record description entry written as follows:

```cobol
01 RECORD-ENTRY.
   05 GROUP-1.
      10 SUBGROUP-1.
         15 ELEM-1 PIC... .
         15 ELEM-2 PIC... .
      10 SUBGROUP-2.
         15 ELEM-3 PIC... .
         15 ELEM-4 PIC... .
   05 GROUP-2.
      15 SUBGROUP-3.
         25 ELEM-5 PIC... .
         25 ELEM-6 PIC... .
      15 SUBGROUP-4 PIC... .
   05 ELEM-7 PIC... .
```

is subdivided as indicated below:

```
+-----------------+-----------------+-----------------+
| This entry includes | This entry includes | This entry includes |
+-----------------+-----------------+-----------------+
| This entry includes | This entry includes | This entry includes |
+-----------------+-----------------+-----------------+
| This entry includes | This entry includes | This entry includes |
+-----------------+-----------------+-----------------+
| This entry includes itself. | This entry includes itself. | |
```

The storage arrangement of the record description entry is illustrated below:

```
+-----------------+-----------------+-----------------+
| GROUP 1          | GROUP 2          | SUBGROUP-3       |
+-----------------+-----------------+-----------------+
| SUBGROUP-1      | SUBGROUP-2      | SUBGROUP-3      |
+-----------------+-----------------+-----------------+
| ELEM-1        | ELEM-2        | ELEM-3        | ELEM-4        | ELEM-5        | ELEM-6        | SUBGROUP-4   | ELEM-7       |
```

Chapter 16. DATA DIVISION overview 151
You can also define groups with subordinate items that have different level-numbers for the same level in the hierarchy. For example, 05 EMPLOYEE-NAME and 04 EMPLOYEE-ADDRESS in EMPLOYEE-RECORD below define the same level in the hierarchy. The compiler renumbers the levels in a relative fashion, as shown in MAP output.

```
01 EMPLOYEE-RECORD.
  05 EMPLOYEE-NAME.
    10 FIRST-NAME PICTURE X(10).
    10 LAST-NAME PICTURE X(10).
  04 EMPLOYEE-ADDRESS.
    08 STREET PICTURE X(10).
    08 CITY PICTURE X(10).
```

The following record description entry defines the same data hierarchy as the preceding record description entry:

```
01 EMPLOYEE-RECORD.
  02 EMPLOYEE-NAME.
    03 FIRST-NAME PICTURE X(10).
    03 LAST-NAME PICTURE X(10).
  02 EMPLOYEE-ADDRESS.
    03 STREET PICTURE X(10).
    03 CITY PICTURE X(10).
```

Elementary items can be specified at any level within the hierarchy.

**Special level-numbers**

Special level-numbers identify items that do not structure a record.

The special level-numbers are:

- **66**
  - Identifies items that must contain a RENAMES clause; such items regroup previously defined data items. (For details, see “RENAMES clause” on page 210.)

- **77**
  - Identifies data item description entries that are independent WORKING-STORAGE, LOCAL-STORAGE, or LINKAGE SECTION items; they are not subdivisions of other items and are not subdivided themselves. Level-77 items must begin in Area A.

- **88**
  - Identifies any condition-name entry that is associated with a particular value of a conditional variable. (For details, see “VALUE clause” on page 226.)

Level-77 and level-01 entries in the WORKING-STORAGE, LOCAL-STORAGE, or LINKAGE SECTION that are referenced in a program or method must be given unique data-names because level-77 and level-01 entries cannot be qualified. Subordinate data-names that are referenced in the program or method must be either uniquely defined, or made unique through qualification. Unreferenced data-names need not be uniquely defined.

**Indentation**

Successive data description entries can begin in the same column as preceding entries, or can be indented.

Indentation is useful for documentation but does not affect the action of the compiler.

**Classes and categories of group items**

Enterprise COBOL has three types of groups: alphanumeric groups, national groups, and UTF-8 groups.

Groups that do not specify a GROUP-USAGE clause are alphanumeric groups. An alphanumeric group has class and category alphanumeric and is treated as though its usage were DISPLAY, regardless of the representation of the elementary data items that are contained within the group. In many operations, such as moves and compares, alphanumeric groups are treated as though they were elementary items of
category alphanumeric, except that no editing or conversion of data representation takes place. In other operations, such as MOVE CORRESPONDING and ADD CORRESPONDING, the subordinate data items are processed as separate elementary items.

National groups are defined by a GROUP-USAGE clause with the NATIONAL phrase at the group level. All subordinate data items must be explicitly or implicitly described with usage NATIONAL, and subordinate groups must be explicitly or implicitly described with GROUP-USAGE NATIONAL.

Unless stated otherwise, a national group item is processed exactly as though it were an elementary data item of usage national, class and category national, described with PICTURE N(m), where m is the length of the group in national character positions. Because national groups contain only national characters, data is converted as necessary for moves and compares. The compiler ensures proper truncation and padding. In other operations, such as MOVE CORRESPONDING and ADD CORRESPONDING, the subordinate data items are processed as separate elementary items. See “GROUP-USAGE clause” on page 180 for details.

UTF-8 groups are defined by a GROUP-USAGE clause with the UTF-8 phrase at the group level. All subordinate data items must be explicitly or implicitly described with USAGE UTF-8 and must be defined with the BYTE-LENGTH phrase of the PICTURE clause. Subordinate groups must be explicitly or implicitly described with GROUP-USAGE UTF-8.

Unless stated otherwise, a UTF-8 group item is processed exactly as though it were an elementary data item of usage UTF-8, class and category UTF-8, described with PICTURE U BYTE-LENGTH m, where m is the length of the group in bytes. Because UTF-8 groups contain only UTF-8 characters, data is converted as necessary for moves and compares. The compiler ensures proper truncation and padding. In other operations, such as MOVE CORRESPONDING, the subordinate data items are processed as separate elementary items. See “GROUP-USAGE clause” on page 180 for details.

The table below summarizes the classes and categories of group items.

<table>
<thead>
<tr>
<th>Group description</th>
<th>Class of group</th>
<th>Category of group</th>
<th>USAGE of elementary items within a group</th>
<th>USAGE of a group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without a GROUP-USAGE clause</td>
<td>Alphanumeric</td>
<td>Alphanumeric</td>
<td>Any</td>
<td>Treated as DISPLAY when usage is relevant</td>
</tr>
<tr>
<td>With explicit or implicit GROUP-USAGE clause</td>
<td>National</td>
<td>National</td>
<td>NATIONAL</td>
<td>NATIONAL</td>
</tr>
<tr>
<td>With explicit or implicit GROUP-USAGE clause</td>
<td>UTF-8</td>
<td>UTF-8</td>
<td>UTF-8</td>
<td>UTF-8</td>
</tr>
</tbody>
</table>

**Classes and categories of data**

Most data and all literals used in a COBOL program are divided into classes and categories. Data classes are groupings of data categories. Data categories are determined by the attributes of data description entries or function definitions.

For more information about data categories, see “Category descriptions” on page 155.

The following elementary data items do not have a class and category:

- Index data items
- Items described with USAGE POINTER, USAGE FUNCTION-POINTER, USAGE PROCEDURE-POINTER, or USAGE OBJECT REFERENCE
All other types of elementary data items have a class and category as shown in Table 8 on page 154.

A function references an elementary data item and belongs to the data class and category associated with the type of the function, as shown in Table 9 on page 155.

Literals have a class and category as shown in Table 10 on page 155. Figurative constants (except NULL) have a class and category that depends on the literal or value represented by the figurative constant in the context of its use. For details, see “Figurative constants” on page 13.

All group items have a class and category, even if the subordinate elementary items belong to another class and category. For the classification of group items, see “Classes and categories of group items” on page 152.

<table>
<thead>
<tr>
<th>Class of elementary data items</th>
<th>Category</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alphabetic</td>
<td>Alphabetic</td>
<td>DISPLAY</td>
</tr>
<tr>
<td>Alphanumeric</td>
<td>Alphanumeric</td>
<td>DISPLAY</td>
</tr>
<tr>
<td>Alphanumeric-edited</td>
<td>Alphanumeric-edited</td>
<td>DISPLAY</td>
</tr>
<tr>
<td>Numeric-edited</td>
<td>Numeric-edited</td>
<td>DISPLAY</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class of elementary data items</th>
<th>Category</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timestamp</td>
<td>DISPLAY</td>
<td></td>
</tr>
<tr>
<td>DBCS</td>
<td>DBCS</td>
<td>DISPLAY-1</td>
</tr>
<tr>
<td>National</td>
<td>National</td>
<td>NATIONAL</td>
</tr>
<tr>
<td>National-edited</td>
<td>National-edited</td>
<td>NATIONAL</td>
</tr>
<tr>
<td>Numeric-edited</td>
<td>National-edited</td>
<td>NATIONAL</td>
</tr>
<tr>
<td>UTF-8</td>
<td>UTF-8</td>
<td>UTF-8</td>
</tr>
<tr>
<td>Numeric</td>
<td>Numeric</td>
<td>DISPLAY (type zoned decimal)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NATIONAL (type national decimal)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PACKED-DECIMAL (type internal decimal)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COMP-3 (type internal decimal)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BINARY</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COMP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COMP-4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COMP-5</td>
</tr>
<tr>
<td>Internal floating-point</td>
<td>COMP-1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMP-2</td>
<td></td>
</tr>
<tr>
<td>External floating-point</td>
<td>DISPLAY</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NATIONAL</td>
<td></td>
</tr>
</tbody>
</table>
Table 9. Classes and categories of functions

<table>
<thead>
<tr>
<th>Function type</th>
<th>Class and category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alphanumeric</td>
<td>Alphanumeric</td>
</tr>
<tr>
<td>National</td>
<td>National</td>
</tr>
<tr>
<td>UTF-8</td>
<td>UTF-8</td>
</tr>
<tr>
<td>Integer</td>
<td>Numeric</td>
</tr>
<tr>
<td>Numeric</td>
<td>Numeric</td>
</tr>
</tbody>
</table>

Table 10. Classes and categories of literals

<table>
<thead>
<tr>
<th>Literal</th>
<th>Class and category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alphanumeric (including hexadecimal formats)</td>
<td>Alphanumeric</td>
</tr>
<tr>
<td>DBCS</td>
<td>DBCS</td>
</tr>
<tr>
<td>National (including hexadecimal formats)</td>
<td>National</td>
</tr>
<tr>
<td>UTF-8 (including hexadecimal formats)</td>
<td>UTF-8</td>
</tr>
<tr>
<td>Numeric (fixed-point and floating-point)</td>
<td>Numeric</td>
</tr>
</tbody>
</table>

Category descriptions

The category of a data item is established by the attributes of its data description entry (such as its PICTURE character-string or USAGE clause) or by its function definition.

The meaning of each category is given below.

Alphabetic

A data item is described as category alphabetic by its PICTURE character-string. For PICTURE character-string details, see “Alphabetic items” on page 194.

A data item of category alphabetic is referred to as an alphabetic data item.

Alphanumeric

Each of the following data items is of category alphanumeric:

- An elementary data item described as alphanumeric by its PICTURE character-string. For PICTURE character-string details, see “Alphanumeric items” on page 194.
- An alphanumeric group item.
- An alphanumeric function.
- The following special registers:
  - DEBUG-ITEM
  - SHIFT-OUT
Alphanumeric-edited

A data item is described as category alphanumeric-edited by its PICTURE character-string. For PICTURE character-string details, see “Alphanumeric-edited items” on page 195.
A data item of category alphanumeric-edited is referred to as an alphanumeric-edited data item.

DBCS

A data item is described as category DBCS by its PICTURE character-string and the NSYMBOL(DBCS) compiler option or by an explicit USAGE DISPLAY-1 clause. For PICTURE character-string details, see “DBCS items” on page 195.
A data item of category DBCS is referred to as a DBCS data item.

External floating-point

A data item is described as category external floating-point by its PICTURE character-string. For PICTURE character-string details, see “External floating-point items” on page 196. An external floating-point data item can be described with USAGE DISPLAY or USAGE NATIONAL.
When the usage is DISPLAY, the item is referred to as a display floating-point data item.
When the usage is NATIONAL, the item is referred to as a national floating-point data item.
An external floating-point data item is of class numeric and, unless specifically excluded, is included in a reference to a numeric data item.

Internal floating-point

A data item is described as category internal floating-point by a USAGE clause with the COMP-1 or COMP-2 phrase.
A data item of category internal floating-point is referred to as an internal floating-point data item. An internal floating-point data item is of class numeric and, unless specifically excluded, is included in a reference to a numeric data item.

National

Each of the following data items is of category national:

- A data item that is described as category national by its PICTURE character-string and the NSYMBOL(NATIONAL) compiler option or by an explicit USAGE NATIONAL clause. For PICTURE character-string details, see “National items” on page 197.
- A group item explicitly or implicitly described with a GROUP-USAGE NATIONAL clause.
- A national function.
- The special register XML-NTEXT.

National-edited

A data item is described as category national-edited by its PICTURE character-string. For PICTURE character-string details, see “National-edited items” on page 197.
A data item of category national-edited is referred to as a national-edited data item.

**UTF-8**

Each of the following data items is of category UTF-8:

- A data item is described as category UTF-8 when its PICTURE character-string contains one or more U symbols. The explicit USAGE UTF-8 clause may be present for items containing U in their PICTURE character-string but is not required. For PICTURE character-string details, see “UTF-8 items” on page 200.
- A group item explicitly or implicitly described with a GROUP-USAGE UTF-8 clause.
- A UTF-8 function.

**Numeric**

Each of the following data items is of category numeric:

- An elementary data item described as numeric by its PICTURE character-string and not described with a BLANK WHEN ZERO clause. For PICTURE character-string details, see “Numeric items” on page 198.
- An elementary data item described with one of the following usages:
  - BINARY, COMPUTATIONAL, COMPUTATIONAL-4, COMPUTATIONAL-5, COMP, COMP-4, or COMP-5
  - PACKED-DECIMAL, COMPUTATIONAL-3, or COMP-3
- A special register of numeric type:
  - JSON-CODE
  - JSON-STATUS
  - LENGTH OF
  - LINAGE-COUNTER
  - RETURN-CODE
  - SORTCORE-SIZE
  - SORT-FILE-SIZE
  - SORT-MODE-SIZE
  - SORT-RETURN
  - TALLY
  - XML-CODE
- A numeric function.
- An integer function.

A data item of category numeric is referred to as a numeric data item.

**Numeric-edited**

Each of the following data items is of category numeric-edited:

- A data item described as numeric-edited by its PICTURE character-string. For PICTURE character-string details, see “Numeric-edited items” on page 199.
- A data item described as numeric by its PICTURE character-string and described with a BLANK WHEN ZERO clause.
Alignment rules
The standard alignment rules for positioning data in an elementary item depend on the category of a receiving item.

A receiving item is an item into which the data is moved. For more details about a receiving item, see “Elementary moves” on page 364.

Numeric
For numeric receiving items, the following rules apply:

1. The data is aligned on the assumed decimal point and, if necessary, truncated or padded with zeros. (An assumed decimal point is one that has logical meaning but that does not exist as an actual character in the data.)
2. If an assumed decimal point is not explicitly specified, the receiving item is treated as though an assumed decimal point is specified immediately to the right of the field. The data is then treated according to the preceding rule.

Numeric-edited
The data is aligned on the decimal point, and (if necessary) truncated or padded with zeros at either end except when editing causes replacement of leading zeros.

Internal floating-point
A decimal point is assumed immediately to the left of the field. The data is then aligned on the leftmost digit position that follows the decimal point, with the exponent adjusted accordingly.

External floating-point
The data is aligned on the leftmost digit position; the exponent is adjusted accordingly.

Alphanumeric, alphanumeric-edited, alphabetic, DBCS
For these receiving items, the following rules apply:

1. The data is aligned at the leftmost character position, and (if necessary) truncated or padded with spaces at the right.
2. If the JUSTIFIED clause is specified for this receiving item, the above rule is modified as described in “JUSTIFIED clause” on page 180.

National, national-edited
For these receiving items, the following rules apply:

1. The data is aligned at the leftmost character position, and (if necessary) truncated or padded with default Unicode spaces (NX'0020') at the right. Truncation occurs at the boundary of a national character position.
2. If the JUSTIFIED clause is specified for this receiving item, the above rule is modified as described in “JUSTIFIED clause” on page 180.

UTF-8
For these receiving items, the following rules apply:

1. The data is aligned at the leftmost character position, and (if necessary) truncated or padded with default UTF-8 spaces (UX'20') at the right. Truncation occurs at the boundary of a UTF-8 character position.
2. If the JUSTIFIED clause is specified for this receiving item, the above rule is modified as described in “JUSTIFIED clause” on page 180.

Character-string and item size
For items described with a PICTURE clause, the size of an elementary item is expressed in source code by the number of character positions described in the PICTURE character-string and a SIGN clause (if applicable). Storage size, however, is determined by the actual number of bytes the item occupies as determined by the combination of its PICTURE character-string, SIGN IS SEPARATE clause (if specified), and USAGE clause.
For items described with USAGE DISPLAY (categories alphabetic, alphanumeric, alphanumeric-edited, numeric-edited, numeric, and external floating-point), 1 byte of storage is reserved for each character position described by the item's PICTURE character-string and SIGN IS SEPARATE clause (if applicable).

For items described with USAGE DISPLAY-1 (category DBCS), 2 bytes of storage are reserved for each character position described by the item's PICTURE character-string.

For items described with USAGE NATIONAL (categories national, national-edited, numeric-edited, numeric, and external floating-point), 2 bytes of storage are reserved for each character position described by the item's PICTURE character-string and SIGN IS SEPARATE clause (if specified).

For items described with USAGE UTF-8 (category UTF-8), when the BYTE-LENGTH phrase of the PICTURE clause and the DYNAMIC LENGTH clause are not specified in the item's definition, 4 bytes of storage are reserved for each character position described by the item's PICTURE character-string. Note, however, that due to the varying-length nature of UTF-8 characters, many UTF-8 strings of the specified character length can be represented using fewer bytes than the maximum reserved. In such a case, any unused bytes in the data item are padded with UTF-8 blanks.

For internal floating-point items, the size of the item in storage is determined by its USAGE clause. USAGE COMPUTATIONAL-1 reserves 4 bytes of storage for the item; USAGE COMPUTATIONAL-2 reserves 8 bytes of storage.

Normally, when an arithmetic item is moved from a longer field into a shorter one, the compiler truncates the data to the number of digits represented in the shorter item's PICTURE character-string by truncating leading digits. For example, if a sending field with PICTURE S99999 that contains the value +12345 is moved to a BINARY receiving field with PICTURE S99, the data is truncated to +45. For additional information, see “USAGE clause” on page 218.

The TRUNC compiler option can affect the value of a binary numeric item. For information about TRUNC, see TRUNC in the Enterprise COBOL Programming Guide.

Signed data
There are two categories of algebraic signs used in COBOL: operational signs and editing signs.

Operational signs
Operational signs are associated with signed numeric items, and indicate their algebraic properties.

The internal representation of an algebraic sign depends on the item's USAGE clause, its SIGN clause (if present), and the operating environment. (For further details about the internal representation, see Examples: numeric data and internal representation in the Enterprise COBOL Programming Guide.)

Editing signs
Editing signs are associated with numeric-edited items. Editing signs are PICTURE symbols that identify the sign of the item in edited output.

Dynamic-length items
Enterprise COBOL supports usage of dynamic-length items. Generally, dynamic-length items are items whose logical length might change at runtime.

Dynamic-length elementary items
A dynamic-length item is a data item whose logical length might change at runtime.

If a dynamic-length elementary item is used as a sending operand (which might be reference-modified), the item is treated as a fixed-length data item with a length that is the same as the current length of the dynamic-length elementary item. If a dynamic-length elementary item is used as a receiving operand and is not reference-modified, the sender contents are moved into the receiver's content buffer.

If the length of the sender is longer than the length of the receiver, then a larger receiver content buffer might be allocated at runtime to contain the contents of the sender, before moving any data. The length of the receiver is then set to the length of the sender. If the length of the sender is zero, no data is moved.
and the length of the receiver is set to zero. If the sender is a figurative constant, the length of the operand is as specified in “Figurative constants” on page 13.

If a dynamic-length elementary item is used as a receiving operand and is reference-modified, the item is treated as a fixed-length item with a length that is the same as the current length of the dynamic-length elementary item. The dynamic-length elementary item receiver buffer will not be allocated or reallocated if it is reference-modified.

For items described with the DYNAMIC LENGTH clause, the number of bytes allocated for the item depends on the length of the item at program runtime.

The following statements support dynamic-length elementary items:

- ACCEPT statement - Format 2 (See “System date-related information transfer” on page 284)
- “CALL statement” on page 293
- “CANCEL statement” on page 300
- “DISPLAY statement” on page 307
- “EVALUATE statement” on page 313
- “MOVE statement” on page 363
- “SET statement” on page 401
- “STOP statement” on page 418
- “STRING statement” on page 418
- “UNSTRING statement” on page 426

Dynamic-length group items

A dynamic-length group item is a group item that contains a subordinate dynamic-length elementary item and whose logical length might change at runtime.

A dynamic-length group item's data description entry contains one or more subordinate dynamic-length elementary items. A group item that is not a dynamic-length group item is considered to be a fixed-length group item, such as group items that do not contain subordinate dynamic-length elementary items. Fixed-length group items can contain variable length tables whose data description entry contains the OCCURS DEPENDING ON clause.

Dynamic-length group items may not be compared to, or moved to, other group items (dynamic-length or not).

Fixed-length group items are always compatible and comparable with other fixed-length group items.
Chapter 17. DATA DIVISION--file description entries

In a COBOL program, the File Description (FD) Entry (or Sort File Description (SD) Entry for sort/merge files) represents the highest level of organization in the FILE SECTION. The order in which the optional clauses follow the FD or SD entry is not important.
clause 2

- VARYING
- IN
- SIZE
- FROM integer-6
- TO integer-7
- CHARACTERS
- FOOTING
- AT data-name-6
- integer-9
- LINES
- AT data-name-7
- integer-10
- BOTTOM
- AT data-name-8
- integer-11
Format 2: relative or indexed file description entry

FD file-name-1

EXTERNAL IS
GLOBAL IS

BLOCK CONTAINS integer-1 TO integer-2 CHARACTERS RECORDS

RECORD CONTAINS integer-3 CHARACTERS
CONTAINS integer-4 TO integer-5 CHARACTERS

clause 1 DEPENDING ON data-name-1

LABEL RECORD IS
RECORDS ARE

VALUE OF system-name-1 IS data-name-3 literal-1

DATA RECORD IS
RECORDS ARE

clause 1

VARYING IS IN SIZE FROM integer-6
TO integer-7 CHARACTERS
Format 3: line-sequential file description entry

FD file-name-1
  IS EXTERNAL
  IS GLOBAL

RECORD
  CONTAINS integer-3
  CHARACTERS

clause 1
  DEPENDING ON data-name-1

clause 1
  VARYING
  IN SIZE
  FROM integer-6

  TO integer-7
  CHARACTERS
Format 4: sort/merge file description entry

clause 1

clause 2
FILE SECTION

The FILE SECTION must contain a level-indicator for each input and output file. For all files except sort or merge files, the FILE SECTION must contain an FD entry. For each sort or merge file, the FILE SECTION must contain an SD entry.

file-name

Must follow the level indicator (FD or SD), and must be the same as that specified in the associated SELECT clause. file-name must adhere to the rules of formation for a user-defined word; at least one character must be alphabetic. file-name must be unique within this program.

One or more record description entries must follow file-name. When more than one record description entry is specified, each entry implies a redefinition of the same storage area.

The clauses that follow file-name are optional, and they can appear in any order.

FD (formats 1, 2, and 3)
The last clause in the FD entry must be immediately followed by a separator period.

SD (format 4)
An SD entry must be written for each sort or merge file in the program. The last clause in the SD entry must be immediately followed by a separator period.

The following example illustrates the FILE SECTION entries needed for a sort or merge file:

```
SD  SORT-FILE.
01  SORT-RECORD  PICTURE X(80).
```

A record in the FILE SECTION must be described as an alphanumeric group item, a national group item, or an elementary item of class alphabetic, alphanumeric, DBCS, national, or numeric.

EXTERNAL clause

The EXTERNAL clause specifies that a file connector is external, and permits communication between two programs by the sharing of files.

A file connector is external if the storage associated with that file is associated with the run unit rather than with any particular program within the run unit. An external file can be referenced by any program in the run unit that describes the file. References to an external file from different programs that use separate descriptions of the file are always to the same file. In a run unit, there is only one representative of an external file.

In the FILE SECTION, the EXTERNAL clause can be specified only in file description entries.

The records appearing in the file description entry need not have the same name in corresponding external file description entries. In addition, the number of such records need not be the same in corresponding file description entries.
Use of the EXTERNAL clause does not imply that the associated file-name is a global name. See Sharing data by using the EXTERNAL clause in the Enterprise COBOL Programming Guide for specific information about the use of the EXTERNAL clause.

**GLOBAL clause**

The GLOBAL clause specifies that the file connector named by a file-name is a global name. A global file-name is available to the program that declares it and to every program that is contained directly or indirectly in that program.

A file-name is global if the GLOBAL clause is specified in the file description entry for that file-name. A record-name is global if the GLOBAL clause is specified in the record description entry by which the record-name is declared or, in the case of record description entries in the FILE SECTION, if the GLOBAL clause is specified in the file description entry for the file-name associated with the record description entry. For details on using the GLOBAL clause, see Using data in input and output operations and Scope of names in the Enterprise COBOL Programming Guide.

Two programs in a run unit can reference global file connectors in the following circumstances:

- An external file connector can be referenced from any program that describes that file connector.
- If a program is contained within another program, both programs can refer to a global file connector by referring to an associated global file-name either in the containing program or in any program that directly or indirectly contains the containing program.

**BLOCK CONTAINS clause**

The BLOCK CONTAINS clause specifies the size of the physical records.

The CHARACTERS phrase indicates that the integer specified in the BLOCK CONTAINS clause reflects the number of bytes in the record. For example, if you have a block with 10 DBCS characters or 10 national characters, the BLOCK CONTAINS clause should say BLOCK CONTAINS 20 CHARACTERS.

If the records in the file are not blocked, the BLOCK CONTAINS clause can be omitted. When it is omitted, the compiler assumes that records are not blocked. Even if each physical record contains only one complete logical record, coding BLOCK CONTAINS 1 RECORD would result in fixed blocked records.

The BLOCK CONTAINS clause can be omitted when the associated file-control entry specifies a VSAM file. The concept of blocking has no meaning for VSAM files. The BLOCK CONTAINS clause is syntax checked but has no effect on the execution of the program.

For external files, the value of all BLOCK CONTAINS clauses of corresponding external files must match within the run unit. This conformance is in terms of bytes and does not depend upon whether the value was specified as CHARACTERS or as RECORDS.

**integer-1, integer-2**

Must be unsigned integers. They specify:

CHARACTERS

Specifies the number of bytes required to store the physical record, no matter what USAGE the data items have within the data record.

If only integer-2 is specified, it specifies the exact number of bytes in the physical record. When integer-1 and integer-2 are both specified, they represent the minimum and maximum number of bytes in the physical record, respectively.

integer-1 and integer-2 must include any control bytes and padding contained in the physical record. (Logical records do not include padding.)

The CHARACTERS phrase is the default. CHARACTERS must be specified when:

- The physical record contains padding.
Logical records are grouped so that an inaccurate physical record size could be implied. For example, suppose you describe a variable-length record of 100 bytes, yet each time you write a block of 4, one 50-byte record is written followed by three 100-byte records. If the RECORDS phrase were specified, the compiler would calculate the block size as 420 bytes instead of the actual size, 370 bytes. (This calculation includes block and record descriptors.)

**RECORDS**

Specifies the number of logical records contained in each physical record.

The compiler assumes that the block size must provide for integer-2 records of maximum size, and provides any additional space needed for control bytes.

BLOCK CONTAINS 0 can be specified for QSAM files. If BLOCK CONTAINS 0 is specified for a QSAM file, then:

- The block size is determined at run time from the DD parameters or the data set label of the file. For output data sets, the DCB used by Language Environment will have a zero block size value. When the DCB has a zero block size value, the operating system might select a system-determined block size (SDB). See the operating system specifications for further information about SDB.

- BLOCK CONTAINS can be omitted for SYSIN files and for SYSOUT files. The blocking is determined by the operating system.

For a way to apply BLOCK CONTAINS 0 to QSAM files that do not already have a BLOCK CONTAINS clause, see the description of the compiler option, BLOCK0 in the *Enterprise COBOL Programming Guide.*

The BLOCK CONTAINS clause is syntax checked but has no effect on the execution of the program when specified under an SD.

The BLOCK CONTAINS clause cannot be used with the RECORDING MODE U clause.

**RECORD clause**

When the RECORD clause is used, the record size must be specified as the number of bytes needed to store the record internally, regardless of the USAGE of the data items contained within the record.

For example, if you have a record with 10 DBCS characters, the RECORD clause should say RECORD CONTAINS 20 CHARACTERS. For a record with 10 national characters, the RECORD clause should say the same, RECORD CONTAINS 20 CHARACTERS.

The size of a record is determined according to the rules for obtaining the size of a group item. (See “USAGE clause” on page 218 and “SYNCHRONIZED clause” on page 212.)

When the RECORD clause is omitted, the compiler determines the record lengths from the record descriptions. When one of the entries within a record description contains an OCCURS DEPENDING ON clause, the compiler uses the maximum value of the variable-length item to calculate the number of bytes needed to store the record internally.

If the associated file connector is an external file connector, all file description entries in the run unit that are associated with that file connector must specify the same maximum number of bytes.

The following sections describe the formats of the RECORD clause:

**Format 1**

Format 1 specifies the number of bytes for fixed-length records.
integer-3
Must be an unsigned integer that specifies the number of bytes contained in each record in the file.

The RECORD CONTAINS 0 CHARACTERS clause can be specified for input QSAM files containing fixed-length records; the record size is determined at run time from the DD statement parameters or the data set label. If, at run time, the actual record is larger than the 01 record description, then only the 01 record length is available. If the actual record is shorter, then only the actual record length can be referred to. Otherwise, uninitialized data or an addressing exception can be produced.

Usage note: If the RECORD CONTAINS 0 clause is specified, then the SAME AREA, SAME RECORD AREA, or APPLY WRITE-ONLY clauses cannot be specified.

Do not specify the RECORD CONTAINS 0 clause for an SD entry.

Format 2
Format 2 specifies the number of bytes for either fixed-length or variable-length records.

Fixed-length records are obtained when all 01 record description entry lengths are the same. The format-2 RECORD CONTAINS clause is never required, because the minimum and maximum record lengths are determined from the record description entries.

Format 2

\[
\text{RECORD} \quad \text{CONTAINS} \quad \text{integer-4} \quad \text{TO} \quad \text{integer-5} \quad \text{CHARACTERS}
\]

integer-4, integer-5
Must be unsigned integers. integer-4 specifies the size of the smallest data record, and integer-5 specifies the size of the largest data record.

Format 3
Format 3 is used to specify variable-length records.

Format 3

\[
\text{RECORD} \quad \text{IS} \quad \text{VARYING} \quad \text{IN} \quad \text{SIZE} \\
\text{FROM} \quad \text{integer-6} \quad \text{TO} \quad \text{integer-7} \quad \text{CHARACTERS} \\
\text{DEPENDING} \quad \text{ON} \quad \text{data-name-1}
\]

integer-6
Specifies the minimum number of bytes to be contained in any record of the file. If integer-6 is not specified, the minimum number of bytes to be contained in any record of the file is equal to the least number of bytes described for a record in that file.

integer-7
Specifies the maximum number of bytes in any record of the file. If integer-7 is not specified, the maximum number of bytes to be contained in any record of the file is equal to the greatest number of bytes described for a record in that file.
The number of bytes associated with a record description is determined by the sum of the number of bytes in all elementary data items (excluding redefinitions and renamings), plus any implicit FILLER due to synchronization. If a table is specified:

- The minimum number of table elements described in the record is used in the summation above to determine the minimum number of bytes associated with the record description.
- The maximum number of table elements described in the record is used in the summation above to determine the maximum number of bytes associated with the record description.

If `data-name-1` is specified:

- `data-name-1` must be an elementary unsigned integer.
- The number of bytes in the record must be placed into the data item referenced by `data-name-1` before any RELEASE, REWRITE, or WRITE statement is executed for the file.
- The execution of a DELETE, RELEASE, REWRITE, START, or WRITE statement or the unsuccessful execution of a READ or RETURN statement does not alter the content of the data item referenced by `data-name-1`.
- After the successful execution of a READ or RETURN statement for the file, the contents of the data item referenced by `data-name-1` indicate the number of bytes in the record just read.

During the execution of a RELEASE, REWRITE, or WRITE statement, the number of bytes in the record is determined by the following conditions:

- If `data-name-1` is specified, by the content of the data item referenced by `data-name-1`
- If `data-name-1` is not specified and the record does not contain a variable occurrence data item, by the number of bytes positions in the record
- If `data-name-1` is not specified and the record contains a variable occurrence data item, by the sum of the fixed position and that portion of the table described by the number of occurrences at the time of execution of the output statement

During the execution of a READ ... INTO or RETURN ... INTO statement, the number of bytes in the current record that participate as the sending data items in the implicit MOVE statement is determined by the following conditions:

- If `data-name-1` is specified, by the content of the data item referenced by `data-name-1`
- If `data-name-1` is not specified, by the value that would have been moved into the data item referenced by `data-name-1` had `data-name-1` been specified

**LABEL RECORDS clause**

For sequential, relative, or indexed files, and for sort/merge SDs, the LABEL RECORDS clause is syntax checked, but has no effect on the execution of the program.

The LABEL RECORDS clause documents the presence or absence of labels.

**STANDARD**
- Labels conforming to system specifications exist for this file.
- STANDARD is permitted for mass storage devices and tape devices.

**OMITTED**
- No labels exist for this file.
- OMITTED is permitted for tape devices.

**data-name-2**
- User labels are present in addition to standard labels. `data-name-2` specifies the name of a user label record. `data-name-2` must appear as the subject of a record description entry associated with the file.
VALUE OF clause

The VALUE OF clause describes an item in the label records associated with the file.

**data-name-3**
Should be qualified when necessary, but cannot be subscripted. It must be described in the WORKING-STORAGE SECTION. It cannot be described with the USAGE IS INDEX clause.

**literal-1**
Can be numeric or alphanumeric, or a figurative constant of category numeric or alphanumeric. Cannot be a floating-point literal.

The VALUE OF clause is syntax checked, but has no effect on the execution of the program.

DATA RECORDS clause

The DATA RECORDS clause is syntax checked but serves only as documentation for the names of data records associated with the file.

**data-name-4**
The names of record description entries associated with the file.

The data-name need not have an associated 01 level number record description with the same name.

LINAGE clause

The LINAGE clause specifies the depth of a logical page in number of lines. Optionally, it also specifies the line number at which the footing area begins and the top and bottom margins of the logical page. (The logical page and the physical page cannot be the same size.)

The LINAGE clause is effective for sequential files opened as OUTPUT or EXTEND.

All integers must be unsigned. All data-names must be described as unsigned integer data items.

**data-name-5 , integer-8**
The number of lines that can be written or spaced on this logical page. The area of the page that these lines represent is called the page body. The value must be greater than zero.

**WITH FOOTING AT**

**integer-9** or the value of the data item in **data-name-6** specifies the first line number of the footing area within the page body. The footing line number must be greater than zero, and not greater than the last line of the page body. The footing area extends between those two lines.

**LINES AT TOP**

**integer-10** or the value of the data item in **data-name-7** specifies the number of lines in the top margin of the logical page. The value can be zero.

**LINES AT BOTTOM**

**integer-11** or the value of the data item in **data-name-8** specifies the number of lines in the bottom margin of the logical page. The value can be zero.

The following figure illustrates the use of each phrase of the LINAGE clause.
The logical page size specified in the LINAGE clause is the sum of all values specified in each phrase except the FOOTING phrase. If the LINES AT TOP phrase is omitted, the assumed value for the top margin is zero. Similarly, if the LINES AT BOTTOM phrase is omitted, the assumed value for the bottom margin is zero. Each logical page immediately follows the preceding logical page, with no additional spacing provided.

If the FOOTING phrase is omitted, its assumed value is equal to that of the page body (integer-8 or data-name-5).

At the time an OPEN OUTPUT statement is executed, the values of integer-8, integer-9, integer-10, and integer-11, if specified, are used to determine the page body, first footing line, top margin, and bottom margin of the logical page for this file. (See the figure above.) These values are then used for all logical pages printed for this file during a given execution of the program.

At the time an OPEN statement with the OUTPUT phrase is executed for the file, data-name-5, data-name-6, data-name-7, and data-name-8 determine the page body, first footing line, top margin, and bottom margin for the first logical page only.

At the time a WRITE statement with the ADVANCING PAGE phrase is executed or a page overflow condition occurs, the values of data-name-5, data-name-6, data-name-7, and data-name-8 if specified, are used to determine the page body, first footing line, top margin, and bottom margin for the next logical page.

If an external file connector is associated with this file description entry, all file description entries in the run unit that are associated with this file connector must have:

- A LINAGE clause, if any file description entry has a LINAGE clause
- The same corresponding values for integer-8, integer-9, integer-10, and integer-11, if specified
- The same corresponding external data items referenced by data-name-5, data-name-6, data-name-7, and data-name-8

See “ADVANCING phrase” on page 434 for the behavior of carriage control characters in external files.

A LINAGE clause under an SD is syntax checked, but has no effect on the execution of the program.

**LINAGE-COUNTER special register**

For information about the LINAGE-COUNTER special register, see “LINAGE-COUNTER” on page 20.
RECORDING MODE clause

The RECORDING MODE clause specifies the format of the physical records in a QSAM file. The clause is ignored for a VSAM file.

Permitted values for RECORDING MODE are:

**Recording mode F (fixed)**

All the records in a file are the same length and each is wholly contained within one block. Blocks can contain more than one record, and there is usually a fixed number of records for each block. In this mode, there are no record-length or block-descriptor fields.

**Recording mode V (variable)**

The records can be either fixed-length or variable-length, and each must be wholly contained within one block. Each data record includes a record-length field and each block includes a block-descriptor field. These fields are not described in the DATA DIVISION. They are each 4 bytes long and provision is automatically made for them. These fields are not available to you.

**Recording mode U (fixed or variable)**

The records can be either fixed-length or variable-length. However, there is only one record for each block. There are no record-length or block-descriptor fields.

You cannot use RECORDING MODE U if you are using the BLOCK CONTAINS clause.

**Recording mode S (spanned)**

The records can be either fixed-length or variable-length, and can be larger than a block. If a record is larger than the remaining space in a block, a segment of the record is written to fill the block. The remainder of the record is stored in the next block (or blocks, if required). Only complete records are made available to you. Each segment of a record in a block, even if it is the entire record, includes a segment-descriptor field, and each block includes a block-descriptor field. These fields are not described in the DATA DIVISION; provision is automatically made for them. These fields are not available to you.

When recording mode S is used, the BLOCK CONTAINS CHARACTERS clause must be used. Recording mode S is not allowed for ASCII files.

If the RECORDING MODE clause is not specified for a QSAM file, the Enterprise COBOL compiler determines the recording mode as follows:

**F**

The compiler determines the recording mode to be F if the largest level-01 record associated with the file is not greater than the block size specified in the BLOCK CONTAINS clause, and you do one of the following things:

- Use the RECORD CONTAINS integer clause. (For more information, see the Enterprise COBOL Migration Guide.)
- Omit the RECORD clause and make sure that all level-01 records associated with the file are the same size and none contains an OCCURS DEPENDING ON clause.

**V**

The compiler determines the recording mode to be V if the largest level-01 record associated with the file is not greater than the block size specified in the BLOCK CONTAINS clause, and you do one of the following things:

- Use the RECORD IS VARYING clause.
- Omit the RECORD clause and make sure that all level-01 records associated with the file are not the same size or some contain an OCCURS DEPENDING ON clause.
- Use the RECORD CONTAINS integer-1 TO integer-2 clause, with integer-1 the minimum length and integer-2 the maximum length of the level-01 records associated with the file. The two integers must be different, with values matching minimum and maximum length of either different length records or records with an OCCURS DEPENDING ON clause.
**S**

The compiler determines the recording mode to be S if the maximum block size is smaller than the largest record size.

**U**

Recording mode U is never obtained by default. The RECORDING MODE U clause must be explicitly specified to get recording mode U.

---

**CODE-SET clause**

The CODE-SET clause specifies the character code used to represent data on a magnetic tape file. When the CODE-SET clause is specified, an alphabet-name identifies the character code convention used to represent data on the input-output device.

alphabet-name must be defined in the SPECIAL-NAMES paragraph as STANDARD-1 (for ASCII-encoded files), STANDARD-2 (for ISO 7-bit encoded files), EBCDIC (for EBCDIC-encoded files), or NATIVE. When NATIVE is specified, the CODE-SET clause is syntax checked but has no effect on the execution of the program.

The CODE-SET clause also specifies the algorithm for converting the character codes on the input-output medium from and to the internal EBCDIC character set.

When the CODE-SET clause is specified for a file, all data in the file must have USAGE DISPLAY; and if signed numeric data is present, it must be described with the SIGN IS SEPARATE clause.

When the CODE-SET clause is omitted, the EBCDIC character set is assumed for the file.

If the associated file connector is an external file connector, all CODE-SET clauses in the run unit that are associated with the file connector must have the same character set.

The CODE-SET clause is valid only for magnetic tape files.

The CODE-SET clause is syntax checked but has no effect on the execution of the program when specified under an SD.
Chapter 18. DATA DIVISION--data description entry

A data description entry specifies the characteristics of a data item. In the sections that follow, sets of data description entries are called record description entries. The term data description entry refers to data and record description entries.

Data description entries that define independent data items do not make up a record. These entries are known as data item description entries.

Data description entries have three general formats, and all data description entries must end with a separator period.

Format 1

Format 1 is used for data description entries in all DATA DIVISION sections.

The clauses can be written in any order, with the following exceptions:

• data-name-1 or FILLER, if specified, must immediately follow the level-number.

• When the REDEFINES clause is specified, it must immediately follow data-name-1 or FILLER, if either is specified. If data-name-1 or FILLER is not specified, the REDEFINES clause must immediately follow the level-number.

The level-number in format 1 can be any number in the range 01–49, or 77.

A space, a comma, or a semicolon must separate clauses.
Format 2

Format 2 regroups previously defined items.

**Format 2: renames**

irenames 66 data-name-1 renames-clause.

A level-66 entry cannot rename another level-66 entry, nor can it rename a level-01, level-77, or level-88 entry.

All level-66 entries associated with one record must immediately follow the last data description entry in that record.

See “RENAMES clause” on page 210 for further details.

Format 3

Format 3 describes condition-names.

**Format 3: condition-name**

irenames 88 condition-name-1 value-clause.

**condition-name-1**

A user-specified name that associates a value, a set of values, or a range of values with a conditional variable.

Level-88 entries must immediately follow the data description entry for the conditional variable with which the condition-names are associated.

Format 3 can be used to describe elementary items, national group items, or alphanumeric group items. Additional information about condition-name entries can be found under “VALUE clause” on page 226 and “Condition-name condition” on page 248.

Level-numbers

The level-number specifies the hierarchy of data within a record, and identifies special-purpose data entries. A level-number begins a data description entry, a renamed or redefined item, or a condition-name entry.

A level-number has an integer value between 1 and 49, inclusive, or one of the special level-number values 66, 77, or 88.

**level-number**

01 and 77 must begin in Area A and be followed either by a separator period or by a space followed by its associated data-name, FILLER, or appropriate data description clause.
Level numbers 02 through 49 can begin in Areas A or B and must be followed by a space or a separator period.

Level numbers 66 and 88 can begin in Areas A or B and must be followed by a space.

Single-digit level-numbers 1 through 9 can be substituted for level-numbers 01 through 09.

Successive data description entries can start in the same column as the first entry or can be indented according to the level-number. Indentation does not affect the magnitude of a level-number.

When level-numbers are indented, each new level-number can begin any number of spaces to the right of Area A. The extent of indentation to the right is limited only by the width of Area B.

For more information, see “Levels of data” on page 150.

**data-name-1**

Explicitly identifies the data being described.

*data-name-1*, if specified, identifies a data item used in the program. *data-name-1* must be the first word following the level-number.

The data item can be changed during program execution.

*data-name-1* must be specified for level-66 and level-88 items. It must also be specified for any entry containing the GLOBAL or EXTERNAL clause, and for record description entries associated with file description entries that have the GLOBAL or EXTERNAL clauses.

**FILLER**

A data item that is not explicitly referred to in a program. The keyword FILLER is optional. If specified, FILLER must be the first word following the level-number.

The keyword FILLER can be used with a conditional variable if explicit reference is never made to the conditional variable but only to values that it can assume. FILLER cannot be used with a condition-name.

In a MOVE CORRESPONDING statement or in an ADD CORRESPONDING or SUBTRACT CORRESPONDING statement, FILLER items are ignored.

In an INITIALIZE statement:

- When the FILLER phrase is not specified, elementary FILLER items are ignored.
- When the FILLER phrase is specified, the receiving elementary data items that have an explicit or implicit FILLER clause will be initialized.

If *data-name-1* or the FILLER clause is omitted, the data item being described is treated as though FILLER had been specified.

**BLANK WHEN ZERO clause**

The BLANK WHEN ZERO clause specifies that an item contains only spaces when its value is zero.

The BLANK WHEN ZERO clause may be specified only for an elementary item described by its picture character string as category numeric-edited or numeric, without the picture symbol S or *. These items must be described, either implicitly or explicitly, as USAGE DISPLAY or USAGE NATIONAL.

A BLANK WHEN ZERO clause that is specified for an item defined as numeric by its picture character string defines the item as category numeric-edited.
DYNAMIC LENGTH clause

The DYNAMIC LENGTH clause specifies a dynamic-length elementary item. The LENGTH keyword is optional.

Format

\[
\text{DYNAMIC}\begin{array}{c}
\text{LENGTH} \quad \text{LIMIT} \quad \text{integer-1} \\
\text{IS} \quad \text{integer-1}
\end{array}
\]

LIMIT

If the LIMIT phrase is not specified, then the limit is set to the maximum value according to the data item class. Attempts to receive into a dynamic-length elementary item more characters over the limit will result in truncation at a character boundary on the right.

**Note:** In practice, the maximum length of a dynamic-length elementary item might be limited by available runtime memory or other runtime environment limits.

integer-1

An integer specifies the maximum number of alphanumeric or national characters that the data item can contain.

When the PICTURE clause is PIC U, integer-1 represents the maximum bytes that the data item can contain. integer-1 must be an integer greater than or equal to 1, and less than or equal to 999999999 for alphanumeric class and 999999999 for UTF-8 class dynamic-length elementary items.

A dynamic-length elementary item is a data item whose length can vary at runtime. The length of a data item is the current number of characters contained by this data item.

A dynamic-length elementary item has a minimum length of zero, and a maximum length that is the smallest of the following limitations:

- integer-1 characters of the LIMIT phrase
- 999999999 characters if the PICTURE clause is 'X' (alphanumeric class) or 999999999 bytes if the PICTURE clause is 'U' (UTF-8 class)
- The available runtime memory.

The PICTURE clause of a dynamic-length elementary item must be either PIC X or PIC U, making this a data item of class alphanumeric or UTF-8 respectively. No other PICTURE clause strings other than a single instance of 'X' or 'U' are allowed. PIC N is not supported in a dynamic-length elementary item.

Dynamic-length elementary items can be specified in the WORKING-STORAGE SECTION or LOCAL-STORAGE SECTION.

Dynamic-length elementary items can be specified as subordinate items within a group. The existence of a dynamic-length elementary item within a group item will make it a dynamic-length group item (See “Dynamic-length group items” on page 160).

Dynamic-length elementary items cannot be variably located or a subordinate data item within a table containing the OCCURS DEPENDING ON phrase. When a dynamic-length elementary item appears within a group, the content of the item can be considered logically inline within this group, even if the physical location of the content is remotely located. Comparisons and moves between groups where one or both groups contains a subordinate dynamic-length elementary item is not allowed. Moving a dynamic-length elementary item to a fixed-length group is allowed, and moving any group (dynamic-length or not) to a dynamic-length elementary item is allowed.
**Usage note:** Dynamic-length elementary items in the LOCAL-STORAGE section may not be freed when the program executes unstructured GOTOs such as an EXEC CICS® HANDLE statement or the Language Environment service CEEMRCE (Move resume cursor explicit).

**EXTERNAL clause**

The EXTERNAL clause specifies that the storage associated with a data item is associated with the run unit rather than with any particular program or method within the run unit.

An external data item can be referenced by any program or method in the run unit that describes the data item. References to an external data item from different programs or methods using separate descriptions of the data item are always to the same data item. In a run unit, there is only one representative of an external data item.

The EXTERNAL clause can be specified only on data description entries whose level-number is 01. It can be specified only on data description entries that are in the WORKING-STORAGE SECTION of a program or method. It cannot be specified in LINKAGE SECTION, LOCAL-STORAGE SECTION, or FILE SECTION data description entries. Any data item described by a data description entry subordinate to an entry that describes an external record also attains the external attribute. Indexes in an external data record do not possess the external attribute.

The data contained in the record named by the data-name clause is external and can be accessed and processed by any program or method in the run unit that describes and, optionally, redefines it. This data is subject to the following rules:

- If two or more programs or methods within a run unit describe the same external data record, each record-name of the associated record description entries must be the same, and the records must define the same number of bytes. However, a program or method that describes an external record can contain a data description entry including the REDEFINES clause that redefines the complete external record, and this complete redefinition need not occur identically in other programs or methods in the run unit.
- Use of the EXTERNAL clause does not imply that the associated data-name is a global name.

**GLOBAL clause**

The GLOBAL clause specifies that a data-name is available to every program contained within the program that defines it, as long as the contained program does not itself have a definition for that name. All data-names subordinate to or condition-names or indexes associated with a global name are global names.

A data-name is global if the GLOBAL clause is specified either in the data description entry by which the data-name is defined or in another entry to which that data description entry is subordinate. The GLOBAL clause can be specified in the WORKING-STORAGE SECTION, the FILE SECTION, the LINKAGE SECTION, and the LOCAL-STORAGE SECTION, but only in data description entries whose level-number is 01.

In the same DATA DIVISION, the data description entries for any two data items for which the same data-name is specified must not include the GLOBAL clause.

A statement in a program contained directly or indirectly within a program that describes a global name can reference that name without describing it again.

Two programs in a run unit can reference common data in the following circumstances:

- The data content of an external data record can be referenced from any program that describes the data record as external.
- If a program is contained within another program, both programs can refer to data that possesses the global attribute either in the containing program or in any program that directly or indirectly contains the containing program.
JUSTIFIED clause

The JUSTIFIED clause overrides standard positioning rules for receiving items of category alphabetic, alphanumeric, DBCS, or national.

<table>
<thead>
<tr>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>JUSTIFIED</td>
</tr>
<tr>
<td>JUST</td>
</tr>
</tbody>
</table>

You can specify the JUSTIFIED clause only at the elementary level. JUST is an abbreviation for JUSTIFIED, and has the same meaning.

You cannot specify the JUSTIFIED clause:
- For data items of category numeric, numeric-edited, alphanumeric-edited, or national-edited
- For edited DBCS items
- For index data items
- For items described as USAGE FUNCTION-POINTER, USAGE POINTER, USAGE PROCEDURE-POINTER, or USAGE OBJECT REFERENCE
- For external floating-point or internal floating-point items
- With level-66 (RENAMES) and level-88 (condition-name) entries

When the JUSTIFIED clause is specified for a receiving item, the data is aligned at the rightmost character position in the receiving item. Also:
- If the sending item is larger than the receiving item, the leftmost character positions are truncated.
- If the sending item is smaller than the receiving item, the unused character positions at the left are filled with spaces. For a DBCS item, each unused position is filled with a DBCS space (X'4040'); for an item described with usage NATIONAL, each unused position is filled with the default UTF-16 space (NX'0020'); for an item described with usage UTF-8, each unused position is filled with the default UTF-8 space (UX'20'); otherwise, each unused position is filled with an alphanumeric space.

If you omit the JUSTIFIED clause, the rules for standard alignment are followed (see “Alignment rules” on page 158).

The JUSTIFIED clause does not affect initial settings as determined by the VALUE clause.

GROUP-USAGE clause

A GROUP-USAGE clause with the NATIONAL phrase specifies that the group item defined by the entry is a national group item. A national group item contains national characters in all subordinate data items and subordinate group items. A GROUP-USAGE clause with the UTF-8 phrase specifies that the group item defined by the entry is a UTF-8 group item. A UTF-8 group item contains UTF-8 characters in all subordinate data items and subordinate group items.

<table>
<thead>
<tr>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>GROUP-USAGE</td>
</tr>
<tr>
<td>IS</td>
</tr>
</tbody>
</table>

When GROUP-USAGE NATIONAL is specified:
- The subject of the entry is a national group item. The class and category of a national group are national.
• A USAGE clause must not be specified for the subject of the entry. A USAGE NATIONAL clause is implied.
• A USAGE NATIONAL clause is implied for any subordinate elementary data items that are not described with a USAGE NATIONAL clause.
• All subordinate elementary data items must be explicitly or implicitly described with USAGE NATIONAL.
• Any signed numeric data items must be described with the SIGN IS SEPARATE clause.
• A GROUP-USAGE NATIONAL clause is implied for any subordinate group items that are not described with a GROUP-USAGE NATIONAL clause.
• All subordinate group items must be explicitly or implicitly described with a GROUP-USAGE NATIONAL clause.
• The JUSTIFIED clause must not be specified.

Unless stated otherwise, a national group item is processed as though it were an elementary data item of usage national, class and category national, described with PICTURE N(m), where m is the length of the group in national character positions.

Usage note: When you use national groups, the compiler can ensure proper truncation and padding of group items for statements such as MOVE and INSPECT. Groups defined without a GROUP-USAGE NATIONAL clause are alphanumeric groups. The content of alphanumeric groups, including any national characters, is treated as alphanumeric data, possibly leading to invalid truncation or mishandling of national character data.

The table below summarizes the cases where a national and UTF-8 group items are processed as a group item.

When GROUP-USAGE UTF-8 is specified:
• The subject of the entry is a UTF-8 group item. The class and category of a UTF-8 group are UTF-8.
• A USAGE clause must not be specified for the subject of the entry. A USAGE UTF-8 clause is implied.
• A USAGE UTF-8 clause is implied for any subordinate elementary data items that are not described with a USAGE UTF-8 clause.
• All subordinate elementary data items must be explicitly or implicitly described with USAGE UTF-8 and must be defined with the BYTE-LENGTH phrase of the PICTURE clause.
• A GROUP-USAGE UTF-8 clause is implied for any subordinate group items that are not described with a GROUP-USAGE UTF-8 clause.
• All subordinate group items must be explicitly or implicitly described with a GROUP-USAGE UTF-8 clause.
• The JUSTIFIED clause must not be specified.

Unless stated otherwise, a UTF-8 group item is processed as though it were an elementary data item of usage UTF-8, class and category UTF-8, and defined with PICTURE U BYTE-LENGTH m, where m is the length of the group in bytes.

Usage note: When you use UTF-8 groups, the compiler can ensure proper truncation and padding of group items for statements such as MOVE. Groups defined without a GROUP-USAGE UTF-8 or GROUP-USAGE NATIONAL clause are alphanumeric groups. The content of alphanumeric groups, including any UTF-8 characters, is treated as alphanumeric data, possibly leading to invalid truncation or mishandling of UTF-8 character data.

The table below summarizes the cases where a national and UTF-8 group items are processed as a group item.
Table 11. Where national and UTF-8 group items are processed as groups

<table>
<thead>
<tr>
<th>Language feature</th>
<th>Processing of national group items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name qualification</td>
<td>The names of national and UTF-8 group items can be used to qualify the names of elementary data items and subordinate group items in national and UTF-8 groups. The rules of qualification for national and UTF-8 groups are the same as the rules of qualification for an alphanumeric group.</td>
</tr>
<tr>
<td>RENAMES clause</td>
<td>The rules for national and UTF-8 group items specified in the THROUGH phrase are the same as the rules for an alphanumeric group item specified in the THROUGH phrase. The result is an alphanumeric group item.</td>
</tr>
<tr>
<td>CORRESPONDING phrase</td>
<td>National and UTF-8 group items are processed as a group in accordance with the rules of the CORRESPONDING phrase. Elementary data items within a national or UTF-8 group are processed the same as they would be if defined within an alphanumeric group.</td>
</tr>
<tr>
<td>INITIALIZE statement</td>
<td>National and UTF-8 group items are processed as a group in accordance with the rules of the INITIALIZE statement. Elementary items within the national or UTF-8 group are initialized the same as they would be if defined within an alphanumeric group.</td>
</tr>
<tr>
<td>XML GENERATE statement</td>
<td>A national group item specified in the FROM phrase is processed as a group in accordance with the rules of the XML GENERATE statement. Elementary items within the national group are processed the same as they would be if defined within an alphanumeric group.</td>
</tr>
<tr>
<td>JSON GENERATE statement</td>
<td>A national group item specified in the FROM phrase is processed as a group in accordance with the rules of the JSON GENERATE statement. Elementary items within the national group are processed the same as they would be if defined within an alphanumeric group.</td>
</tr>
</tbody>
</table>

**OCCURS clause**

The DATA DIVISION language elements used for table handling are the OCCURS clause and the INDEXED BY phrase.

For the INDEXED BY phrase description, see "INDEXED BY phrase" on page 184.

The OCCURS clause specifies tables whose elements can be referred to by indexing or subscripting. It also eliminates the need for separate entries for repeated data items.

Formats for the OCCURS clause include fixed-length tables and variable-length tables.

The subject of an OCCURS clause is the data-name of the data item that contains the OCCURS clause. Except for the OCCURS clause itself, data description clauses used with the subject apply to each occurrence of the item described.

Whenever the subject of an OCCURS clause or any data-item subordinate to it is referenced, it must be subscripted or indexed, with the following exceptions:

- When the subject of the OCCURS clause is used as the subject of a SEARCH statement.
- When the subject of the OCCURS clause is used as the subject of a format 2 SORT statement.
- When the subject or a subordinate data item is the object of the ASCENDING/DESCENDING KEY phrase.
- When the subordinate data item is the object of the REDEFINES clause.
- When the subordinate data item is used as the subject of a LENGTH OF special register. For details, see “LENGTH OF” on page 19.

When subscripted or indexed, the subject refers to one occurrence within the table, unless the ALL subscript is used in an intrinsic function.
The OCCURS clause cannot be specified in a data description entry that:

- Has a level number of 01, 66, 77, or 88.
- Describes a redefined data item. (However, a redefined item can be subordinate to an item that contains an OCCURS clause.) See “REDEFINES clause” on page 206.

**Fixed-length tables**

Fixed-length tables are specified using the OCCURS clause.

Because seven subscripts or indexes are allowed, six nested levels and one outermost level of the format-1 OCCURS clause are allowed. The format-1 OCCURS clause can be specified as subordinate to the OCCURS DEPENDING ON clause. In this way, a table of up to seven dimensions can be specified.

**Format 1: fixed-length tables**

```
OCCURS integer-2 TIMES
  ASCENDING KEY IS data-name-2
  DESCENDING KEY
  INDEXED BY index-name-1
```

*integer-2*

The exact number of occurrences. *integer-2* must be greater than zero.

**ASCENDING KEY and DESCENDING KEY phrases**

Data is arranged in ascending or descending order, depending on the keyword specified, according to the values contained in *data-name-2*. The data-names are listed in their descending order of significance.

The order is determined by the rules for comparison of operands (see “Relation conditions” on page 249). The ASCENDING KEY and DESCENDING KEY data items are used in OCCURS clauses, the SEARCH ALL statements for a binary search of the table element, and the format 2 SORT statements. As an alternative, keys can be specified with the format 2 SORT statements.

*data-name-2*

Must be the name of the subject entry or the name of an entry subordinate to the subject entry. *data-name-2* can be qualified.

If *data-name-2* names the subject entry, that entire entry becomes the ASCENDING KEY or DESCENDING KEY and is the only key that can be specified for this table element.

If *data-name-2* does not name the subject entry, then *data-name-2*:

- Must be subordinate to the subject of the table entry itself
- Must *not* be subordinate to, or follow, any other entry that contains an OCCURS clause
- Must not contain an OCCURS clause

*data-name-2* must not have subordinate items that contain OCCURS DEPENDING ON clauses.
When the ASCENDING KEY or DESCENDING KEY phrase is specified, the following rules apply:

- Keys must be listed in decreasing order of significance.
- The total number of keys for a given table element must not exceed 12.
- The data in the table must be arranged in ascending or descending sequence according to the collating sequence in use.
- The key must be described with one of the following usages:
  - BINARY
  - DISPLAY
  - DISPLAY-1
  - NATIONAL
  - UTF-8
  - PACKED-DECIMAL
  - COMPUTATIONAL
  - COMPUTATIONAL-1
  - COMPUTATIONAL-2
  - COMPUTATIONAL-3
  - COMPUTATIONAL-4
  - COMPUTATIONAL-5
- A key described with usage NATIONAL can have one of the following categories: national, national-edited, numeric-edited, numeric, or external floating-point.
- The sum of the lengths of all the keys associated with one table element must not exceed 256.
- If a key is specified without qualifiers and it is not a unique name, the key will be implicitly qualified with the subject of the OCCURS clause and all qualifiers of the OCCURS clause subject.

The following example illustrates the specification of ASCENDING KEY data items:

```
WORKING-STORAGE SECTION.
01  TABLE-RECORD.
   05  EMPLOYEE-TABLE OCCURS 100 TIMES
       ASCENDING KEY IS WAGE-RATE EMPLOYEE-NO
       INDEXED BY A, B.
   10  EMPLOYEE-NAME                     PIC X(20).
   10  EMPLOYEE-NO                       PIC 9(6).
   10  WAGE-RATE                         PIC 9999V99.
   10  WEEK-RECORD OCCURS 52 TIMES
       ASCENDING KEY IS WEEK-NO INDEXED BY C.
   15  WEEK-NO                         PIC 99.
   15  AUTHORIZED-ABSENCES             PIC  9.
   15  UNAUTHORIZED-ABSENCES           PIC  9.
   15  LATE-ARRIVALS                   PIC  9.
```

The keys for EMPLOYEE-TABLE are subordinate to that entry, and the key for WEEK-RECORD is subordinate to that subordinate entry.

In the preceding example, records in EMPLOYEE-TABLE must be arranged in ascending order of WAGE-RATE, and in ascending order of EMPLOYEE-NO within WAGE-RATE. Records in WEEK-RECORD must be arranged in ascending order of WEEK-NO. If they are not, results of any SEARCH ALL statement are unpredictable.

**INDEXED BY phrase**

The INDEXED BY phrase specifies the indexes that can be used with a table. A table without an INDEXED BY phrase can be referred to through indexing by using an index-name associated with another table.

For more information about using indexing, see “Subscripting using index-names (indexing)” on page 70.
Indexes normally are allocated in static memory associated with the program that contains the table. Thus indexes are in the last-used state when a program is reentered. However, in the following cases, indexes are allocated on a per-invocation basis. Thus you must set the value of the index on every entry for indexes on tables in the following sections:

- The LOCAL-STORAGE SECTION
- The WORKING-STORAGE SECTION of:
  - A program with the IS INITIAL clause on the PROGRAM-ID phrase
  - A class definition (object instance variables)
- The LINKAGE SECTION of:
  - Methods
  - Programs compiled with the RECURSIVE clause
  - Programs compiled with the THREAD option

Indexes specified in an external data record do not possess the external attribute.

`index-name-1`

Each index-name specifies an index to be created by the compiler for use by the program. These index-names are not data-names and are not identified elsewhere in the COBOL program; instead, they can be regarded as private special registers for the use of this object program only. They are not data and are not part of any data hierarchy.

Unreferenced index names need not be uniquely defined.

In one table entry, up to 12 index-names can be specified.

If a data item that possesses the global attribute includes a table accessed with an index, that index also possesses the global attribute. Therefore, the scope of an index-name is the same as that of the data-name that names the table in which the index is defined.

**Variable-length tables**

You can specify variable-length tables by using the OCCURS DEPENDING ON clause.

---

**Format 2: variable-length tables**

```
OCCURS integer-1 TO integer-2 UNBOUNDED TIMES DEPENDING ON data-name-1

ASCENDING DESCENDING KEY IS data-name-2

INDEXED BY index-name-1
```
**integer-1**

The minimum number of occurrences.

The value of **integer-1** must be greater than or equal to zero, and it must also be less than the value of **integer-2**.

If **integer-1** is omitted, a value of 1 is assumed and the keyword TO must also be omitted.

If **integer-1** is omitted, you will receive warning messages if the RULES (NOOMITODOMIN) compiler option is in effect. For more information, see **RULES** in the *Enterprise COBOL Programming Guide*.

**integer-2**

The maximum number of occurrences.

**integer-2** must be greater than **integer-1**.

The length of the subject item is fixed. Only the number of repetitions of the subject item is variable.

**UNBOUNDED**

Unbounded maximum number of occurrences.

**Unbounded table**

A table with an OCCURS clause that specifies UNBOUNDED.

You can reference unbounded tables in COBOL syntax anywhere a table can be referenced.

**Unbounded group**

A group that contains at least one unbounded table.

You can define unbounded groups only in the LINKAGE SECTION. Either alphanumeric groups or national groups can be unbounded.

You can reference unbounded groups in COBOL syntax anywhere an alphanumeric or national group can be referenced, with the following exceptions:

- You cannot specify unbounded groups as a BY CONTENT argument in a CALL statement.
- You cannot specify unbounded groups as **data-name-2** on the PROCEDURE DIVISION RETURNING phrase.
- You cannot specify unbounded groups as arguments to intrinsic functions, except as an argument to the LENGTH intrinsic function.

The total size of an unbounded group at run time must be less than 999,999,999 bytes.

For unbounded tables and groups, the effect of the SSRANGE compiler option is limited. For more information, see **SSRANGE** in the *Enterprise COBOL Programming Guide*.

For references about working with unbounded tables and groups, see Working with unbounded tables and groups in the *Enterprise COBOL Programming Guide*.

**OCCURS DEPENDING ON clause**

The OCCURS DEPENDING ON clause specifies variable-length tables.

**data-name-1**

Identifies the object of the OCCURS DEPENDING ON clause; that is, the data item whose current value represents the current number of occurrences of the subject item. The contents of items whose occurrence numbers exceed the value of the object are undefined.

The object of the OCCURS DEPENDING ON clause (**data-name-1**) must describe an integer data item.

The object of the OCCURS DEPENDING ON clause must not occupy any storage position within the range of the table (that is, any storage position from the first character position in the table through the last character position in the table).

The object of the OCCURS DEPENDING ON clause cannot be variably located; the object cannot follow an item that contains an OCCURS DEPENDING ON clause.
If the OCCURS clause is specified in a data description entry included in a record description entry that contains the EXTERNAL clause, \textit{data-name-1}, if specified, must reference a data item that possesses the external attribute. \textit{data-name-1} must be described in the same DATA DIVISION as the subject of the entry.

If the OCCURS clause is specified in a data description entry subordinate to one that contains the GLOBAL clause, \textit{data-name-1}, if specified, must be a global name. \textit{data-name-1} must be described in the same DATA DIVISION as the subject of the entry.

All data-names used in the OCCURS clause can be qualified; they cannot be subscripted or indexed.

At the time that the group item, or any data item that contains a subordinate OCCURS DEPENDING ON item or that follows but is not subordinate to the OCCURS DEPENDING ON item, is referenced, the value of the object of the OCCURS DEPENDING ON clause must fall within the range \textit{integer-1} through \textit{integer-2}, if \textit{integer-2} is specified (that is, if the table is not UNBOUNDED).

The behavior is undefined if the value of the object is outside of the range \textit{integer-1} through \textit{integer-2}.

When a group item that contains a subordinate OCCURS DEPENDING ON item is referred to, the part of the table area used in the operation is determined as follows:

- If the object is outside the group, only that part of the table area that is specified by the object at the start of the operation is used.
- If the object is included in the same group and the group data item is referenced as a sending item, only that part of the table area that is specified by the value of the object at the start of the operation is used in the operation.
- If the object is included in the same group and the group data item is referenced as a receiving item, the maximum length of the group item is used in the operation.

The following statements are affected by the maximum length rule:

- \texttt{ACCEPT identifier (format 1 and 2)}
- \texttt{CALL ... USING BY REFERENCE identifier}
- \texttt{INVOKE ... USING BY REFERENCE identifier}
- \texttt{MOVE ... TO identifier}
- \texttt{READ ... INTO identifier}
- \texttt{RELEASE identifier FROM ...}
- \texttt{RETURN ... INTO identifier}
- \texttt{REWRITE identifier FROM ...}
- \texttt{STRING ... INTO identifier}
- \texttt{UNSTRING ... INTO identifier DELIMITER IN identifier}
- \texttt{WRITE identifier FROM ...}

If a variable-length group item is not followed by a nonsubordinate item, the maximum length of the group is used when it appears as the identifier in \texttt{CALL ... USING BY REFERENCE identifier}. Therefore, the object of the OCCURS DEPENDING ON clause does not need to be set unless the group is variably located.

If the group item is followed by a nonsubordinate item, the actual length, rather than the maximum length, is used. At the time the subject of entry is referenced, or any data item subordinate or superordinate to the subject of entry is referenced, the object of the OCCURS DEPENDING ON clause must fall within the range \textit{integer-1} through \textit{integer-2}, if \textit{integer-2} is specified.

\textbf{Note:}

The maximum length rule does not apply to unbounded groups. For unbounded groups, based on the current run time value of the OCCURS DEPENDING ON objects, the actual length of the group is used for all references to the group. Consequently, before any COBOL statement that references an unbounded group runs, you must set the OCCURS DEPENDING ON objects for that group.
Certain uses of the OCCURS DEPENDING ON clause result in complex OCCURS DEPENDING ON (ODO) items. The following items constitute complex ODO items:

- A data item described with an OCCURS DEPENDING ON clause that is followed by a nonsubordinate elementary data item, described with or without an OCCURS clause
- A data item described with an OCCURS DEPENDING ON clause that is followed by a nonsubordinate group item
- A group item that contains one or more subordinate items described with an OCCURS DEPENDING ON clause
- A data item described with an OCCURS clause or an OCCURS DEPENDING ON clause that contains a subordinate data item described with an OCCURS DEPENDING ON clause (a table that contains variable-length elements)
- An index-name associated with a table that contains variable-length elements

The object of an OCCURS DEPENDING ON clause cannot be a nonsubordinate item that follows a complex ODO item.

Any nonsubordinate item that follows an item described with an OCCURS DEPENDING ON clause is a variably located item. That is, its location is affected by the value of the OCCURS DEPENDING ON object.

When implicit redefinition is used in a File Description (FD) entry, subordinate level items can contain OCCURS DEPENDING ON clauses.

The INDEXED BY phrase can be specified for a table that has a subordinate item that contains an OCCURS DEPENDING ON clause.

For more information about complex OCCURS DEPENDING ON, see Complex OCCURS DEPENDING ON in the Enterprise COBOL Programming Guide.

The ASCENDING KEY phrase, the DESCENDING KEY phrase, and the INDEXED BY clause are described under “Fixed-length tables” on page 183.

**PICTURE clause**

The PICTURE clause specifies the general characteristics and editing requirements of an elementary item.

---

**Format**

```
PICTURE
    PIC character-string IS
    BYTE-LENGTH IS integer-1
```

**PICTURE or PIC**

The PICTURE clause must be specified for every elementary item except the following ones:

- Index data items
- The subject of the RENAMES clause
- Items described with USAGE POINTER, USAGE FUNCTION-POINTER, USAGE PROCEDURE-POINTER, or USAGE OBJECT REFERENCE
- Internal floating-point data items

In these cases, use of the PICTURE clause is prohibited.
The PICTURE clause can be specified only at the elementary level.

PIC is an abbreviation for PICTURE and has the same meaning.

**character-string**

*character-string* is made up of certain COBOL characters used as picture symbols. The allowable combinations determine the category of the elementary data item.

*character-string* can contain a maximum of 50 characters.

**BYTE-LENGTH integer-1**

The **BYTE-LENGTH** phrase is only allowed for UTF-8 data items (i.e., data items defined with the U picture symbol) and indicates that the UTF-8 data item has a fixed byte-length but a varying number of characters. The number of bytes occupied by a UTF-8 data item defined with the **BYTE-LENGTH** phrase of the PICTURE clause is indicated by integer-1. The number of UTF-8 characters that can be stored in a data item defined with the **BYTE-LENGTH** phrase of the PICTURE clause depends on the size of each character. The maximum number of characters that can be stored is integer-1, which happens when each UTF-8 character is a single byte in length.

The **BYTE-LENGTH** phrase can only be specified when the picture string consists of a single ‘U’ symbol. The **DYNAMIC LENGTH** clause must not also be specified.

UTF-8 data items defined with the **BYTE-LENGTH** phrase of the PICTURE clause are truncated on UTF-8 character boundaries when truncation is needed and are always padded with UTF-8 spaces (x’20’) to a byte length of integer-1.

Only UTF-8 data items defined with the **BYTE-LENGTH** phrase of the PICTURE clause can be used as Db2® host variables and only UTF-8 data items defined with the **BYTE-LENGTH** phrase of the PICTURE clause can be part of a group defined with the **GROUP-USAGE UTF-8** clause.

UTF-8 data items defined with the **BYTE-LENGTH** phrase of the PICTURE clause are strongly recommended for UTF-8 items that need to function as a sort key or record key.

**Symbols used in the PICTURE clause**

Any punctuation character that appears within the PICTURE character-string is not considered a punctuation character, but rather is a PICTURE character-string symbol.

When specified in the **SPECIAL-NAMES** paragraph, **DECIMAL-POINT IS COMMA** exchanges the functions of the period and the comma in PICTURE character-strings and in numeric literals.

The lowercase letters that correspond to the uppercase letters that represent the following PICTURE symbols are equivalent to their uppercase representations in a PICTURE character-string:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Equivalent Uppercase Letter</th>
</tr>
</thead>
</table>

All other lowercase letters are not equivalent to their corresponding uppercase representations.

Table 12 on page 190 defines the meaning of each PICTURE clause symbol. The heading **Size** indicates how the item is counted in determining the number of character positions in the item. The type of the character positions depends on the **USAGE** clause specified for the item, as follows:

<table>
<thead>
<tr>
<th>Usage</th>
<th>Type of character positions</th>
<th>Number of bytes per character</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISPLAY</td>
<td>Alphanumeric</td>
<td>1</td>
</tr>
<tr>
<td>DISPLAY-1</td>
<td>DBCS</td>
<td>2</td>
</tr>
<tr>
<td>NATIONAL</td>
<td>National</td>
<td>2</td>
</tr>
<tr>
<td>UTF-8</td>
<td>UTF-8</td>
<td>1 to 4 bytes</td>
</tr>
<tr>
<td>All others</td>
<td>Conceptual</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>
Table 12. **PICTURE clause symbol meanings**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A character position that can contain only a letter of the Latin alphabet or a space.</td>
<td>Each 'A' is counted as one character position in the size of the data item.</td>
</tr>
<tr>
<td>B</td>
<td>For usage DISPLAY, a character position into which an alphanumeric space is inserted. For usage DISPLAY-1, a character position into which a DBCS space is inserted. For usage NATIONAL, a character position into which a national space is inserted.</td>
<td>Each 'B' is counted as one character position in the size of the data item.</td>
</tr>
<tr>
<td>E</td>
<td>Marks the start of the exponent in an external floating-point item. For additional details of external floating-point items, see “Data categories and PICTURE rules” on page 194.</td>
<td>Each 'E' is counted as one character position in the size of the data item.</td>
</tr>
<tr>
<td>G</td>
<td>A DBCS character position.</td>
<td>Each 'G' is counted as one character position in the size of the data item.</td>
</tr>
<tr>
<td>N</td>
<td>A DBCS character position when specified with usage DISPLAY-1 or when usage is unspecified and the NSYMBOL(DBCS) compiler option is in effect. For category national, a national character position when specified with usage NATIONAL or when usage is unspecified and the NSYMBOL(NATIONAL) compiler option is in effect. For category national-edited, a national character position.</td>
<td>Each 'N' is counted as one character position in the size of the data item.</td>
</tr>
<tr>
<td>P</td>
<td>An assumed decimal scaling position. Used to specify the location of an assumed decimal point when the point is not within the number that appears in the data item. See “P symbol” on page 192 for further details.</td>
<td>Not counted in the size of the data item. Scaling position characters are counted in determining the maximum number of digit positions in numeric-edited items or in items that are used as arithmetic operands. The size of the value is the number of digit positions represented by the PICTURE character-string.</td>
</tr>
<tr>
<td>S</td>
<td>An indicator of the presence (but not the representation, and not necessarily the position) of an operational sign. An operational sign indicates whether the value of an item involved in an operation is positive or negative.</td>
<td>Not counted in the size of the elementary item, unless an associated SIGN clause specifies the SEPARATE CHARACTER phrase (which would be counted as one character position).</td>
</tr>
<tr>
<td>Symbol</td>
<td>Meaning</td>
<td>Size</td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>U</td>
<td>A UTF-8 character position. USAGE UTF-8 is assumed whenever a U symbol appears in the PICTURE character-string of a data item. If the BYTE-LENGTH phrase of the PICTURE clause is specified, then only one 'U' symbol can be used in the PICTURE clause and the byte length of the data item is indicated in the BYTE-LENGTH phrase, and the number of characters that can be stored in the data item varies and depends on the size of each character.</td>
<td>Each 'U' counts as one UTF-8 character position in the size of the data item.</td>
</tr>
<tr>
<td>V</td>
<td>An indicator of the location of the assumed decimal point. Does not represent a character position. When the assumed decimal point is to the right of the rightmost symbol in the string, the V is redundant.</td>
<td>Not counted in the size of the elementary item.</td>
</tr>
<tr>
<td>X</td>
<td>A character position that can contain any allowable character from the alphanumeric character set of the computer.</td>
<td>Each 'X' is counted as one character position in the size of the data item.</td>
</tr>
<tr>
<td>Z</td>
<td>A leading numeric character position. When that position contains a zero, a space character replaces the zero.</td>
<td>Each 'Z' is counted as one character position in the size of the data item.</td>
</tr>
<tr>
<td>9</td>
<td>A character position that contains a numeral. For usages DISPLAY and NATIONAL, each nine is counted as one character position in the size of the data item.</td>
<td>Each nine specifies one decimal digit in the value of the item.</td>
</tr>
<tr>
<td>0</td>
<td>A character position into which the numeral zero is inserted.</td>
<td>Each zero is counted as one character position in the size of the data item.</td>
</tr>
<tr>
<td>/</td>
<td>A character position into which the slash character is inserted.</td>
<td>Each slash character is counted as one character position in the size of the data item.</td>
</tr>
<tr>
<td>,</td>
<td>A character position into which a comma is inserted.</td>
<td>Each comma is counted as one character position in the size of the data item.</td>
</tr>
<tr>
<td>.</td>
<td>An editing symbol that represents the decimal point for alignment purposes. In addition, it represents a character position into which a period is inserted.</td>
<td>Each period is counted as one character position in the size of the data item.</td>
</tr>
<tr>
<td>+ - CR DB</td>
<td>Editing sign control symbols. Each represents the character position into which the editing sign control symbol is placed.</td>
<td>Each character used in the editing sign symbol is counted as one character position in the size of the data item.</td>
</tr>
<tr>
<td>*</td>
<td>A check protect symbol: a leading numeric character position into which an asterisk is placed when that position contains a zero.</td>
<td>Each asterisk is counted as one character position in the size of the item.</td>
</tr>
</tbody>
</table>
Table 12. PICTURE clause symbol meanings (continued)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>cs</td>
<td>cs can be any valid currency symbol. A currency symbol represents a character position into which a currency sign value is placed. The default currency symbol is the character assigned the value X’5B’ in the code page in effect at compile time. In this document, the default currency symbol is represented by the dollar sign ($) and cs stands for any valid currency symbol. For details, see “Currency symbol” on page 193.</td>
<td>The first occurrence of a currency symbol adds the number of characters in the currency sign value to the size of the data item. Each subsequent occurrence adds one character position to the size of the data item.</td>
</tr>
</tbody>
</table>

The following figure shows the sequences in which picture symbols can be specified to form picture character-strings. More detailed explanations of PICTURE clause symbols follow the figure.

The P symbol

The symbol P specifies a scaling position and implies an assumed decimal point (to the left of the Ps if the Ps are leftmost PICTURE characters; to the right of the Ps if the Ps are rightmost PICTURE characters).

The assumed decimal point symbol V is redundant as either the leftmost or rightmost character within such a PICTURE description.
The symbol P can be specified only as a continuous string of Ps in the leftmost or rightmost digit positions within a PICTURE character-string.

In certain operations that reference a data item whose PICTURE character-string contains the symbol P, the algebraic value of the data item is used rather than the actual character representation of the data item. This algebraic value assumes the decimal point in the prescribed location and zero in place of the digit position specified by the symbol P. The size of the value is the number of digit positions represented by the PICTURE character-string. These operations are any of the following ones:

- Any operation that requires a numeric sending operand
- A MOVE statement where the sending operand is numeric and its PICTURE character-string contains the symbol P
- A MOVE statement where the sending operand is numeric-edited and its PICTURE character-string contains the symbol P, and the receiving operand is numeric or numeric-edited
- A comparison operation where both operands are numeric

In all other operations, the digit positions specified with the symbol P are ignored and are not counted in the size of the operand.

**Currency symbol**

The currency symbol in a picture character-string is represented by the default currency symbol $ or by a single character specified either in the CURRENCY compiler option or in the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION.

Although the default currency symbol is represented by $ in this document, the actual default currency symbol is the character with the value X'5B' in the EBCDIC code page in effect at compile time.

If the CURRENCY SIGN clause is specified, the CURRENCY and NOCURRENCY compiler options are ignored. 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. For more information about the CURRENCY SIGN clause, see “CURRENCY SIGN clause” on page 113. For more information about the CURRENCY and NOCURRENCY compiler options, see CURRENCY in the Enterprise COBOL Programming Guide.

A currency symbol can be repeated within the PICTURE character-string to specify floating insertion. Different currency symbols must not be used in the same PICTURE character-string.

Unlike all other picture symbols, currency symbols are case sensitive. For example, 'D' and 'd' specify different currency symbols.

A currency symbol can be used only to define a numeric-edited item with USAGE DISPLAY.

**Character-string representation**

The topic lists symbols that can appear once or more than once in the PICTURE character-string.

**Symbols that can appear more than once**

The following symbols can appear more than once in one PICTURE character-string:

- At least one of the symbols A, G, N, X, U, Z, 9, or *, or at least two of the symbols +, -, or cs must be present in a PICTURE string.

An unsigned nonzero integer enclosed in parentheses immediately following any of these symbols specifies the number of consecutive occurrences of that symbol.
Example: The following two PICTURE clause specifications are equivalent:

```
PICTURE IS $99999.99CR
PICTURE IS $9(5).9(2)CR
```

Symbols that can appear only once
The following symbols can appear only once in one PICTURE character-string:

```
E  S  V  .  CR  DB
```

Except for the PICTURE symbol V, each occurrence of any of the above symbols in a given PICTURE character-string represents an occurrence of that character or set of allowable characters in the data item.

Data categories and PICTURE rules
The allowable combinations of PICTURE symbols determine the data category of the item.

The data categories are:

- Alphabetic
- Alphanumeric
- Alphanumeric-edited
- DBCS
- External floating-point
- National
- National-edited
- Numeric
- Numeric-edited
- UTF-8

Note: Category internal floating point is defined by a USAGE clause that specifies the COMP-1 or COMP-2 phrase.

Alphabetic items
The PICTURE character-string can contain only the symbol A.

The content of the item must consist only of letters of the Latin alphabet and the space character.

Other clauses

- USAGE DISPLAY must be specified or implied.
- Any associated VALUE clause must specify an alphanumeric literal containing only alphabetic characters, SPACE, or a symbolic-character as the value of a figurative constant.
- Do not include a single byte character in a DBCS data item.
- When padding is required for a DBCS data item, the following rules apply:
  - Padding is done using double-byte space characters until the data area is filled (based on the number of double-byte character positions allocated for the data item).
  - Padding is done using single-byte space characters when the padding needed is not an even number of bytes (for example, when an alphanumeric group item is moved to a DBCS data item).

Alphanumeric items
The PICTURE character-string must consist of certain symbols.
The symbols are:
- One or more occurrences of the symbol X.
- Combinations of the symbols A, X, and 9. (A character-string containing all As or all 9s does not define an alphanumeric item.)

The item is treated as if the character-string contained only the symbol X.

The contents of the item in standard data format can be any allowable characters from the character set of the computer.

**Other clauses**

USAGE DISPLAY must be specified or implied.

Any associated VALUE clause must specify an alphanumeric literal or one of the following figurative constants:
- ZERO
- SPACE
- QUOTE
- HIGH-VALUE
- LOW-VALUE
- symbolic-character
- ALL alphanumeric-literal

**Alphanumeric-edited items**

The PICTURE character-string can contain the following symbols: A X 9 B 0 /

The string must contain at least one A or X, and at least one B or 0 (zero) or /.

The contents of the item in standard data format must be two or more characters from the character set of the computer.

**Other clauses**

USAGE DISPLAY must be specified or implied.

Any associated VALUE clause must specify an alphanumeric literal or one of the following figurative constants:
- ZERO
- SPACE
- QUOTE
- HIGH-VALUE
- LOW-VALUE
- symbolic-character
- ALL alphanumeric-literal

The literal is treated exactly as specified; no editing is done.

**DBCS items**

The PICTURE character-string can contain the symbols G, G and B, or N. Each G, B, or N represents a single DBCS character position.

Any associated VALUE clause must contain a DBCS literal, the figurative constant SPACE, or the figurative constant ALL DBCS-literal.
Other clauses
When PICTURE symbol G is used, USAGE DISPLAY-1 must be specified. When PICTURE symbol N is used and the NSYMBOL(DBCS) compiler option is in effect, USAGE DISPLAY-1 is implied if the USAGE clause is omitted.

External floating-point items
A data item is described as category external floating-point by its PICTURE character-string.

The PICTURE character-string details are described below.

Format

```
+  mantissa  E  exponent  
```

+ or -
A sign character must immediately precede both the mantissa and the exponent.
A + sign indicates that a positive sign will be used in the output to represent positive values and that a negative sign will represent negative values.
A - sign indicates that a blank will be used in the output to represent positive values and that a negative sign will represent negative values.
Each sign position occupies one byte of storage.

mantissa
The mantissa can contain the symbols:

```
9 . V
```

An actual decimal point can be represented with a period (.) while an assumed decimal point is represented by a V.
Either an actual or an assumed decimal point must be present in the mantissa; the decimal point can be leading, embedded, or trailing.
The mantissa can contain from 1 to 16 numeric characters.

E
Indicates the exponent.

exponent
The exponent must consist of the symbol 99.
Example: Pic -9v9(9)E-99
The DISPLAY phrase of the USAGE clause and a floating-point picture character-string define the item as a display floating-point data item.
The NATIONAL phrase of the USAGE clause and a floating-point picture character-string define the item as a national floating-point data item.
For items defined with usage DISPLAY, each picture symbol except V defines one alphanumeric character position in the item.
For items defined with usage NATIONAL, each picture symbol except V defines one national character position in the item.

Other clauses
The DISPLAY phrase or the NATIONAL phrase of the USAGE clause must be specified or implied.
The OCCURS, REDEFINES, and RENAMES clauses can be associated with external floating-point items.
The SIGN clause is accepted as documentation and has no effect on the representation of the sign. The SYNCHRONIZED clause is treated as documentation.

The following clauses are invalid with external floating-point items:

- BLANK WHEN ZERO
- JUSTIFIED
- VALUE

**National items**

The PICTURE character-string can contain one or more occurrences of the picture symbol N.

These rules apply when the NSYMBOL(NATIONAL) compiler option is in effect or the USAGE NATIONAL clause is specified. In the absence of a USAGE NATIONAL clause, if the NSYMBOL(DBCS) compiler option is in effect, picture symbol N represents a DBCS character and the rules of the PICTURE clause for a DBCS item apply.

Each N represents a single national character position.

Any associated VALUE clause must specify an alphanumeric literal, a national literal, or one of the following figurative constants:

- ZERO
- SPACE
- QUOTE
- HIGH-VALUE
- LOW-VALUE
- symbolic-character
- ALL alphanumeric-literal
- ALL national-literal

**Other clauses**

Only the NATIONAL phrase can be specified in the USAGE clause. When PICTURE symbol N is used and the NSYMBOL(NATIONAL) compiler option is in effect, USAGE NATIONAL is implied if the usage clause is omitted.

The following clauses can be used:

- JUSTIFIED
- EXTERNAL
- GLOBAL
- OCCURS
- REDEFINES
- RENAMES
- SYNCHRONIZED
- VOLATILE

The following clauses cannot be used:

- BLANK WHEN ZERO
- SIGN

**National-edited items**

The PICTURE character-string must contain at least one symbol N, and at least one instance of one of these symbols: B 0 (zero) or / (slash).
Each symbol represents a single national character position.

Any associated VALUE clause must specify an alphanumeric literal, a national literal, or one of the following figurative constants:

- ZERO
- SPACE
- QUOTE
- HIGH-VALUE
- LOW-VALUE
- symbolic-character
- ALL alphanumeric-literal
- ALL national-literal

The literal is treated exactly as specified; no editing is done.

The NSYMBOL(NATIONAL) compiler option has no effect on the definition of a data item of category national-edited.

**Other clauses**

USAGE NATIONAL must be specified or implied.

The following clauses can be used:

- JUSTIFIED
- EXTERNAL
- GLOBAL
- OCCURS
- REDEFINES
- RENAMES
- SYNCHRONIZED
- VOLATILE

The following clauses cannot be used:

- BLANK WHEN ZERO
- SIGN

**Numeric items**

There are several types of numeric items.

The types are:

- Binary
- Packed decimal (internal decimal)
- Zoned decimal (external decimal)
- National decimal (external decimal)

The type of a numeric item is defined by the usage clause as shown in the table below.

<table>
<thead>
<tr>
<th>Type</th>
<th>USAGE clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>BINARY, COMP, COMP-4, or COMP-5</td>
</tr>
<tr>
<td>Internal decimal</td>
<td>PACKED-DECIMAL, COMP-3</td>
</tr>
</tbody>
</table>
Table 13. Numeric types (continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>USAGE clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoned decimal (external decimal)</td>
<td>DISPLAY</td>
</tr>
<tr>
<td>National decimal (external decimal)</td>
<td>NATIONAL</td>
</tr>
</tbody>
</table>

For all numeric fields, the PICTURE character-string can contain only the symbols 9, P, S, and V.

The symbol S can be written only as the leftmost character in the PICTURE character-string.

The symbol V can be written only once in a given PICTURE character-string.

For binary items, the number of digit positions must range from 1 through 18 inclusive. For packed decimal and zoned decimal items the number of digit positions must range from 1 through 18, inclusive, when the ARITH(COMPAT) compiler option is in effect, or from 1 through 31, inclusive, when the ARITH(EXTEND) compiler option is in effect.

If unsigned, the contents of the item in standard data format must contain a combination of the Arabic numerals 0-9. If signed, it can also contain a +, -, or other representation of the operational sign.

Examples of valid ranges

<table>
<thead>
<tr>
<th>PICTURE character-string</th>
<th>Valid range of values</th>
</tr>
</thead>
<tbody>
<tr>
<td>9999</td>
<td>0 through 9999</td>
</tr>
<tr>
<td>S99</td>
<td>-99 through +99</td>
</tr>
<tr>
<td>S999V</td>
<td>-999.9 through +999.9</td>
</tr>
<tr>
<td>PPP999</td>
<td>0 through .000999</td>
</tr>
<tr>
<td>S999PPP</td>
<td>-1000 through -999000 and +1000 through +999000 or zero</td>
</tr>
</tbody>
</table>

Other clauses

The USAGE of the item can be DISPLAY, NATIONAL, BINARY, COMPUTATIONAL, PACKED-DECIMAL, COMPUTATIONAL-3, COMPUTATIONAL-4, or COMPUTATIONAL-5.

For signed numeric items described with usage NATIONAL, the SIGN IS SEPARATE clause must be specified or implied.

The NUMPROC and TRUNC compiler options can affect the use of numeric data items. For details, see NUMPROC and TRUNC in the Enterprise COBOL Programming Guide.

Numeric-edited items

The PICTURE character-string can contain certain symbols.

The symbols are:

B  P  V  Z  9  0  /  ,  .  +  -  CR  DB  *  cs

The combinations of symbols allowed are determined from the PICTURE clause symbol order allowed (see the figure in “Symbols used in the PICTURE clause” on page 189), and the editing rules (see “PICTURE clause editing” on page 201).

The following rules apply:

• Either the BLANK WHEN ZERO clause must be specified for the item, or the string must contain at least one of the following symbols:

B / Z 0 , . * + - CR DB cs

• Only one of the following symbols can be written in a given PICTURE character-string:

+ - CR DB

• If the ARITH(COMPAT) compiler option is in effect, then the number of digit positions represented in the character-string must be in the range 1 through 18, inclusive. If the ARITH(EXTEND) compiler option is
in effect, then the number of digit positions represented in the character-string must be in the range 1 through 31, inclusive.

- The total number of character positions in the string (including editing-character positions) must not exceed 249.
- The contents of those character positions representing digits in standard data format must be one of the 10 Arabic numerals.

Other clauses

USAGE DISPLAY or NATIONAL must be specified or implied.

If the usage of the item is DISPLAY, any associated VALUE clause must specify an alphanumeric literal or a figurative constant. The value is assigned without editing.

If the usage of the item is NATIONAL, any associated VALUE clause must specify an alphanumeric literal, a national literal, or a figurative constant. The value is assigned without editing.

UTF-8 items

The PICTURE character-string can contain one or more occurrences of the picture symbol \U. Each \U represents a single UTF-8 character position.

Any associated VALUE clause must specify an alphanumeric literal, a UTF-8 literal, or one of the following figurative constants:

- ZERO
- SPACE
- QUOTE
- HIGH-VALUE
- LOW-VALUE
- symbolic-character
- ALL utf-8-literal

Other clauses

Only the UTF-8 phrase can be specified in the USAGE clause. USAGE UTF-8 is implied if the usage clause is omitted.

The following clauses can be used:

- JUSTIFIED
- EXTERNAL
- GLOBAL
- OCCURS
- REDEFINES
- RENAMES
- SYNCHRONIZED
- VOLATILE

The following clauses cannot be used:

- BLANK WHEN ZERO
- SIGN
PICTURE clause editing

There are two general methods of editing in a PICTURE clause, insertion editing, and suppression and replacement editing.

Insertion editing includes the following types of editing:
- Simple insertion
- Special insertion
- Fixed insertion
- Floating insertion

Suppression and replacement editing includes the following types of editing:
- Zero suppression and replacement with asterisks
- Zero suppression and replacement with spaces

The type of editing allowed for an item depends on its data category. The type of editing that is valid for each category is shown in the following table. cs indicates any valid currency symbol.

<table>
<thead>
<tr>
<th>Data category</th>
<th>Type of editing</th>
<th>Insertion symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alphabetic</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Alphanumeric</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Alphanumeric-edited</td>
<td>Simple insertion</td>
<td>B 0 /</td>
</tr>
<tr>
<td>DBCS</td>
<td>Simple insertion</td>
<td>B</td>
</tr>
<tr>
<td>External floating-point</td>
<td>Special insertion</td>
<td>.</td>
</tr>
<tr>
<td>National</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>National-edited</td>
<td>Simple insertion</td>
<td>B 0 /</td>
</tr>
<tr>
<td>Numeric</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Numeric-edited</td>
<td>Simple insertion</td>
<td>B 0 / ,</td>
</tr>
<tr>
<td></td>
<td>Special insertion</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>Fixed insertion</td>
<td>cs + - CR DB</td>
</tr>
<tr>
<td></td>
<td>Floating insertion</td>
<td>cs + -</td>
</tr>
<tr>
<td></td>
<td>Zero suppression</td>
<td>Z *</td>
</tr>
<tr>
<td></td>
<td>Replacement</td>
<td>Z * + - cs</td>
</tr>
<tr>
<td>UTF-8</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>

Types of editing are described in the following sections:
- “Simple insertion editing” on page 202
- “Special insertion editing” on page 202
- “Fixed insertion editing” on page 203
- “Floating insertion editing” on page 204
- “Zero suppression and replacement editing” on page 205
Simple insertion editing

This type of editing is valid for alphanumeric-edited, numeric-edited, and DBCS items. Each insertion symbol is counted in the size of the item, and represents the position within the item where the equivalent character is to be inserted. For edited DBCS items, each insertion symbol (B) is counted in the size of the item and represents the position within the item where the DBCS space is to be inserted.

For example:

<table>
<thead>
<tr>
<th>PICTURE</th>
<th>Value of data</th>
<th>Edited result</th>
</tr>
</thead>
<tbody>
<tr>
<td>X(10)/XX</td>
<td>ALPHANUMER01</td>
<td>ALPHANUMER/01</td>
</tr>
<tr>
<td>X(5)BX(7)</td>
<td>ALPHANUMERIC</td>
<td>ALPHA NUMERIC</td>
</tr>
<tr>
<td>99,B999,B000</td>
<td>1234</td>
<td>01,b234,b0001</td>
</tr>
<tr>
<td>99,999</td>
<td>12345</td>
<td>12,345</td>
</tr>
<tr>
<td>GGBBGG</td>
<td>D1D2D3D4</td>
<td>D1D2bbbdD3D41</td>
</tr>
</tbody>
</table>

Notes:
1. The symbol b represents a space.

Special insertion editing

This type of editing is valid for either numeric-edited items or external floating-point items. The period (.) is the special insertion symbol; it also represents the actual decimal point for alignment purposes.

Note: If the DECIMAL-POINT IS COMMA clause is specified, then a comma will be used in place of the period.

The period insertion symbol is counted in the size of the item, and represents the position within the item where the actual decimal point is inserted.

Either the actual decimal point or the symbol V as the assumed decimal point, but not both, must be specified in one PICTURE character-string.

For example:

<table>
<thead>
<tr>
<th>PICTURE</th>
<th>Value of data</th>
<th>Edited result</th>
</tr>
</thead>
<tbody>
<tr>
<td>999.99</td>
<td>1.234</td>
<td>001.23</td>
</tr>
<tr>
<td>999.99</td>
<td>12.34</td>
<td>012.34</td>
</tr>
<tr>
<td>999.99</td>
<td>123.45</td>
<td>123.45</td>
</tr>
<tr>
<td>999.99</td>
<td>1234.5</td>
<td>234.50</td>
</tr>
<tr>
<td>+999.99E+99</td>
<td>12345</td>
<td>+123.45E+02</td>
</tr>
</tbody>
</table>
**Fixed insertion editing**

Fixed insertion editing is valid only for numeric-edited items.

The following insertion symbols are used:

- **cs**
- **+ - CR DB** (editing-sign control symbols)

In fixed insertion editing, only one currency symbol and one editing-sign control symbol can be specified in a PICTURE character-string.

Unless it is preceded by a + or - symbol, the currency symbol must be the first character in the character-string.

When either + or - is used as a symbol, it must be the first or last character in the character-string.

When CR or DB is used as a symbol, it must occupy the rightmost two character positions in the character-string. If these two character positions contain the symbols CR or DB, the uppercase letters are the insertion characters.

Editing sign control symbols produce results that depend on the value of the data item, as shown below:

<table>
<thead>
<tr>
<th>Editing symbol in PICTURE character-string</th>
<th>Result: data item positive or zero</th>
<th>Result: data item negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>space</td>
<td>-</td>
</tr>
<tr>
<td>CR</td>
<td>2 spaces</td>
<td>CR</td>
</tr>
<tr>
<td>DB</td>
<td>2 spaces</td>
<td>DB</td>
</tr>
</tbody>
</table>

For example:

<table>
<thead>
<tr>
<th>PICTURE</th>
<th>Value of data</th>
<th>Edited result</th>
</tr>
</thead>
<tbody>
<tr>
<td>999.99+</td>
<td>+6555.556</td>
<td>555.55+</td>
</tr>
<tr>
<td>+9999.99</td>
<td>-6555.555</td>
<td>-6555.55</td>
</tr>
<tr>
<td>9999.99</td>
<td>+1234.56</td>
<td>1234.56</td>
</tr>
<tr>
<td>$999.99</td>
<td>-123.45</td>
<td>$123.45</td>
</tr>
<tr>
<td>-$999.99</td>
<td>-123.456</td>
<td>-$123.45</td>
</tr>
<tr>
<td>-$999.99</td>
<td>+123.456</td>
<td>$123.45</td>
</tr>
<tr>
<td>$9999.99CR</td>
<td>+123.45</td>
<td>$0123.45</td>
</tr>
</tbody>
</table>
Floating insertion editing

Floating insertion editing is valid only for numeric-edited items.

The following symbols are used:

| cs | + | - |

Within one PICTURE character-string, these symbols are mutually exclusive as floating insertion characters.

Floating insertion editing is specified by using a string of at least two of the allowable floating insertion symbols to represent leftmost character positions into which the actual characters can be inserted.

The leftmost floating insertion symbol in the character-string represents the leftmost limit at which the actual character can appear in the data item. The rightmost floating insertion symbol represents the rightmost limit at which the actual character can appear.

The second leftmost floating insertion symbol in the character-string represents the leftmost limit at which numeric data can appear within the data item. Nonzero numeric data can replace all characters at or to the right of this limit.

Any simple-insertion symbols (B 0 / ,) within or to the immediate right of the string of floating insertion symbols are considered part of the floating character-string. If the period (.) special-insertion symbol is included within the floating string, it is considered to be part of the character-string.

To avoid truncation, the minimum size of the PICTURE character-string must be:

- The number of character positions in the sending item, plus
- The number of nonfloating insertion symbols in the receiving item, plus
- One character position for the floating insertion symbol

Representing floating insertion editing

In a PICTURE character-string, there are two ways to represent floating insertion editing and thus two ways in which editing is performed:

1. Any or all leading numeric character positions to the left of the decimal point are represented by the floating insertion symbol. When editing is performed, a single floating insertion character is placed to the immediate left of the first nonzero digit in the data, or of the decimal point, whichever is farther to the left. The character positions to the left of the inserted character are filled with spaces.

   If all numeric character positions in the PICTURE character-string are represented by the insertion character, then at least one of the insertion characters must be to the left of the decimal point.

2. All the numeric character positions are represented by the floating insertion symbol. When editing is performed, then:
   - If the value of the data is zero, the entire data item will contain spaces.
   - If the value of the data is nonzero, the result is the same as in rule 1.

For example:

<table>
<thead>
<tr>
<th>PICTURE</th>
<th>Value of data</th>
<th>Edited result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$9999.99CR</td>
<td>-123.45</td>
<td>$0123.45CR</td>
</tr>
<tr>
<td>PICTURE</td>
<td>Value of data</td>
<td>Edited result</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------</td>
<td>---------------</td>
</tr>
<tr>
<td>$.$$9.99</td>
<td>.12</td>
<td>$0.12</td>
</tr>
<tr>
<td>$$,$$$,999.99</td>
<td>-1234.56</td>
<td>$1,234.56</td>
</tr>
<tr>
<td>++$+,999.99</td>
<td>-123456.789</td>
<td>-123,456.78</td>
</tr>
<tr>
<td>$$, $$,$$,.99CR</td>
<td>-1234567</td>
<td>$1,234,567.00CR</td>
</tr>
<tr>
<td>++,++,+++</td>
<td>0000.00</td>
<td></td>
</tr>
</tbody>
</table>

**Zero suppression and replacement editing**

Zero suppression and replacement editing is valid only for numeric-edited items.

In zero suppression editing, the symbols Z and * are used. These symbols are mutually exclusive in one PICTURE character-string.

The following symbols are mutually exclusive as floating replacement symbols in one PICTURE character-string:

\[ Z \ast + - cs \]

Specify zero suppression and replacement editing with a string of one or more of the allowable symbols to represent leftmost character positions in which zero suppression and replacement editing can be performed.

Any simple insertion symbols (B 0 / ,) within or to the immediate right of the string of floating editing symbols are considered part of the string. If the period (.) special insertion symbol is included within the floating editing string, it is considered to be part of the character-string.

**Representing zero suppression**

In a PICTURE character-string, there are two ways to represent zero suppression, and two ways in which editing is performed:

1. Any or all of the leading numeric character positions to the left of the decimal point are represented by suppression symbols. When editing is performed, the replacement character replaces any leading zero in the data that appears in the same character position as a suppression symbol. Suppression stops at the leftmost character:
   - That does not correspond to a suppression symbol
   - That contains nonzero data
   - That is the decimal point

2. All the numeric character positions in the PICTURE character-string are represented by the suppression symbols. When editing is performed and the value of the data is nonzero, the result is the same as in the preceding rule. If the value of the data is zero, then:
   - If Z has been specified, the entire data item will contain spaces.
   - If * has been specified, the entire data item except the actual decimal point will contain asterisks.

For example:
### PICTURE

<table>
<thead>
<tr>
<th>PICTURE</th>
<th>Value of data</th>
<th>Edited result</th>
</tr>
</thead>
<tbody>
<tr>
<td>**<strong>.</strong></td>
<td>0000.00</td>
<td>**<strong>.</strong></td>
</tr>
<tr>
<td>ZZZZ.ZZ</td>
<td>0000.00</td>
<td></td>
</tr>
<tr>
<td>ZZZZ.99</td>
<td>0000.00</td>
<td>.00</td>
</tr>
<tr>
<td>****.99</td>
<td>0000.00</td>
<td>****.00</td>
</tr>
<tr>
<td>ZZ99.99</td>
<td>0000.00</td>
<td>00.00</td>
</tr>
<tr>
<td>Z,ZZZ.ZZ+</td>
<td>+123.456</td>
<td>123.45+</td>
</tr>
<tr>
<td><em>,</em><strong>.</strong>*</td>
<td>-123.45</td>
<td>**123.45-</td>
</tr>
<tr>
<td><strong>,</strong>,<strong>,</strong></td>
<td>+12345678.9</td>
<td>12,345,678.90+</td>
</tr>
<tr>
<td>$Z,ZZZ.ZZCR</td>
<td>+12345.67</td>
<td>$ 12,345.67</td>
</tr>
<tr>
<td>$B*,***,**BBDB</td>
<td>-12345.67</td>
<td>$ ***12,345.67  DB</td>
</tr>
</tbody>
</table>

Do not specify both the asterisk (*) as a suppression symbol and the BLANK WHEN ZERO clause for the same entry.

### REDEFINES clause

The REDEFINES clause allows you to use different data description entries to describe the same computer storage area.

#### Format

```
| level-number | data-name-1 | REDEFINES | data-name-2 |
```

(\(level\)-number, \(data\)-name-1, and FILLER are not part of the REDEFINES clause, and are included in the format only for clarity.)

When specified, the REDEFINES clause must be the first entry following \(data\)-name-1 or FILLER. If \(data\)-name-1 or FILLER is not specified, the REDEFINES clause must be the first entry following the level-number.
**data-name-1, FILLER**

Identifies an alternate description for the data area identified by `data-name-2`; `data-name-1` is the redefining item or the REDEFINES subject.

Neither `data-name-1` nor any of its subordinate entries can contain a VALUE clause.

**data-name-2**

Identifies the redefined item or the REDEFINES object.

The data description entry for `data-name-2` can contain a REDEFINES clause.

The data description entry for `data-name-2` cannot contain an OCCURS clause. However, `data-name-2` can be subordinate to an item whose data description entry contains an OCCURS clause; in this case, the reference to `data-name-2` in the REDEFINES clause must not be subscripted.

Neither `data-name-1` nor `data-name-2` can contain an OCCURS DEPENDING ON clause.

`data-name-1` and `data-name-2` must have the same level in the hierarchy; however, the level numbers need not be the same. Neither `data-name-1` nor `data-name-2` can be defined with level number 66 or 88.

`data-name-1` and `data-name-2` can each be described with any usage.

Redefinition begins at `data-name-1` and ends when a level-number less than or equal to that of `data-name-1` is encountered. No entry that has a level-number numerically lower than those of `data-name-1` and `data-name-2` can occur between these entries. In the following example:

```
05  A PICTURE X(6).
05  B REDEFINES A.
   10 B-1  PICTURE X(2).
   10 B-2  PICTURE 9(4).
05  C  PICTURE 99V99.
```

A is the redefined item, and B is the redefining item. Redefinition begins with B and includes the two subordinate items B-1 and B-2. Redefinition ends when the level-05 item C is encountered.

If the GLOBAL clause is used in the data description entry that contains the REDEFINES clause, only `data-name-1` (the redefining item) possesses the global attribute. For example, in the following description, only item B possesses the GLOBAL attribute:

```
05  A PICTURE X(6).
05  B REDEFINES A GLOBAL PICTURE X(4).
```

The EXTERNAL clause must not be specified in the same data description entry as a REDEFINES clause.

If the redefined data item (`data-name-2`) is declared to be an external data record, the size of the redefining data item (`data-name-1`) must not be greater than the size of the redefined data item. If the redefined data item is not declared to be an external data record, there is no such constraint.

The following example shows that the redefining item, B, can occupy more storage than the redefined item, A. The size of storage for the REDEFINED clause is determined in number of bytes. Item A occupies 6 bytes of storage and item B, a data item of category national, occupies 8 bytes of storage.

```
05  A PICTURE X(6).
05  B REDEFINES A GLOBAL PICTURE N(4).
```

One or more redefinitions of the same storage area are permitted. The entries that give the new descriptions of the storage area must immediately follow the description of the redefined area without intervening entries that define new character positions. Multiple redefinitions can, but need not, all use the data-name of the original entry that defined this storage area. For example:

```
05  A  PICTURE 9999.
```
Also, multiple redefinitions can use the name of the preceding definition as shown in the following example:

```
05 A          PICTURE 999.
05 B REDEFINES A PICTURE 9V999.
05 C REDEFINES B PICTURE 99V99.
```

When more than one level-01 entry is written subordinate to an FD entry, a condition known as implicit redefinition occurs. That is, the second level-01 entry implicitly redefines the storage allotted for the first entry. In such level-01 entries, the REDEFINES clause must not be specified.

When the data item implicitly redefines multiple 01-level records in a file description (FD) entry, items subordinate to the redefining or redefined item can contain an OCCURS DEPENDING ON clause.

**REDEFINES clause considerations**

The topic lists considerations of using the REDEFINES clause.

When an area is redefined, all descriptions of the area are always in effect; that is, redefinition does not supersede a previous description. Thus, if B REDEFINES C has been specified, either of the two procedural statements MOVE X TO B or MOVE Y TO C could be executed at any point in the program. In the first case, the area described as B would receive the value and format of X. In the second case, the same physical area (described now as C) would receive the value and format of Y. Note that if the second statement is executed immediately after the first, the value of Y replaces the value of X in the one storage area.

The usage of a redefining data item need not be the same as that of a redefined item. This does not, however, cause any change in the format or content of existing data. For example:

```
05 B          PICTURE  99 USAGE DISPLAY VALUE 8.
05 C REDEFINES B PICTURE S99 USAGE COMPUTATIONAL-4.
05 A          PICTURE S99 USAGE COMPUTATIONAL-4.
```

Redefining B does not change the bit configuration of the data in the storage area. Therefore, the following two statements produce different results:

```
ADD B TO A
ADD C TO A
```

In the first case, the value 8 is added to A (because B has USAGE DISPLAY). In the second statement, the value -3848 is added to A (because C has USAGE COMPUTATIONAL-4), and the bit configuration of the storage area has the binary value -3848. This example demonstrates how the improper use of redefinition can give unexpected or incorrect results.

**REDEFINES clause examples**

The REDEFINES clause can be specified for an item within the scope of (subordinate to) an area that is redefined.

In the following example, WEEKLY-PAY redefines SEMI-MONTHLY-PAY (which is within the scope of REGULAR-EMPLOYEE, while REGULAR-EMPLOYEE is redefined by TEMPORARY-EMPLOYEE).

```
05 REGULAR-EMPLOYEE.
  10 LOCATION           PICTURE A(8).
  10 GRADE              PICTURE X(4).
  10 SEMI-MONTHLY-PAY   PICTURE 9999V99.
  10 WEEKLY-PAY REDEFINES SEMI-MONTHLY-PAY
                      PICTURE 999V999.
  05 TEMPORARY-EMPLOYEE REDEFINES REGULAR-EMPLOYEE.
  10 LOCATION           PICTURE A(8).
```
The REDEFINES clause can also be specified for an item subordinate to a redefining item, as shown for CODE-H REDEFINES HOURLY-PAY in the following example:

```
05 REGULAR-EMPLOYEE.
  10 LOCATION       PICTURE A(8).
  10 GRADE          PICTURE X(4).
  10 SEMI-MONTHLY-PAY PICTURE 9999V999.
05 TEMPORARY-EMPLOYEE REDEFINES REGULAR-EMPLOYEE.
  10 LOCATION       PICTURE A(8).
  10 FILLER         PICTURE X(6).
  10 HOURLY-PAY     PICTURE 99V99.
  10 CODE-H REDEFINES HOURLY-PAY    PICTURE 9999.
```

Data items within an area can be redefined without changing their lengths. For example:

```
05 NAME-2.
  10 SALARY       PICTURE XXX.
  10 SO-SEC-NO   PICTURE X(9).
  10 MONTH       PICTURE XX.
05 NAME-1 REDEFINES NAME-2.
  10 WAGE        PICTURE XXX.
  10 EMP-NO     PICTURE X(9).
  10 YEAR       PICTURE XX.
```

Data item lengths and types can also be respecified within an area. For example:

```
05 NAME-2.
  10 SALARY       PICTURE XXX.
  10 SO-SEC-NO   PICTURE X(9).
  10 MONTH       PICTURE XX.
05 NAME-1 REDEFINES NAME-2.
  10 WAGE        PICTURE 9999V999.
  10 EMP-NO     PICTURE X(6).
  10 YEAR       PICTURE X(4).
```

Data items can also be respecified with a length that is greater than the length of the redefined item. For example:

```
05 NAME-2.
  10 SALARY       PICTURE XXX.
  10 SO-SEC-NO   PICTURE X(9).
  10 MONTH       PICTURE XX.
05 NAME-1 REDEFINES NAME-2.
  10 WAGE        PICTURE 9999V999.
  10 EMP-NO     PICTURE X(6).
  10 YEAR       PICTURE X(4).
```

This does not change the length of the redefined item NAME-2.

**Undefined results**

Undefined results can occur in the conditions as listed in the topic.

- A redefining item is moved to a redefined item (that is, if B REDEFINES C and the statement MOVE B TO C is executed).
- A redefined item is moved to a redefining item (that is, if B REDEFINES C and the statement MOVE C TO B is executed).
RENAMES clause

The RENAMES clause specifies alternative and possibly overlapping groupings of elementary data items.

**Format**

```

66   data-name-1   RENAMES   data-name-2

THRU   data-name-3

```

The special level-number 66 must be specified for data description entries that contain the RENAMES clause. (Level-number 66 and `data-name-1` are not part of the RENAMES clause, and are included in the format only for clarity.)

One or more RENAMES entries can be written for a logical record. All RENAMES entries associated with one logical record must immediately follow the last data description entry of that record.

*data-name-1*

Identifies an alternative grouping of data items.

A level-66 entry cannot rename a level-01, level-77, level-88, or another level-66 entry.

`data-name-1` cannot be used as a qualifier; it can be qualified only by the names of level indicator entries or level-01 entries.

*data-name-2, data-name-3*

Identify the original grouping of elementary data items; that is, they must name elementary or group items within the associated level-01 entry and must not be the same data-name. Both data-names can be qualified.

`data-name-2` and `data-name-3` can each reference any of the following items:

- An elementary data item
- An alphanumeric group item
- A national group item

When `data-name-2` or `data-name-3` references a national group item, the referenced item is processed as a group (not as an elementary data item of category national).

The OCCURS clause must not be specified in the data entries for `data-name-2` and `data-name-3`, or for any group entry to which they are subordinate. In addition, the OCCURS DEPENDING clause must not be specified for any item defined between `data-name-2` and `data-name-3`.

The keywords THROUGH and THRU are equivalent.

When the THROUGH phrase is specified:

- `data-name-1` defines an alphanumeric group item that includes all the elementary items that:
  - Start with `data-name-2` if it is an elementary item, or the first elementary item within `data-name-2` if it is a group item
  - End with `data-name-3` if it is an elementary item, or the last elementary item within `data-name-3` if it is an alphanumeric group item or national group item
- The storage area occupied by the starting item through the ending item becomes the storage area occupied by `data-name-1`.

**Usage note:** The group defined with the THROUGH phrase can include data items of usage NATIONAL.
The leftmost character position in `data-name-3` must not precede the leftmost character position in `data-name-2`, and the rightmost character position in `data-name-3` must not precede the rightmost character position in `data-name-2`. This means that `data-name-3` cannot be totally subordinate to `data-name-2`.

When the THROUGH phrase is not specified:

- The storage area occupied by `data-name-2` becomes the storage area occupied by `data-name-1`.
- All of the data attributes of `data-name-2` become the data attributes for `data-name-1`. That is:
  - When `data-name-2` is an alphanumeric group item, `data-name-1` is an alphanumeric group item.
  - When `data-name-2` is a national group item, `data-name-1` is a national group item.
  - When `data-name-2` is an elementary item, `data-name-1` is an elementary item.

The following figure illustrates valid and invalid RENAMES clause specifications.

**COBOL Specifications**

<table>
<thead>
<tr>
<th>Example 1 (Valid)</th>
<th>01 RECORD-I</th>
</tr>
</thead>
<tbody>
<tr>
<td>05 DN-1...</td>
<td></td>
</tr>
<tr>
<td>05 DN-2...</td>
<td></td>
</tr>
<tr>
<td>05 DN-3...</td>
<td></td>
</tr>
<tr>
<td>05 DN-4...</td>
<td></td>
</tr>
<tr>
<td>66 DN-6 RENAMES DN-1 THROUGH DN-3.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example 2 (Valid)</th>
<th>01 RECORD-II</th>
</tr>
</thead>
<tbody>
<tr>
<td>05 DN-1.</td>
<td></td>
</tr>
<tr>
<td>10 DN-2...</td>
<td></td>
</tr>
<tr>
<td>10 DN-2A...</td>
<td></td>
</tr>
<tr>
<td>05 DN-1A REDEFINES DN-1.</td>
<td></td>
</tr>
<tr>
<td>10 DN-3A...</td>
<td></td>
</tr>
<tr>
<td>10 DN-3...</td>
<td></td>
</tr>
<tr>
<td>10 DN-3B...</td>
<td></td>
</tr>
<tr>
<td>05 DN-5...</td>
<td></td>
</tr>
<tr>
<td>66 DN-6 RENAMES DN-2 THROUGH DN-3.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example 3 (Invalid)</th>
<th>01 RECORD-III</th>
</tr>
</thead>
<tbody>
<tr>
<td>05 DN-2.</td>
<td></td>
</tr>
<tr>
<td>10 DN-3...</td>
<td></td>
</tr>
<tr>
<td>10 DN-4...</td>
<td></td>
</tr>
<tr>
<td>05 DN-5...</td>
<td></td>
</tr>
<tr>
<td>66 DN-6 RENAMES DN-2 THROUGH DN-3.</td>
<td>DN-6 is indeterminate</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example 4 (Invalid)</th>
<th>01 RECORD-IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>05 DN-1.</td>
<td></td>
</tr>
<tr>
<td>10 DN-2A...</td>
<td></td>
</tr>
<tr>
<td>10 DN-2B...</td>
<td></td>
</tr>
<tr>
<td>10 DN-2C REDEFINES DN-2B.</td>
<td></td>
</tr>
<tr>
<td>15 DN-2CA...</td>
<td></td>
</tr>
<tr>
<td>05 DN-3...</td>
<td></td>
</tr>
<tr>
<td>66 DN-4 RENAMES DN-1 THROUGH DN-2CA.</td>
<td>DN-4 is indeterminate</td>
</tr>
</tbody>
</table>

**SIGN clause**

The SIGN clause specifies the position and mode of representation of the operational sign for the signed numeric item to which it applies.

The SIGN clause is required only when an explicit description of the properties or position of the operational sign is necessary.
The SIGN clause can be specified only for the following items:

- An elementary numeric data item of usage DISPLAY or NATIONAL that is described with an S in its picture character string, or
- A group item that contains at least one such elementary entry as a subordinate item

When the SIGN clause is specified at the group level, that SIGN clause applies only to subordinate signed numeric elementary data items of usage DISPLAY or NATIONAL. Such a group can also contain items that are not affected by the SIGN clause. If the SIGN clause is specified for a group or elementary entry that is subordinate to a group item that has a SIGN clause, the SIGN clause for the subordinate entry takes precedence for that subordinate entry.

The SIGN clause is treated as documentation for external floating-point items.

When the SIGN clause is specified without the SEPARATE phrase, USAGE DISPLAY must be specified explicitly or implicitly. When SIGN IS SEPARATE is specified, either USAGE DISPLAY or USAGE NATIONAL can be specified.

If you specify the CODE-SET clause in an FD entry, any signed numeric data description entries associated with that file description entry must be described with the SIGN IS SEPARATE clause.

If the SEPARATE CHARACTER phrase is not specified, then:

- The operational sign is presumed to be associated with the LEADING or TRAILING digit position, whichever is specified, of the elementary numeric data item. (In this instance, specification of SIGN IS TRAILING is the equivalent of the standard action of the compiler.)
- The character S in the PICTURE character string is not counted in determining the size of the item (in terms of standard data format characters).

If the SEPARATE CHARACTER phrase is specified, then:

- The operational sign is presumed to be the LEADING or TRAILING character position, whichever is specified, of the elementary numeric data item. This character position is not a digit position.
- The character S in the PICTURE character string is counted in determining the size of the data item (in terms of standard data format characters).
- + is the character used for the positive operational sign.
- - is the character used for the negative operational sign.

**SYNCHRONIZED clause**

The SYNCHRONIZED clause specifies the alignment of an elementary item on a natural boundary in storage.

SYNC is an abbreviation for SYNCHRONIZED and has the same meaning.
The SYNCHRONIZED clause is never required, but can improve performance on some systems for binary items used in arithmetic.

The SYNCHRONIZED clause can be specified for elementary items and for level-01 group items, in which case every elementary item within the group item is synchronized.

**LEFT**

Specifies that the elementary item is to be positioned so that it will begin at the left character position of the natural boundary in which the elementary item is placed.

**RIGHT**

Specifies that the elementary item is to be positioned such that it will terminate on the right character position of the natural boundary in which it has been placed.

When specified, the LEFT and the RIGHT phrases are syntax checked but have no effect on the execution of the program.

The length of an elementary item is not affected by the SYNCHRONIZED clause.

The following table lists the effect of the SYNCHRONIZE clause on other language elements.

<table>
<thead>
<tr>
<th>Language element</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCCURS clause</td>
<td>When specified for an item within the scope of an OCCURS clause, each occurrence of the item is synchronized.</td>
</tr>
<tr>
<td>USAGE DISPLAY or PACKED-DECIMAL</td>
<td>Each item is syntax checked, but the SYNCHRONIZED clause has no effect on execution.</td>
</tr>
<tr>
<td>USAGE NATIONAL</td>
<td>Each item is syntax checked, but the SYNCHRONIZED clause has no effect on execution.</td>
</tr>
<tr>
<td>USAGE BINARY or COMPUTATIONAL</td>
<td>When the item is the first elementary item subordinate to an item that contains a REDEFINES clause, the item must not require the addition of unused character positions. When the synchronized clause is not specified for a subordinate data item (one with a level number of 02 through 49):</td>
</tr>
<tr>
<td></td>
<td>• The item is aligned at a displacement that is a multiple of 2 relative to the beginning of the record if its USAGE is BINARY and its PICTURE is in the range of S9 through S9(4).</td>
</tr>
<tr>
<td></td>
<td>• The item is aligned at a displacement that is a multiple of 4 relative to the beginning of the record if its USAGE is BINARY and its PICTURE is in the range of S9(5) through S9(18), or its USAGE is INDEX.</td>
</tr>
<tr>
<td></td>
<td>When SYNCHRONIZED is not specified for binary items, no space is reserved for slack bytes.</td>
</tr>
<tr>
<td>USAGE POINTER, PROCEDURE-POINTER, FUNCTION-POINTER, OBJECT REFERENCE, INDEX</td>
<td>The data is aligned on a fullword boundary when the LP(32) compiler option is in effect. The data is aligned on a doubleword boundary when the LP(64) compiler option is in effect.</td>
</tr>
<tr>
<td>USAGE COMPUTATIONAL-1</td>
<td>The data is aligned on a fullword boundary.</td>
</tr>
<tr>
<td>USAGE COMPUTATIONAL-2</td>
<td>The data is aligned on a doubleword boundary.</td>
</tr>
<tr>
<td>USAGE COMPUTATIONAL-3</td>
<td>The data is treated the same as the SYNCHRONIZED clause for a PACKED-DECIMAL item.</td>
</tr>
</tbody>
</table>
Table 15. **SYNCHRONIZE clause effect on other language elements** (continued)

<table>
<thead>
<tr>
<th>Language element</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>USAGE COMPUTATIONAL-4</td>
<td>The data is treated the same as the SYNCHRONIZED clause for a COMPUTATIONAL item.</td>
</tr>
<tr>
<td>USAGE COMPUTATIONAL-5</td>
<td>The data is treated the same as the SYNCHRONIZED clause for a COMPUTATIONAL item.</td>
</tr>
<tr>
<td>USAGE UTF-8</td>
<td>Each item is syntax checked, but the SYNCHRONIZED clause has no effect on execution.</td>
</tr>
<tr>
<td>DBCS and external floating-point items</td>
<td>Each item is syntax checked, but the SYNCHRONIZED clause has no effect on execution.</td>
</tr>
<tr>
<td>REDEFINES clause</td>
<td>For an item that contains a REDEFINES clause, the data item that is redefined must have the proper boundary alignment for the data item that redefines it. For example, if you write the following, be sure that data item A begins on a fullword boundary:</td>
</tr>
<tr>
<td></td>
<td>02 A PICTURE X(4). 02 B REDEFINES A PICTURE S9(9) BINARY SYNC.</td>
</tr>
</tbody>
</table>

In the FILE SECTION, the compiler assumes that all level-01 records that contain SYNCHRONIZED items are aligned on doubleword boundaries in the buffer. You must provide the necessary slack bytes between records to ensure alignment when there are multiple records in a block.

In the WORKING-STORAGE SECTION, the compiler aligns all level-01 entries on a doubleword boundary.

For the purposes of aligning binary items in the LINKAGE SECTION, all level-01 items are assumed to begin on doubleword boundaries. Therefore, if you issue a CALL statement, such operands of any USING phrase within it must be aligned correspondingly.

**Slack bytes**

There are two types of slack bytes.

- Slack bytes **within** records: unused character positions that precede each synchronized item in the record
- Slack bytes **between** records: unused character positions added between blocked logical records

If the RULES=(NOSLACKBYTES) option is in effect, warning messages are issued for any SYNCHRONIZED data items that cause the compiler to add slack bytes, either slack bytes within records or slack bytes between records. For details about the RULES option, see RULES in the *Enterprise COBOL Programming Guide*.

**Slack bytes within records**

For any data description that has binary items that are not on their natural boundaries, the compiler inserts slack bytes within a record to ensure that all SYNCHRONIZED items are on their proper boundaries.

Because it is important that you know the length of the records in a file, you need to determine whether slack bytes are required and, if so, how many bytes the compiler will add. The algorithm that the compiler uses is as follows:

- The total number of bytes occupied by all elementary data items that precede the binary item are added together, including any slack bytes that are previously added.
- This sum is divided by \( m \), where:
  - \( m = 2 \) for binary items of four-digit length or less
  - \( m = 4 \) for binary items of five-digit length or more and for COMPUTATIONAL-1 data items
– $m = 4$ for data items described with USAGE INDEX, USAGE POINTER, USAGE PROCEDURE-POINTER, USAGE OBJECT REFERENCE, or USAGE FUNCTION-POINTER

– $m = 8$ for COMPUTATIONAL-2 data items

• If the remainder ($r$) of this division is equal to zero, no slack bytes are required. If the remainder is not equal to zero, the number of slack bytes that must be added is equal to $m - r$.

These slack bytes are added to each record immediately following the elementary data item that precedes the binary item. They are defined as if they constitute an item with a level-number equal to that of the elementary item that immediately precedes the SYNCHRONIZED binary item, and are included in the size of the group that contains them.

For example:

```
01 FIELD-A.
  05 FIELD-B            PICTURE X(5).
  05 FIELD-C.
    10 FIELD-D          PICTURE XX.
    [10 SLACK-BYTES      PICTURE X. INSERTED BY COMPILER]
    10 FIELD-E COMPUTATIONAL  PICTURE S9(6) SYNC.
 01 FIELD-L.
  05 FIELD-M            PICTURE X(5).
  05 FIELD-N            PICTURE XX.
  [05 SLACK-BYTES       PICTURE X. INSERTED BY COMPILER]
  05 FIELD-O.
    10 FIELD-P COMPUTATIONAL  PICTURE S9(6) SYNC.
```

Slack bytes can also be added by the compiler when a group item is defined with an OCCURS clause and contains within it a SYNCHRONIZED binary data item. To determine whether slack bytes are to be added, the following action is taken:

• The compiler calculates the size of the group, including all the necessary slack bytes within a record.
• This sum is divided by the largest $m$ required by any elementary item within the group.
• If $r$ is equal to zero, no slack bytes are required. If $r$ is not equal to zero, $m - r$ slack bytes must be added.

The slack bytes are inserted at the end of each occurrence of the group item that contains the OCCURS clause. For example, a record defined as follows appears in storage, as shown, in the figure after the record:

```
01 WORK-RECORD.
  05 WORK-CODE           PICTURE X.
  05 COMP-TABLE OCCURS 10 TIMES.
    10 COMP-TYPE          PICTURE X.
    [10 SLACK-BYTES       PIC XX. INSERTED BY COMPILER]
    10 COMP-PAY           PICTURE S9(4)V99 COMP SYNC.
    10 COMP-HOURS         PICTURE S9(3) COMP SYNC.
    10 COMP-NAME          PICTURE X(5).
```
In order to align COMP-PAY and COMP-HOURS on their proper boundaries, the compiler added 2 slack bytes within the record.

In the previous example, without further adjustment, the second occurrence of COMP-TABLE would begin 1 byte before a doubleword boundary, and the alignment of COMP-PAY and COMP-HOURS would not be valid for any occurrence of the table after the first. Therefore, the compiler must add slack bytes at the end of the group, as though the record had been written as follows:

```
01 WORK-RECORD.  
  05 WORK-CODE         PICTURE X.  
  05 COMP-TABLE OCCURS 10 TIMES.  
      10 COMP-TYPE          PICTURE X.  
      10 SLACK-BYTES        PIC XX.  INSERTED BY COMPILER]  
      10 COMP-PAY            PICTURE S9(4)V99 COMP SYNC.  
      10 COMP-HOURS          PICTURE S9(3) COMP SYNC.  
      10 COMP-NAME           PICTURE X(5).  
      [10 SLACK-BYTES        PIC XX.  INSERTED BY COMPILER]  
```

In this example, the second and each succeeding occurrence of COMP-TABLE begins 1 byte beyond a doubleword boundary. The storage layout for the first occurrence of COMP-TABLE now appears as shown in the following figure:

Each succeeding occurrence within the table will now begin at the same relative position as the first.
Slack bytes between records

If the file contains blocked logical records that are to be processed in a buffer, and any of the records contain binary entries for which the SYNCHRONIZED clause is specified, you can improve performance by adding any needed slack bytes between records for proper alignment.

The lengths of all the elementary data items in the record, including all slack bytes, are added. (For variable-length records, it is necessary to add an additional 4 bytes for the count field.) The total is then divided by the highest value of \( m \) for any one of the elementary items in the record.

If \( r \) (the remainder) is equal to zero, no slack bytes are required. If \( r \) is not equal to zero, \( m - r \) slack bytes are required. These slack bytes can be specified by writing a level-02 FILLER at the end of the record.

Consider the following record description:

```
01  COMP-RECORD.
   05  A-1    PICTURE X(5).
   05  A-2    PICTURE X(3).
   05  A-3    PICTURE X(3).
   05  B-1    PICTURE S9999 USAGE COMP SYNCHRONIZED.
   05  B-2    PICTURE S99999 USAGE COMP SYNCHRONIZED.
   05  B-3    PICTURE S9999 USAGE COMP SYNCHRONIZED.
```

The number of bytes in A-1, A-2, and A-3 totals 11. B-1 is a four-digit COMPUTATIONAL item and 1 slack byte must therefore be added before B-1. With this byte added, the number of bytes that precede B-2 totals 14. Because B-2 is a COMPUTATIONAL item of five digits in length, 2 slack bytes must be added before it. No slack bytes are needed before B-3.

The revised record description entry now appears as:

```
01  COMP-RECORD.
   05  A-1            PICTURE X(5).
   05  A-2            PICTURE X(3).
   05  A-3            PICTURE X(3).
   [05  SLACK-BYTE-1   PICTURE X.   INSERTED BY COMPILER]
   05  B-1            PICTURE S9999 USAGE COMP SYNCHRONIZED.
   [05  SLACK-BYTE-2   PICTURE XX.  INSERTED BY COMPILER]
   05  B-2            PICTURE S99999 USAGE COMP SYNCHRONIZED.
   05  B-3            PICTURE S9999 USAGE COMP SYNCHRONIZED.
```

There is a total of 22 bytes in COMP-RECORD, but from the rules above, it appears that \( m = 4 \) and \( r = 2 \). Therefore, to attain proper alignment for blocked records, you must add 2 slack bytes at the end of the record.

The final record description entry appears as:

```
01  COMP-RECORD.
   05  A-1            PICTURE X(5).
   05  A-2            PICTURE X(3).
   05  A-3            PICTURE X(3).
   [05  SLACK-BYTE-1    PICTURE X.   INSERTED BY COMPILER]
   [05  SLACK-BYTE-2    PICTURE XX.  INSERTED BY COMPILER]
   05  B-1            PICTURE S9999 USAGE COMP SYNCHRONIZED.
   05  B-2            PICTURE S99999 USAGE COMP SYNCHRONIZED.
   05  B-3            PICTURE S9999 USAGE COMP SYNCHRONIZED.
   05  FILLER          PICTURE XX.  [SLACK BYTES YOU ADD]
```
The USAGE clause specifies the format in which data is represented in storage.

**Format 1**

**Note:** NATIVE is treated as a comment in all phrases for which NATIVE is shown in the USAGE clause.
The USAGE clause can be specified for a data description entry with any level-number other than 66 or 88.

When specified at the group level, the USAGE clause applies to each elementary item in the group. The usage of elementary items must not contradict the usage of a group to which the elementary items belongs.

A USAGE clause must not be specified in a group level entry for which a GROUP-USAGE NATIONAL clause is specified.

When a GROUP-USAGE NATIONAL clause is specified or implied for a group level entry, USAGE NATIONAL must be specified or implied for every elementary item within the group. For details, see “GROUP-USAGE clause” on page 180.

When the USAGE clause is not specified at either the group or elementary level, a usage clause is implied with:

• Usage DISPLAY when the PICTURE clause contains only symbols other than G or N
• Usage NATIONAL when the PICTURE clause contains only one or more of the symbol N and the NSYM(NATIONAL) compiler option is in effect
• Usage DISPLAY-1 when the PICTURE clause contains one or more of the symbol N and the NSYM(DBCS) compiler option is in effect

Computational items

A computational item is a value used in arithmetic operations. It must be numeric. If a group item is described with a computational usage, the elementary items within the group have that usage.

The maximum length of a computational item is 18 decimal digits, except for a PACKED-DECIMAL item. If the ARITH(COMPAT) compiler option is in effect, then the maximum length of a PACKED-DECIMAL item is 18 decimal digits. If the ARITH(EXTEND) compiler option is in effect, then the maximum length of a PACKED-DECIMAL item is 31 decimal digits.

The PICTURE of a computational item can contain only:

9  
One or more numeric character positions
S  
One operational sign
V  
One implied decimal point
P  
One or more decimal scaling positions

COMPUTATIONAL-1 and COMPUTATIONAL-2 items (internal floating-point) cannot have PICTURE strings.

BINARY

Specified for binary data items. Such items have a decimal equivalent consisting of the decimal digits 0 through 9, plus a sign. Negative numbers are represented as the two's complement of the positive number with the same absolute value.

The amount of storage occupied by a binary item depends on the number of decimal digits defined in its PICTURE clause:

<table>
<thead>
<tr>
<th>Digits in PICTURE clause</th>
<th>Storage occupied</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 through 4</td>
<td>2 bytes (halfword)</td>
</tr>
<tr>
<td>5 through 9</td>
<td>4 bytes (fullword)</td>
</tr>
<tr>
<td>10 through 18</td>
<td>8 bytes (doubleword)</td>
</tr>
</tbody>
</table>
Binary data is **big-endian**: the operational sign is contained in the leftmost bit.

BINARY, COMPUTATIONAL, and COMPUTATIONAL-4 data items can be affected by the TRUNC compiler option. For information about the effect of this compiler option, see TRUNC in the *Enterprise COBOL Programming Guide*.

**PACKED-DECIMAL**

Specified for internal decimal items. Such an item appears in storage in packed decimal format. There are two digits for each character position, except for the trailing character position, which is occupied by the low-order digit and the sign. Such an item can contain any of the digits 0 through 9, plus a sign, representing a value not exceeding 18 decimal digits, unless the ARTIH(EXTEND) compiler option is in effect, in which case up to 31 digits might be represented.

The sign representation uses the same bit configuration as the 4-bit sign representation in zoned decimal fields. For details, see *Sign representation of zoned and packed-decimal data* in the *Enterprise COBOL Programming Guide*.

The most efficient use of packed-decimal data items is to define them with an odd number of digits, to use all of the bits. This can also result in more efficient generated code. To find out if you have packed-decimal data items with an even number of digits in your program, you can use the RULES(NOEVENPACK) compiler option. For details, see RULES in the *Enterprise COBOL Programming Guide*.

**COMPUTATIONAL or COMP (binary)**

This is the equivalent of BINARY. The COMPUTATIONAL phrase is synonymous with BINARY.

**COMPUTATIONAL-1 or COMP-1 (floating-point)**

Specified for internal floating-point items (single precision). COMP-1 items are 4 bytes long.

**COMPUTATIONAL-2 or COMP-2 (long floating-point)**

Specified for internal floating-point items (double precision). COMP-2 items are 8 bytes long.

**COMPUTATIONAL-3 or COMP-3 (internal decimal)**

This is the equivalent of PACKED-DECIMAL.

**COMPUTATIONAL-4 or COMP-4 (binary)**

This is the equivalent of BINARY.

**COMPUTATIONAL-5 or COMP-5 (native binary)**

These data items are represented in storage as binary data. The data items can contain values 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 nines in the picture for the item (as is the case for USAGE BINARY data). When numeric data is moved or stored into a COMP-5 item, truncation occurs at the binary field size rather than at the COBOL picture size limit. When a COMP-5 item is referenced, the full binary field size is used in the operation.

The TRUNC(BIN) compiler option causes all binary data items (USAGE BINARY, COMP, COMP-4) to be handled as if they were declared USAGE COMP-5.

The following table shows several picture character strings, the resulting storage representation, and the range of values for data items described with USAGE COMP-5.

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

The picture for a COMP-5 data item can specify a scaling factor (that is, decimal positions or implied integer positions). In this case, the maximal capacities listed in the table above must be scaled appropriately. For example, a data item described with PICTURE S99V99 COMP-5 is represented in storage as a binary halfword, and supports a range of values from -327.68 to +327.67.

**USAGE NOTE:** When the ON SIZE ERROR phrase is used on an arithmetic statement and a receiver is defined with USAGE COMP-5, the maximum value that the receiver can contain is the value implied by the item's decimal PICTURE character-string. Any attempt to store a value larger than this maximum will result in a size error condition.

**DISPLAY phrase**

The data item is stored in character form, one character for each 8-bit byte. This corresponds to the format used for printed output. DISPLAY can be explicit or implicit.

**USAGE IS DISPLAY** is valid for the following types of items:

- Alphabetic
- Alphanumeric
- Alphanumeric-edited
- Numeric-edited
- External floating-point
- External decimal

Alphabetic, alphanumeric, alphanumeric-edited, and numeric-edited items are discussed in “Data categories and PICTURE rules” on page 194.

External decimal items with USAGE DISPLAY are sometimes referred to as zoned decimal items. Each digit of a number is represented by a single byte. The 4 high-order bits of each byte are zone bits; the 4 high-order bits of the low-order byte represent the sign of the item. The 4 low-order bits of each byte contain the value of the digit.

If the ARITH(COMPAT) compiler option is in effect, then the maximum length of an external decimal item is 18 digits. If the ARITH(EXTEND) compiler option is in effect, then the maximum length of an external decimal item is 31 digits.

The PICTURE character-string of an external decimal item can contain only:

- One or more of the symbol 9
- The operational-sign, S
- The assumed decimal point, V
- One or more of the symbol P

**DISPLAY-1 phrase**

The DISPLAY-1 phrase defines an item as DBCS. The data item is stored in character form, with each character occupying 2 bytes of storage.

**FUNCTION-POINTER phrase**

The FUNCTION-POINTER phrase defines an item as a function-pointer data item. A function-pointer data item can contain the address of a descriptor for a procedure entry point.

A function-pointer is a 4-byte elementary item or an 8-byte elementary item depending on whether the LP(32) or LP(64) is in effect. If LP(32) is in effect, 4 bytes are allocated for the item; otherwise, 8 bytes are
allocated for the item. Function-pointers have the same capabilities as procedure-pointers. Function-pointers are thus more easily interoperable with C function pointers.

A function-pointer can point to a function descriptor for one of the following or can contain NULL:

- The primary entry point of a COBOL program, defined by the PROGRAM-ID paragraph of the outermost program
- An alternate entry point of a COBOL program, defined by a COBOL ENTRY statement
- An entry point in a non-COBOL program

A VALUE clause for a function-pointer data item can contain only NULL or NULLS.

The GLOBAL, EXTERNAL, OCCURS and VOLATILE clauses can be used with USAGE IS FUNCTION-POINTER.

A function-pointer can be used in the same contexts as a procedure-pointer, as defined in “PROCEDURE-POINTER phrase” on page 225.

INDEX phrase

A data item defined with the INDEX phrase is an index data item.

An index data item is a 4-byte elementary item or an 8-byte elementary item depending on whether the LP(32) or LP(64) is in effect, that can be used to save index-name values for future reference. An index data item is not necessarily connected with any specific table. Through a SET statement, an index data item can be assigned an index-name value. Such a value corresponds to the occurrence number in a table.

Direct references to an index data item can be made only in a SEARCH statement, a SET statement, a relation condition, the USING phrase of the PROCEDURE DIVISION header, or the USING phrase of the CALL or ENTRY statement.

An index data item can be part of an alphanumeric group item that is referenced in a MOVE statement or an input/output statement.

An index data item saves values that represent table occurrences, yet is not necessarily defined as part of any table. There is no conversion of values when an index data item is referenced in the following circumstances:

- directly in a SEARCH or SET statement
- indirectly in a MOVE statement
- indirectly in an input or output statement

An index data item cannot be a conditional variable.

The JUSTIFIED, PICTURE, BLANK WHEN ZERO, or VALUE clauses cannot be used to describe a group item or elementary items described with the USAGE IS INDEX clause.

SYNCHRONIZED can be used with USAGE IS INDEX to obtain efficient use of the index data item.

NATIONAL phrase

The NATIONAL phrase defines an item whose content is represented in storage in UTF-16 (CCSID 1200). The class and category of the data item depend on the picture symbols that are specified in the associated PICTURE clause.

OBJECT Reference phrase

A data item defined with the OBJECT REFERENCE phrase is an object reference. An object reference data item is a 4-byte elementary item or an 8-byte elementary item depending on whether the LP(32) or LP(64) is in effect. If LP(32) is in effect, 4 bytes are allocated for the item; otherwise, 8 bytes are allocated for the item.

class-name-1
An optional class name.
You must define class-name-1 in the REPOSITORY paragraph in the configuration section of the containing class or outermost program.

If specified, class-name-1 indicates that data-name-1 always refers to an object-instance of class class-name-1 or a class derived from class-name-1.

**Important:** The programmer must ensure that the referenced object meets this requirement; violations are not diagnosed.

If class-name-1 is not specified, the object reference can refer to an object of any class. In this case, data-name-1 is a universal object reference.

You can specify data-name-1 within an alphanumeric group item without affecting the semantics of the group item. There is no conversion of values or other special handling of the object references when statements are executed that operate on the group. The group continues to behave as an alphanumeric group item.

An object reference can be defined in any section of the DATA DIVISION of a factory definition, object definition, method, or program. An object-reference data item can be used in only:

- A SET statement (format 7 only)
- A relation condition
- An INVOKE statement
- The USING or RETURNING phrase of an INVOKE statement
- The USING or RETURNING phrase of a CALL statement
- A program procedure division or ENTRY statement USING or RETURNING phrase
- A method procedure division USING or RETURNING phrase

Object-reference data items:

- Are ignored in CORRESPONDING operations
- Are unaffected by INITIALIZE statements
- Can be the subject or object of a REDEFINES clause
- Cannot be a conditional variable
- Can be written to a file (but upon subsequent reading of the record the content of the object reference is undefined)

A VALUE clause for an object-reference data item can contain only NULL or NULLS.

You can use the SYNCHRONIZED clause with the USAGE OBJECT REFERENCE clause to obtain efficient alignment of the object-reference data item.

The JUSTIFIED, PICTURE, and BLANK WHEN ZERO clauses cannot be used to describe group or elementary items defined with the USAGE OBJECT REFERENCE clause.

**POINTER phrase**

A data item defined with USAGE IS POINTER is a pointer data item or data-pointer. A pointer data item is a 4-byte elementary item or an 8-byte elementary item depending on whether the LP(32) or LP(64) is in effect. If LP(32) is in effect, 4 bytes are allocated for the item; otherwise, 8 bytes are allocated for the item.

You can use pointer data items to accomplish limited base addressing. Pointer data items can be compared for equality or moved to other pointer items.

A pointer data item can be used only:

- In an ALLOCATE statement
- In a FREE statement
- In a SET statement (format 5 only)
- In a relation condition
In the USING phrase of a CALL statement, an ENTRY statement, or the PROCEDURE DIVISION header, a POINTER data item can be set to a POINTER-32 data item, and vice versa. When the LP(32) compiler option is in effect, USAGE POINTER and USAGE POINTER-32 are synonyms. When the LP(64) compiler option is in effect, the following statements apply:

**Note:** The size of the POINTER data item is 8 bytes in the following cases.

- A POINTER data item can be SET to a value from a POINTER-32 data item. The high-order word of the POINTER data item is cleared to zero.
- Pointer data items can be part of an alphanumeric group that is referred to in a MOVE statement or an input/output statement. However, if a pointer data item is part of a group, there is no conversion of values when the statement is executed.
- A pointer data item can be the subject or object of a REDEFINES clause.
- SYNCHRONIZED can be used with USAGE IS POINTER to obtain efficient use of the pointer data item.
- A VALUE clause for a pointer data item can contain only NULL or NULLS.
- A pointer data item cannot be a conditional variable.
- A pointer data item does not belong to any class or category.

The following table lists clauses that can or cannot be used to describe group or elementary items defined with the USAGE IS POINTER.

<table>
<thead>
<tr>
<th>Can be used with USAGE IS POINTER</th>
<th>Cannot be used with USAGE IS POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLOBAL clause</td>
<td>JUSTIFIED clause</td>
</tr>
<tr>
<td>EXTERNAL clause</td>
<td>PICTURE clause</td>
</tr>
<tr>
<td>OCCURS clause</td>
<td>BLANK WHEN ZERO clause</td>
</tr>
<tr>
<td>VOLATILE clause</td>
<td></td>
</tr>
</tbody>
</table>

Pointer data items are ignored in the processing of a CORRESPONDING phrase.

A pointer data item can be written to a data set, but upon subsequent reading of the record that contains the pointer, the address contained might no longer represent a valid pointer.

USAGE IS POINTER is implicitly specified for the ADDRESS OF special register. For more information, see *Using tables (arrays) and pointers* in the *Enterprise COBOL Programming Guide*.

**POINTER-32 phrase**

A data item defined with USAGE IS POINTER-32 is a *pointer data item* or *data-pointer*. A pointer data item is a 4-byte elementary item regardless of the LP compiler option setting.

You can use POINTER-32 data items to accomplish limited base addressing. Pointer data items can be compared for equality or moved to other pointer items.

A POINTER-32 data item can be used only:

- In an ALLOCATE statement
- In a FREE statement
- In a SET statement (format 5 only)
- In a relation condition
- In the USING phrase of a CALL statement, an ENTRY statement, or the PROCEDURE DIVISION header

A POINTER-32 data item can be set to a POINTER data item, and vice versa. When the LP(32) compiler option is in effect, USAGE POINTER and USAGE POINTER-32 are synonyms. When the LP(64) compiler option is in effect, the following statements apply:
• A POINTER-32 data item can be SET to a value from a POINTER data item. Only the low-order word of the POINTER data item is used in this case. It is a programming error if the high-order word of the 64-bit pointer data item is not zero.

• A POINTER-32 data item can be compared with a POINTER data item in a relation condition. The bit representation in the POINTER-32 data item is extended to 64-bit if the high-order word is zero before the comparison.

• POINTER-32 data items can be part of an alphanumeric group that is referred to in a MOVE statement or an input/output statement. However, if a pointer data item is part of a group, there is no conversion of values when the statement is executed.

• A POINTER-32 data item can be the subject or object of a REDEFINES clause.

• SYNCHRONIZED can be used with USAGE IS POINTER-32 to obtain efficient use of the pointer data item.

• A VALUE clause for a POINTER-32 data item can contain only NULL or NULLS.

• A POINTER-32 data item cannot be a conditional variable.

• A POINTER-32 data item does not belong to any class or category.

The following table lists clauses that can or cannot be used to describe group or elementary items defined with the USAGE IS POINTER-32.

<table>
<thead>
<tr>
<th>Can be used with USAGE IS POINTER-32</th>
<th>Cannot be used with USAGE IS POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLOBAL clause</td>
<td>JUSTIFIED clause</td>
</tr>
<tr>
<td>EXTERNAL clause</td>
<td>PICTURE clause</td>
</tr>
<tr>
<td>OCCURS clause</td>
<td>BLANK WHEN ZERO clause</td>
</tr>
<tr>
<td>VOLATILE clause</td>
<td></td>
</tr>
</tbody>
</table>

POINTER-32 data items are ignored in the processing of a CORRESPONDING phrase.

A POINTER-32 data item can be written to a data set, but upon the subsequent reading of the record that contains the pointer, the address contained might no longer represent a valid pointer.

**PROCEDURE-POINTER phrase**

The PROCEDURE-POINTER phrase defines an item as a *procedure-pointer data item*. A procedure-pointer data item can contain the address of a descriptor for a procedure entry point.

A procedure-pointer data item is an 8-byte elementary item regardless of the LP compiler option setting.

A procedure-pointer data item can point to a function descriptor for one of the following or can contain NULL:

• The primary entry point of a COBOL program as defined by the program-ID paragraph of the outermost program of a compilation unit
• An alternate entry point of a COBOL program as defined by a COBOL ENTRY statement
• An entry point in a non-COBOL program

A procedure-pointer data item can be used only:

• In a SET statement (format 6 only)
• In a CALL statement (from a high-level language or LE-conforming assembler program)
• In a relation condition
• In the USING phrase of an ENTRY statement or the PROCEDURE DIVISION header

Procedure-pointer data items can be compared for equality or moved to other procedure-pointer data items.

Procedure-pointer data items can be part of a group that is referred to in a MOVE statement or an input/output statement. However, there is no conversion of values when the statement is executed. If a
A procedure-pointer data item can be the subject or object of a REDEFINES clause.
SYNCHRONIZED can be used with USAGE IS PROCEDURE-POINTER to obtain efficient alignment of the procedure-pointer data item.
The GLOBAL, EXTERNAL, OCCURS, and VOLATILE clauses can be used with USAGE IS PROCEDURE-POINTER.
A VALUE clause for a procedure-pointer data item can contain only NULL or NULLS.
The JUSTIFIED, PICTURE, and BLANK WHEN ZERO clauses cannot be used to describe group or elementary items defined with the USAGE IS PROCEDURE-POINTER clause.
A procedure-pointer data item cannot be a conditional variable.
A procedure-pointer data item does not belong to any class or category.
Procedure-pointer data items are ignored in CORRESPONDING operations.

NATIVE phrase
The NATIVE phrase is syntax checked, but has no effect on the execution of the program.

UTF-8 phrase
The UTF-8 phrase defines an item whose content is represented in storage in UTF-8 (CCSID 1208). The class and category of the data item are UTF-8.

VALUE clause
The VALUE clause specifies the initial contents of a data item or the values associated with a condition-name. The use of the VALUE clause differs depending on the DATA DIVISION section in which it is specified.
In the WORKING-STORAGE SECTION and the LOCAL-STORAGE SECTION, the VALUE clause can be used in condition-name entries or in specifying the initial value of any data item. The data item assumes the specified value at the beginning of program execution. If the initial value is not explicitly specified, the value is unpredictable.

Format 1
Format 1 specifies the initial value of a data item. Initialization is independent of any BLANK WHEN ZERO or JUSTIFIED clause that is specified.

A format-1 VALUE clause specified in a data description entry that contains or is subordinate to an OCCURS clause causes every occurrence of the associated data item to be assigned the specified value. Each structure that contains the DEPENDING ON phrase of the OCCURS clause is assumed to contain the maximum number of occurrences for the purposes of VALUE initialization.
The VALUE clause must not be specified for a data description entry that contains or is subordinate to an entry that contains either an EXTERNAL or a REDEFINES clause. This rule does not apply to condition-name entries.
A format-1 VALUE clause can be specified for an elementary data item or for a group item. When the VALUE clause is specified at the group level, the group area is initialized without consideration for the
subordinate entries within the group. In addition, a VALUE clause must not be specified for subordinate entries within the group.

For group items, the VALUE clause must not be specified if any subordinate entries contain a JUSTIFIED or SYNCHRONIZED clause.

If the VALUE clause is specified for an alphanumeric group, all subordinate items must be explicitly or implicitly described with USAGE DISPLAY.

The VALUE clause must not conflict with other clauses in the data description entry or in the data description of that entry's hierarchy.

The functions of the editing characters in a PICTURE clause are ignored in determining the initial value of the item described. However, editing characters are included in determining the size of the item. Therefore, any editing characters must be included in the literal. For example, if the item is defined as PICTURE +999.99 and the value is to be +12.34, then the VALUE clause should be specified as VALUE "+012.34".

A VALUE clause cannot be specified for external floating-point items.

A data item cannot contain a VALUE clause if the prior data item contains an OCCURS clause with the DEPENDING ON phrase.

**Rules for literal values**

- Wherever a literal is specified, a figurative constant can be substituted, in accordance with the rules specified in “Figurative constants” on page 13.

- If the item is class numeric, the VALUE clause literal must be numeric. If the literal defines the value of a WORKING-STORAGE item or LOCAL-STORAGE item, the literal is aligned according to the rules for numeric moves, with one additional restriction: The literal must not have a value that requires truncation of nonzero digits. If the literal is signed, the associated PICTURE character-string must contain a sign symbol.

- With some exceptions, numeric literals in a VALUE clause must have a value within the range of values indicated by the PICTURE clause for the item. For example, for PICTURE 99PPP, the literal must be zero or within the range 1000 through 99000. For PICTURE PPP99, the literal must be within the range 0.00000 through 0.00099.

  The exceptions are the following ones:
  - Data items described with usage COMP-5 that do not have a picture symbol P in their PICTURE clause.
  - When the TRUNC(BIN) compiler option is in effect, data items described with usage BINARY, COMP, or COMP-4 that do not have a picture symbol P in their PICTURE clause.

  A VALUE clause for these items can have a value up to the capacity of the native binary representation.

- If the VALUE clause is specified for an elementary alphabetic, alphanumeric, alphanumeric-edited, or numeric-edited item described with usage DISPLAY, the VALUE clause literal must be an alphanumeric literal or a figurative constant. The literal is aligned according to the alphanumeric alignment rules, with one additional restriction: the number of characters in the literal must not exceed the size of the item.

- If the VALUE clause is specified for an elementary national, national-edited, or numeric-edited item described with usage NATIONAL, the VALUE clause literal must be a national or alphanumeric literal or a figurative constant as specified in “Figurative constants” on page 13. The value of an alphanumeric literal is converted from its source code representation to UTF-16 representation. The literal is aligned according to the national alignment rules, with one additional restriction: the number of characters in the literal must not exceed the size, in character positions, of the item.

- If the VALUE clause is specified for an elementary UTF-8 item, the VALUE clause literal must be a UTF-8 or an alphanumeric literal or a figurative constant as specified in “Figurative constants” on page 13. The value of an alphanumeric or national literal is converted from its source code representation to UTF-8 representation. The literal is aligned according to the UTF-8 alignment rules, with one additional restriction: the number of characters in the literal must not exceed the size, in character positions, of the item.
restriction: if the UTF-8 item is a fixed character-length UTF-8 item (i.e., it was not defined with the
BYTE-LENGTH phrase of the PICTURE clause or the DYNAMIC LENGTH clause), then the number of
characters in the literal must not exceed the size, in character positions, of the item. Otherwise, the
number of bytes of the literal must not exceed the maximum byte length allowed by the item.

• If the VALUE clause is specified at the group level for an alphanumeric group, the literal must be an
alphanumeric literal or a figurative constant as specified in “Figurative constants” on page 13, other
than ALL national-literal or ALL utf-8-literal. The size of the literal must not exceed the size of the group
item.

• If the VALUE clause is specified at the group level for a national group, the literal can be an
alphanumeric literal, a national literal, or one of the figurative constants ZERO, SPACE, QUOTES, HIGH-
VALUE, LOW-VALUE, symbolic character, ALL national-literal, or ALL -literal. The value of an
alphanumeric literal is converted from its source code representation to UTF-16 representation. Each
figurative constant represents a national character value. The size of the literal must not exceed the size
of the group item.

• If the VALUE clause is specified at the group level for a UTF-8 group, the literal can be an
alphanumeric literal, a UTF-8 literal, or one of the figurative constants ZERO, SPACE, QUOTES, HIGH-
VALUE, LOW-VALUE, symbolic character, ALL utf-8-literal, or ALL -literal. The value of an
alphanumeric literal is converted from its source code representation to UTF-8 representation. Each
figurative constant represents a UTF-8 character value. The size of the literal must not exceed the size
of the group item.

• A VALUE clause associated with a DBCS item must contain a DBCS literal, the
figurative
constant SPACE,
or the
figurative
constant ALL DBCS-literal. The length of the literal must not exceed the size indicated
by the data item’s PICTURE clause.

• A VALUE clause that specifies a national literal can be associated only with a data item of class national.

• A VALUE clause that specifies a UTF-8 literal can be associated only with a data item of class UTF-8.

• A VALUE clause that specifies a DBCS literal can be associated only with a data item of class DBCS.

• A VALUE clause associated with a COMPUTATIONAL-1 or COMPUTATIONAL-2 (internal floating-point)
item must specify a floating-point literal. In addition, the figurative constant ZERO and both integer and
decimal forms of the zero literal can be specified in a floating-point VALUE clause.

You cannot specify a floating-point format numeric literal in the VALUE clause of a fixed-point numeric
item.

For information about floating-point literal values, see “Rules for floating-point literal values” on page
41.

Format 2
This format associates a value, values, or ranges of values with a condition-name. Each such condition-
name requires a separate level-88 entry. Level-number 88 and the condition-name are not part of the
format-2 VALUE clause itself. They are included in the format only for clarity.
**Format 2: condition-name value**

```
format-2: condition-name value

condition-name-1
  VALUE
    IS
    VALUES
      ARE

literal-1
  THROUGH
    literal-2

  WHEN
    SET
    TO
    FALSE
    IS
  literal-3
```

**condition-name-1**
A user-specified name that associates a value with a conditional variable. If the associated conditional variable requires subscripts or indexes, each procedural reference to the condition-name must be subscripted or indexed as required for the conditional variable.

Condition-names are tested procedurally in condition-name conditions (see “Conditional expressions” on page 246).

**literal-1**
Associates the condition-name with a single value.

The class of literal-1 must be a valid class for assignment to the associated conditional variable.

**literal-1 THROUGH literal-2**
Associates the condition-name with at least one range of values. When the THROUGH phrase is used, literal-1 must be less than literal-2. For details, see “Rules for condition-name entries” on page 230.

literal-1 and literal-2 must be of the same class. The class of literal-1 and literal-2 must be a valid class for assignment to the associated conditional variable.

When literal-1 and literal-2 are DBCS literals, the range of DBCS values specified by the THROUGH phrase is based on the binary collating sequence of the hexadecimal values of the DBCS characters.

When literal-1 and literal-2 are NATIONAL literals, the range of national character values specified by the THROUGH phrase is based on the binary collating sequence of the hexadecimal values of the national characters represented by the literals.

When literal-1 and literal-2 are UTF-8 literals, the range of UTF-8 character values specified by the THROUGH phrase is based on the binary collating sequence of the hexadecimal values of the UTF-8 characters represented by the literals.

If the associated conditional variable is of class DBCS, literal-1 and literal-2 must be DBCS literals. The figurative constant SPACE or the figurative constant ALL DBCS-literal can be specified.

If the associated conditional variable is of class NATIONAL, literal-1 and literal-2 must be either both national literals or both alphanumeric literals for a given condition-name. The figurative constants ZERO, SPACE, QUOTE, HIGH-VALUE, LOW-VALUE, symbolic-character, ALL national-literal, or ALL literal can be specified.

If the associated conditional variable is of class UTF-8, literal-1 and literal-2 must be either both UTF-8 literals or both alphanumeric literals for a given condition-name. The figurative constants...
ZERO, SPACE, QUOTE, HIGH-VALUE, LOW-VALUE, symbolic-character, ALL utf-8-literal, or ALL literal can be specified.

**WHEN SET TO FALSE**

Allows specification of a FALSE condition value. This value is moved to the associated conditional variable when the SET TO FALSE statement is executed for the associated condition-name.

**literal-3**

When a condition-name is referenced in a SET TO FALSE statement, the value of literal-3 from the FALSE phrase is placed in the associated conditional variable.

*Note:* The true values of a conditional variable are all the values associated with its condition-names, and all other values are false values. The WHEN SET TO FALSE phrase specifies just one of possibly many false values. The purpose is to define a single value to be used by the SET TO FALSE statement. When a condition-name condition is tested, all the non-true values give a false result, not just the one false value defined by the WHEN SET TO FALSE phrase.

**Rules for condition-name entries**

There are certain rules for condition-name entries.

The rules are:

- The VALUE clause is required in a condition-name entry, and must be the only clause in the entry. Each condition-name entry is associated with a preceding conditional variable. Thus every level-88 entry must always be preceded either by the entry for the conditional variable or by another level-88 entry when several condition-names apply to one conditional variable. Each such level-88 entry implicitly has the PICTURE characteristics of the conditional variable.
- A space, a separator comma, or a separator semicolon must separate successive operands.
  - Each entry must end with a separator period.
- The keywords THROUGH and THRU are equivalent.
- The condition-name entries associated with a particular conditional variable must immediately follow the conditional variable entry. The conditional variable can be any elementary data description entry except the following ones:
  - Another condition-name
  - A RENAMES clause (level-66 item)
  - An item described with USAGE IS INDEX
  - An item described with USAGE POINTER, USAGE PROCEDURE-POINTER, USAGE FUNCTION-POINTER, or USAGE OBJECT REFERENCE
- Condition-names can be specified both at the group level and at subordinate levels within an alphanumeric group, national group or UTF-8 group.
- When the condition-name is specified for an alphanumeric group data description entry:
  - The value of literal-1 (or literal-1 and literal-2) must be specified as an alphanumeric literal or figurative constant.
  - The group can contain items of any usage.
- When the condition-name is specified for a national group data description entry:
  - The value of literal-1 (or literal-1 and literal-2) must be specified as an alphanumeric literal, a national literal, or a figurative constant.
  - The group can contain only items of usage national, as specified for the “GROUP-USAGE clause” on page 180.
- When the condition-name is specified for a UTF-8 group data description entry:
  - The value of literal-1 (or literal-1 and literal-2) must be specified as an alphanumeric literal, a UTF-8 literal, or a figurative constant.
– The group can contain only items of usage UTF-8, as specified for the “GROUP-USAGE clause” on page 180.

• When the condition-name is associated with an alphanumeric group data description entry or a national group data description entry or a UTF-8 group data description entry:
  – The size of each literal value must not exceed the sum of the sizes of all the elementary items within the group.
  – No element within the group can contain a JUSTIFIED or SYNCHRONIZED clause.

• Relation tests implied by the definition of a condition-name are performed in accordance with the rules referenced in the table below.

<table>
<thead>
<tr>
<th>Table 18. Relation test references for condition-names</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type of conditional variable</strong></td>
</tr>
<tr>
<td>---------------------------------</td>
</tr>
<tr>
<td>Alphanumeric group item</td>
</tr>
<tr>
<td>National group item (treated as elementary data item of class national)</td>
</tr>
<tr>
<td>Elementary data item of class alphanumeric</td>
</tr>
<tr>
<td>Elementary data item of class national</td>
</tr>
<tr>
<td>Elementary data item of class numeric</td>
</tr>
<tr>
<td>Elementary data item of class DBCS</td>
</tr>
<tr>
<td>Elementary data item of class UTF-8</td>
</tr>
</tbody>
</table>

• A VALUE clause that specifies a national literal can be associated with a condition-name defined only for a data item of class national.

• A VALUE clause that specifies a DBCS literal can be associated with a condition-name defined only for a data item of class DBCS.

• The literals in a condition-name entry for an elementary data item of class national or a national group item must be either national literals or alphanumeric literals, and literal-1 and literal-2 must be of the same class. For alphanumeric groups or elementary data items of other classes, the type of literal must be consistent with the data type of the conditional variable. In the following example:

  – CITY-COUNTY-INFO, COUNTY-NO, and CITY are conditional variables.

  The PICTURE associated with COUNTY-NO limits the condition-name value to a two-digit numeric literal.

  The PICTURE associated with CITY limits the condition-name value to a three-character alphanumeric literal.

  – The associated condition-names are level-88 entries.

  Any values for the condition-names associated with CITY-COUNTY-INFO cannot exceed five characters.

  Because this is an alphanumeric group item, the literal must be alphanumeric.
Format 3
This format assigns an invalid address as the initial value of an item defined as USAGE POINTER, USAGE
PROCEDURE-POINTER, or USAGE FUNCTION-POINTER. It also assigns an invalid object reference as the
initial value of an item defined as USAGE OBJECT REFERENCE.

<table>
<thead>
<tr>
<th>Format 3: NULL value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALUE IS NULL</td>
</tr>
</tbody>
</table>

VALUE IS NULL can be specified only for elementary items described implicitly or explicitly as USAGE
POINTER, USAGE PROCEDURE-POINTER, USAGE FUNCTION-POINTER, or USAGE OBJECT REFERENCE.

VOLATILE clause
The VOLATILE clause indicates that a data item’s value can be modified or referenced in ways that the
compiler cannot detect, such as by a Language Environment (LE) condition handler routine or by some
other asynchronous process or thread. Thus, optimization is restricted for the data item.

<table>
<thead>
<tr>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>VOLATILE</td>
</tr>
</tbody>
</table>

In particular, the compiler will enforce the following restrictions:

- A volatile data item is loaded from memory each time it is referenced and stored to memory each time it
  is modified.
- Loads and stores to the data item are never reordered or eliminated.
- Storage is always allocated for the data item and initialized where necessary, even when no references
to the data item are in the compilation unit.

**Note:** The STGOPT option is ignored for data items that have the VOLATILE clause.

The VOLATILE clause can be specified on data items that are defined in the FILE SECTION, WORKING-
STORAGE SECTION, LOCAL-STORAGE SECTION, and LINKAGE SECTION. This clause can be specified
together with any other clauses. For example, VOLATILE can be specified on tables, group data items,
elementary data items, record descriptions and variably located data items.

There are additional considerations for groups:

- When a group item is explicitly defined with the VOLATILE clause, all items subordinate to the group
  item are treated as volatile by the compiler.
- When a group item has one or more subordinate items that are explicitly defined with the VOLATILE
  clause, the group item is treated as volatile by the compiler.

The VOLATILE clause cannot be specified on level-66 or level-88 data items.

It is not possible to indicate that all memory associated with a class instance is volatile. However,
individual members of a class can be defined with the VOLATILE clause.

**Example of using VOLATILE with groups:**
Consider the following group definition:
In this example:

- **DATA-ITEMS-A** and **DATA-B2** are considered volatile because they are defined with the VOLATILE clause.
- **DATA-A1** and **DATA-A2** are treated as volatile because they are both subordinate to a group item (DATA-ITEMS-A) that has the VOLATILE clause.
- **DATA-COLLECTION** and **DATA-ITEMS-B** are treated as volatile because they are group items that have subordinates that are defined with the VOLATILE clause. For example:

  ```plsql
  MOVE DATA-ITEMS-B TO DATA-ITEMS-C.
  ```

  In this case, by treating DATA-ITEMS-B as volatile, the compiler ensures that the latest value of its subordinate member DATA-B2 is used in the memory copy operation.

In the following LE condition handler scenario, it is necessary to specify the "STEP" data item with the VOLATILE clause to achieve correct results. In particular, if the VOLATILE clause is not used, the compiler might assume that "STEP" is never referenced between the assignment of "2" to "STEP" and the assignment of "3" to "STEP" and might therefore decide to eliminate the first assignment during optimization. Unfortunately, this could result in a problem because if a divide-by-zero condition occurs during execution of the subsequent line of code, the condition handler will execute and reference the external variable "STEP", which might have the incorrect value.

**Main program:**

```plsql
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 USER-HANDLER PROCEDURE-POINTER.
77 TOKEN   PIC S9(9) COMP.
01 QTY     PIC 9(8)  BINARY.
01 DIVISOR PIC 9(8)  BINARY VALUE 0.
01 ANSWER  PIC 9(8)  BINARY.
01 STEP    PIC 9(8)  BINARY VALUE 0 EXTERNAL VOLATILE.
):
SET USER-HANDLER TO ENTRY 'HANDLER'
CALL 'CEEHDLR' USING USER-HANDLER, TOKEN, NULL
COMPUTE STEP = 2
COMPUTE ANSWER = NUMBER / DIVISOR
COMPUTE STEP = 3
COMPUTE ANSWER = QTY + 2
 :
```

**Condition handler program:**

```plsql
IDENTIFICATION DIVISION.
PROGRAM-ID. HANDLER.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 STEP PIC 9(8) BINARY EXTERNAL.
PROCEDURE DIVISION.
):
DISPLAY 'ERROR: A PROBLEM WAS ENCOUNTERED IN STEP ' STEP.
```
Part 6. Procedure division
Chapter 19. Procedure division structure

The PROCEDURE DIVISION is an optional division.

**Program procedure division**
The program procedure division consists of optional declaratives, and procedures that contain sections, paragraphs, sentences, and statements.

**Factory procedure division**
The factory procedure division contains only factory method definitions.

**Object procedure division**
The object procedure division contains only object method definitions.

**Method procedure division**
A method procedure division consists of optional declaratives, and procedures that contain sections, paragraphs, sentences, and statements. A method can INVOKE other methods, be recursively invoked, and issue a CALL to a program. A method procedure division cannot contain nested programs or methods.

For additional details on a method procedure division, see “Requirements for a method procedure division” on page 238.

---

**Format: procedure division**

```
procedure-division-header

factory-or-object-procedure-division-header

method-procedure-division-header

DECLARATIVES.

sect . use-statement 1 para

section-name 2 SECTION priority-number 3 para

sect

section-name . SECTION priority-number 3 para

para

paragraph-name.

sentence
```

Notes:
1. The USE statement is described under “USE statement” on page 541.
2. Section-name can be omitted. If you omit section-name, paragraph-name can be omitted.

© Copyright IBM Corp. 1991, 2020
Requirements for a method procedure division

There are specific requirements when you code a method procedure division.

The requirements are:

- You can use the EXIT METHOD statement or the GOBACK statement to return control to the invoking method or program. An implicit EXIT METHOD statement is generated as the last statement of every method procedure division. For details on the EXIT METHOD statement, see “Format 3 (method)” on page 317.
- You can use the STOP RUN statement (which terminates the run unit) in a method.
- You can use the RETURN-CODE special register within a method procedure division to access return codes from subprograms that are called with the CALL statement, but the RETURN-CODE value is not returned to the invoker of the current method. Use the procedure division RETURNING data name to return a value to the invoker of the current method. For details, see the discussion of RETURNING data-name-2 under “The PROCEDURE DIVISION header” on page 238.

You cannot specify the following statements or clauses in a method procedure division:

- ALTER
- ENTRY
- EXIT PROGRAM
- GO TO without a specified procedure name
- SEGMENT-LIMIT
- USE FOR DEBUGGING

The PROCEDURE DIVISION header

The PROCEDURE DIVISION, if specified, is identified by one of the following headers, depending on whether you are specifying a program, a factory definition, an object definition, or a method definition.

The following syntax diagram shows the format for a PROCEDURE DIVISION header in a program.
The following syntax diagram shows the format for a PROCEDURE DIVISION header in a factory paragraph or object paragraph.

**Format: factory and object procedure division header**

```
PROCEDURE DIVISION.
```

The following syntax diagram shows the format for a PROCEDURE DIVISION header in a method.

**Format: method procedure division header**

```
PROCEDURE DIVISION
```

The **USING phrase**

The USING phrase specifies the parameters that a program or method receives when the program is called or the method is invoked.

The USING phrase is valid in the PROCEDURE DIVISION header of a called subprogram or invoked method entered at the beginning of the nondeclaratives portion. Each USING identifier must be defined as a level-01 or level-77 item in the LINKAGE SECTION of the called subprogram or invoked method.

In a called subprogram entered at the first executable statement following an ENTRY statement, the USING phrase is valid in the ENTRY statement. Each USING identifier must be defined as a level-01 or level-77 item in the LINKAGE SECTION of the called subprogram.

However, a data item specified in the USING phrase of the CALL statement can be a data item of any level in the DATA DIVISION of the calling COBOL program or method. A data item specified in the USING
phrase of an INVOKE statement can be a data item of any level in the DATA DIVISION of the invoking COBOL program or method.

A data item in the USING phrase of the header can have a REDEFINES clause in its data description entry. It is possible to call COBOL programs from non-COBOL programs or to pass user parameters from a system command to a COBOL main program. COBOL methods can be invoked only from Java or COBOL.

The order of appearance of USING identifiers in both calling and called subprograms, or invoking methods or programs and invoked methods, determines the correspondence of single sets of data available to both. The correspondence is positional and not by name. For calling and called subprograms, corresponding identifiers must contain the same number of bytes although their data descriptions need not be the same.

For index-names, no correspondence is established. Index-names in calling and called programs, or invoking method or program and invoked methods, always refer to separate indexes.

The identifiers specified in a CALL USING or INVOKE USING statement name the data items available to the calling program or invoking method or program that can be referred to in the called program or invoked method. These items can be defined in any DATA DIVISION section.

A given identifier can appear more than once in a USING phrase. The last value passed to it by a CALL or INVOKE statement is used.

The BY REFERENCE or BY VALUE phrase applies to all parameters that follow until overridden by another BY REFERENCE or BY VALUE phrase.

**BY REFERENCE (for programs only)**

When an argument is passed BY CONTENT or BY REFERENCE, BY REFERENCE must be specified or implied for the corresponding formal parameter on the PROCEDURE or ENTRY USING phrase.

BY REFERENCE is the default if neither BY REFERENCE nor BY VALUE is specified.

If the reference to the corresponding data item in the CALL statement declares the parameter to be passed BY REFERENCE (explicit or implicit), the program executes as if each reference to a USING identifier in the called subprogram is replaced by a reference to the corresponding USING identifier in the calling program.

If the reference to the corresponding data item in the CALL statement declares the parameter to be passed BY CONTENT, the value of the item is moved when the CALL statement is executed and placed into a system-defined storage item that possesses the attributes declared in the LINKAGE SECTION for data-name-1. The data description of each parameter in the BY CONTENT phrase of the CALL statement must be the same, meaning no conversion or extension or truncation, as the data description of the corresponding parameter in the USING phrase of the header.

**BY VALUE**

When an argument is passed BY VALUE, the value of the argument is passed, not a reference to the sending data item. The receiving subprogram or method has access only to a temporary copy of the sending data item. Any modifications made to the formal parameters that correspond to an argument passed BY VALUE do not affect the argument.

Parameters specified in the USING phrase of a method procedure division header must be passed to the method BY VALUE.

See **Passing data** in the *Enterprise COBOL Programming Guide* for examples that illustrate these concepts.

**data-name-1**

data-name-1 must be a level-01 or level-77 item in the LINKAGE SECTION.

When data-name-1 is an object reference in a method procedure division header, an explicit class-name must be specified in the data description entry for that object reference; that is, data-name-1 must not be a universal object reference.

For methods, the parameter data types are restricted to the data types that are interoperable between COBOL and Java, as listed in "Interoperable data types for COBOL and Java" on page 339.
RETURNING phrase

The RETURNING phrase specifies a data item that is to receive the program or method result.

\textit{data-name-2}

\textit{data-name-2} is the RETURNING data item. \textit{data-name-2} must be a level-01 or level-77 item in the LINKAGE SECTION.

\textbf{Note:} An unbounded group cannot be specified as \textit{data-name-2}.

In a method procedure division header, the data type of \textit{data-name-2} must be one of the types supported for Java interoperation, as listed in “Interoperable data types for COBOL and Java” on page 339.

The RETURNING data item is an output-only parameter. On entry to the method, the initial state of the RETURNING data item has an undefined and unpredictable value. You must initialize the PROCEDURE DIVISION RETURNING data item before you reference its value. The value that is returned to the invoking routine is the value that the data item has at the point of exit from the method. See “RETURNING phrase” on page 337 for further details on conformance requirements for the INVOKE RETURNING identifier and the method RETURNING data item.

Do not use the PROCEDURE DIVISION RETURNING phrase in:

- Programs that contain the ENTRY statement.
- Nested programs.
- Main programs: Results of specifying PROCEDURE DIVISION RETURNING on a main program are undefined. You should specify the PROCEDURE DIVISION RETURNING phrase only on called subprograms. For main programs, use the RETURN-CODE special register to return a value to the operating environment.

References to items in the LINKAGE SECTION

Data items defined in the LINKAGE SECTION of the called program or invoked method can be referenced within the PROCEDURE DIVISION of that program if and only if they satisfy one of the conditions as listed in the topic.

- They are operands of the USING phrase of the PROCEDURE DIVISION header or the ENTRY statement.
- They are operands of \textit{SET ADDRESS OF}, \textit{ALLOCATE}, \textit{CALL \ldots\ BY REFERENCE ADDRESS OF}, or \textit{INVOKE \ldots\ BY REFERENCE ADDRESS OF}.
- They are defined with a \textit{REDEFINES} or \textit{RENAMES} clause, the object of which satisfies the above conditions.
- They are items subordinate to any item that satisfies the condition in the rules above.
- They are condition-names or index-names associated with data items that satisfy any of the above conditions.

\textbf{Declaratives}

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

When declarative sections are specified, they must be grouped at the beginning of the procedure division and the entire PROCEDURE DIVISION must be divided into sections.

Each declarative section starts with a USE statement that identifies the section's function. The series of procedures that follow specify the actions that are to be taken when the exceptional condition occurs. Each declarative section ends with another section-name followed by a USE statement, or with the keywords END DECLARATIVES.

The entire group of declarative sections is preceded by the keyword DECLARATIVES written on the line after the PROCEDURE DIVISION header. The group is followed by the keywords END DECLARATIVES. The
keywords DECLARATIVES and END DECLARATIVES must each begin in Area A and be followed by a separator period. No other text can appear on the same line.

In the declaratives part of the PROCEDURE DIVISION, each section header must be followed by a separator period, and must be followed by a USE statement followed by a separator period. No other text can appear on the same line.

The USE statement has the following formats:

- “EXCEPTION/ERROR declarative” on page 541
- “DEBUGGING declarative” on page 543

The USE statement itself is never executed; instead, the USE statement defines the conditions that execute the succeeding procedural paragraphs, which specify the actions to be taken. After the procedure is executed, control is returned to the routine that activated it.

A declarative procedure can be performed from a nondeclarative procedure.
A nondeclarative procedure can be performed from a declarative procedure.
A declarative procedure can be referenced in a GO TO statement in a declarative procedure.
A nondeclarative procedure can be referenced in a GO TO statement in a declarative procedure.

You can include a statement that executes a previously called USE procedure that is still in control. However, to avoid an infinite loop, you must be sure there is an eventual exit at the bottom.

The declarative procedure is exited when the last statement in the procedure is executed.

### Procedures

Within the PROCEDURE DIVISION, a procedure consists of a section or a group of sections, and a paragraph or group of paragraphs.

A procedure-name is a user-defined name that identifies a section or a paragraph.

**Section**

A section-header optionally followed by one or more paragraphs.

**Section-header**

A section-name followed by the keyword SECTION, optionally followed by a priority-number, followed by a separator period.

Section-headers are optional after the keywords END DECLARATIVES or if there are no declaratives.

**Section-name**

A user-defined word that identifies a section. A referenced section-name, because it cannot be qualified, must be unique within the program in which it is defined.

**Priority-number**

An integer or a positive signed numeric literal ranging in value from 0 through 99. Priority-number identifies a fixed segment or an independent segment that is to contain the section.

Sections in the declaratives portion must contain priority numbers in the range of 0 through 49.

You cannot specify priority-numbers:

- In a method definition
- In a program that is declared with the RECURSIVE attribute
- In a program compiled with the THREAD compiler option

A section ends immediately before the next section header, or at the end of the PROCEDURE DIVISION, or, in the declaratives portion, at the keywords END DECLARATIVES.
Segments
A segment consists of all sections in a program that have the same priority-number. Priority-number determines whether a section is stored in a fixed segment or an independent segment at run time.

Segments with a priority-number of 0 through 49 are fixed segments. Segments with a priority-number of 50 through 99 are independent segments.

The type of segment (fixed or independent) controls the segmentation feature.

In fixed segments, procedures are always in last-used state. In independent segments, procedures are in initial state each time the segment receives control from a segment with a different priority-number, except when the transfer of control results from the execution of a GOBACK or EXIT PROGRAM statement. Restrictions on the use of ALTER, SORT, and MERGE statements in independent segments are described under those statements.

Enterprise COBOL does not support the overlay feature of the 85 COBOL Standard segmentation module.

Paragraph
A paragraph-name followed by a separator period, optionally followed by one or more sentences.

Paragraphs must be preceded by a period because paragraphs always follow either the IDENTIFICATION DIVISION header, a section, or another paragraph, all of which must end with a period.

Paragraph-name
A user-defined word that identifies a paragraph. A paragraph-name, because it can be qualified, need not be unique.

If there are no declaratives (format 2), a paragraph-name is not required in the PROCEDURE DIVISION.

A paragraph ends immediately before the next paragraph-name or section header, or at the end of the PROCEDURE DIVISION, or, in the declaratives portion, at the keywords END DECLARATIVES.

Paragraphs need not all be contained within sections, even if one or more paragraphs are so contained.

Sentence
One or more statements terminated by a separator period.

Statement
A syntactically valid combination of identifiers and symbols (literals, relational-operators, and so forth) beginning with a COBOL statement.

Identifier
The word or words necessary to make unique reference to a data item, optionally including qualification, subscripting, indexing, and reference-modification. In any PROCEDURE DIVISION reference (except the class test), the contents of an identifier must be compatible with the class specified through its PICTURE clause, otherwise results are unpredictable.

Execution begins with the first statement in the PROCEDURE DIVISION, excluding declaratives. Statements are executed in the order in which they are presented for compilation, unless the statement rules dictate some other order of execution.

The end of the PROCEDURE DIVISION is indicated by one of the following items:

- An IDENTIFICATION DIVISION header that indicates the start of a nested source program
- An END PROGRAM, END METHOD, END FACTORY, or END OBJECT marker
- The physical end of a program; that is, the physical position in a source program after which no further source program lines occur
**Arithmetic expressions**

Arithmetic expressions are used as operands of certain conditional and arithmetic statements.

An arithmetic expression can consist of any of the following items:

1. An identifier described as a numeric elementary item (including numeric functions)
2. A numeric literal
3. The figurative constant ZERO
4. Identifiers and literals, as defined in items 1, 2, and 3, separated by arithmetic operators
5. Two arithmetic expressions, as defined in items 1, 2, 3, or 4, separated by an arithmetic operator
6. An arithmetic expression, as defined in items 1, 2, 3, 4, or 5, enclosed in parentheses

Any arithmetic expression can be preceded by a unary operator.

Identifiers and literals that appear in arithmetic expressions must represent either numeric elementary items or numeric literals on which arithmetic can be performed.

If an exponential expression is evaluated as both a positive and a negative number, the result is always the positive number. For example, the square root of 4:

```
4 ** 0.5
```

is evaluated as +2 and -2. Enterprise COBOL always returns +2.

If the value of an expression to be raised to a power is zero, the exponent must have a value greater than zero. Otherwise, the size error condition exists. In any case where no real number exists as the result of an evaluation, the size error condition exists.

**Arithmetic operators**

Five binary arithmetic operators and two unary arithmetic operators can be used in arithmetic expressions. These operators are represented by specific characters that must be preceded and followed by a space.

These binary and unary arithmetic operators are shown in Table 19 on page 244.

<table>
<thead>
<tr>
<th>Binary operator</th>
<th>Meaning</th>
<th>Unary operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>+</td>
<td>Multiplication by +1</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>-</td>
<td>Multiplication by -1</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td></td>
<td></td>
</tr>
<tr>
<td>**</td>
<td>Exponentiation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Limitation:** Exponents in fixed-point exponential expressions cannot contain more than nine digits. The compiler will truncate any exponent with more than nine digits. In the case of truncation, the compiler will issue a diagnostic message if the exponent is a literal or constant; if the exponent is a variable or data-name, a diagnostic message is issued at run time.

Parentheses can be used in arithmetic expressions to specify the order in which elements are to be evaluated.

Expressions within parentheses are evaluated first. When expressions are contained within nested parentheses, evaluation proceeds from the least inclusive to the most inclusive set.
When parentheses are not used, or parenthesized expressions are at the same level of inclusiveness, the following hierarchic order is implied:

1. Unary operator
2. Exponentiation
3. Multiplication and division
4. Addition and subtraction

Parentheses either eliminate ambiguities in logic where consecutive operations appear at the same hierarchic level, or modify the normal hierarchic sequence of execution when this is necessary. When the order of consecutive operations at the same hierarchic level is not completely specified by parentheses, the order is from left to right.

An arithmetic expression can begin only with a left parenthesis, a unary operator, or an operand (that is, an identifier or a literal). It can end only with a right parenthesis or an operand. An arithmetic expression must contain at least one reference to an identifier or a literal.

There must be a one-to-one correspondence between left and right parentheses in an arithmetic expression, with each left parenthesis placed to the left of its corresponding right parenthesis.

If the first operator in an arithmetic expression is a unary operator, it must be immediately preceded by a left parenthesis if that arithmetic expression immediately follows an identifier or another arithmetic expression.

The following table shows permissible arithmetic symbol pairs. An arithmetic symbol pair is the combination of two such symbols in sequence. In the table:

Yes
  Indicates a permissible pairing.

No
  Indicates that the pairing is not permitted.

<table>
<thead>
<tr>
<th>Table 20. <strong>Valid arithmetic symbol pairs</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier or literal first symbol</td>
</tr>
<tr>
<td>Identifier or literal second symbol</td>
</tr>
<tr>
<td>* / ** + - first symbol</td>
</tr>
<tr>
<td>Unary + or unary - first symbol</td>
</tr>
<tr>
<td>( first symbol</td>
</tr>
<tr>
<td>) first symbol</td>
</tr>
</tbody>
</table>
Conditional expressions

A conditional expression causes the object program to select alternative paths of control, depending on the truth value of a test. Conditional expressions are specified in EVALUATE, IF, PERFORM, and SEARCH statements.

A conditional expression can be specified in either simple conditions or complex conditions. Both simple and complex conditions can be enclosed within any number of paired parentheses; the parentheses do not change whether the condition is simple or complex.

Simple conditions

There are five simple conditions.

The simple conditions are:

• Class condition
• Condition-name condition
• Relation condition
• Sign condition
• Switch-status condition

A simple condition has a truth value of either true or false.

Class condition

The class condition determines whether the content of a data item is alphabetic, alphabetic-lower, alphabetic-upper, numeric, DBCS, KANJI, or contains only the characters in the set of characters specified by the CLASS clause as defined in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION.

Format

\[
\begin{align*}
\text{identifier-1} & \quad \text{IS} \quad \text{NOT} \\
& \quad \text{NUMERIC} \\
& \quad \text{ALPHABETIC} \\
& \quad \text{ALPHABETIC-LOWER} \\
& \quad \text{ALPHABETIC-UPPER} \\
& \quad \text{class-name} \\
& \quad \text{DBCS} \\
& \quad \text{KANJI}
\end{align*}
\]

identifier-1

Must reference a data item described with one of the following usages:

• DISPLAY, NATIONAL, COMPUTATIONAL-3, or PACKED-DECIMAL when NUMERIC is specified
• DISPLAY-1 when DBCS or KANJI is specified
• DISPLAY or NATIONAL when ALPHABETIC, ALPHABETIC-UPPER, or ALPHABETIC-LOWER is specified
• DISPLAY when class-name is specified

Must not be of class alphabetic when NUMERIC is specified.

Must not be of class numeric when ALPHABETIC, ALPHABETIC-UPPER, or ALPHABETIC-LOWER is specified.
Table 21 on page 248 lists the forms of class condition that are valid for each type of identifier. If `identifier-1` is a function-identifier, it must reference an alphanumeric or national function. An alphanumeric group item can be used in a class condition where an elementary alphanumeric item can be used, except that the NUMERIC class condition cannot be used if the group contains one or more signed elementary items.

When `identifier-1` is described with usage NATIONAL, the class-condition tests for the national character representation of the characters associated with the specified character class. For example, specifying a class condition of the form `IF national-item IS ALPHABETIC` is a test for the lowercase and uppercase letters Latin capital letter A through Latin capital letter Z and the space, as represented in national characters. Specifying `IF national-item IS NUMERIC` is a test for the characters 0 through 9.

When `identifier-1` is described with usage UTF-8, the class-condition tests for the UTF-8 character representation of the characters associated with the specified character class. For example, specifying a class condition of the form `IF utf8-item IS ALPHABETIC` is a test for the lowercase and uppercase letters Latin capital letter A through Latin capital letter Z and the space, as represented by UTF-8 characters.

**NOT**
When used, NOT and the next keyword define the class test to be executed for truth value. For example, NOT NUMERIC is a truth test for determining that the result of a NUMERIC class test is false (in other words, the item contains data that is nonnumeric).

**NUMERIC**
`identifier-1` consists entirely of the characters 0 through 9, with or without an operational sign.
If its PICTURE does not contain an operational sign, the identifier being tested is determined to be numeric only if the contents are numeric and an operational sign is not present.
If its PICTURE does contain an operational sign, the identifier being tested is determined to be numeric only if the item is an elementary item, the contents are numeric, and a valid operational sign is present.

**Usage note:** Valid operational signs are determined from the setting of the NUMCLS installation option and the NUMPROC compiler option. For more information, see Checking for incompatible data (numeric class test) in the Enterprise COBOL Programming Guide.

**ALPHABETIC**
`identifier-1` consists entirely of any combination of the lowercase or uppercase Latin alphabetic characters A through Z and the space.

**ALPHABETIC-LOWER**
`identifier-1` consists entirely of any combination of the lowercase Latin alphabetic characters a through z and the space.

**ALPHABETIC-UPPER**
`identifier-1` consists entirely of any combination of the uppercase Latin alphabetic characters A through Z and the space.

**class-name**
`identifier-1` consists entirely of the characters listed in the definition of class-name in the SPECIAL-NAMES paragraph.

**DBCS**
`identifier-1` consists entirely of DBCS characters.
A range check is performed on the item for valid character representation. The valid range is X'41' through X'FE' for both bytes of each DBCS character and X'4040' for the DBCS blank.

**KANJI**
`identifier-1` consists entirely of DBCS characters.
A range check is performed on the item for valid character representation. The valid range is from X'41' through X'7E' for the first byte, from X'41' through X'FE' for the second byte, and X'4040' for the DBCS blank.

### Table 21. Valid forms of the class condition for different types of data items

<table>
<thead>
<tr>
<th>Type of data item referenced by identifier-1</th>
<th>Valid forms of the class condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alphabetic</td>
<td>ALPHABETIC</td>
</tr>
<tr>
<td></td>
<td>ALPHABETIC-LOWER</td>
</tr>
<tr>
<td></td>
<td>ALPHABETIC-UPPER</td>
</tr>
<tr>
<td></td>
<td>class-name</td>
</tr>
<tr>
<td></td>
<td>NOT ALPHABETIC</td>
</tr>
<tr>
<td></td>
<td>NOT ALPHABETIC-LOWER</td>
</tr>
<tr>
<td></td>
<td>NOT ALPHABETIC-UPPER</td>
</tr>
<tr>
<td></td>
<td>NOT class-name</td>
</tr>
<tr>
<td>Alphanumeric, alphanumeric-edited, or numeric-edited</td>
<td>ALPHABETIC</td>
</tr>
<tr>
<td></td>
<td>ALPHABETIC-LOWER</td>
</tr>
<tr>
<td></td>
<td>ALPHABETIC-UPPER</td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
</tr>
<tr>
<td></td>
<td>class-name</td>
</tr>
<tr>
<td></td>
<td>NOT ALPHABETIC</td>
</tr>
<tr>
<td></td>
<td>NOT ALPHABETIC-LOWER</td>
</tr>
<tr>
<td></td>
<td>NOT ALPHABETIC-UPPER</td>
</tr>
<tr>
<td></td>
<td>NOT NUMERIC</td>
</tr>
<tr>
<td></td>
<td>NOT class-name</td>
</tr>
<tr>
<td>External-decimal or internal-decimal</td>
<td>NUMERIC</td>
</tr>
<tr>
<td></td>
<td>NOT NUMERIC</td>
</tr>
<tr>
<td>DBCS</td>
<td>DBCS</td>
</tr>
<tr>
<td></td>
<td>KANJI</td>
</tr>
<tr>
<td></td>
<td>NOT DBCS</td>
</tr>
<tr>
<td></td>
<td>NOT KANJI</td>
</tr>
<tr>
<td>National</td>
<td>NUMERIC</td>
</tr>
<tr>
<td></td>
<td>ALPHABETIC</td>
</tr>
<tr>
<td></td>
<td>ALPHABETIC-LOWER</td>
</tr>
<tr>
<td></td>
<td>ALPHABETIC-UPPER</td>
</tr>
<tr>
<td></td>
<td>class-name</td>
</tr>
<tr>
<td></td>
<td>NOT NUMERIC</td>
</tr>
<tr>
<td></td>
<td>NOT ALPHABETIC</td>
</tr>
<tr>
<td></td>
<td>NOT ALPHABETIC-LOWER</td>
</tr>
<tr>
<td></td>
<td>NOT ALPHABETIC-UPPER</td>
</tr>
<tr>
<td>Numeric</td>
<td>NUMERIC</td>
</tr>
<tr>
<td></td>
<td>class-name</td>
</tr>
<tr>
<td></td>
<td>NOT NUMERIC</td>
</tr>
<tr>
<td></td>
<td>NOT class-name</td>
</tr>
</tbody>
</table>

## Condition-name condition

A condition-name condition tests a conditional variable to determine whether its value is equal to any values that are associated with the condition-name.

### Format

```
condition-name-1
```

A condition-name is used in conditions as an abbreviation for the relation condition. The rules for comparing a conditional variable with a condition-name value are the same as those specified for relation conditions.

If `condition-name-1` has been associated with a range of values (or with several ranges of values), the conditional variable is tested to determine whether its value falls within the ranges, including the end values. The result of the test is true if one of the values that corresponds to the condition-name equals the value of its associated conditional variable.

Condition-names are allowed for alphanumeric, DBCS, national, UTF-8, and floating-point data items, as well as others, as defined for the condition-name format of the VALUE clause.
The following example illustrates the use of conditional variables and condition-names:

```
01 AGE-GROUP      PIC 99.
 88 INFANT        VALUE 0.
 88 BABY          VALUE 1, 2.
 88 CHILD         VALUE 3 THRU 12.
 88 TEENAGER      VALUE 13 THRU 19.
```

AGE-GROUP is the conditional variable; INFANT, BABY, CHILD, and TEENAGER are condition-names. For individual records in the file, only one of the values specified in the condition-name entries can be present.

The following IF statements can be added to the above example to determine the age group of a specific record:

```
IF INFANT...         (Tests for value 0)
IF BABY...           (Tests for values 1, 2)
IF CHILD...          (Tests for values 3 through 12)
IF TEENAGER...       (Tests for values 13 through 19)
```

Depending on the evaluation of the condition-name condition, alternative paths of execution are taken by the object program.

**Relation conditions**

A relation condition specifies the comparison of two operands. The relational operator that joins the two operands specifies the type of comparison. The relation condition is true if the specified relation exists between the two operands; the relation condition is false if the specified relation does not exist.

Comparisons are defined for the following cases:

- Two operands of class alphabetic
- Two operands of class alphanumeric
- Two operands of class DBCS
- Two operands of class national
- Two operands of class numeric
- Two operands of class UTF-8.
- Two operands of different classes where each operand is one of the classes alphabetic, alphanumeric, national or UTF-8
- Two operands where one is a numeric integer and the other is class alphanumeric or national
- Two operands where one is class DBCS and the other is class national
- Comparisons involving indexes or index data items
- Two data pointer operands
- Two procedure pointer operands
- Two function pointer operands
- Two object reference operands
- An alphanumeric group and any operand that has usage DISPLAY, DISPLAY-1, NATIONAL, or UTF-8.

The following relation condition formats are defined:

- A general relation condition, for comparisons that involve only data items, literals, index-names, or index data items. For details, see “General relation conditions” on page 250.
- A data pointer relation condition. For details, see “Data pointer relation conditions” on page 258.
- A program pointer relation condition, for comparison of procedure pointers or function pointers. For details, see “Procedure-pointer and function-pointer relation conditions” on page 259.
• An object-reference relation condition. For details, see “Object-reference relation conditions” on page 260.

**General relation conditions**

A general relation condition compares two operands, either of which can be an identifier, literal, arithmetic expression, or index-name.

**Format 1: general relation condition**

```
operand-1
    IS NOT GREATER THAN
    NOT GREATER
    LESS THAN
    MINUS
    EQUAL
    GREATER THAN OR EQUAL
    GREATER THAN
    LESS THAN OR EQUAL
    LESS THAN

operand-2
```

**operand-1**

The subject of the relation condition. Can be an identifier, literal, function-identifier, arithmetic expression, or index-name.

**operand-2**

The object of the relation condition. Can be an identifier, literal, function-identifier, arithmetic expression, or index-name.

An alphanumeric literal can be enclosed in parentheses within a relation condition.

The relation condition must contain at least one reference to an identifier.

The relational operators, shown in Table 22 on page 250, specify the type of comparison to be made. Each relational operator must be preceded and followed by a space. The two characters of the relational operators >= and <= must not have a space between them.

**Table 22. Relational operators and their meanings**

<table>
<thead>
<tr>
<th>Relational operator</th>
<th>Can be written</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS GREATER THAN</td>
<td>IS &gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>IS NOT GREATER THAN</td>
<td>IS NOT &gt;</td>
<td>Not greater than</td>
</tr>
<tr>
<td>IS LESS THAN</td>
<td>IS &lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>IS NOT LESS THAN</td>
<td>IS NOT &lt;</td>
<td>Not less than</td>
</tr>
</tbody>
</table>
Table 22. *Relational operators and their meanings* (continued)

<table>
<thead>
<tr>
<th>Relational operator</th>
<th>Can be written</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS EQUAL TO</td>
<td>IS =</td>
<td>Equal to</td>
</tr>
<tr>
<td>IS NOT EQUAL TO</td>
<td>IS NOT =</td>
<td>Not equal to</td>
</tr>
<tr>
<td>IS GREATER THAN OR EQUAL TO</td>
<td>IS &gt;=</td>
<td>Is greater than or equal to</td>
</tr>
<tr>
<td>IS LESS THAN OR EQUAL TO</td>
<td>IS &lt;=</td>
<td>Is less than or equal to</td>
</tr>
</tbody>
</table>

In a general relation condition, data items, literals, and figurative constants of class alphabetic, alphanumeric, DBCS, national, UTF-8, and numeric are compared using the following comparison types:

**Comparison type** | **Meaning**                                                                 |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Alphanumeric</td>
<td>Comparison of the alphanumeric character value of two operands</td>
</tr>
<tr>
<td>DBCS</td>
<td>Comparison of the DBCS character value of two operands</td>
</tr>
<tr>
<td>National</td>
<td>Comparison of the national character value of two operands</td>
</tr>
<tr>
<td>UTF-8</td>
<td>Comparison of the UTF-8 character value of two operands</td>
</tr>
<tr>
<td>Numeric</td>
<td>Comparison of the algebraic value of two operands</td>
</tr>
<tr>
<td>Group</td>
<td>Comparison of the alphanumeric character value of two operands, where one or both operands is an alphanumeric group item</td>
</tr>
</tbody>
</table>

Table 23 on page 252 and Table 24 on page 253 show the permissible pairs for comparisons with different types of operands. The comparison type is indicated at the row and column intersection for permitted comparisons, using the following key:

- **Alph**
  - Comparison of alphanumeric characters (further described in “Alphanumeric comparisons” on page 253)

- **DBCS**
  - Comparison of DBCS characters (further described in “DBCS comparisons” on page 254)

- **Nat**
  - Comparison of national characters (further described in “National comparisons” on page 255)

- **UTF-8**
  - Comparison of UTF-8 characters (further described in “UTF-8 comparisons” on page 256)

- **Num**
  - Comparison of algebraic value (further described in “Numeric comparisons” on page 257)

- **Group**
  - Comparison of alphanumeric characters involving an alphanumeric group (further described in “Group comparisons” on page 257)

- **(Int)**
  - Integer items only (combined with comparison type Alph, Nat, Num, or Group)

- **Blank**
  - Comparison is not allowed

For rules and restrictions for comparisons involving index-names and index data items, see “Comparison of index-names and index data items” on page 257.

**Introduction to Table 23 on page 252:** This table is organized in the following manner:

- In the first column, under “Type of data item or literal”, each row identifies a type of operand. In some cases, the type of operand references a grouping of operands that have common properties for
comparison. For example, the row for "Alphanumeric character items" references all the types of operands that are listed in the cell, as follows:

- Data items of category:
  - Alphanumeric
  - Alphanumeric-edited
  - Numeric-edited with usage DISPLAY
- Alphanumeric functions

  - Subsequent column headings refer to the type of an operand or a grouping of operands. For example, the column heading "Alphabetic and alphanumeric character items" refers to the types of operands identified as "Alphabetic data items" and all the types of operands that are grouped under the operand titled "Alphanumeric character items".
  - Literals are listed as a type of operand only in the first column. They do not appear as column headings because literals cannot be used as both operands of a relation condition.

<p>| Table 23. Comparisons involving data items and literals |
|----------------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|</p>
<table>
<thead>
<tr>
<th>Type of data item or literal</th>
<th>Alphanumeric group items</th>
<th>Alphabetic and alphanumeric character items</th>
<th>Zoned decimal items</th>
<th>Native numeric items</th>
<th>Alphanumeric floating-point items</th>
<th>National character items</th>
<th>National decimal items</th>
<th>National floating-point items</th>
<th>DBCS items</th>
<th>UTF-8 character items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alphanumeric group item</td>
<td>Group</td>
<td>Group</td>
<td>Group (Int)</td>
<td>Group</td>
<td>Group</td>
<td>Group (Int)</td>
<td>Group</td>
<td>Group (Int)</td>
<td>Group</td>
<td>UTF-8 character items</td>
</tr>
<tr>
<td>Alphabetic data items</td>
<td>Group</td>
<td>Alph</td>
<td>Alph (Int)</td>
<td>Alph</td>
<td>Nat</td>
<td>Alph (Int)</td>
<td>Nat</td>
<td>Nat</td>
<td>Nat</td>
<td>UTF-8 character items</td>
</tr>
<tr>
<td>Alphanumeric character items:</td>
<td>Group</td>
<td>Alph</td>
<td>Alph (Int)</td>
<td>Alph</td>
<td>Nat</td>
<td>Alph (Int)</td>
<td>Nat</td>
<td>Nat</td>
<td>Nat</td>
<td>UTF-8 character items</td>
</tr>
<tr>
<td>• Data items of category:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Alphanumeric</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Alphanumeric-edited</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Numeric-edited with usage DISPLAY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Alphanumeric functions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alphanumeric literals</td>
<td>Group</td>
<td>Alph</td>
<td>Alph (Int)</td>
<td>Alph</td>
<td>Nat</td>
<td>Alph (Int)</td>
<td>Nat</td>
<td>Nat</td>
<td>Nat</td>
<td>UTF-8 character items</td>
</tr>
<tr>
<td>Numeric literals</td>
<td>Group (Int)</td>
<td>Alph (Int)</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num (Int)</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>UTF-8 character items</td>
</tr>
<tr>
<td>Zoned decimal data items</td>
<td>Group (Int)</td>
<td>Alph (Int)</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num (Int)</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>UTF-8 character items</td>
</tr>
<tr>
<td>Native numeric items:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Binary</td>
<td></td>
<td></td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td></td>
</tr>
<tr>
<td>- Arithmetic expression</td>
<td></td>
<td></td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td></td>
</tr>
<tr>
<td>- Internal decimal</td>
<td></td>
<td></td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td></td>
</tr>
<tr>
<td>- Internal floating point</td>
<td></td>
<td></td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td></td>
</tr>
<tr>
<td>Numeric and integer intrinsic functions</td>
<td></td>
<td></td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td></td>
</tr>
<tr>
<td>Display floating-point items</td>
<td>Group</td>
<td>Alph</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Nat</td>
<td>Num</td>
<td>Num</td>
<td></td>
</tr>
<tr>
<td>Floating-point literals</td>
<td></td>
<td></td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td></td>
</tr>
<tr>
<td>Type of data item or literal</td>
<td>Alphanumeric group items</td>
<td>Alphabetic and alphanumeric character items</td>
<td>Zoned decimal items</td>
<td>Native numeric items</td>
<td>Alphanumeric floating-point items</td>
<td>National character items</td>
<td>National decimal items</td>
<td>National floating-point items</td>
<td>DBCS items</td>
<td>UTF-8 character items</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>--------------------------</td>
<td>---------------------------------------------</td>
<td>---------------------</td>
<td>---------------------</td>
<td>----------------------------------</td>
<td>------------------------</td>
<td>-----------------------</td>
<td>-------------------------</td>
<td>------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>National character items:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Data items of category:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• National</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• National- edited</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Numeric- edited with</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>usage NATIONAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• National intrinsic functions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• National groups (treated as elementary item)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

National literals

National decimal items

National floating-point items

DBCS data items

DBCS literals

UTF-8 character items

<table>
<thead>
<tr>
<th>Figurative constant</th>
<th>Alphanumeric group items</th>
<th>Alphabetic and alphanumeric character items</th>
<th>Zoned decimal items</th>
<th>Native numeric items</th>
<th>Alphanumeric floating-point items</th>
<th>National character items</th>
<th>National decimal items</th>
<th>National floating-point items</th>
<th>DBCS items</th>
<th>UTF-8 data items</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZERO</td>
<td>Group</td>
<td>Alph</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Nat</td>
<td>Num</td>
<td>Num</td>
<td></td>
<td>UTF-8</td>
</tr>
<tr>
<td>SPACE</td>
<td>Group</td>
<td>Alph</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Nat</td>
<td>Nat (Int)</td>
<td>Nat</td>
<td>DBCS</td>
<td>UTF-8</td>
</tr>
<tr>
<td>HIGH-VALUE, LOW-VALUE QUOTE</td>
<td>Group</td>
<td>Alph</td>
<td>Alph (Int)</td>
<td>Alph</td>
<td>Nat</td>
<td>Nat (Int)</td>
<td>Nat</td>
<td>Nat</td>
<td></td>
<td>UTF-8</td>
</tr>
</tbody>
</table>

Symbolic character

ALL alphanumeric literal

ALL national literal

ALL UTF-8 literal

ALL DBCS literal

**Alphanumeric comparisons**

An alphanumeric comparison is a comparison of the single-byte character values of two operands. When one of the operands is neither class alphanumeric nor class alphabetic, that operand is processed as follows:

- A display floating-point data item is treated as though it were a data item of category alphanumeric, rather than as a numeric value.
• A zoned decimal integer operand is treated as though it were moved to a temporary elementary data item of category alphanumeric with a length the same as the total number of digits in the number, according to the rules of the MOVE statement.

When the ZWB compiler option is in effect, the unsigned value of the integer operand is moved to the temporary data item. When the NOZWB compiler option is specified, the signed value is moved to the temporary data item. See ZWB in the Enterprise COBOL Programming Guide for more details about the ZWB (NOZWB) compiler option.

Comparison then proceeds with the temporary data item of category alphanumeric.

Comparison of two alphanumeric operands

Alphanumeric comparisons are made with respect to the collating sequence of the character set in use as follows:

• For the EBCDIC character set, the EBCDIC collating sequence is used.
• For the ASCII character set, the ASCII collating sequence is used. (See Appendix C, “EBCDIC and ASCII collating sequences,” on page 579.)
• When the PROGRAM COLLATING SEQUENCE clause is specified in the object-computer paragraph, the collating sequence used is the one associated in the special-names paragraph with the specified alphabet-name.

The size of each operand is the total number of character positions in that operand; the size affects the result of the comparison. There are two cases to consider:

Operands of equal size

Characters in corresponding positions of the two operands are compared, beginning with the leftmost character and continuing through the rightmost character.

If all pairs of characters through the last pair evaluate as equal, the operands are equal.

If a pair of unequal characters is encountered, the characters are tested to determine their relative positions in the collating sequence. The operand that contains the character higher in the sequence is considered the greater operand.

Operands of unequal size

If the operands are of unequal size, the comparison is made as though the shorter operand were extended to the right with appropriate space characters to make the operands equal in size.

The higher collating value is determined using the hexadecimal value of characters.

Standard comparison

A standard comparison is any comparison that is not based on a locale. The standard comparison method depends on whether the operands to be compared are of equal length or unequal length.

If the operands are of unequal length, the comparison proceeds as though the shorter operand were padded on the right with appropriate space characters to make the operands of equal length. The comparison then proceeds according to the rules for the comparison of operands of equal length.

If the operands are of equal length, the comparison proceeds by comparing corresponding character positions in the two operands, starting from the leftmost position, until either unequal characters are encountered or the rightmost character position is reached, whichever comes first. The operands are determined to be equal if all corresponding characters are equal.

The first-encountered unequal character in the operands is compared to determine the relation of the operands. The operand that contains the character with the higher collating value is the greater operand.

DBCS comparisons

A DBCS comparison is a comparison of two DBCS operands.
The following rules apply to a DBCS comparison:

- If the DBCS operands are not the same length, the comparison is made as though the shorter operand were padded on the right with DBCS spaces to the length of the longer operand.
- The comparison is based on the binary collating sequence of the hexadecimal values of the DBCS characters.

**Dynamic-length elementary item comparisons**

A dynamic-length elementary item comparison is a comparison of two dynamic-length elementary item operands.

When comparing two dynamic-length elementary items in a relational condition, their current lengths are compared followed possibly by a comparison of their contents. The following rules apply:

- Operands of equal length: for dynamic-length elementary items of equal length, their contents are compared in the same manner as operands of equal size for two fixed-length items.
- Operands of unequal length: for dynamic-length elementary items of unequal length, the item with the larger length is considered the greater operand, regardless of either contents of operands.

When comparing a dynamic-length elementary item to a fixed-length item, the dynamic-length elementary item is treated as though it were a fixed-length item whose length is the same as the current length of the dynamic-length elementary item. An example that compares two dynamic-length elementary items:

```
01 dy-1 PIC X DYNAMIC.
01 dy-2 PIC X DYNAMIC.
```

The following PROCEDURE DIVISION statement compares content in terms of the *Alphanumeric comparison* rules. Notice the additional space character at the end of the first literal.

```
MOVE 'abcd ' to dy-1.
MOVE 'abcd' to dy-2.
IF dy-1 IS EQUAL TO dy-2 THEN
   DISPLAY 'IS EQUAL'
ELSE
   DISPLAY 'NOT EQUAL'
END-IF
```

When executed the program above will display 'NOT EQUAL' since the length of `dy-1` is 5, while the length of `dy-2` is 4. The contents of the items are not inspected.

**National comparisons**

A national comparison is a comparison of the national character value of two operands of class national.

When the relation condition specifies an operand that is neither class national nor class UTF-8, that operand is converted to a data item of category national before the comparison. When a UTF-8 item is compared with a national item, the comparison is always done in UTF-8. The following list describes the conversion of operands to category national.

**DBCS**

A DBCS operand is treated as though it were moved to a temporary data item of category national of the same length as the DBCS operand. DBCS characters are converted to the corresponding national characters. The source code page used for the conversion is the one in effect for the CODEPAGE compiler option when the source code was compiled.

**Alphabetic, alphanumeric, alphanumeric-edited, and numeric-edited with usage DISPLAY and UTF-8**

The operand is treated as though it were moved to a temporary data item of category national of the length needed to represent the number of character positions in that operand. Alphanumeric characters are converted to the corresponding national characters. The source code page used for the conversion is the one in effect for the CODEPAGE compiler option when the source code was compiled.
**Numeric integer**
A numeric integer operand is treated as though it were moved to a temporary data item of category alphanumeric of a length the same as the number of digits in the integer. The unsigned value is used. The resulting temporary data item is then converted as an alphanumeric operand.

**External floating-point**
A display floating-point item is treated as though it were a data item of category alphanumeric, rather than as a numeric value, and then converted as an alphanumeric operand.

A national floating-point item is treated as though it were a data item of category national, rather than as a numeric value.

The implicit moves for the conversions are carried out in accordance with the rules of the MOVE statement.

The resulting category national data item is used in the comparison of two national operands.

**Comparison of two national operands**
If the operands are of unequal length, the comparison proceeds as though the shorter operand were padded on the right with the default national space character (NX'0020') to make the operands of equal length. The comparison then proceeds according to the rules for the comparison of operands of equal length.

If the operands are of equal length, the comparison proceeds by comparing corresponding national character positions in the two operands, starting from the leftmost position, until either unequal national characters are encountered or the rightmost national character position is reached, whichever comes first. The operands are determined to be equal if all corresponding national characters are equal.

The first-encountered unequal national character in the operands is compared to determine the relation of the operands. The operand that contains the national character with the higher collating value is the greater operand.

The higher collating value is determined using the hexadecimal value of characters.

The PROGRAM COLLATING SEQUENCE clause has no effect on comparisons of national operands.

**UTF-8 comparisons**
A UTF-8 comparison is a comparison of two operands of class UTF-8. When the relation condition specifies an operand that is not class UTF-8, that operand is converted to a data item of category UTF-8 before the comparison.

**Alphabetic, alphanumeric, alphanumeric-edited, numeric-edited with usage DISPLAY, national, national-edited and numeric-edited with usage NATIONAL**
The operand is treated as though it were moved to a temporary data item of category UTF-8 of the length needed to represent the number of character positions in that operand. Alphanumeric characters are converted to the corresponding UTF-8 characters. The source code page used for the conversion is the one in effect for the CODEPAGE compiler option when the source code was compiled.

The implicit moves for the conversions are carried out in accordance with the rules of the MOVE statement.

The resulting category UTF-8 data item is used in the comparison of two UTF-8 operands.

**Comparison of two UTF-8 operands**
If the operands are of unequal length, the comparison proceeds as though the shorter operand were padded on the right with the default UTF-8 space character (UX'20') to make the operands of equal length. The comparison then proceeds according to the rules for the comparison of operands of equal length.

If the operands are of equal length, the comparison proceeds by comparing corresponding UTF-8 character positions in the two operands, starting from the leftmost position, until either unequal
UTF-8 characters are encountered or the rightmost UTF-8 character position is reached, whichever comes first. The operands are determined to be equal if all corresponding UTF-8 characters are equal.

The first-encountered unequal UTF-8 character in the operands is compared to determine the relation of the operands. The operand that contains the UTF-8 character with the higher collating value is the greater operand.

**Note:** The higher collating value is determined using the hexadecimal value of characters, and the PROGRAM COLLATING SEQUENCE clause has no effect on comparisons of UTF-8 operands.

**Numeric comparisons**
A numeric comparison is a comparison of the algebraic value of two operands of class numeric.

When the algebraic values of numeric operands are compared:

- The length (number of digits) of the operands is not significant.
- The usage of the operands is not significant.
- Unsigned numeric operands are considered positive.
- All zero values compare equal; the presence or absence of a sign does not affect the result.

The behavior of numeric comparisons depends on the settings of the NUMPROC and ZONEDATA compiler options. For details, see [NUMPROC](#) and [ZONEDATA](#) in the Enterprise COBOL Programming Guide.

**Group comparisons**
A group comparison is a comparison of the alphanumeric character values of two group item operands.

For the comparison operation between fixed-length groups, each operand is treated as though it were an elementary data item of category alphanumeric of the same size as the operand, in bytes. The comparison then proceeds as for two elementary operands of category alphanumeric, as described in “Alphanumeric comparisons” on page 253.

Comparing a dynamic-length group with another group is not allowed.

**Usage note:** There is no conversion of data for group comparisons. The comparison operates on bytes of data without regard to data representation. The result of comparing an elementary item or literal operand to an alphanumeric group item is predictable when that operand and the content of the group item have the same data representation.

**Comparison of index-names and index data items**
Comparisons involving index-names, index data items, or both conform to rules.

The rules for comparisons are:

- The comparison of two index-names is actually the comparison of the corresponding occurrence numbers.
- In the comparison of an index-name with a data item (other than an index data item), or in the comparison of an index-name with a literal, the occurrence number that corresponds to the value of the index-name is compared with the data item or literal.
- In the comparison of an index-name with an arithmetic expression, the occurrence number that corresponds to the value of the index-name is compared with the arithmetic expression.

Because an integer function can be used wherever an arithmetic expression can be used, you can compare an index-name to an integer or numeric function.

- In the comparison of an index data item with an index-name or another index data item, the actual values are compared without conversion. Results of any other comparison involving an index data item are undefined.

Valid comparisons for index-names and index data items are shown in the following table.
Table 25. Comparisons for index-names and index data items

<table>
<thead>
<tr>
<th>Operands compared</th>
<th>Index-name</th>
<th>Index data item</th>
<th>Data-name (numeric integer only)</th>
<th>Literal (numeric integer only)</th>
<th>Arithmetic Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index-name</td>
<td>Compare occurrence number</td>
<td>Compare without conversion</td>
<td>Compare occurrence number with content of referenced data item</td>
<td>Compare occurrence number with literal</td>
<td>Compare occurrence number with arithmetic expression</td>
</tr>
<tr>
<td>Index data item</td>
<td>Compare without conversion</td>
<td>Compare without conversion</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
</tbody>
</table>

**Data pointer relation conditions**

Only EQUAL and NOT EQUAL are allowed as relational operators when specifying pointer data items.

Pointer data items are items defined explicitly as USAGE POINTER, or are ADDRESS OF special registers, which are implicitly defined as USAGE POINTER.

The operands are equal if the two addresses used in the comparison would both result in the same storage location.

This relation condition is allowed in IF, PERFORM, EVALUATE, and SEARCH format-1 statements. It is not allowed in SEARCH format-2 (SEARCH ALL) statements because there is no meaningful ordering that can be applied to pointer data items.

**Format 2: data-pointer relation condition**

- **identifier-1, identifier-3**
  Can specify any level item defined in the LINKAGE SECTION, except 66 and 88.

- **identifier-2, identifier-4**
  Must be described as USAGE POINTER.

- **NULL, NULLS**
  Can be used only if the other operand is defined as USAGE POINTER. That is, NULL=NULL is not allowed.

The following table summarizes the permissible comparisons for USAGE POINTER, NULL, and ADDRESS OF.
Table 26. *Permissible comparisons for USAGE POINTER, NULL, and ADDRESS OF*

<table>
<thead>
<tr>
<th>Permissible comparisons</th>
<th>USAGE POINTER second operand</th>
<th>ADDRESS OF second operand</th>
<th>NULL or NULLS second operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>USAGE POINTER first operand</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ADDRESS OF first operand</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>NULL/NULLS first operand</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Yes
Comparison allowed only for EQUAL, NOT EQUAL

No
No comparison allowed

**Procedure-pointer and function-pointer relation conditions**

Only EQUAL and NOT EQUAL are allowed as relational operators when specifying procedure-pointer or function-pointer data items in a relation condition.

Procedure-pointer data items are defined explicitly as USAGE PROCEDURE-POINTER. Function-pointer data items are defined explicitly as USAGE FUNCTION-POINTER.

The operands are equal if the two addresses used in the comparison would both result in the same storage location.

This relation condition is allowed in IF, PERFORM, EVALUATE, and SEARCH format-1 statements. It is not allowed in SEARCH format-2 (SEARCH ALL) statements, because there is no meaningful ordering that can be applied to procedure-pointer data items.

**Format 3: procedure-pointer and function-pointer relation condition**

```
identifier-1 IS NOT EQUAL TO identifier-2
```

*identifier-1, identifier-2*

Must be described as USAGE PROCEDURE-POINTER or USAGE FUNCTION-POINTER. *identifier-1* and *identifier-2* need not be described the same.

**NULL, NULLS**

Can be used only if the other operand is defined as USAGE FUNCTION-POINTER or USAGE PROCEDURE-POINTER. That is, NULL=NULL is not allowed.
Object-reference relation conditions
A data item of usage OBJECT REFERENCE can be compared for equality or inequality with another data item of usage OBJECT REFERENCE or with NULL, NULLS, or SELF.

Format 4: object-reference relation condition

A comparison with SELF is allowed only in a method.
Two object-references compare equal only if the data items identify the same object.

Sign condition
The sign condition determines whether the algebraic value of a numeric operand is greater than, less than, or equal to zero.

Format: sign condition

operand-1

Must be defined as a numeric identifier, or as an arithmetic expression that contains at least one reference to a variable. operand-1 can be defined as a floating-point identifier.

The operand is:
• POSITIVE if its value is greater than zero
• NEGATIVE if its value is less than zero
• ZERO if its value is equal to zero

An unsigned operand is either POSITIVE or ZERO.

NOT
One algebraic test is executed for the truth value of the sign condition. For example, NOT ZERO is regarded as true when the operand tested is positive or negative in value.

The results of the sign condition test depend on the setting of the NUMPROC compiler option. For details, see NUMPROC in the Enterprise COBOL Programming Guide.
Switch-status condition

The switch-status condition determines the on or off status of a UPSI switch.

**Format**

```
condition-name
```

(condition-name)

Must be defined in the special-names paragraph as associated with the on or off value of an UPSI switch. (See “SPECIAL-NAMES paragraph” on page 108.)

The switch-status condition tests the value associated with condition-name. (The value is considered to be alphanumeric.) The result of the test is true if the UPSI switch is set to the value (0 or 1) corresponding to condition-name.

Complex conditions

A complex condition is formed by combining simple conditions, combined conditions, or complex conditions with logical operators, or negating those conditions with logical negation.

Each logical operator must be preceded and followed by a space. The following table shows the logical operators and their meanings.

<table>
<thead>
<tr>
<th>Logical operator</th>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>Logical conjunction</td>
<td>The truth value is true when both conditions are true.</td>
</tr>
<tr>
<td>OR</td>
<td>Logical inclusive OR</td>
<td>The truth value is true when either or both conditions are true.</td>
</tr>
<tr>
<td>NOT</td>
<td>Logical negation</td>
<td>Reversal of truth value (the truth value is true if the condition is false).</td>
</tr>
</tbody>
</table>

Unless modified by parentheses, the following list is the order of precedence (from highest to lowest):

1. Arithmetic operations
2. Simple conditions
3. NOT
4. AND
5. OR

The truth value of a complex condition (whether parenthesized or not) is the truth value that results from the interaction of all the stated logical operators on either of the following options:

- The individual truth values of simple conditions
- The intermediate truth values of conditions logically combined or logically negated

A complex condition can be either of the following options:

- A negated simple condition
- A combined condition (which can be negated)
Negated simple conditions

A simple condition is negated through the use of the logical operator NOT.

**Format**

```
NOT condition-1
```

The negated simple condition gives the opposite truth value of the simple condition. That is, if the truth value of the simple condition is true, then the truth value of that same negated simple condition is false, and vice versa.

Placing a negated simple condition within parentheses does not change its truth value. That is, the following two statements are equivalent:

```
NOT A IS EQUAL TO B.
NOT (A IS EQUAL TO B).
```

Combined conditions

Two or more conditions can be logically connected to form a combined condition.

**Format**

```
condition-1 AND condition-2
```

The condition to be combined can be any of the following ones:

- A simple-condition
- A negated simple-condition
- A combined condition
- A negated combined condition (that is, the NOT logical operator followed by a combined condition enclosed in parentheses)
- A combination of the preceding conditions that is specified according to the rules in the following table

<table>
<thead>
<tr>
<th>Combined condition element</th>
<th>Leftmost</th>
<th>When not leftmost, can be immediately preceded by:</th>
<th>Rightmost</th>
<th>When not rightmost, can be immediately followed by:</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple-condition</td>
<td>Yes</td>
<td>OR NOT AND (</td>
<td>Yes</td>
<td>OR AND )</td>
</tr>
<tr>
<td>OR AND</td>
<td>No</td>
<td>simple-condition</td>
<td>No</td>
<td>simple-condition NOT (</td>
</tr>
</tbody>
</table>

Table 28. Combined conditions—permissible element sequences
Table 28. Combined conditions—permissible element sequences (continued)

<table>
<thead>
<tr>
<th>Combined condition element</th>
<th>Leftmost</th>
<th>When not leftmost, can be immediately preceded by:</th>
<th>Rightmost</th>
<th>When not rightmost, can be immediately followed by:</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>Yes</td>
<td>OR AND</td>
<td>No</td>
<td>simple-condition (</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>Yes</td>
<td>OR NOT AND</td>
<td>No</td>
<td>simple-condition NOT (</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>No</td>
<td>simple-condition</td>
<td>Yes</td>
<td>OR AND</td>
</tr>
</tbody>
</table>

Parentheses are never needed when either ANDs or ORs (but not both) are used exclusively in one combined condition. However, parentheses might be needed to modify the implicit precedence rules to maintain the correct logical relation of operators and operands.

There must be a one-to-one correspondence between left and right parentheses, with each left parenthesis to the left of its corresponding right parenthesis.

The following table illustrates the relationships between logical operators and conditions C1 and C2.

Table 29. Logical operators and evaluation results of combined conditions

<table>
<thead>
<tr>
<th>Value for C1</th>
<th>Value for C2</th>
<th>C1 AND C2</th>
<th>C1 OR C2</th>
<th>NOT (C1 AND C2)</th>
<th>NOT C1 AND C2</th>
<th>NOT (C1 OR C2)</th>
<th>NOT C1 OR C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

Order of evaluation of conditions

Parentheses, both explicit and implicit, define the level of inclusiveness within a complex condition. Two or more conditions connected by only the logical operators AND or OR at the same level of inclusiveness establish a hierarchical level within a complex condition. Therefore an entire complex condition is a nested structure of hierarchical levels, with the entire complex condition being the most inclusive hierarchical level.

Within this context, the evaluation of the conditions within an entire complex condition begins at the left of the condition. The constituent connected conditions within a hierarchical level are evaluated in order from left to right, and evaluation of that hierarchical level terminates as soon as a truth value for it is determined, regardless of whether all the constituent connected conditions within that hierarchical level have been evaluated.
Values are established for arithmetic expressions and functions if and when the conditions that contain them are evaluated. Similarly, negated conditions are evaluated if and when it is necessary to evaluate the complex condition that they represent. For example:

```
NOT A IS GREATER THAN B OR A + B IS EQUAL TO C AND D IS POSITIVE
```

is evaluated as if parenthesized as follows:

```
(NOT (A IS GREATER THAN B)) OR
(((A + B) IS EQUAL TO C) AND (D IS POSITIVE))
```

**Order of evaluation:**

1. (NOT (A IS GREATER THAN B)) is evaluated, giving some intermediate truth value, \( t_1 \). If \( t_1 \) is true, the combined condition is true, and no further evaluation takes place. If \( t_1 \) is false, evaluation continues as follows.
2. (A + B) is evaluated, giving some intermediate result, \( x \).
3. (x IS EQUAL TO C) is evaluated, giving some intermediate truth value, \( t_2 \). If \( t_2 \) is false, the combined condition is false, and no further evaluation takes place. If \( t_2 \) is true, the evaluation continues as follows.
4. (D IS POSITIVE) is evaluated, giving some intermediate truth value, \( t_3 \). If \( t_3 \) is false, the combined condition is false. If \( t_3 \) is true, the combined condition is true.

**Abbreviated combined relation conditions**

When relation-conditions are written consecutively, any relation-condition after the first can be abbreviated by omission of the subject, or by omission of the subject and relational operator.

<table>
<thead>
<tr>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>relation-condition</td>
</tr>
</tbody>
</table>

In any consecutive sequence of relation-conditions, both forms of abbreviation can be specified. The abbreviated condition is evaluated as if:

1. The last stated subject is the missing subject.
2. The last stated relational operator is the missing relational operator.

The resulting combined condition must comply with the rules for element sequences in combined conditions, as shown in “Combined conditions” on page 262.

If NOT is immediately followed by GREATER THAN, >, LESS THAN, <, EQUAL TO, or =, then the NOT participates as part of the relational operator. NOT in any other position is considered a logical operator (and thus results in a negated relation condition).

**Using parentheses**

You can use parentheses in combined relation conditions to specify an intended order of evaluation. Using parentheses can also help improve the readability of conditional expressions.

The following rules govern the use of parentheses in abbreviated combined relation conditions:
1. Parentheses can be used to change the order of evaluation of the logical operators AND and OR.
2. The word NOT participates as part of the relational operator when it is immediately followed by GREATER THAN, >, LESS THAN, <, EQUAL TO, or =.
3. NOT in any other position is considered a logical operator and thus results in a negated relation condition. If you use NOT as a logical operator, only the relation condition immediately following the NOT is negated; the negation is not propagated through the abbreviated combined relation condition along with the subject and relational operator.
4. The logical NOT operator can appear within a parenthetical expression that immediately follows a relational operator.
5. When a left parenthesis appears immediately after the relational operator, the relational operator is distributed to all objects enclosed in the parentheses. In the case of a "distributed" relational operator, the subject and relational operator remain current after the right parenthesis which ends the distribution. The following three restrictions apply to cases where the relational operator is distributed throughout the expression:
   a. A simple condition cannot appear within the scope of the distribution.
   b. Another relational operator cannot appear within the scope of the distribution.
   c. The logical operator NOT cannot appear immediately after the left parenthesis, which defines the scope of the distribution.
6. Evaluation proceeds from the least to the most inclusive condition.
7. There must be a one-to-one correspondence between left and right parentheses, with each left parenthesis to the left of its corresponding right parenthesis. If the parentheses are unbalanced, the compiler inserts a parenthesis and issues an E-level message. However, if the compiler-inserted parenthesis results in the truncation of the expression, you will receive an S-level diagnostic message.
8. The last stated subject is inserted in place of the missing subject.
9. The last stated relational operator is inserted in place of the missing relational operator.
10. Insertion of the omitted subject or relational operator ends when:
    a. Another simple condition is encountered.
    b. A condition-name is encountered.
    c. A right parenthesis is encountered that matches a left parenthesis that appears to the left of the subject.
11. In any consecutive sequence of relation conditions, you can use both abbreviated relation conditions that contain parentheses and those that do not.
12. Consecutive logical NOT operators cancel each other and result in an S-level message. Note, however, that an abbreviated combined relation condition can contain two consecutive NOT operators when the second NOT is part of a relational operator. For example, you can abbreviate the first condition as the second condition listed below.

\[
A = B \text{ and not } A \text{ not } = C \\
A = B \text{ and not not } = C
\]

The following table summarizes the rules for forming an abbreviated combined relation condition.

<table>
<thead>
<tr>
<th>Combined condition element</th>
<th>Left- most</th>
<th>When not leftmost, can be immediately preceded by:</th>
<th>Right- most</th>
<th>When not rightmost, can be immediately followed by:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject</td>
<td>Yes</td>
<td>NOT</td>
<td>No</td>
<td>Relational operator</td>
</tr>
</tbody>
</table>
### Table 30. Abbreviated combined conditions: permissible element sequences (continued)

<table>
<thead>
<tr>
<th>Combined condition element</th>
<th>Left-most</th>
<th>When not leftmost, can be immediately preceded by:</th>
<th>Right-most</th>
<th>When not rightmost, can be immediately followed by:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>No</td>
<td>Relational operator AND OR NOT (</td>
<td>Yes</td>
<td>AND OR</td>
</tr>
<tr>
<td>Relational operator</td>
<td>No</td>
<td>Subject AND OR NOT (</td>
<td>No</td>
<td>Object</td>
</tr>
<tr>
<td>AND OR</td>
<td>No</td>
<td>Object</td>
<td>No</td>
<td>Object Relational operator NOT (</td>
</tr>
<tr>
<td>NOT</td>
<td>Yes</td>
<td>AND OR (</td>
<td>No</td>
<td>Subject Object Relational operator NOT (</td>
</tr>
<tr>
<td>(</td>
<td>Yes</td>
<td>Relational operator AND OR NOT (</td>
<td>No</td>
<td>Subject Object NOT (</td>
</tr>
<tr>
<td>)</td>
<td>No</td>
<td>Object</td>
<td>Yes</td>
<td>AND OR</td>
</tr>
</tbody>
</table>

The following table shows examples of abbreviated combined relation conditions, with and without parentheses, and their unabbreviated equivalents.

### Table 31. Abbreviated combined conditions: unabbreviated equivalents

<table>
<thead>
<tr>
<th>Abbreviated combined relation condition</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = B AND NOT &lt; C OR D</td>
<td>((A = B) AND (A NOT &lt; C)) OR (A NOT &lt; D)</td>
</tr>
<tr>
<td>A NOT &gt; B OR C</td>
<td>(A NOT &gt; B) OR (A NOT &gt; C)</td>
</tr>
<tr>
<td>NOT A = B OR C</td>
<td>(NOT (A = B)) OR (A = C)</td>
</tr>
<tr>
<td>NOT (A = B OR &lt; C)</td>
<td>NOT ((A = B) OR (A &lt; C))</td>
</tr>
<tr>
<td>NOT (A NOT = B AND C AND NOT D)</td>
<td>NOT (((A NOT = B) AND (A NOT = C)) AND (NOT (A NOT = D))))</td>
</tr>
</tbody>
</table>

266  Enterprise COBOL for z/OS: Enterprise COBOL for z/OS, V6.3 Language Reference
Statement categories

There are four categories of COBOL statements: imperative statements, conditional statements, delimited scope statements, and compiler-directing statements. See the following topics for more details.

Imperative statements

An imperative statement either specifies an unconditional action to be taken by the program, or is a conditional statement terminated by its explicit scope terminator.

A series of imperative statements can be specified wherever an imperative statement is allowed. A conditional statement that is terminated by its explicit scope terminator is also classified as an imperative statement.

For more information about explicit scope terminator, see “Delimited scope statements” on page 270.

The following lists contain the COBOL imperative statements.

Arithmetic

- ADD
- COMPUTE
- DIVIDE
- MULTIPLY
- SUBTRACT

1. Without the ON SIZE ERROR or the NOT ON SIZE ERROR phrase.

Data movement

- ACCEPT (DATE, DAY, DAY-OF-WEEK, TIME)
- INITIALIZE
- INSPECT
- JSON GENERATE
- JSON PARSE
- MOVE
- SET
- STRING
- UNSTRING
- XML GENERATE
- XML PARSE

2. Without the ON OVERFLOW or the NOT ON OVERFLOW phrase.

3. Without the ON EXCEPTION or NOT ON EXCEPTION phrase.

Ending

- STOP RUN
- EXIT PROGRAM
- EXIT METHOD
- GOBACK
**Input-output**
- ACCEPT identifier
- CLOSE
- DELETE
- DISPLAY
- OPEN
- READ
- REWRITE
- START
- STOP literal
- WRITE

4. Without the INVALID KEY or the NOT INVALID KEY phrase.
5. Without the AT END or NOT AT END, and INVALID KEY or NOT INVALID KEY phrases.
6. Without the INVALID KEY or NOT INVALID KEY, and END-OF-PAGE or NOT END-OF-PAGE phrases.

**Ordering**
- ALLOCATE
- Format 1 SORT
- FREE
- MERGE
- RELEASE
- RETURN

7. Without the AT END or NOT AT END phrase.

**Procedure-branching**
- ALTER
- CONTINUE
- Format 1 EXIT
- GO TO
- PERFORM

**Program or method linkage**
- CALL
- CANCEL
- INVOKE

8. Without the ON OVERFLOW phrase, and without the ON EXCEPTION or NOT ON EXCEPTION phrase.

**Table-handling**
- Format 2 SORT (table SORT)
- SET
Conditional statements

A conditional statement specifies that the truth value of a condition is to be determined and that the subsequent action of the object program is dependent on this truth value.

For more information about conditional expressions, see “Conditional expressions” on page 246.

The following lists contain COBOL statements that become conditional when a condition (for example, ON SIZE ERROR or ON OVERFLOW) is included and when the statement is not terminated by its explicit scope terminator.

Arithmetic

• ADD ... ON SIZE ERROR
• ADD ... NOT ON SIZE ERROR
• COMPUTE ... ON SIZE ERROR
• COMPUTE ... NOT ON SIZE ERROR
• DIVIDE ... ON SIZE ERROR
• DIVIDE ... NOT ON SIZE ERROR
• MULTIPLY ... ON SIZE ERROR
• MULTIPLY ... NOT ON SIZE ERROR
• SUBTRACT ... ON SIZE ERROR
• SUBTRACT ... NOT ON SIZE ERROR

Data movement

• JSON GENERATE ... ON EXCEPTION
• JSON GENERATE ... NOT ON EXCEPTION
• JSON PARSE ... ON EXCEPTION
• JSON PARSE ... NOT ON EXCEPTION
• STRING ... ON OVERFLOW
• STRING ... NOT ON OVERFLOW
• UNSTRING ... ON OVERFLOW
• UNSTRING ... NOT ON OVERFLOW
• XML GENERATE ... ON EXCEPTION
• XML GENERATE ... NOT ON EXCEPTION
• XML PARSE ... ON EXCEPTION
• XML PARSE ... NOT ON EXCEPTION

Decision

• IF
• EVALUATE

Input-output

• DELETE ... INVALID KEY
• DELETE ... NOT INVALID KEY
• READ ... AT END
• READ ... NOT AT END
• READ ... INVALID KEY
Delimited scope statements

In general, a DELIMITED SCOPE statement uses an explicit scope terminator to turn a conditional statement into an imperative statement.

The resulting imperative statement can then be nested. Explicit scope terminators can also be used to terminate the scope of an imperative statement. Explicit scope terminators are provided for all COBOL statements that can have conditional phrases.

Unless explicitly specified otherwise, a delimited scope statement can be specified wherever an imperative statement is allowed by the rules of the language.

Explicit scope terminators

An explicit scope terminator marks the end of certain PROCEDURE DIVISION statements.

A conditional statement that is delimited by its explicit scope terminator is considered an imperative statement and must follow the rules for imperative statements.

These are the explicit scope terminators:

- END-ADD
- END-CALL
- END-COMPUTE
- END-DELETE
- END-DIVIDE
- END-EVALUATE
- END-IF
Implicit scope terminators

At the end of any sentence, an implicit scope terminator is a separator period that terminates the scope of all previous statements not yet terminated.

An unterminated conditional statement cannot be contained by another statement.

Except for nesting conditional statements within IF statements, nested statements must be imperative statements and must follow the rules for imperative statements. You should not nest conditional statements.

Related references

RULES (Enterprise COBOL Programming Guide)

Compiler-directing statements

A compiler-directing statement is a statement that causes the compiler to take a specific action during compilation.

For more information about statements that direct the compiler to take a specified action, see Chapter 22, “Compiler-directing statements,” on page 521.

Statement operations

The topic shows types of operations performed by COBOL statements.

COBOL statements perform the following types of operations:

• Arithmetic
• Data manipulation
• Input/output
• Procedure branching

There are several phrases common to arithmetic and data manipulation statements, such as:

• CORRESPONDING phrase
• GIVING phrase
• ROUNDED phrase
• SIZE ERROR phrases
**CORRESPONDING phrase**

The CORRESPONDING (CORR) phrase causes ADD, SUBTRACT, and MOVE operations to be performed on elementary data items of the same name if the alphanumeric group item or national group item to which they belong is specified.

A national group is processed as a group item when the CORRESPONDING phrase is used.

Both identifiers that follow the keyword CORRESPONDING must name group items. In this discussion, these identifiers are referred to as identifier-1 and identifier-2. identifier-1 references the sending group item. identifier-2 references the receiving group item.

Two subordinate data items, one from identifier-1 and one from identifier-2, correspond if the following conditions are true:

- In an ADD or SUBTRACT statement, both of the data items are elementary numeric data items. Other data items are ignored.
- In a MOVE statement, at least one of the data items is an elementary item, and the move is permitted by the move rules.
- The two subordinate items have the same name and the same qualifiers up to but not including identifier-1 and identifier-2.
- The subordinate items are not identified by the keyword FILLER.
- Neither identifier-1 nor identifier-2 is described as a level 66, 77, or 88 item, and neither is described as an index data item. Neither identifier-1 nor identifier-2 can be reference-modified.
- Neither identifier-1 nor identifier-2 is described with USAGE POINTER, USAGE FUNCTION-POINTER, USAGE PROCEDURE-POINTER, or USAGE OBJECT REFERENCE.
- The subordinate items do not include a REDEFINES, RENAMES, OCCURS, USAGE INDEX, USAGE POINTER, USAGE PROCEDURE-POINTER, USAGE FUNCTION-POINTER, or USAGE OBJECT REFERENCE clause in their descriptions. However, identifier-1 and identifier-2 themselves can contain or be subordinate to items that contain a REDEFINES or OCCURS clause in their descriptions.
- The name of each subordinate data item that satisfies these conditions is unique after application of implicit qualifiers.

identifier-1, identifier-2, or both can be subordinate to a FILLER item.

For example, consider two data hierarchies defined as follows:

```cobol
05 ITEM-1 OCCURS 6.
   10 ITEM-A PIC S9(3).
   10 ITEM-B PIC +99.9.
   10 ITEM-C PIC X(4).
   10 ITEM-D REDEFINES ITEM-C PIC 9(4).
   10 ITEM-E USAGE COMP-1.
   10 ITEM-F USAGE INDEX.
05 ITEM-2.
   10 ITEM-A PIC 99.
   10 ITEM-B PIC +9V9.
   10 ITEM-C PIC A(4).
   10 ITEM-D PIC 9(4).
   10 ITEM-E PIC 9(9) USAGE COMP.
   10 ITEM-F USAGE INDEX.
```

If ADD CORR ITEM-2 TO ITEM-1(x) is specified, ITEM-A and ITEM-A(x), ITEM-B and ITEM-B(x), and ITEM-E and ITEM-E(x) are considered to be corresponding and are added together. ITEM-C and ITEM-C(x) are not included because they are not numeric. ITEM-D and ITEM-D(x) are not included because ITEM-D(x) includes a REDEFINES clause in its data description. ITEM-F and ITEM-F(x) are not included because they are index data items. Note that ITEM-1 is valid as either identifier-1 or identifier-2.
If any of the individual operations in the ADD CORRESPONDING statement produces a size error condition, *imperative-statement-1* in the ON SIZE ERROR phrase is not executed until all of the individual additions are completed.

**GIVING phrase**

For arithmetic statements, the value of the identifier that follows the word GIVING is set equal to the calculated result of the arithmetic operation. Because this identifier is not involved in the computation, it can be a numeric-edited item.

**ROUNDED phrase**

After decimal point alignment, the number of places in the fraction of the result of an arithmetic operation is compared with the number of places provided for the fraction of the resultant identifier.

When the size of the fractional result exceeds the number of places provided for its storage, truncation occurs unless ROUNDED is specified. When ROUNDED is specified, the least significant digit of the resultant identifier is increased by 1 whenever the most significant digit of the excess is greater than or equal to 5.

When the resultant identifier is described by a PICTURE clause that contains rightmost Ps and when the number of places in the calculated result exceeds the number of integer positions specified, rounding or truncation occurs relative to the rightmost integer position for which storage is allocated.

In a floating-point arithmetic operation, the ROUNDED phrase has no effect; the result of a floating-point operation is always rounded. For more information on floating-point arithmetic expressions, see *Fixed-point contrasted with floating-point arithmetic* in the Enterprise COBOL Programming Guide.

When the ARITH(EXTEND) compiler option is in effect, the ROUNDED phrase is not supported for arithmetic receivers with 31 digit positions to the right of the decimal point. For example, neither X nor Y below is valid as a receiver with the ROUNDED phrase:

```cobol
01  X PIC V31.
01  Y PIC P(30)9(1).

   COMPUTE X ROUNDED = A + B
   COMPUTE Y ROUNDED = A - B
```

Otherwise, the ROUNDED phrase is fully supported for extended-precision arithmetic statements.

**SIZE ERROR phrases**

A size error condition can occur in different ways.

- When the absolute value of the result of an arithmetic evaluation, after decimal point alignment, exceeds the largest value that can be contained in the result field.
- When division by zero occurs.
- In an exponential expression, as indicated in the following table:

<table>
<thead>
<tr>
<th>Size error</th>
<th>Action taken when a SIZE ERROR clause is present</th>
<th>Action taken when a SIZE ERROR clause is not present</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero raised to zero power</td>
<td>The SIZE ERROR imperative is executed.</td>
<td>The value returned is 1, and a message is issued.</td>
</tr>
<tr>
<td>Zero raised to a negative number</td>
<td>The SIZE ERROR imperative is executed.</td>
<td>The program is terminated abnormally.</td>
</tr>
<tr>
<td>A negative number raised to a fractional power</td>
<td>The SIZE ERROR imperative is executed.</td>
<td>The absolute value of the base is used, and a message is issued.</td>
</tr>
</tbody>
</table>
The size error condition applies only to final results, not to any intermediate results.

If the resultant identifier is defined with usage BINARY, COMPUTATIONAL, COMPUTATIONAL-4, or COMPUTATIONAL-5, the largest value that the resultant data item can contain is the value implied by the item's decimal PICTURE character-string, regardless of the TRUNC compiler option in effect.

If the ROUNDED phrase is specified, rounding takes place before size error checking.

When a size error occurs, the subsequent action of the program depends on whether the ON SIZE ERROR phrase is specified.

If the ON SIZE ERROR phrase is not specified and a size error condition occurs, truncation rules apply and the value of the affected resultant identifier is computed.

If the ON SIZE ERROR phrase is specified and a size error condition occurs, the value of the resultant identifier affected by the size error is not altered; that is, the error results are not placed in the receiving identifier. After completion of the execution of the arithmetic operation, the imperative statement in the ON SIZE ERROR phrase is executed, control is transferred to the end of the arithmetic statement, and the NOT ON SIZE ERROR phrase, if specified, is ignored.

For ADD CORRESPONDING and SUBTRACT CORRESPONDING statements, if an individual arithmetic operation causes a size error condition, the ON SIZE ERROR imperative statement is not executed until all the individual additions or subtractions have been completed.

If the NOT ON SIZE ERROR phrase has been specified and, after execution of an arithmetic operation, a size error condition does not exist, the NOT ON SIZE ERROR phrase is executed.

When both the ON SIZE ERROR and NOT ON SIZE ERROR phrases are specified, and the statement in the phrase that is executed does not contain any explicit transfer of control, then if necessary an implicit transfer of control is made after execution of the phrase to the end of the arithmetic statement.

## Arithmetic statements

The arithmetic statements are used for computations. Individual operations are specified by the ADD, SUBTRACT, MULTIPLY, and DIVIDE statements. These individual operations can be combined symbolically in a formula that uses the COMPUTE statement.

### Arithmetic statement operands

The data descriptions of operands in an arithmetic statement need not be the same. Throughout the calculation, the compiler performs any necessary data conversion and decimal point alignment.

#### Size of operands

If the ARITH(COMPAT) compiler option is in effect, the maximum size of each operand is 18 decimal digits. If the ARITH(EXTEND) compiler option is in effect, the maximum size of each operand is 31 decimal digits.

The composite of operands is a hypothetical data item resulting from aligning the operands at the decimal point and then superimposing them on one another.

If the ARITH(COMPAT) compiler option is in effect, the composite of operands can be a maximum of 30 digits. If the ARITH(EXTEND) compiler option is in effect, the composite of operands can be a maximum of 31 digits.

The following table shows how the composite of operands is determined for arithmetic statements:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Determination of the composite of operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Superimposing all operands in a given statement except those following the word GIVING</td>
</tr>
</tbody>
</table>
Table 33. How the composite of operands is determined (continued)

<table>
<thead>
<tr>
<th>Statement</th>
<th>Determination of the composite of operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULTIPLY</td>
<td>Superimposing all receiving data items</td>
</tr>
<tr>
<td>DIVIDE</td>
<td>Superimposing all receiving data items except the REMAINDER data item</td>
</tr>
<tr>
<td>COMPUTE</td>
<td>Restriction does not apply</td>
</tr>
</tbody>
</table>

For example, assume that each item is defined as follows in the DATA DIVISION:

```plaintext
A  PICTURE 9(7)V9(5).
B  PICTURE 9(11)V99.
C  PICTURE 9(12)V9(3).
```

If the following statement is executed, the composite of operands consists of 17 decimal digits:

```plaintext
ADD A B TO C
```

It has the following implicit description:

```plaintext
COMPOSITE-OF-OPERANDS PICTURE 9(12)V9(5).
```

In the ADD and SUBTRACT statements, if the composite of operands is 30 digits or less with the ARITH(COMPAT) compiler option, or 31 digits or less with the ARITH(EXTEND) compiler option, the compiler ensures that enough places are carried so that no significant digits are lost during execution.

In all arithmetic statements, it is important to define data with enough digits and decimal places to ensure the required accuracy in the final result. For more information, see Appendix A. Intermediate results and arithmetic precision in the Enterprise COBOL Programming Guide.

**Overlapping operands**

When operands in an arithmetic statement share part of their storage (that is, when the operands overlap), the result of the execution of such a statement is unpredictable.

**Multiple results**

When an arithmetic statement has multiple results, execution conceptually proceeds as follows:

1. The statement performs all arithmetic operations to find the result to be placed in the receiving items, and stores that result in a temporary location.

2. A sequence of statements transfers or combines the value of this temporary result with each single receiving field. The statements are considered to be written in the same left-to-right order in which the multiple results are listed.

For example, executing the following statement:

```plaintext
ADD A, B, C, TO C, D(C), E.
```

is equivalent to executing the following series of statements:

```plaintext
ADD A, B, C GIVING TEMP.
ADD TEMP TO C.
ADD TEMP TO D(C).
ADD TEMP TO E.
```

In the above example, TEMP is a compiler-supplied temporary result field. When the addition operation for D(C) is performed, the subscript C contains the new value of C.
Data manipulation statements
The following COBOL statements move and inspect data: ACCEPT, INITIALIZE, INSPECT, JSON GENERATE, JSON PARSE, MOVE, READ, RELEASE, RETURN, REWRITE, SET, STRING, UNSTRING, WRITE, XML PARSE, and XML GENERATE.

Overlapping operands
When the sending and receiving fields of a data manipulation statement share a part of their storage (that is, when the operands overlap), the result of the execution of such a statement is unpredictable.

Input-output statements
COBOL input-output statements transfer data to and from files stored on external media, and also control low-volume data that is obtained from or sent to an input/output device.

In COBOL, the unit of file data made available to the program is a record. You need only be concerned with such records. Provision is automatically made for such operations as the movement of data into buffers, internal storage, validity checking, error correction (where feasible), blocking and deblocking, and volume-switching procedures.

The description of the file in the ENVIRONMENT DIVISION and the DATA DIVISION governs which input-output statements are allowed in the PROCEDURE DIVISION. Permissible statements for sequential files are shown in Table 47 on page 374, and permissible statements for indexed files and relative files are shown in Table 48 on page 375. Permissible statements for line sequential files are shown in Table 49 on page 376.

Common processing facilities
Several common processing facilities apply to more than one input-output statement.

The common processing facilities provided are:

• “File status key” on page 276
• “Invalid key condition” on page 280
• “INTO and FROM phrases” on page 281
• “File position indicator” on page 282

Discussions in the following sections use the terms volume and reel. The term volume refers to all non-unit-record input-output devices. The term reel applies only to tape devices. Treatment of direct-access devices in the sequential access mode is logically equivalent to the treatment of tape devices.

File status key
If the FILE STATUS clause is specified in the file-control entry, a value is placed in the specified file status key (the two-character data item named in the FILE STATUS clause) during execution of any request on that file; the value indicates the status of that request.

The value is placed in the file status key before execution of any EXCEPTION/ERROR declarative, INVALID KEY phrase, or AT END phrase associated with the request.

There are two file status key data-names. One is described by data-name-1 in the FILE STATUS clause of the file-control entry. This is a two-character data item with the first character known as file status key 1 and the second character known as file status key 2. The combinations of possible values and their meanings are shown in Table 34 on page 277.

The other file status key is described by data-name-8 in the FILE STATUS clause of the file-control entry. data-name-8 does not apply to QSAM files. For more information about data-name-8, see “FILE STATUS clause” on page 137.
### Table 34. *File status key values and meanings*

<table>
<thead>
<tr>
<th>High-order digit</th>
<th>Meaning</th>
<th>Low-order digit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful completion</td>
<td>0</td>
<td>No further information</td>
</tr>
<tr>
<td>2</td>
<td>This file status value applies only to indexed files with alternate keys that allow duplicates. The input-output statement was successfully executed, but a duplicate key was detected. For a READ statement, the key value for the current key of reference was equal to the value of the same key in the next record within the current key of reference. For a REWRITE or WRITE statement, the record just written created a duplicate key value for at least one alternate record key for which duplicates are allowed.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>A READ statement was successfully executed, but the number of character positions that were read was less than the minimum size or was greater than the maximum size specified by the record description entries associated with the FD for the file. <strong>Note:</strong> If the VLR(COMPAT) option is in effect, you will get the status value of 00 when READ statements encounter a record length conflict. For details about the VLR option, see VLR in the Enterprise COBOL Programming Guide.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>An OPEN statement was successfully executed, but the referenced optional file was unavailable at the time the OPEN statement was executed. The file had been created if the open mode was I-O or EXTEND. This does not apply to VSAM sequential files.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>For a CLOSE statement with the NO REWIND, REEL/UNIT, or FOR REMOVAL phrase or for an OPEN statement with the NO REWIND phrase, the referenced file was on a non-reel/unit medium.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>At-end condition</td>
<td>0</td>
<td>A sequential READ statement was attempted and no next logical record existed in the file because the end of the file had been reached. Or the first READ was attempted on an optional input file that was unavailable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>A sequential READ statement was attempted for a relative file, and the number of significant digits in the relative record number was larger than the size of the relative key data item described for the file.</td>
</tr>
<tr>
<td>High-order digit</td>
<td>Meaning</td>
<td>Low-order digit</td>
<td>Meaning</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------</td>
<td>-----------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>2</td>
<td>Invalid key condition</td>
<td>1</td>
<td>A sequence error exists for a sequentially accessed indexed file. The prime record key value was changed by the program between the successful execution of a READ statement and the execution of the next REWRITE statement for that file. Or the ascending requirements for successive record key values were violated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>An attempt was made to write a record that would create a duplicate key in a relative file. Or an attempt was made to write or rewrite a record that would create a duplicate prime record key or a duplicate alternate record key without the DUPLICATES phrase in an indexed file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>An attempt was made to randomly access a record that does not exist in the file. Or a START or random READ statement was attempted on an optional input file that was unavailable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>An attempt was made to write beyond the externally defined boundaries of a relative or indexed file. Or a sequential WRITE statement was attempted for a relative file and the number of significant digits in the relative record number was larger than the size of the relative key data item described for the file.</td>
</tr>
<tr>
<td>3</td>
<td>Permanent error condition</td>
<td>0</td>
<td>No further information</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>A permanent error exists because of a boundary violation; an attempt was made to write beyond the externally defined boundaries of a sequential file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>An OPEN statement with the INPUT, I-O, or EXTEND phrase was attempted on a nonoptional file that was unavailable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>An OPEN statement was attempted on a file that would not support the open mode specified in the OPEN statement. Possible violations are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• The EXTEND or OUTPUT phrase was specified but the file would not support write operations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• The I-O phrase was specified but the file would not support the input and output operations permitted.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• The INPUT phrase was specified but the file would not support read operations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>An OPEN statement was attempted on a file previously closed with lock.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>The OPEN statement was unsuccessful because a conflict was detected between the fixed file attributes and the attributes specified for that file in the program. These attributes include the organization of the file (sequential, relative, or indexed), the prime record key, the alternate record keys, the code set, the maximum record size, the record type (fixed or variable), and the blocking factor.</td>
</tr>
<tr>
<td>High-order digit</td>
<td>Meaning</td>
<td>Low-order digit</td>
<td>Meaning</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------</td>
<td>-----------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>4</td>
<td>Logic error condition</td>
<td>1</td>
<td>An OPEN statement was attempted for a file in the open mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>A CLOSE statement was attempted for a file not in the open mode.</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1</td>
<td>For a mass storage file in the sequential access mode, the last input-output statement executed for the associated file prior to the execution of a REWRITE statement was not a successfully executed READ statement.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>For relative and indexed files in the sequential access mode, the last input-output statement executed for the file prior to the execution of a DELETE or REWRITE statement was not a successfully executed READ statement.</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>1</td>
<td>A boundary violation exists because an attempt was made to rewrite a record to a file and the record was not the same size as the record being replaced. Or an attempt was made to write or rewrite a record that was larger than the largest or smaller than the smallest record allowed by the RECORD IS VARYING clause of the associated file-name.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>A sequential READ statement was attempted on a file open in the input or I-O mode and no valid next record had been established because:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• The preceding READ statement was unsuccessful but did not cause an at-end condition.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• The preceding READ statement caused an at-end condition.</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>1</td>
<td>The execution of a READ statement was attempted on a file not open in the input or I-O mode.</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>1</td>
<td>The execution of a WRITE statement was attempted on a file not open in the I-O, output, or extend mode.</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>1</td>
<td>The execution of a DELETE or REWRITE statement was attempted on a file not open in the I-O mode.</td>
</tr>
</tbody>
</table>
Table 34. File status key values and meanings (continued)

<table>
<thead>
<tr>
<th>High-order digit</th>
<th>Meaning</th>
<th>Low-order digit</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| 9                | Implementor-defined condition | 0 | • For multithreading only: A CLOSE of a VSAM or QSAM file was attempted on a thread that did not open the file.  
• Without multithreading: For VSAM only: See the information about VSAM return codes in Using VSAM status codes (VSAM files only) in the Enterprise COBOL Programming Guide.  
• QSAM files: No further information available. See the DFSMS error message for more information. |
| 1                | For VSAM only: Password failure | 2 | Logic error |
| 3                | For all files, except QSAM: Resource unavailable | 5 | For all files except QSAM: Invalid or incomplete file information |
| 6                | For VSAM file: An OPEN statement with the OUTPUT phrase was attempted, or an OPEN statement with the I-O or EXTEND phrase was attempted for an optional file but no DD statement was specified for the file.  
For QSAM file: An OPEN statement with the OUTPUT phrase was attempted, or an OPEN statement with the I-O or EXTEND phrase was attempted for an optional file but no DD statement was specified for the file and the CBLQDA(OFF) runtime option was specified. | 7 | For VSAM only: OPEN statement execution successful: File integrity verified  
**Note:** If the VSAMOPENFS(SUCC) option is in effect, you will get the status value of 00 when a VSAM OPEN statement is successfully verified. For details about the VSAMOPENFS option, see VSAMOPENFS in the Enterprise COBOL Programming Guide. |
| 8                | Open failed due to the invalid contents of an environment variable specified in a SELECT ... ASSIGN clause or due to dynamic allocation failure. For more information about the contents of environment variables, see “ASSIGN clause” on page 126. |

Invalid key condition

The invalid key condition can occur during execution of a START, READ, WRITE, REWRITE, or DELETE statement. When an invalid key condition occurs, the input-output statement that caused the condition is unsuccessful.

When the invalid key condition is recognized, actions are taken in the following order:

1. If the FILE STATUS clause is specified in the file-control entry, a value is placed into the file status key to indicate an invalid key condition, as shown in Table 34 on page 277.

2. If the INVALID KEY phrase is specified in the statement that caused the condition, control is transferred to the INVALID KEY imperative statement. Any EXCEPTION/ERROR declarative procedure specified for this file is not executed. Execution then continues according to the rules for each statement specified in the imperative statement.
3. If the INVALID KEY phrase is not specified in the input-output statement for a file and an applicable EXCEPTION/ERROR procedure exists, that procedure is executed. The NOT INVALID KEY phrase, if specified, is ignored.

Both the INVALID KEY phrase and the EXCEPTION/ERROR procedure can be omitted.

If the invalid key condition does not exist after execution of the input-output operation, the INVALID KEY phrase is ignored, if specified, and the following actions are taken:

- If an exception condition that is not an invalid key condition exists, control is transferred according to the rules of the USE statement following the execution of any USE AFTER EXCEPTION procedure.
- If no exception condition exists, control is transferred to the end of the input-output statement or the imperative statement specified in the NOT INVALID KEY phrase, if it is specified.

**INTO and FROM phrases**

The INTO and FROM phrases are valid for READ, RETURN, RELEASE, REWRITE, and WRITE statements. You must specify an identifier that is the name of an entry in the WORKING-STORAGE SECTION or the LINKAGE SECTION, or of a record description for another previously opened file.

### Format: INTO and FROM phrases of input-output statements

- **READ**
  - file-name-1
- **RETURN**
- **RELEASE**
  - record-name-1
- **REWRITE**
- **WRITE**
  - INTO — identifier-1
  - FROM — identifier-1

- record-name-1 and identifier-1 must not refer to the same storage area.
- If record-name-1 or identifier-1 refers to a national group item, the item is processed as an elementary data item of category national.
- The INTO phrase can be specified in a READ or RETURN statement.

The result of the execution of a READ or RETURN statement with the INTO phrase is equivalent to the application of the following rules in the order specified:

- The execution of the same READ or RETURN statement without the INTO phrase.
- The current record is moved from the record area to the area specified by identifier-1 according to the rules for the MOVE statement without the CORRESPONDING phrase. The size of the current record is determined by rules specified in the RECORD clause. If the file description entry contains a RECORD IS VARYING clause, the implied move is a group move. The implied MOVE statement does not occur if the execution of the READ or RETURN statement was unsuccessful. Any subscripting or reference-modification associated with identifier-1 is evaluated after the record has been read or returned and immediately before it is moved to the data item. The record is available in both the record area and the data item referenced by identifier-1.
- The FROM phrase can be specified in a RELEASE, REWRITE, or WRITE statement.

The result of the execution of a RELEASE, REWRITE, or WRITE statement with the FROM phrase is equivalent to the execution of the following statements in the order specified:

1. MOVE identifier-1 TO record-name-1
2. The same RELEASE, REWRITE, or WRITE statement without the FROM phrase

After the execution of the RELEASE, REWRITE or WRITE statement is complete, the information in the area referenced by identifier-1 is available even though the information in the area referenced by record-name-1 is unavailable, except as specified by the SAME RECORD AREA clause.
**File position indicator**

The file position indicator is a conceptual entity used in this document to facilitate exact specification of the next record to be accessed within a given file during certain sequences of input-output operations. The setting of the file position indicator is affected only by the OPEN, CLOSE, READ and START statements. The concept of a file position indicator has no meaning for a file opened in the output or extend mode.
Chapter 20. PROCEDURE DIVISION statements

Statements, sentences, and paragraphs in the PROCEDURE DIVISION are executed sequentially except when a procedure branching statement such as EXIT, GO TO, PERFORM, GOBACK, or STOP is used.

ACCEPT statement

The ACCEPT statement transfers data or system date-related information into the data area referenced by the specified identifier. There is no editing or error checking of the incoming data.

Data transfer

Format 1 transfers data from an input source into the data item referenced by identifier-1 (the receiving area). When the FROM phrase is omitted, the system input device is assumed.

Format 1: data transfer

```
ACCEPT    identifier-1
         FROM mnemonic-name-1
         environment-name
```

Format 1 is useful for exceptional situations in a program when operator intervention (to supply a given message, code, or exception indicator) is required. The operator must of course be supplied with the appropriate messages with which to reply.

identifier-1

The receiving area. Can be:
- An alphanumeric group item
- A national group item
- An elementary data item of usage DISPLAY, DISPLAY-1, or NATIONAL

A national group item is processed as an elementary data item of category national.

identifier-1 must not be a dynamic-length group item or a dynamic-length elementary item.

mnemonic-name-1

Specifies the input device. mnemonic-name-1 must be associated in the SPECIAL-NAMES paragraph with an environment-name. See “SPECIAL-NAMES paragraph” on page 108.

- System input device

The length of a data transfer is the same as the length of the record on the input device, with a maximum of 32,760 bytes.

The system input device is read until the receiving area is filled or EOF is encountered. If the length of the receiving area is not an even multiple of the system input device record length, the final record will be truncated as required. If EOF is encountered after data has been moved and before the receiving area has been filled, the receiving area is padded with spaces of the appropriate representation for the receiving area. If EOF is encountered before any data has been moved to the
receiving area, padding will not take place and the contents of the receiving area are unchanged. Each input record is concatenated with the previous input record.

If the input record is of a fixed-length format, the entire input record is used. No editing is performed to remove trailing or leading blanks.

If the input record is of the variable-length format, the actual record length is used to determine the amount of data received. With variable-format records, the Record Definition Word (RDW) is removed from the beginning of the input record. Only the actual input data is transferred to identifier-1.

If the data item referenced by identifier-1 is of usage national, data is transferred without conversion and without checking for validity. The input data is assumed to be in UTF-16 format.

- **Console**

  1. A system-generated message code is automatically displayed, followed by the literal AWAITING REPLY.
     
     The maximum length of an input message is 114 characters.
  2. Execution is suspended.
  3. After the message code (the same code as in item 1) is entered from the console and recognized by the system, ACCEPT statement execution is resumed. The message is moved to the receiving area and left-justified regardless of its PICTURE clause.

     If identifier-1 references a data item of usage NATIONAL, the message is converted from the native code page representation to national character representation. The native code page is the one that was specified by the CODEPAGE compiler option when the source code was compiled.

     The ACCEPT statement is terminated if any of the following conditions occurs:

     - No data is received from the console; for example, if the operator hits the Enter key.
     - The receiving data item is filled with data.
     - Fewer than 114 characters of data are entered.

     If 114 bytes of data are entered and the receiving area is still not filled with data, more requests for data are issued to the console.

     If more than 114 characters of data are entered, only the first 114 characters will be recognized by the system.

     If the receiving area is longer than the incoming message, the rightmost characters are padded with spaces of the appropriate representation for the receiving area.

     If the incoming message is longer than the receiving area, the character positions beyond the length of the receiving area are truncated.

     For information about obtaining ACCEPT input from a z/OS UNIX file or stdin, see Assigning input from a screen or file (ACCEPT) in the Enterprise COBOL Programming Guide.

**environment-name**

Identifies the source of input data. An environment-name from the names given in Table 5 on page 110 can be specified.

If the device is the same as that used for READ statements for a LINE SEQUENTIAL file, results are unpredictable.

**System date-related information transfer**

System information contained in the specified conceptual data items DATE, DATE YYYYMMDD, DAY, DAY YYYYDDD, DAY-OF-WEEK, or TIME, can be transferred into the data item referenced by identifier-2. The transfer must follow the rules for the MOVE statement without the CORRESPONDING phrase.

For more information, see “MOVE statement” on page 363.
**Format 2: system information transfer**

![Diagram]

*identifier-2*

The receiving area. Can be:

- An alphanumeric group item
- A national group item
- An elementary data item of one of the following categories:
  - alphanumeric
  - alphanumeric-edited
  - numeric-edited (with usage DISPLAY or NATIONAL)
  - national
  - national-edited
  - numeric
  - internal floating-point
  - external floating-point (with usage DISPLAY or NATIONAL)

A national group item is processed as an elementary data item of category national.

*identifier-2* might not be a dynamic-length group item, but can be a dynamic-length elementary item.

Format 2 accesses the current date in two formats: the day of the week or the time of day as carried by the system (which can be useful in identifying when a particular run of an object program was executed). You can also use format 2 to supply the date in headings and footings.

The current date and time can also be accessed with the intrinsic function CURRENT-DATE, which also supports four-digit year values and provides additional information (see “CURRENT-DATE” on page 476).

**DATE, DATE YYYYMMDD, DAY, DAY YYYYDDD, DAY-OF-WEEK, and TIME**

The conceptual data items DATE, DATE YYYYMMDD, DAY, DAY YYYYDDD, DAY-OF-WEEK, and TIME implicitly have USAGE DISPLAY. Because these are conceptual data items, they cannot be described in the COBOL program.

The content of the conceptual data items is moved to the receiving area using the rules of the MOVE statement. If the receiving area is of usage NATIONAL, the data is converted to national character representation.

**DATE**

Has the implicit PICTURE 9(6).

The sequence of data elements (from left to right) is:

- Two digits for the year
- Two digits for the month
- Two digits for the day

Thus 27 April 2003 is expressed as 030427.
DATE YYYYMMDD
Has the implicit PICTURE 9(8).

The sequence of data elements (from left to right) is:

- Four digits for the year
- Two digits for the month
- Two digits for the day

Thus 27 April 2003 is expressed as 20030427.

DAY
Has the implicit PICTURE 9(5).

The sequence of data elements (from left to right) is:

- Two digits for the year
- Three digits for the day

Thus 27 April 2003 is expressed as 03117.

DAY YYYYDDD
Has the implicit PICTURE 9(7).

The sequence of data elements (from left to right) is:

- Four digits for the year
- Three digits for the day

Thus 27 April 2003 is expressed as 2003117.

DAY-OF-WEEK
Has the implicit PICTURE 9(1).

The single data element represents the day of the week according to the following values:

- 1 represents Monday
- 2 represents Tuesday
- 3 represents Wednesday
- 4 represents Thursday
- 5 represents Friday
- 6 represents Saturday
- 7 represents Sunday

Thus Wednesday is expressed as 3.

TIME
Has the implicit PICTURE 9(8).

The sequence of data elements (from left to right) is:

- Two digits for hour of day
- Two digits for minute of hour
- Two digits for second of minute
- Two digits for hundredths of second

Thus 2:41 PM is expressed as 14410000.
ADD statement

The ADD statement sums two or more numeric operands and stores the result.

**Format 1: ADD statement**

```
ADD identifier-1 TO identifier-2 ROUNDED
```

- `ADD` - The keyword ADD.
- `identifier-1` - An identifier or literal that precedes the keyword TO.
- `literal-1` - A literal that precedes the keyword TO.
- `TO` - The keyword TO.
- `identifier-2` - An identifier that follows the keyword TO.
- `ROUNDED` - The keyword ROUNDED.
- `imperative-statement-1` - An imperative statement that may be specified.
- `SIZE ERROR ON` - Error handling.
- `NOT ON` - Error handling.
- `END-ADD` - The keyword END-ADD.

All identifiers or literals that precede the keyword TO are added together, and this sum is added to and stored in `identifier-2`. This process is repeated for each successive occurrence of `identifier-2` in the left-to-right order in which `identifier-2` is specified.

**Format 2: ADD statement with GIVING phrase**

```
ADD identifier-1 TO identifier-2 literal-2 GIVING identifier-3 ROUNDED
```

- `ADD` - The keyword ADD.
- `identifier-1` - An identifier or literal that precedes the word GIVING.
- `literal-1` - A literal that precedes the word GIVING.
- `TO` - The keyword TO.
- `identifier-2` - An identifier that follows the keyword TO.
- `literal-2` - A literal that follows the keyword TO.
- `GIVING` - The keyword GIVING.
- `identifier-3` - An identifier that follows the word GIVING.
- `ROUNDED` - The keyword ROUNDED.
- `imperative-statement-1` - An imperative statement that may be specified.
- `SIZE ERROR ON` - Error handling.
- `NOT ON` - Error handling.
- `END-ADD` - The keyword END-ADD.

The values of the operands that precede the word GIVING are added together, and the sum is stored as the new value of each data item referenced by `identifier-3`. 
Elementary data items within \textit{identifier-1} are added to and stored in the corresponding elementary items within \textit{identifier-2}.

For all formats:

\textbf{\textit{identifier-1, identifier-2}}

In format 1, must name an elementary numeric item.

In format 2, must name an elementary numeric item except when following the word \textit{GIVING}. Each identifier that follows the word \textit{GIVING} must name an elementary numeric or numeric-edited item.

In format 3, must name an alphanumeric group item or national group item.

\textbf{\textit{literal}}

Must be a numeric literal.

Floating-point data items and literals can be used anywhere that a numeric data item or literal can be specified.

When the \textit{ARITH(COMPAT)} compiler option is in effect, the composite of operands can contain a maximum of 30 digits. When the \textit{ARITH(EXTEND)} compiler option is in effect, the composite of operands can contain a maximum of 31 digits. For more information, see “Arithmetic statement operands” on page 274 and the details on arithmetic intermediate results in \textsl{Appendix A. Intermediate results and arithmetic precision} in the \textsl{Enterprise COBOL Programming Guide}.

\textbf{ROUNDED phrase}

For formats 1, 2, and 3, see “ROUNDED phrase” on page 273.

\textbf{SIZE ERROR phrases}

For formats 1, 2, and 3, see “SIZE ERROR phrases” on page 273.

\textbf{CORRESPONDING phrase (format 3)}

See “CORRESPONDING phrase” on page 272.

\textbf{END-ADD phrase}

This explicit scope terminator serves to delimit the scope of the ADD statement. END-ADD permits a conditional ADD statement to be nested in another conditional statement. END-ADD can also be used with an imperative ADD statement.

For more information, see “Delimited scope statements” on page 270.
ALLOCATE statement

The ALLOCATE statement obtains dynamic storage.

Format

\[
\text{ALLOCATE} \quad \text{arithmetic-expression-1} \quad \text{CHARACTERS} \quad \text{data-name-1} \quad \text{INITIALIZED} \\
\text{LOC} \quad \text{integer-1} \quad \text{RETURNING} \quad \text{data-name-2}
\]

If \text{data-name-1} is specified, the address of the data item is set to the address of the obtained storage, as if the "SET ADDRESS OF \text{data-name-1} TO address" statement was used. If a RETURNING data item is also specified, the pointer data item will contain that address.

If \text{arithmetic-expression-1} CHARACTERS is specified, the RETURNING \text{data-item-2} will be set to the address of the obtained storage.

\text{data-name-1}

- Must be a level-01 or level-77 item defined in the LINKAGE SECTION.
- If \text{data-name-1} is specified, the RETURNING phrase can be omitted. Otherwise, the RETURNING phrase must be specified.
- Cannot be reference modified.
- Cannot be a group item that contains an unbounded table.

\text{integer-1}

- Must be an unsigned integer with value of 24, 31 or 64. The default for LP(64) is 64, and for LP(32) is 31. The value 64 is not supported in LP(32).

\text{data-name-2}

- Must be defined as USAGE POINTER or USAGE POINTER-32.
- Can be qualified or subscripted.

\text{arithmetic-expression-1}

- Specifies a number of bytes of storage to be allocated:
  - If \text{arithmetic-expression-1} does not evaluate to an integer, the result is rounded up to the next whole number.
  - If \text{arithmetic-expression-1} evaluates to 0 or a negative value, the data item referenced by \text{data-name-2} is set to the predefined address NULL.

\text{INITIALIZED phrase}

The \text{INITIALIZED phrase} initializes the allocated storage:

- If the \text{INITIALIZED phrase} is not specified, the content of the allocated storage is undefined.
- If both \text{arithmetic-expression-1} and the \text{INITIALIZED phrase} are specified, all bytes of the allocated storage are initialized to binary zeros.
- If both \text{data-name-1} and the \text{INITIALIZED phrase} are specified, the allocated storage is initialized as if an INITIALIZE \text{data-name-1} WITH FILLER ALL TO VALUE THEN TO DEFAULT statement were executed.
**LOC phrase**

The LOC phrase controls how ALLOCATE acquires storage:

- **LOC 24** causes ALLOCATE to acquire storage from below the 16 MB line, regardless of the setting of the DATA compiler option.
- **LOC 31** causes ALLOCATE to attempt to acquire storage from above the 16 MB line, regardless of the setting of the DATA compiler option.

**Note:** It is still possible that storage is acquired below the 16 MB line with LOC 31 if storage above the 16 MB line is exhausted.

- **LOC 64** causes ALLOCATE to attempt to acquire storage from above the 2 GB bar, regardless of the setting of the DATA compiler option.

When the LOC phrase is not specified:

- **LOC 31** is assumed to be specified whenever the DATA(31) compiler option is in effect.
- **LOC 24** is assumed to be specified whenever the DATA(24) compiler option is in effect.
- **LOC 64** is assumed to be specified whenever the LP(64) compiler option is in effect.

**Note:** It is recommended to use the default value of LOC unless dynamic storage below the 16 MB line is required when the DATA(31) option is in effect, or dynamic storage above the 16 MB line is desired when the DATA(24) option is in effect.

If LOC 64 is specified and `data-name-2` references a USAGE POINTER-32 data item, a diagnostic message will be issued, because the size of the data item is too small for the 64-bit address.

If LOC 64 is specified and LP(32) is in effect, a diagnostic message will be issued.

If `data-name-1` is specified, the amount of storage to be allocated is the number of bytes required to hold an item as described by `data-name-1`. If a data description entry that is subordinate to `data-name-1` contains an OCCURS DEPENDING ON clause, the maximum length of the record is allocated.

If the LOC phrase is omitted and `data-name-2` references a USAGE POINTER data item:

- If LP(64) is in effect, the allocated storage will be above the 2 GB bar.
- If LP(32) is in effect, the allocated storage will be below the 2 GB bar.

If the LOC phrase is omitted and `data-name-2` references a USAGE POINTER-32 data item, the allocated storage will always be below the 2 GB bar regardless of the setting of the LP compiler option.

If the specified amount of storage is available for allocation:

- If the RETURNING phrase is specified, the data item referenced by `data-name-2` is set to the address of that storage.
- If `data-name-1` is specified, the address of the 01 or 77 LINKAGE SECTION data item referenced by `data-name-1` is set to the address of that storage, as if the “SET ADDRESS OF `data-name-1` TO address-of-obtained-storage” statement was used.

If the specified amount of storage is not available for allocation:

- If the RETURNING phrase is specified, the data item referenced by `data-name-2` is set to the predefined address NULL.
- If `data-name-1` is specified, the address of the 01 or 77 LINKAGE SECTION data item referenced by `data-name-1` is set to the predefined address NULL.

The allocated storage persists until explicitly released with a FREE statement or the run unit is terminated, whichever occurs first.

You can use ALLOCATE to dynamically increase the size of an UNBOUNDED table, see “Example: ALLOCATE and FREE storage for UNBOUNDED tables” on page 291.

**Related references**

- “FREE statement” on page 319
- “INITIALIZE statement” on page 324
Example: ALLOCATE and FREE storage for UNBOUNDED tables

This example illustrates one way to manage an UNBOUNDED table that needs to be dynamically increased in size by using the ALLOCATE and FREE statements.

```
ID DIVISION.
PROGRAM-ID. ALLOC.
ENVIRONMENT DIVISION.
DATA DIVISION.
  WORKING-STORAGE SECTION.
    77 X PIC 9(2) PACKED-DECIMAL.
    77 NUM-ELEMENTS PIC 9(4) BINARY.
    77 SIZE-NEEDED PIC 9(4) BINARY.
    77 VPTR POINTER.

LINKAGE SECTION.
  01 VARGRP.
    02 OBJ PIC 9(4) COMP.
    02 TABGRP.
      03 VARTAB OCCURS 1 TO UNBOUNDED DEPENDING ON OBJ.
        04 T1 PIC 9(4).
        04 T2 PIC X(8).
        04 T3 PIC 9(4) COMP.
    01 BUFFER PIC X(1000).

PROCEDURE DIVISION.
  DISPLAY 'Starting testcase ALLOC'
  SET VPTR To NULL

*****************************************************************************
* Allocate a table with 20 elements
*****************************************************************************
  COMPUTE NUM-ELEMENTS = 20
  PERFORM ALLOC-VARGRP

*****************************************************************************
* Set some ‘test’ values to validate re-allocated table
*****************************************************************************
  COMPUTE T1(12) = 9999
  MOVE ‘HI MOM’ TO T2 (17)
  DISPLAY ‘ ’
  DISPLAY ‘VARTAB(12) = ’ VARTAB(12)
  DISPLAY ‘VARTAB(17) = ’ VARTAB(17)
  DISPLAY ‘ ’

*****************************************************************************
* Need a bigger table! Allocate a larger one and copy data
*****************************************************************************
  COMPUTE NUM-ELEMENTS = 30
  PERFORM ALLOC-VARGRP

*****************************************************************************
* Ensure that new table has correct data from original
*****************************************************************************
  DISPLAY ‘VARTAB(12) = ’ VARTAB(12)
  DISPLAY ‘VARTAB(17) = ’ VARTAB(17)

GOBACK.

*****************************************************************************
* The first time allocate the original table. If the table
* has already been allocated, assume that we are allocating
* a larger one and want to copy the data over to it
*****************************************************************************
  ALLOC-VARGRP.
    IF VPTR = NULL Then ※ If first time, allocate the table
      COMPUTE SIZE-NEEDED = LENGTH OF OBJ +
                        LENGTH OF VARTAB * NUM-ELEMENTS
      ALLOCATE SIZE-NEEDED CHARACTERS INITIALIZED RETURNING VPTR
      SET ADDRESS OF VARGRP TO VPTR
```
**ALTER statement**

The ALTER statement changes the transfer point specified in a GO TO statement. The ALTER statement encourages the use of unstructured programming practices; the EVALUATE statement provides the same function as the ALTER statement but helps to ensure that a program is well-structured.

**Note:** When the LP(64) compiler option is in effect, a compiler diagnostic message will be issued if the ALTER statement is specified. The EVALUATE statement should be used instead.

```
ALTER procedure-name-1 TO PROCEED TO procedure-name-2
```

The ALTER statement modifies the GO TO statement in the paragraph named by `procedure-name-1`. Subsequent executions of the modified GO TO statement transfer control to `procedure-name-2`.

**procedure-name-1**

Must name a PROCEDURE DIVISION paragraph that contains only one sentence: a GO TO statement without the DEPENDING ON phrase.

Related references

- “ALLOCATE statement” on page 289
- “FREE statement” on page 319
- “MOVE statement” on page 363
- Working with unbounded tables and groups (Enterprise COBOL Programming Guide)
**procedure-name-2**

Must name a PROCEDURE DIVISION section or paragraph.

Before the ALTER statement is executed, when control reaches the paragraph specified in procedure-name-1, the GO TO statement transfers control to the paragraph specified in the GO TO statement. After execution of the ALTER statement however, the next time control reaches the paragraph specified in procedure-name-1, the GO TO statement transfers control to the paragraph specified in procedure-name-2.

The ALTER statement acts as a program switch, allowing, for example, one sequence of execution during initialization and another sequence during the bulk of file processing.

Altered GO TO statements in programs with the INITIAL attribute are returned to their initial states each time the program is entered.

Do not use the ALTER statement in programs that have the RECURSIVE attribute, in methods, or in programs compiled with the THREAD option.

**Segmentation considerations**

A GO TO statement that is coded in an independent segment must not be referenced by an ALTER statement in a segment with a different priority-number. All other uses of the ALTER statement are valid and are performed even if the GO TO referenced by the ALTER statement is in a fixed segment.

Altered GO TO statements in independent segments are returned to their initial state when control is transferred to the independent segment that contains the ALTERED GO TO from another independent segment with a different priority-number.

This transfer of control can take place because of:

- The effect of previous statements
- An explicit transfer of control with a PERFORM or GO TO statement
- A sort or merge statement with the INPUT or OUTPUT phrase specified

**CALL statement**

The CALL statement transfers control from one object program to another within the run unit.

The program containing the CALL statement is the calling program; the program identified in the CALL statement is the called subprogram. Called programs can contain CALL statements; however, only programs defined with the RECURSIVE clause can execute a CALL statement that directly or indirectly calls itself.
**Format**

```
CALL identifier-1 literal-1 procedure-pointer-1 function-pointer-1

USING

identifier-2
REFERENCE
BY ADDRESS OF file-name-1
OMITTED

identifier-2
CONTENT
BY ADDRESS OF LENGTH OF identifier-3
fixed-phrase

identifier-3
VALUE
BY ADDRESS OF LENGTH OF literal-2
OMITTED

identifier-4
fixed-phrase

RETURNING identifier-5
exception-phrases
END-CALL
```

**fixed-phrases**

```
AS FIXED LENGTH integer-4
```

**exception-phrases**

```
EXCEPTION imperative-statement-1
not-exception-phrase

OVERFLOW imperative-statement-3
```

**not-exception-phrase**

```
NOT
EXCEPTION imperative-statement-2
```

**identifier-1, literal-1**

`literal-1` must be an alphanumeric literal. `identifier-1` must be an alphanumeric, alphabetic, or numeric data item described with USAGE DISPLAY such that its value can be a program-name.

The rules of formation for program-names are dependent on the PGMNAME compiler option. For details, see the discussion of program-names in “PROGRAM-ID paragraph” on page 96 and also the description of PGMNAME in the Enterprise COBOL Programming Guide.

**Usage note:** Do not specify the name of a class or method in the CALL statement.
**procedure-pointer-1**
Must be defined with USAGE IS PROCEDURE-POINTER and must be set to a valid program entry point; otherwise, the results of the CALL statement are undefined.

After a program has been canceled by COBOL, released by PL/I or C, or deleted by assembler, any procedure-pointers that had been set to that program's entry point are no longer valid.

**function-pointer-1**
Must be defined with USAGE IS FUNCTION-POINTER and must be set to a valid function or program entry point; otherwise, the results of the CALL statement are undefined.

After a program has been canceled by COBOL, released by PL/I or C, or deleted by the assembler, any function-pointers that had been set to that function or program's entry point are no longer valid.

When the called subprogram is to be entered at the beginning of the PROCEDURE DIVISION, literal-1 or the contents of identifier-1 must specify the program-name of the called subprogram.

When the called subprogram is entered through an ENTRY statement, literal-1 or the contents of identifier-1 must be the same as the name specified in the called subprogram's ENTRY statement.

**AMODE 64 considerations**
Static and dynamic call supports calling other AMODE 64 Language Environment conforming programs.

AMODE 64 COBOL programs cannot be called by non-Language Environment conforming programs. CALL using file-name is not supported with AMODE 64. An assembler program using LOAD and then branch to an entry point of an LP(64) COBOL subprogram will not work. Instead, use the LE macro CEEFETCH to fetch and call 64-bit COBOL programs.

Parameter passing convention is XPLINK.

The AS FIXED LENGTH phrase is not currently supported for programs compiled with LP(64).

**USING phrase**
The USING phrase specifies arguments that are passed to the target program.

Include the USING phrase in the CALL statement only if there is a USING phrase in the PROCEDURE DIVISION header or the ENTRY statement through which the called program is run. The number of operands in each USING phrase must be identical.

For more information about the USING phrase, see “The PROCEDURE DIVISION header” on page 238.

The sequence of the operands in the USING phrase of the CALL statement and in the corresponding USING phrase in the called subprogram's PROCEDURE DIVISION header or ENTRY statement determines the correspondence between the operands used by the calling and called programs. This correspondence is positional.

The values of the parameters referenced in the USING phrase of the CALL statement are made available to the called subprogram at the time the CALL statement is executed. The description of the data items in the called program must describe the same number of character positions as the description of the corresponding data items in the calling program.

The BY CONTENT, BY REFERENCE, and BY VALUE phrases apply to parameters that follow them until another BY CONTENT, BY REFERENCE, or BY VALUE phrase is encountered. BY REFERENCE is assumed if you do not specify a BY CONTENT, BY REFERENCE, or BY VALUE phrase prior to the first parameter.

**BY REFERENCE phrase**
If the BY REFERENCE phrase is either specified or implied for a parameter, the corresponding data item in the calling program occupies the same storage area as the data item in the called program.

**identifier-2**
Can be any data item of any level in the DATA DIVISION. identifier-2 cannot be a function-identifier.
**Note:** identifier-2 can only be a dynamic length elementary item if the AS FIXED LENGTH phrase is specified.

If it is defined in the LINKAGE SECTION or FILE SECTION, you must have already provided addressability for identifier-2 prior to invocation of the CALL statement. You can do this by coding either one of the following: SET ADDRESS OF identifier-2 TO pointer or PROCEDURE/ENTRY USING.

**file-name-1**

A file-name for a QSAM file. See Passing data in the Enterprise COBOL Programming Guide for details on using file-name with the CALL statement. This phrase is not supported with AMODE 64.

**ADDRESS OF identifier-2**

identifier-2 must be a level-01 or level-77 item defined in the LINKAGE SECTION.

**OMITTED**

Indicates that no argument is passed.

### AS FIXED LENGTH phrase

If the AS FIXED LENGTH phrase is specified, then the corresponding data item must be a dynamic-length elementary item.

**integer-4**

The value of integer-4 represents the number of characters for items of class alphanumeric or national, or bytes for items of class UTF-8. integer-4 must be an integer greater than zero, and less than the value implied or specified on the LIMIT phrase of the dynamic-length elementary item.

If the item is of class alphanumeric, national, or UTF-8, then the padding character is EBCDIC blanks, UTF-16 blanks, or UTF-8 blanks respectively.

When the AS FIXED LENGTH phrase is specified, the address of the dynamic-length elementary item is passed to the callee as if the equivalent fixed-length alphanumeric, national, or UTF-8 class data item would be whose length is integer-4. If integer-4 is larger than the current length of the dynamic-length elementary item, then the item is extended and padded on the right with blanks to either integer-4 characters. After execution of the CALL statement the current length of the dynamic-length elementary item remains as it was before the execution of the CALL statement.

When the BY CONTENT phrase is specified or implied for a dynamic-length elementary item parameter, then the AS FIXED LENGTH phrase must be specified.

### BY CONTENT phrase

If the BY CONTENT phrase is specified or implied for a parameter, the called program cannot change the value of this parameter as referenced in the CALL statement's USING phrase, though the called program can change the value of the data item referenced by the corresponding data-name in the called program's PROCEDURE DIVISION header. Changes to the parameter in the called program do not affect the corresponding argument in the calling program.

**identifier-3**

Can be any data item of any level in the DATA DIVISION. identifier-3 cannot be a function identifier or an unbounded group.

If defined in the LINKAGE SECTION or FILE SECTION, you must have already provided addressability for identifier-3 prior to invocation of the CALL statement. You can do this by coding one of the following phrases:

- SET ADDRESS OF identifier-3 TO pointer
- PROCEDURE DIVISION USING
- ENTRY USING
**literal-2**

Can be:

- An alphanumeric literal
- A figurative constant (except ALL literal or NULL/NULLS)
- A DBCS literal
- A national literal

**LENGTH OF special register**

*For information about the LENGTH OF special register, see “LENGTH OF” on page 19.*

**ADDRESS OF identifier-3**

*identifier-3 must be a data item of any level except 66 or 88 defined in the LINKAGE SECTION, the WORKING-STORAGE SECTION, or the LOCAL-STORAGE SECTION.*

**OMITTED**

Indicates that no argument is passed.

For alphanumeric literals, the called subprogram should describe the parameter as PIC X(n) USAGE DISPLAY, where n is the number of characters in the literal.

For DBCS literals, the called subprogram should describe the parameter as PIC G(n) USAGE DISPLAY-1, or PIC N(n) with implicit or explicit USAGE DISPLAY-1, where n is the length of the literal.

For national literals, the called subprogram should describe the parameter as PIC N(n) with implicit or explicit USAGE NATIONAL, where n is the length of the literal.

**BY VALUE phrase**

The BY VALUE phrase applies to all arguments that follow until overridden by another BY REFERENCE or BY CONTENT phrase.

If the BY VALUE phrase is specified or implied for an argument, the value of the argument is passed, not a reference to the sending data item. The called program can modify the formal parameter that corresponds to the BY VALUE argument, but any such changes do not affect the argument because the called program has access to a temporary copy of the sending data item.

Although BY VALUE arguments are primarily intended for communication with non-COBOL programs (such as C), they can also be used for COBOL-to-COBOL invocations. In this case, BY VALUE must be specified or implied for both the argument in the CALL USING phrase and the corresponding formal parameter in the PROCEDURE DIVISION USING phrase.

**identifier-4**

Must be an elementary data item in the DATA DIVISION. It must be one of the following items:

- Binary (USAGE BINARY, COMP, COMP-4, or COMP-5)
- Floating point (USAGE COMP-1 or COMP-2)
- Function-pointer (USAGE FUNCTION-POINTER)
- Pointer (USAGE POINTER)
- Procedure-pointer (USAGE PROCEDURE-POINTER)
- Object reference (USAGE OBJECTREFERENCE)
- One single-byte alphanumeric character (such as PIC X or PIC A)
- One national character (PIC N), described as an elementary data item of category national.

The following items can also be passed BY VALUE:

- Reference-modified item of USAGE DISPLAY and length 1
- Reference-modified item of USAGE NATIONAL and length 1
- SHIFT-IN and SHIFT-OUT special registers
- LINAGE-COUNTER special register when it is USAGE BINARY
ADDRESS OF identifier-4

identifier-4 must be a data item of any level except 66 or 88 defined in the LINKAGE SECTION, the WORKING-STORAGE SECTION, or the LOCAL-STORAGE SECTION.

LENGTH OF special register

A LENGTH OF special register passed BY VALUE is treated as a PIC 9(9) binary. For information about the LENGTH OF special register, see “LENGTH OF” on page 19.

literal-3

Must be of one of the following types:

- A numeric literal
- A figurative constant ZERO
- A one-character alphanumeric literal
- A one-character national literal
- A symbolic character
- A single-byte figurative constant
  - SPACE
  - QUOTE
  - HIGH-VALUE
  - LOW-VALUE

ZERO is treated as a numeric value; a fullword binary zero is passed.

If literal-3 is a fixed-point numeric literal, it must have a precision of nine or fewer digits. In this case, a fullword binary representation of the literal value is passed.

If literal-3 is a floating-point numeric literal, an 8-byte internal floating-point (COMP-2) representation of the value is passed.

literal-3 must not be a DBCS literal.

RETURNING phrase

identifier-5

The RETURNING data item, which can be any data item (except the dynamic-length elementary item) defined in the DATA DIVISION. The return value of the called program is implicitly stored into identifier-5.

You can specify the RETURNING phrase for calls to functions written in COBOL, C, or in other programming languages that use C linkage conventions. If you specify the RETURNING phrase on a CALL to a COBOL subprogram:

- The called subprogram must specify the RETURNING phrase on its PROCEDURE DIVISION header.
- identifier-5 and the corresponding PROCEDURE DIVISION RETURNING identifier in the target program must have the same PICTURE, USAGE, SIGN, SYNCHRONIZE, JUSTIFIED, and BLANK WHEN ZERO clauses (except that PICTURE clause currency symbols can differ, and periods and commas can be interchanged due to the DECIMAL POINT IS COMMA clause).

When the target returns, its return value is assigned to identifier-5 using the rules for the SET statement if identifier-6 is of usage INDEX, POINTER, FUNCTION-POINTER, PROCEDURE-POINTER, or OBJECT REFERENCE. When identifier-5 is of any other usage, the rules for the MOVE statement are used.

The CALL ... RETURNING data item is an output-only parameter. On entry to the called program, the initial state of the PROCEDURE DIVISION RETURNING data item has an undefined and unpredictable value. You must initialize the PROCEDURE DIVISION RETURNING data item in the called program before you reference its value. The value that is passed back to the calling program is the final value of the PROCEDURE DIVISION RETURNING data item when the called program returns.
Note: If a COBOL program returns a doubleword binary item via a PROCEDURE DIVISION RETURNING header to a calling COBOL program with a CALL ... RETURNING statement, an issue occurs if only one of the programs is recompiled with Enterprise COBOL V6. Both the called and calling programs must be recompiled with Enterprise COBOL V6 together, so that the linkage convention for the RETURNING item is consistent.

If an EXCEPTION or OVERFLOW occurs, identifier-5 is not changed. identifier-5 must not be reference-modified.

The RETURN-CODE special register is not set by execution of CALL statements that include the RETURNING phrase.

ON EXCEPTION phrase

An exception condition occurs when the called subprogram cannot be made available. At that time, one of the following two actions will occur:

1. If the ON EXCEPTION phrase is specified, control is transferred to imperative-statement-1. Execution then continues according to the rules for each statement specified in imperative-statement-1. If a procedure branching or conditional statement that causes explicit transfer of control is executed, control is transferred in accordance with the rules for that statement. Otherwise, upon completion of the execution of imperative-statement-1, control is transferred to the end of the CALL statement and the NOT ON EXCEPTION phrase, if specified, is ignored.

2. If the ON EXCEPTION phrase is not specified in the CALL statement, the NOT ON EXCEPTION phrase, if specified, is ignored.

NOT ON EXCEPTION phrase

If an exception condition does not occur (that is, the called subprogram can be made available), control is transferred to the called program. After control is returned from the called program, control is transferred to:

• imperative-statement-2, if the NOT ON EXCEPTION phrase is specified.
• The end of the CALL statement in any other case. (If the ON EXCEPTION phrase is specified, it is ignored.)

If control is transferred to imperative-statement-2, execution continues according to the rules for each statement specified in imperative-statement-2. If a procedure branching or conditional statement that causes explicit transfer of control is executed, control is transferred in accordance with the rules for that statement. Otherwise, upon completion of the execution of imperative-statement-2, control is transferred to the end of the CALL statement.

ON OVERFLOW phrase

The ON OVERFLOW phrase has the same effect as the ON EXCEPTION phrase.

END-CALL phrase

This explicit scope terminator serves to delimit the scope of the CALL statement. END-CALL permits a conditional CALL statement to be nested in another conditional statement. END-CALL can also be used with an imperative CALL statement.

For more information, see “Delimited scope statements” on page 270.

Related references
PARMCHECK (Enterprise COBOL Programming Guide)
CANCEL statement

The CANCEL statement ensures that the referenced subprogram is entered in initial state the next time that it is called.

### Format

```
CANCEL identifier-1 literal-1
```

**identifier-1, literal-1**

*literal-1* must be an alphanumeric literal. *identifier-1* must be an alphanumeric, alphabetic, or zoned decimal data item such that its value can be a program-name. The rules of formation for program-names are dependent on the PGMNAME compiler option. For details, see the discussion of program-names in “PROGRAM-ID paragraph” on page 96 and the description of PGMNAME in the *Enterprise COBOL Programming Guide*.

*literal-1* or the contents of *identifier-1* must be the same as a literal or the contents of an identifier specified in an associated CALL statement.

Do not specify the name of a class or a method in the CANCEL statement.

After a CANCEL statement for a called subprogram has been executed, that subprogram no longer has a logical connection to the program. The contents of data items in external data records described by the subprogram are not changed when that subprogram is canceled. If a CALL statement is executed later by any program in the run unit naming the same subprogram, that subprogram is entered in its initial state.

When a CANCEL statement is executed, all programs contained within the program referenced in the CANCEL statement are also canceled. The result is the same as if a valid CANCEL were executed for each contained program in the reverse order in which the programs appear in the separately compiled program.

A CANCEL statement frees all allocated buffers for dynamic-length elementary items that are associated with the named program, including buffers for dynamic-length elementary items in its nested programs.

A CANCEL statement closes all open files that are associated with an internal file connector in the program named in an explicit CANCEL statement. USE procedures associated with those files are not executed.

You can cancel a called subprogram in any of the following ways:

- By referencing it as the operand of a CANCEL statement
- By terminating the run unit of which the subprogram is a member
- By executing an EXIT PROGRAM statement or a GOBACK statement in the called subprogram if that subprogram possesses the initial attribute

No action is taken when a CANCEL statement is executed if the specified program:

- Has not been dynamically called in this run unit by another COBOL program
- Has been called and subsequently canceled

In a multithreaded environment, a program cannot execute a CANCEL statement naming a program that is active on any thread. The named program must be completely inactive.

Called subprograms can contain CANCEL statements. However, a called subprogram must not execute a CANCEL statement that directly or indirectly cancels the calling program itself or that cancels any program higher than itself in the calling hierarchy. In such a case, the run unit is terminated.
A program named in a CANCEL statement must be a program that has been called and has executed an EXIT PROGRAM statement or a GOBACK statement.

A program can cancel a program that it did not call, provided that, in the calling hierarchy, the program that executes the CANCEL statement is higher than or equal to the program it is canceling. For example:

A calls B and B calls C    (When A receives control, it can cancel C.)
A calls B and A calls C    (When C receives control, it can cancel B.)

CLOSE statement

The CLOSE statement terminates the processing of volumes and files.

Format 1: CLOSE statement for sequential files

```
CLOSE file-name-1
  REEL 1
  UNIT 1
  FOR
  REMOVAL WITH NO REWIND
  WITH
  NO REWIND 1
  LOCK
```

Notes:

1 The REEL, UNIT, and NO REWIND phrases are not valid for VSAM files.

Format 2: CLOSE statement for indexed and relative files

```
CLOSE file-name-1
  LOCK
  WITH
```

Format 3: CLOSE statement for line-sequential files

```
CLOSE file-name-1
  REEL
  UNIT
  FOR
  REMOVAL WITH NO REWIND
  WITH
  NO REWIND
  LOCK
```
file-name-1
Designates the file upon which the CLOSE statement is to operate. If more than one file-name is specified, the files need not have the same organization or access. file-name-1 must not be a sort or merge file.

REEL and UNIT
You can specify these phrases only for QSAM multivolume or single volume files. The terms REEL and UNIT are interchangeable.

WITH NO REWIND and FOR REMOVAL
These phrases apply only to QSAM tape files. If they are specified for storage devices to which they do not apply, the close operation is successful and a status key value is set to indicate the file was on a non-reel medium.

A CLOSE statement can be executed only for a file in an open mode. After successful execution of a CLOSE statement (without the REEL/UNIT phrase if using format 1):

• The record area associated with the file-name is no longer available. Unsuccessful execution of a CLOSE statement leaves availability of the record data undefined.

• An OPEN statement for the file must be executed before any other input/output statement can be executed for the file and before data is moved to a record description entry associated with the file.

If the FILE STATUS clause is specified in the file-control entry, the associated file status key is updated when the CLOSE statement is executed.

If the file is in an open status and the execution of a CLOSE statement is unsuccessful, the EXCEPTION/ERROR procedure (if specified) for this file is executed.

Effect of CLOSE statement on file types
If the SELECT OPTIONAL clause is specified in the file-control entry for a file, and the file is not available at run time, standard end-of-file processing is not performed. For QSAM files, the file position indicator and current volume pointer are unchanged.

Files are divided into the following types:

Non-reel/unit
A file whose input or output medium is such that rewinding, reels, and units have no meaning. All VSAM files are of non-reel/unit file types. QSAM files can be of non-reel/unit file types.

Sequential single volume
A sequential file that is contained entirely on one volume. More than one file can be contained on this volume. All VSAM files are single volume. QSAM files can be single volume.

Sequential multivolume
A sequential file that is contained on more than one volume. QSAM files are the only files that can be multivolume. The concept of volume has no meaning for VSAM files.

The permissible combinations of CLOSE statement phrases are shown in the following tables:

• For sequential files: Sequential files and CLOSE statement phrases
• For indexed and relative files: Table 36 on page 303
• For line-sequential files: Table 37 on page 303

The meaning of each key letter is shown in Table 38 on page 303.

<table>
<thead>
<tr>
<th>Table 35. Sequential files and CLOSE statement phrases</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CLOSE statement phrases</strong></td>
</tr>
<tr>
<td>-----------------------------</td>
</tr>
<tr>
<td>CLOSE</td>
</tr>
<tr>
<td>CLOSE REEL/UNIT</td>
</tr>
</tbody>
</table>
### Table 35. Sequential files and CLOSE statement phrases (continued)

<table>
<thead>
<tr>
<th>CLOSE statement phrases</th>
<th>Non-reel/ unit</th>
<th>Sequential single-volume</th>
<th>Sequential multivolume</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSE REEL/UNIT WITH NOREWIND</td>
<td>F</td>
<td>B, F</td>
<td>B, F</td>
</tr>
<tr>
<td>CLOSE REEL/UNIT FOR REMOVAL</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>CLOSE WITH NO REWIND</td>
<td>C, H</td>
<td>B, C</td>
<td>A, B, C</td>
</tr>
<tr>
<td>CLOSE WITH LOCK</td>
<td>C, E</td>
<td>C, E, G</td>
<td>A, C, E, G</td>
</tr>
</tbody>
</table>

### Table 36. Indexed and relative file types and CLOSE statement phrases

<table>
<thead>
<tr>
<th>CLOSE statement phrases</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSE</td>
<td>C</td>
</tr>
<tr>
<td>CLOSE WITH LOCK</td>
<td>C,E</td>
</tr>
</tbody>
</table>

### Table 37. Line-sequential file types and CLOSE statement phrases

<table>
<thead>
<tr>
<th>CLOSE statement phrases</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSE</td>
<td>C</td>
</tr>
<tr>
<td>CLOSE WITH LOCK</td>
<td>C,E</td>
</tr>
</tbody>
</table>

### Table 38. Meanings of key letters for sequential file types

<table>
<thead>
<tr>
<th>Key</th>
<th>Actions taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Previous volumes unaffected</td>
</tr>
<tr>
<td></td>
<td><strong>Input and input-output files:</strong> Standard volume-switch processing is performed for all previous volumes (except those controlled by a previous CLOSE REEL/UNIT statement). Any subsequent volumes are not processed.</td>
</tr>
<tr>
<td></td>
<td><strong>Output files:</strong> Standard volume-switch processing is performed for all previous volumes (except those controlled by a previous CLOSE REEL/UNIT statement).</td>
</tr>
<tr>
<td>B</td>
<td>No rewinding of current reel: The current volume is left in its current position.</td>
</tr>
<tr>
<td>C</td>
<td>Close file</td>
</tr>
<tr>
<td></td>
<td>Standard system closing procedures are performed.</td>
</tr>
<tr>
<td>D</td>
<td>Volume removal: Treated as a comment.</td>
</tr>
<tr>
<td>E</td>
<td>File lock: The compiler ensures that this file cannot be opened again during this execution of the object program. If the file is a tape unit, it will be rewound and unloaded.</td>
</tr>
<tr>
<td>Key</td>
<td>Actions taken</td>
</tr>
<tr>
<td>-----</td>
<td>---------------</td>
</tr>
<tr>
<td>F</td>
<td><strong>Close volume</strong></td>
</tr>
<tr>
<td></td>
<td>Input and input-output files: If the current reel/unit is the last or only reel/unit for the file or if the reel is on a non-reel/unit medium, no volume switching is performed. If another reel/unit exists for the file, the following operations are performed: a volume switch, and the first record on the new volume is made available for reading. If no data records exist for the current volume, another volume switch occurs.</td>
</tr>
<tr>
<td></td>
<td>Output (reel/unit media) files: The following operations are performed: a volume switch. The next executed WRITE statement places the next logical record on the next direct access volume available. A close statement with the REEL phrase does not close the output file; only an end-of-volume condition occurs.</td>
</tr>
<tr>
<td></td>
<td>Output (non-reel/unit media) files: Execution of the CLOSE statement is considered successful. The file remains in the open mode and no action takes place except that the value of the I-O status associated with the file is updated.</td>
</tr>
<tr>
<td>G</td>
<td><strong>Rewind</strong>: The current volume is positioned at its physical beginning.</td>
</tr>
<tr>
<td>H</td>
<td><strong>Optional phrases ignored</strong>: The CLOSE statement is executed as if none of the optional phrases were present.</td>
</tr>
</tbody>
</table>

**COMPUTE statement**

The COMcompute statement assigns the value of an arithmetic expression to one or more data items. With the COMPUTE statement, arithmetic operations can be combined without the restrictions on receiving data items imposed by the rules for the ADD, SUBTRACT, MULTIPLY, and DIVIDE statements. When arithmetic operations are combined, the COMPUTE statement can be more efficient than the separate arithmetic statements written in a series.

**Format**

```
COMPUTE identifier-1 = arithmetic-expression ON SIZE ERROR imperative-statement-1
NOT ON SIZE ERROR imperative-statement-2
END-COMPUTE
```

**identifier-1**

Must name an elementary numeric item or an elementary numeric-edited item. Can name an elementary floating-point data item.
arithmetic-expression
Can be any arithmetic expression, as defined in “Arithmetic expressions” on page 244.

When the COMPUTE statement is executed, the value of arithmetic expression is calculated and stored as the new value of each data item referenced by identifier-1.

An arithmetic expression consisting of a single identifier, numeric function, or literal allows the user to set the value of the data items that are referenced by identifier-1 equal to the value of that identifier, function, or literal.

ROUNDED phrase
For a discussion of the ROUNDED phrase, see “ROUNDED phrase” on page 273.

SIZE ERROR phrases
For a discussion of the SIZE ERROR phrases, see “SIZE ERROR phrases” on page 273.

END-COMPUTE phrase
This explicit scope terminator serves to delimit the scope of the COMPUTE statement. END-COMPUTE permits a conditional COMPUTE statement to be nested in another conditional statement. END-COMPUTE can also be used with an imperative COMPUTE statement.

For more information, see “Delimited scope statements” on page 270.

CONTINUE statement
The CONTINUE statement is a no operation statement. CONTINUE indicates that no executable instruction is present.

Format
```
  CONTINUE
```

DELETE statement
The DELETE statement removes a record from an indexed or relative file. For indexed files, the key can then be reused for record addition. For relative files, the space is then available for a new record with the same RELATIVE KEY value.

When the DELETE statement is executed, the associated file must be open in I-O mode.

Format
```
  DELETE  file-name-1
    RECORD
    INVALID  KEY  imperative-statement-1
    NOT INVALID  KEY  imperative-statement-2
    END-DELETE
```
**file-name-1**

Must be defined in an FD entry in the DATA DIVISION and must be the name of an indexed or relative file.

After successful execution of a DELETE statement, the record is removed from the file and can no longer be accessed.

Execution of the DELETE statement does not affect the contents of the record area associated with file-name-1 or the content of the data item referenced by the data-name specified in the DEPENDING ON phrase of the RECORD clause associated with file-name-1.

If the FILE STATUS clause is specified in the file-control entry, the associated file status key is updated when the DELETE statement is executed.

The file position indicator is not affected by execution of the DELETE statement.

**Sequential access mode**

For a file in sequential access mode, the previous input/output statement must be a successfully executed READ statement. When the DELETE statement is executed, the system removes the record that was retrieved by that READ statement.

For a file in sequential access mode, the INVALID KEY and NOT INVALID KEY phrases must not be specified. An EXCEPTION/ERROR procedure can be specified.

**Random or dynamic access mode**

In random or dynamic access mode, DELETE statement execution results depend on the file organization: indexed or relative.

When the DELETE statement is executed, the system removes the record identified by the contents of the prime RECORD KEY data item for indexed files, or the RELATIVE KEY data item for relative files. If the file does not contain such a record, an INVALID KEY condition exists. (See “Invalid key condition” on page 280.)

Both the INVALID KEY phrase and an applicable EXCEPTION/ERROR procedure can be omitted.

Transfer of control after the successful execution of a DELETE statement, with the NOT INVALID KEY phrase specified, is to the imperative statement associated with the phrase.

**END-DELETE phrase**

This explicit scope terminator serves to delimit the scope of the DELETE statement. END-DELETE permits a conditional DELETE statement to be nested in another conditional statement. END-DELETE can also be used with an imperative DELETE statement.

For more information, see “Delimited scope statements” on page 270.
DISPLAY statement

The DISPLAY statement transfers the contents of each operand to the output device. The contents are displayed on the output device in the order, left to right, in which the operands are listed.

**Format**

```
DISPLAY
  identifier-1
  literal-1
  UPON mnemonic-name-1
  environment-name-1
  WITH NO ADVANCING
```

**identifier-1**

*identifier-1* references the data that is to be displayed. *identifier-1* can reference any data item except an item of usage PROCEDURE-POINTER, FUNCTION-POINTER, OBJECT REFERENCE, or INDEX. *identifier-1* cannot be an index-name.

If *identifier-1* is a binary, internal decimal, or internal floating-point data item, *identifier-1* is converted automatically to external format as follows:

- Binary and internal decimal items are converted to zoned decimal. Negative signed values cause a low-order sign overpunch. This can cause unreadable output if the DISPSIGN(COMPAT) compiler option is used. If the DISPSIGN(SEP) compiler option is in effect, the sign is displayed separately from the data, as if SIGN IS SEPARATE was specified. For details, see DISPSIGN in the Enterprise COBOL Programming Guide.
- Internal floating-point numbers are converted to external floating-point numbers for display such that:
  - A COMP-1 item will display as if it had an external floating-point PICTURE clause of -.9(8)E-99.
  - A COMP-2 item will display as if it had an external floating-point PICTURE clause of -.9(17)E-99.

Data items defined with USAGE POINTER are converted to a zoned decimal number that has an implicit PICTURE clause of PIC 9(10).

If the output is directed to CONSOLE, data items described with usage NATIONAL are converted from national character representation to EBCDIC. The conversion uses the EBCDIC code page that was specified in the CODEPAGE compiler option when the source code was compiled. National characters without EBCDIC counterparts are converted to default substitution characters; no exception condition is indicated or raised.

If the output is not directed to CONSOLE, data items described with usage NATIONAL are written without conversion and without data validation.

No other categories of data require conversion.

DBCS data items, explicitly or implicitly defined as USAGE DISPLAY-1, are transferred to the sending field of the output device. For proper results, the output device must have the capability to recognize DBCS shift-out and shift-in control characters.

Both DBCS and non-DBCS operands can be specified in a single DISPLAY statement.

*identifier-1* must not be a dynamic-length group item.
**literal-1**

Can be any literal or any figurative constant as specified in “Figurative constants” on page 13. When a figurative constant is specified, only a single occurrence of that figurative constant is displayed.

**UPON**

`environment-name-1` or the environment name associated with `mnemonic-name-1` must be associated with an output device. See “SPECIAL-NAMES paragraph” on page 108.

A default logical record size is assumed for each device, as follows:

**The system logical output device**

- 120 characters

**The system punch device**

- 80 characters

**The console**

- 100 characters

A maximum logical record size is allowed for each device, as follows:

**The system logical output device**

- 255 characters

**The system punch device**

- 255 characters

**The console**

- 100 characters

On the system punch device, the last eight characters are used for PROGRAM-ID name.

When the UPON phrase is omitted, the system's logical output device is assumed. The list of valid environment-names in a DISPLAY statement is shown in Table 5 on page 110.

For details on routing DISPLAY output to stdout, see *Displaying values on a screen or in a file (DISPLAY)* in the *Enterprise COBOL Programming Guide*.

**WITH NO ADVANCING**

When specified, the positioning of the output device will not be changed in any way following the display of the last operand.

If the WITH NO ADVANCING phrase is not specified, after the last operand has been transferred to the output device, the positioning of the output device will be reset to the leftmost position of the next line of the device.

Enterprise COBOL does not support output devices that are capable of positioning to a specific character position. See *Displaying values on a screen or in a file (DISPLAY)* in the *Enterprise COBOL Programming Guide* for more information about the DISPLAY statement.

The DISPLAY statement transfers the data in the sending field to the output device. The size of the sending field is the total byte count of all operands listed. If the output device is capable of receiving data of the same size as the data item being transferred, then the data item is transferred. If the output device is not capable of receiving data of the same size as the data item being transferred, then one of the following applies:

- If the total count is less than the device maximum, the remaining rightmost positions are padded with spaces.

- If the total count exceeds the maximum, as many records are written as are needed to display all operands. Any operand being printed or displayed when the end of a record is reached is continued in the next record.

If a DBCS operand must be split across multiple records, it will be split only on a double-byte boundary.

Shift code insertion is required for splitting DBCS items. That is, when a DBCS operand is split across multiple records, the shift-in character is inserted at the end of the current record, and the shift-out character is inserted at the beginning of the next record. A space is padded after the shift-in character, if
necessary. These inserted shift codes and spaces are included in the total byte count of the sending data items.

After the last operand has been transferred to the output device, the device is reset to the leftmost position of the next line of the device.

If a DBCS data item or literal is specified in a DISPLAY statement, the size of the sending field is the total byte count of all operands listed, with each DBCS character counted as two bytes, plus the necessary shift codes and spaces for DBCS.

**DIVIDE statement**

The DIVIDE statement divides one numeric data item into or by others and sets the values of data items equal to the quotient and remainder.

```
DIVIDE statement

Format 1: DIVIDE statement

DIVIDE identifier-1 INTO identifier-2 ROUNDED

ON SIZE ERROR imperative-statement-1

NOT ON SIZE ERROR imperative-statement-2

END-DIVIDE

In format 1, the value of identifier-1 or literal-1 is divided into the value of identifier-2, and the quotient is then stored in identifier-2. For each successive occurrence of identifier-2, the division takes place in the left-to-right order in which identifier-2 is specified.
```
Format 2: DIVIDE statement with INTO and GIVING phrases

```
DIVIDE     identifier-1 INTO identifier-2 GIVING identifier-3 
            literal-1                       literal-2 
                                        \        \ 
                                         ROUNDED          
                                           \                
                                            \ SIZE ERROR    -- imperative-statement-1 
                                                \                
                                                 NOT \      \ SIZE ERROR -- imperative-statement-2 
                                                       \                
                                                        END-DIVIDE 
```

In format 2, the value of `identifier-1` or `literal-1` is divided into the value of `identifier-2` or `literal-2`. The value of the quotient is stored in each data item referenced by `identifier-3`.

Format 3: DIVIDE statement with BY and GIVING phrases

```
DIVIDE     identifier-1 BY identifier-2 GIVING identifier-3 
            literal-1                       literal-2 
                                        \        \ 
                                         ROUNDED          
                                           \                
                                            \ SIZE ERROR    -- imperative-statement-1 
                                                \                
                                                 NOT \      \ SIZE ERROR -- imperative-statement-2 
                                                       \                
                                                        END-DIVIDE 
```

In format 3, the value of `identifier-1` or `literal-1` is divided by the value of `identifier-2` or `literal-2`. The value of the quotient is stored in each data item referenced by `identifier-3`.
In format 4, the value of identifier-1 or literal-1 is divided into identifier-2 or literal-2. The value of the quotient is stored in identifier-3, and the value of the remainder is stored in identifier-4.

In format 5, the value of identifier-1 or literal-1 is divided by identifier-2 or literal-2. The value of the quotient is stored in identifier-3, and the value of the remainder is stored in identifier-4.

For all formats:

- **identifier-1, identifier-2**
  - Must name an elementary numeric data item.

- **identifier-3, identifier-4**
  - Must name an elementary numeric or numeric-edited item.

- **literal-1, literal-2**
  - Must be a numeric literal.

In formats 1, 2, and 3, floating-point data items and literals can be used anywhere that a numeric data item or literal can be specified.
In formats 4 and 5, floating-point data items or literals cannot be used.

**ROUNDED phrase**
For formats 1, 2, and 3, see “ROUNDED phrase” on page 273.

For formats 4 and 5, the quotient used to calculate the remainder is in an intermediate field. The value of the intermediate field is truncated rather than rounded.

**REMAINDER phrase**
The result of subtracting the product of the quotient and the divisor from the dividend is stored in identifier-4. If identifier-3, the quotient, is a numeric-edited item, the quotient used to calculate the remainder is an intermediate field that contains the unedited quotient.

The REMAINDER phrase is invalid if the receiver or any of the operands is a floating-point item.

Any subscripts for identifier-4 in the REMAINDER phrase are evaluated after the result of the divide operation is stored in identifier-3 of the GIVING phrase.

**SIZE ERROR phrases**
For formats 1, 2, and 3, see “SIZE ERROR phrases” on page 273.

For formats 4 and 5, if a size error occurs in the quotient, no remainder calculation is meaningful. Therefore, the contents of the quotient field (identifier-3) and the remainder field (identifier-4) are unchanged.

If size error occurs in the remainder, the contents of the remainder field (identifier-4) are unchanged.

In either of these cases, you must analyze the results to determine which situation has actually occurred.

For information about the NOT ON SIZE ERROR phrase, see “SIZE ERROR phrases” on page 273.

**END-DIVIDE phrase**
This explicit scope terminator serves to delimit the scope of the DIVIDE statement. END-DIVIDE turns a conditional DIVIDE statement into an imperative statement that can be nested in another conditional statement. END-DIVIDE can also be used with an imperative DIVIDE statement.

For more information, see “Delimited scope statements” on page 270.

**ENTRY statement**
The ENTRY statement establishes an alternate entry point into a COBOL called subprogram.

The ENTRY statement cannot be used in:

- Programs that specify a return value using the PROCEDURE DIVISION RETURNING phrase. For details, see the discussion of the RETURNING phrase under “The PROCEDURE DIVISION header” on page 238.
- Nested program. See “Nested programs” on page 81 for a description of nested programs.

When a CALL statement that specifies the alternate entry point is executed in a calling program, control is transferred to the next executable statement following the ENTRY statement.
Format

```
ENTRY literal-1
```

**literal-1**

Must be an alphanumeric literal that conform to the rules for the formation of a program-name in an outermost program (see “PROGRAM-ID paragraph” on page 96).

Must not match the program-ID or any other ENTRY literal in this program.

Must not be a figurative constant.

Execution of the called program begins at the first executable statement following the ENTRY statement whose literal corresponds to the literal or identifier specified in the CALL statement.

The entry point name on the ENTRY statement can be affected by the PGMNAME compiler option. For details, see *PGMNAME* in the *Enterprise COBOL Programming Guide*.

**USING phrase**

For a discussion of the USING phrase, see “The PROCEDURE DIVISION header” on page 238.

**EVALUATE statement**

The EVALUATE statement provides a shorthand notation for a series of nested IF statements. The EVALUATE statement can evaluate multiple conditions. The subsequent action depends on the results of these evaluations.
Operands before the WHEN phrase
Are interpreted in one of two ways, depending on how they are specified:
- Individually, they are called selection subjects.
- Collectively, they are called a set of selection subjects.

Operands in the WHEN phrase
Are interpreted in one of two ways, depending on how they are specified:
- Individually, they are called selection objects
- Collectively, they are called a set of selection objects.

ALSO
Separates selection subjects within a set of selection subjects; separates selection objects within a set of selection objects.
**THROUGH and THRU**

Are equivalent.

All identifiers in the EVALUATE statement must not be dynamic-length group items.

Two operands connected by a THRU phrase must be of the same class. The two operands thus connected constitute a single selection object.

The number of selection objects within each set of selection objects must be equal to the number of selection subjects.

Each selection object within a set of selection objects must correspond to the selection subject having the same ordinal position within the set of selection subjects, according to the following rules:

- Identifiers, literals, or arithmetic expressions appearing within a selection object must be valid operands for comparison to the corresponding operand in the set of selection subjects.
- `condition-1, condition-2`, or the word TRUE or FALSE appearing as a selection object must correspond to a conditional expression or the word TRUE or FALSE in the set of selection subjects.
- The word ANY can correspond to a selection subject of any type.

**END-EVALUATE phrase**

This explicit scope terminator serves to delimit the scope of the EVALUATE statement. END-EVALUATE permits a conditional EVALUATE statement to be nested in another conditional statement.

For more information, see “Delimited scope statements” on page 270.

**Determining values**

The execution of the EVALUATE statement operates as if each selection subject and selection object were evaluated and assigned a numeric, alphanumeric, DBCS, or national character value; a range of numeric, alphanumeric, DBCS, or national character values; or a truth value.

These values are determined as follows:

- Any selection subject specified by `identifier-1, identifier-2, ...` and any selection object specified by `identifier-3` or `identifier-5` without the NOT or THRU phrase are assigned the value and class of the data item that they reference.
- Any selection subject specified by `literal-1, literal-2, ...` and any selection object specified by `literal-3` or `literal-5` without the NOT or THRU phrase are assigned the value and class of the specified literal. If `literal-3` or `literal-5` is the figurative constant ZERO, QUOTE, or SPACE, the figurative constant is assigned the class of the corresponding selection subject.
- Any selection subject in which `expression-1, expression-2, ...` is specified as an arithmetic expression, and any selection object without the NOT or THRU phrase in which `arithmetic-expression-1` or `arithmetic-expression-3` is specified, are assigned numeric values according to the rules for evaluating an arithmetic expression. (See “Arithmetic expressions” on page 244.)
- Any selection subject in which `expression-1, expression-2, ...` is specified as a conditional expression, and any selection object in which `condition-1` or `condition-2` is specified, are assigned a truth value according to the rules for evaluating conditional expressions. (See “Conditional expressions” on page 246.)
- Any selection subject or any selection object specified by the words TRUE or FALSE is assigned a truth value. The truth value "true" is assigned to those items specified with the word TRUE, and the truth value "false" is assigned to those items specified with the word FALSE.
- Any selection object specified by the word ANY is not further evaluated.
- If the THRU phrase is specified for a selection object without the NOT phrase, the range of values includes all values that, when compared to the selection subject, are greater than or equal to the first operand and less than or equal to the second operand according to the rules for comparison. If the first operand is greater than the second operand, there are no values in the range.
• If the NOT phrase is specified for a selection object, the values assigned to that item are all values not equal to the value, or range of values, that would have been assigned to the item had the NOT phrase been omitted.

**Comparing selection subjects and objects**

The execution of the EVALUATE statement then proceeds as if the values assigned to the selection subjects and selection objects were compared to determine whether any WHEN phrase satisfies the set of selection subjects.

This comparison proceeds as follows:

1. Each selection object within the set of selection objects for the first WHEN phrase is compared to the selection subject having the same ordinal position within the set of selection subjects. One of the following conditions must be satisfied if the comparison is to be satisfied:
   a. If the items being compared are assigned numeric, alphanumeric, DBCS, or national character values, or a range of numeric, alphanumeric, DBCS, or national character values, the comparison is satisfied if the value, or one value in the range of values, assigned to the selection object is equal to the value assigned to the selection subject according to the rules for comparison.
   b. If the items being compared are assigned truth values, the comparison is satisfied if the items are assigned identical truth values.
   c. If the selection object being compared is specified by the word ANY, the comparison is always satisfied, regardless of the value of the selection subject.

2. If the above comparison is satisfied for every selection object within the set of selection objects being compared, the WHEN phrase containing that set of selection objects is selected as the one satisfying the set of selection subjects.

3. If the above comparison is not satisfied for every selection object within the set of selection objects being compared, that set of selection objects does not satisfy the set of selection subjects.

4. This procedure is repeated for subsequent sets of selection objects in the order of their appearance in the source text, until either a WHEN phrase satisfying the set of selection subjects is selected or until all sets of selection objects are exhausted.

**Executing the EVALUATE statement**

After the comparison operation is completed, execution of the EVALUATE statement proceeds.

• If a WHEN phrase is selected, execution continues with the first imperative-statement-1 following the selected WHEN phrase. Note that multiple WHEN statements are allowed for a single imperative-statement-1.

• If no WHEN phrase is selected and a WHEN OTHER phrase is specified, execution continues with imperative-statement-2.

• If no WHEN phrase is selected and no WHEN OTHER phrase is specified, execution continues with the next executable statement following the scope delimiter.

• The scope of execution of the EVALUATE statement is terminated when execution reaches the end of the scope of the selected WHEN phrase or WHEN OTHER phrase, or when no WHEN phrase is selected and no WHEN OTHER phrase is specified.

**EXIT statement**

The EXIT statement provides a common end point for a series of procedures. It also provides a way to exit from a section, a paragraph, or an inline PERFORM statement.

**Note:** Enterprise COBOL does not yet support the format 4 EXIT statement, EXIT FUNCTION.
**Format 1 (simple)**

The format 1 EXIT statement provides a common end point for a series of procedures.

```
Format 1
  ➡️ paragraph-name . ➡️ EXIT ➡️
```

The format 1 EXIT statement enables you to assign a procedure-name to a given point in a program. The format 1 EXIT statement is treated as a CONTINUE statement. Any statements following the EXIT statement are executed.

**Format 2 (program)**

The EXIT PROGRAM statement specifies the end of a called program and returns control to the calling program.

You can specify EXIT PROGRAM only in the PROCEDURE DIVISION of a program. EXIT PROGRAM must not be used in a declarative procedure in which the GLOBAL phrase is specified.

```
Format 2
  ➡️ EXIT PROGRAM ➡️
```

If control reaches an EXIT PROGRAM statement in a program that does not possess the INITIAL attribute while operating under the control of a CALL statement (that is, the CALL statement is active), control returns to the point in the calling routine (program or method) immediately following the CALL statement. The state of the calling routine is identical to that which existed at the time it executed the CALL statement. The contents of data items and the contents of data files shared between the calling and called routine could have been changed. The state of the called program or method is not altered except that the ends of the ranges of all executed PERFORM statements are considered to have been reached.

The execution of an EXIT PROGRAM statement in a called program that possesses the INITIAL attribute is equivalent also to executing a CANCEL statement referencing that program.

If control reaches an EXIT PROGRAM statement, and no CALL statement is active, control passes through the exit point to the next executable statement.

If a subprogram specifies the PROCEDURE DIVISION RETURNING phrase, the value in the data item referred to by the RETURNING phrase becomes the result of the subprogram invocation.

The EXIT PROGRAM statement should be the last statement in a sequence of imperative statements. When it is not, statements following the EXIT PROGRAM will not be executed if a CALL statement is active.

When there is no next executable statement in a called program, an implicit EXIT PROGRAM statement is executed.

**Format 3 (method)**

The EXIT METHOD statement specifies the end of an invoked method.

```
Format 3
  ➡️ EXIT METHOD ➡️
```

You can specify EXIT METHOD only in the PROCEDURE DIVISION of a method. EXIT METHOD causes the executing method to terminate, and control returns to the invoking statement. If the containing method specifies the PROCEDURE DIVISION RETURNING phrase, the value in the data item referred to by the RETURNING phrase becomes the result of the method invocation.
If you need method-specific data to be in the last-used state on each invocation, define it in method WORKING-STORAGE. If you need method-specific data to be in the initial state on each invocation, define it in method LOCAL-STORAGE.

If control reaches an EXIT METHOD statement in a method definition, control returns to the point that immediately follows the INVOKE statement in the invoking program or method. The state of the invoking program or method is identical to that which existed at the time it executed the INVOKE statement.

The contents of data items and the contents of data files shared between the invoking program or method and the invoked method could have changed. The state of the invoked method is not altered except that the end of the ranges of all PERFORM statements executed by the method are considered to have been reached.

The EXIT METHOD statement does not have to be the last statement in a sequence of imperative statements, but the statements following the EXIT METHOD will not be executed.

When there is no next executable statement in an invoked method, an implicit EXIT METHOD statement is executed.

**Format 5 (inline-perform)**

The EXIT PERFORM statement controls the exit from an inline PERFORM without using a GO TO statement or a PERFORM ... THROUGH statement.

```
Format 5
-< EXIT PERFORM < - CYCLE
```

If you specify an EXIT PERFORM statement outside of an inline PERFORM statement, the EXIT PERFORM is ignored.

When an EXIT PERFORM statement without the CYCLE phrase is executed, control is passed to an implicit CONTINUE statement. This implicit CONTINUE statement immediately follows the END-PERFORM phrase that matches the most closely preceding and unterminated inline PERFORM statement.

When an EXIT PERFORM statement with the CYCLE phrase is executed, control is passed to an implicit CONTINUE statement. This implicit CONTINUE statement immediately precedes the END-PERFORM phrase that matches the most closely preceding and unterminated inline PERFORM statement.

**Format 6 (procedure)**

The EXIT PARAGRAPH statement controls the exit from the middle of a paragraph without executing any following statements within the paragraph. The EXIT SECTION statement controls the exit from a section without executing any following statements within the section.

```
Format 6
-< EXIT PARAGRAPH < -
```

**EXIT PARAGRAPH**

When an EXIT PARAGRAPH statement is executed, control is passed to an implicit CONTINUE statement that immediately follows the last explicit statement of the current paragraph. This return mechanism supersedes any other return mechanisms that are associated with language elements, such as PERFORM, SORT, and USE for that paragraph.
EXIT SECTION

The EXIT SECTION statement can be specified only in a section.

When an EXIT SECTION statement is executed, control is passed to an unnamed empty paragraph that immediately follows the last paragraph of the current section. This return mechanism supersedes any other return mechanisms that are associated with language elements, such as PERFORM, SORT, and USE for that section.

FREE statement

The FREE statement releases dynamic storage that was previously obtained with an ALLOCATE statement.

```
<table>
<thead>
<tr>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREE data-name-1</td>
</tr>
</tbody>
</table>
```

**data-name-1**

Must be defined as USAGE POINTER or USAGE POINTER-32.

Can be qualified or subscripted.

The FREE statement is processed as follows:

- If the pointer referenced by `data-name-1` identifies the start of storage that is currently allocated by an ALLOCATE statement, that storage is released and the pointer referenced by `data-name-1` is set to NULL, the length of the released storage is the length of the storage obtained by the ALLOCATE statement, and the contents of any data items located within the released storage area become undefined.

- If the pointer referenced by `data-name-1` contains the predefined address NULL or the address of storage that is not acquired by the ALLOCATE statement, no storage will be freed. The pointer `data-name-1` will be kept unchanged and the behavior is undefined.

If more than one `data-name-1` is specified in a FREE statement, the result of executing this FREE statement is the same as if a separate FREE statement had been written for each `data-name-1` in the same order as specified in the FREE statement.

**Related references**

- “ALLOCATE statement” on page 289
- “Example: ALLOCATE and FREE storage for UNBOUNDED tables” on page 291
- POINTER phrase
- POINTER-32 phrase

GOBACK statement

The GOBACK statement functions like the EXIT PROGRAM statement when it is coded as part of a called program (or the EXIT METHOD statement when GOBACK is coded as part of an invoked method) and like the STOP RUN statement when coded in a main program.

The GOBACK statement specifies the logical end of a called program or invoked method.

```
<table>
<thead>
<tr>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOBACK</td>
</tr>
</tbody>
</table>
```
A GOBACK statement should appear as the only statement or as the last of a series of imperative statements in a sentence because any statements following the GOBACK are not executed. GOBACK must not be used in a declarative procedure in which the GLOBAL phrase is specified.

If control reaches a GOBACK statement while a CALL statement is active, control returns to the point in the calling program or method immediately following the CALL statement, as in the EXIT PROGRAM statement.

If control reaches a GOBACK statement while an INVOKE statement is active, control returns to the point in the invoking program or method immediately following the INVOKE statement, as in the EXIT METHOD statement.

In addition, the execution of a GOBACK statement in a called program that possesses the INITIAL attribute is equivalent to executing a CANCEL statement referencing that program.

The table below shows the action taken for the GOBACK statement in a main program, a subprogram, and an invoked method.

<table>
<thead>
<tr>
<th>Termination statement</th>
<th>Main program</th>
<th>Subprogram</th>
<th>Invoked method</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOBACK</td>
<td>Returns to the calling program. (Can be the system, which causes the application to end.)</td>
<td>Returns to the calling program.</td>
<td>Returns to the calling method.</td>
</tr>
</tbody>
</table>

**GO TO statement**

The GO TO statement transfers control from one part of the PROCEDURE DIVISION to another.

The types of GO TO statements are:

- Unconditional
- Conditional
- Altered

**Unconditional GO TO**

The unconditional GO TO statement transfers control to the first statement in the paragraph or section identified by procedure-name, unless the GO TO statement has been modified by an ALTER statement.

For more information, see “ALTER statement” on page 292.

**Format 1: unconditional GO TO statement**

```
GO TO procedure-name-1
```

**procedure-name-1**

Must name a procedure or a section in the same PROCEDURE DIVISION as the GO TO statement.

When the unconditional GO TO statement is not the last statement in a sequence of imperative statements, the statements following the GO TO are not executed.

When a paragraph is referred to by an ALTER statement, the paragraph must consist of a paragraph-name followed by an unconditional or altered GO TO statement.
Conditional GO TO

The conditional GO TO statement transfers control to one of a series of procedures, depending on the value of the data item referenced by identifier-1.

Format 2: conditional GO TO statement

\[
\text{GO TO procedure-name-1 DEPENDING ON identifier-1}
\]

**procedure-name-1**

Must be a procedure or a section in the same PROCEDURE DIVISION as the GO TO statement. The number of procedure-names must not exceed 255.

**identifier-1**

Must be a numeric elementary data item that is an integer.

If 1, control is transferred to the first statement in the procedure named by the first occurrence of procedure-name-1.

If 2, control is transferred to the first statement in the procedure named by the second occurrence of procedure-name-1, and so forth.

If the value of identifier is anything other than a value within the range of 1 through n (where n is the number of procedure-names specified in this GO TO statement), no control transfer occurs. Instead, control passes to the next statement in the normal sequence of execution.

Altered GO TO

The altered GO TO statement transfers control to the first statement of the paragraph named in the ALTER statement.

You cannot specify the altered GO TO statement in the following cases:

- A program or method that has the RECURSIVE attribute
- A program compiled with the THREAD compiler option

An ALTER statement referring to the paragraph that contains the altered GO TO statement should be executed before the GO TO statement is executed. Otherwise, the GO TO statement acts like a CONTINUE statement.

Format 3: altered GO TO statement

\[
\text{GO TO paragraph-name}
\]

When an ALTER statement refers to a paragraph, the paragraph can consist only of the paragraph-name followed by an unconditional or altered GO TO statement.
**IF statement**

The IF statement evaluates a condition and provides for alternative actions in the object program, depending on the evaluation.

### Format

```
IF condition-1
  THEN
    statement-1
    NEXT SENTENCE
  ELSE
    statement-2
    NEXT SENTENCE
END-IF
```

Notes:

1. END-IF can be specified with `statement-2` or **NEXT SENTENCE**.

#### condition-1

Can be any simple or complex condition, as described in “Conditional expressions” on page 246.

#### statement-1, statement-2

Can be any one of the following options:

- An imperative statement
- A conditional statement
- An imperative statement followed by a conditional statement

#### NEXT SENTENCE

The NEXT SENTENCE phrase transfers control to an implicit CONTINUE statement immediately following the next separator period.

When NEXT SENTENCE is specified with END-IF, control does not pass to the statement following the END-IF. Instead, control passes to the statement after the closest following period.

#### END-IF phrase

This explicit scope terminator serves to delimit the scope of the IF statement. END-IF permits a conditional IF statement to be nested in another conditional statement. For more information about explicit scope terminators, see “Delimited scope statements” on page 270.

The scope of an IF statement can be terminated by any of the following options:

- An END-IF phrase at the same level of nesting
- A separator period
- If nested, by an ELSE phrase associated with an IF statement at a higher level of nesting

#### Transferring control

The topic describes the actions to take when conditions tested is true or false.

If the condition tested is true, one of the following actions takes place:

- If `statement-1` is specified, `statement-1` is executed. If `statement-1` contains a procedure branching or conditional statement, control is transferred according to the rules for that statement. If `statement-1`
does not contain a procedure-branching statement, the ELSE phrase, if specified, is ignored, and control passes to the next executable statement after the corresponding END-IF or separator period.

- If NEXT SENTENCE is specified, control passes to an implicit CONTINUE statement immediately following the next separator period.

If the condition tested is false, one of the following actions takes place:

- If ELSE statement-2 is specified, statement-2 is executed. If statement-2 contains a procedure-branching or conditional statement, control is transferred, according to the rules for that statement. If statement-2 does not contain a procedure-branching or conditional statement, control is passed to the next executable statement after the corresponding END-IF or separator period.

- If ELSE NEXT SENTENCE is specified, control passes to an implicit CONTINUE STATEMENT immediately preceding the next separator period.

- If neither ELSE statement-2 nor ELSE NEXT SENTENCE is specified, control passes to the next executable statement after the corresponding END-IF or separator period.

When the ELSE phrase is omitted, all statements following the condition and preceding the corresponding END-IF or the separator period for the sentence are considered to be part of statement-1.

**Nested IF statements**

When an IF statement appears as statement-1 or statement-2, or as part of statement-1 or statement-2, that IF statement is *nested*.

When an IF statement appears as statement-1 or statement-2, or as part of statement-1 or statement-2, that IF statement is *nested*.

Nested IF statements are considered to be matched IF, ELSE, and END-IF combinations proceeding from left to right. Thus, any ELSE encountered is matched with the nearest preceding IF that either has not been already matched with an ELSE or has not been implicitly or explicitly terminated. Any END-IF encountered is matched with the nearest preceding IF that has not been implicitly or explicitly terminated.
**INITIALIZE statement**

The INITIALIZE statement sets selected categories of data fields to predetermined values. The INITIALIZE statement is functionally equivalent to one or more MOVE statements.

---

**Format**

```
INITIALIZE identifier-1
WITH FILLER

ALL category-name
TO VALUE

THEN REPLACING category-name
DATA BY identifier-2 literal-1

THEN TO DEFAULT
```

Where `category-name` is:

- ALPHABETIC
- ALPHANUMERIC
- ALPHANUMERIC-EDITED
- DBCS
- EGCS
- NATIONAL
- NATIONAL-EDITED
- NUMERIC
- NUMERIC-EDITED

**identifier-1**

Receiving areas.

`identifier-1` must reference one of the following items:

- An alphanumeric group item
- A national group item
- An elementary data item of one of the following categories:
  - Alphabetic
  - Alphanumeric
  - Alphanumeric-edited
  - DBCS
  - External floating-point
  - Internal floating-point
A special register that is valid as a receiving operand in a MOVE statement with \textit{identifier-2} or \textit{literal-1} as the sending operand.

\textit{identifier-1} references an elementary item or a group item. The effect of the execution of an INITIALIZE statement is as if a series of implicit MOVE statements, each of which has an elementary data item as its receiving operand, were executed.

When \textit{identifier-1} references a national group item, \textit{identifier-1} is processed as a group item.

\textit{identifier-1} cannot be a dynamic-length elementary item.

\textbf{\textit{identifier-2, literal-1}}

\textbf{Sending areas.}

When \textit{identifier-2} references a national group item, \textit{identifier-2} is processed as an elementary data item of category national.

\textit{identifier-2} must reference an elementary data item (or a national group item treated as elementary) that is valid as a sending operand in a MOVE statement with \textit{identifier-1} as the receiving operand.

\textit{literal-1} must be a literal that is valid as a sending operand in a MOVE statement with \textit{identifier-1} as the receiving operand.

A subscripted item can be specified for \textit{identifier-1}. A complete table can be initialized only by specifying \textit{identifier-1} as a group that contains the complete table.

\textbf{Usage note:} The data description entry for \textit{identifier-1} can contain the DEPENDING phrase of the OCCURS clause. However, you cannot use the INITIALIZE statement to initialize a variably-located item or a variable-length item.

The data description entry for \textit{identifier-1} must not contain a RENAMES clause.

Special registers can be specified for \textit{identifier-1} and \textit{identifier-2} only if they are valid receiving fields or sending fields, respectively, for the implied MOVE statements.

\textbf{FILLER phrase}

When the FILLER phrase is specified, the receiving elementary data items that have an explicit or implicit FILLER clause will be initialized.

\textbf{VALUE phrase}

When the VALUE phrase is specified:

- If ALL is specified in the VALUE phrase, it is as if all of the categories listed in \textit{category-name} were specified.
- The same category cannot be repeated in a VALUE phrase.

\textbf{REPLACING phrase}

When the REPLACING phrase is specified:

- \textit{identifier-2} must reference an item of a category that is valid as a sending operand in a MOVE statement to an item of the corresponding category specified in the REPLACING phrase.
- \textit{literal-1} must be of a category that is valid as a sending operand in a MOVE statement to an item of the corresponding category specified in the REPLACING phrase.
- A floating-point literal, a data item of category internal floating-point, or a data item of category external floating point is treated as if it were in the NUMERIC category.
• The same category cannot be repeated in a REPLACING phrase.

With the exception of EGCS, the keyword after the word REPLACING corresponds to a category of data shown in “Classes and categories of data” on page 153.

EGCS in the REPLACING phrase is synonymous with DBCS.

**Related references**

“ALLOCATE statement” on page 289

**INITIALIZE statement rules**

The effect of the execution of an INITIALIZE statement is as if a series of implicit MOVE statements, each of which has an elementary data item as its receiving operand, were executed. The receiving operands of these implicit statements are defined in rule 1 and the sending operands are defined in rule 2.

1. The receiving operand in each implicit MOVE statement is determined by applying the rules a, b, and c in the order they appear below. Note that if a data item is not excluded as a receiver by a particular rule, it may be excluded as a receiver when a subsequent rule is applied. For example, if a data item is not excluded by rule a, that data item may still be excluded by rule b or rule c.

   a. First, the following data items are excluded as receiving operands:
      - Any identifiers that are not valid receiving operands of a MOVE statement.
      - Elementary data items that have an explicit or implicit FILLER clause if the FILLER phrase is not specified.
      - Any elementary data item subordinate to identifier-1 whose data description entry contains a REDEFINES or RENAMES clause or is subordinate to a data item whose data description entry contains a REDEFINES clause. However, identifier-1 might itself have a REDEFINES clause or be subordinate to a data item with a REDEFINES clause.

   b. Second, an elementary data item is a possible receiving item in either of the following cases:
      - It is explicitly referenced by identifier-1.
      - It is contained within the group data item referenced by identifier-1. If the elementary data item is a table element, each occurrence of the elementary data item is a possible receiving operand.

   c. Finally, each possible receiving operand is a receiving operand if at least one of the following conditions is true:
      - The VALUE phrase is specified, the category of the elementary data item is one of the categories specified or implied in the VALUE phrase, and either of the following conditions is true:
         - A data-item format VALUE clause is specified in the data description entry of the elementary data item.
         - A table format VALUE clause is specified in the data description entry of the elementary item and that VALUE clause specifies a value for the particular occurrence of the elementary data item.
      - The REPLACING phrase is specified and the category of the elementary data item is one of the categories specified in the REPLACING phrase.
      - The DEFAULT phrase is specified.
      - Neither the REPLACING phrase nor the VALUE phrase is specified.

2. The sending operand in each implicit MOVE statement is determined as follows:

   - If the data item qualifies as a receiving operand because of the VALUE phrase, the sending operand is determined by the literal in the VALUE clause specified in the data description entry of the data item. If the data item is a table element, the literal in the VALUE clause that corresponds to the occurrence being initialized determines the sending operand. The actual sending operand is a literal that, when moved to the receiving operand with a MOVE statement, produces the same result as the initial value of the data item as produced by the application of the VALUE clause.
• If the data item does not qualify as a receiving operand because of the VALUE phrase, but does qualify because of the REPLACING phrase, the sending operand is the literal-1 or identifier-2 associated with the category specified in the REPLACING phrase.
• If the data item does not qualify in accordance with the preceding two rules, the sending operand used depends on the category of the receiving operand as follows:
  – SPACE is the implied sending item for receiving items of category alphabetic, alphanumeric, alphanumeric-edited, DBCS, EGCS, national, or national-edited.
  – ZERO is the implied sending item for receiving items of category numeric or numeric-edited.

**INSPECT statement**

The INSPECT statement examines characters or groups of characters in a data item.

The INSPECT statement does the following tasks:

• Counts the occurrences of a specific character (alphanumeric, DBCS, or national) in a data item (formats 1 and 3).
• Counts the occurrences of specific characters and fills all or portions of a data item with specified characters, such as spaces or zeros (formats 2 and 3).
• Converts all occurrences of specific characters in a data item to user-supplied replacement characters (format 4).

**Format 1: INSPECT statement with TALLYING phrase**

```
INSPECT identifier-1 TALLYING identifier-2 FOR CHARACTERS phrase 1 ALL LEADING identifier-3 literal-1 phrase 1 BEFORE AFTER INITIAL identifier-4 literal-2
```
**Format 4: INSPECT statement with CONVERTING phrase**

```
INSPECT — identifier-1 — CONVERTING — identifier-6 TO — identifier-7 — literal-4
  — literal-5
```

**identifier-1**

Is the *inspected item* and can be any of the following items:

- An alphanumeric group item or a national group item
- An elementary data item described explicitly or implicitly with usage DISPLAY, DISPLAY-1, or NATIONAL. The item can have any category that is valid for the selected usage.

**identifier-3, identifier-4, identifier-5, identifier-6, identifier-7**

Must reference an elementary data item described explicitly or implicitly with usage DISPLAY, DISPLAY-1, or NATIONAL.

**literal-1, literal-2, literal-3, literal-4**

Must be of category alphanumeric, DBCS, or national.

When **identifier-1** is of usage NATIONAL, literals must be of category national.

When **identifier-1** is of usage DISPLAY-1, literals must be of category alphanumeric.

When **identifier-1** is of usage DISPLAY-1 (DBCS) literals may be the figurative constant SPACE.

When **identifier-1** is of usage DISPLAY or NATIONAL, literals can be any figurative constant that does not begin with the word ALL, as specified in “Figurative constants” on page 13. The figurative constant is treated as a one-character alphanumeric literal when **identifier-1** is of usage DISPLAY, and as a one-character national literal when **identifier-1** is of usage NATIONAL.

All identifiers (except **identifier-2**) must have the same usage as **identifier-1**. All literals must have category alphanumeric, DBCS, or national when **identifier-1** has usage DISPLAY, DISPLAY-1, or NATIONAL, respectively.

All identifiers may not be dynamic-length group or dynamic-length elementary items.

**TALLYING phrase (formats 1 and 3)**

This phrase counts the occurrences of a specific character or special character in a data item.

When **identifier-1** is a DBCS data item, DBCS characters are counted; when **identifier-1** is a data item of usage national, national characters (encoding units) are counted; otherwise, alphanumeric characters (bytes) are counted.

**identifier-2**

Is the *count field*, and must be an elementary integer item defined without the symbol P in its PICTURE character-string.

**identifier-2** cannot be of category external floating-point.

You must initialize **identifier-2** before execution of the INSPECT statement begins.
Usage note: The count field can be an integer data item defined with usage NATIONAL.

**identifier-3 or literal-1**

Is the **tallying field** (the item whose occurrences will be tallied).

**CHARACTERS**

When **CHARACTERS** is specified and neither the BEFORE nor AFTER phrase is specified, the count field (**identifier-2**) is increased by 1 for each character (including the space character) in the inspected item (**identifier-1**). Thus, execution of an INSPECT statement with the TALLYING phrase increases the value in the count field by the number of character positions in the inspected item.

**ALL**

When **ALL** is specified and neither the BEFORE nor AFTER phrase is specified, the count field (**identifier-2**) is increased by 1 for each nonoverlapping occurrence of the tallying comparand (**identifier-3** or **literal-1**) in the inspected item (**identifier-1**), beginning at the leftmost character position and continuing to the rightmost.

**LEADING**

When **LEADING** is specified and neither the BEFORE nor AFTER phrase is specified, the count field (**identifier-2**) is increased by 1 for each contiguous nonoverlapping occurrence of the tallying comparand in the inspected item (**identifier-1**), provided that the leftmost such occurrence is at the point where comparison began in the first comparison cycle for which this tallying comparand is eligible to participate.

**FIRST (format 3 only)**

When **FIRST** is specified and neither the BEFORE nor AFTER phrase is specified, the substitution field replaces the leftmost occurrence of the subject field in the inspected item (**identifier-1**).

**REPLACING phrase (formats 2 and 3)**

This phrase fills all or portions of a data item with specified characters, such as spaces or zeros.

**identifier-3 or literal-1**

Is the **subject field**, which identifies the characters to be replaced.

**identifier-5 or literal-3**

Is the **substitution field** (the item that replaces the subject field).

The subject field and the substitution field must be the same length.

**CHARACTERS BY**

When the **CHARACTERS BY** phrase is used, the substitution field must be one character position in length.

When **CHARACTERS BY** is specified and neither the BEFORE nor AFTER phrase is specified, the substitution field replaces each character in the inspected item (**identifier-1**), beginning at the leftmost character position and continuing to the rightmost.

**ALL**

When **ALL** is specified and neither the BEFORE nor AFTER phrase is specified, the substitution field replaces each nonoverlapping occurrence of the subject field in the inspected item (**identifier-1**), beginning at the leftmost character position and continuing to the rightmost.

**LEADING**

When **LEADING** is specified and neither the BEFORE nor AFTER phrase is specified, the substitution field replaces each contiguous nonoverlapping occurrence of the subject field in the inspected item (**identifier-1**), provided that the leftmost such occurrence is at the point where comparison began in the first comparison cycle for which this substitution field is eligible to participate.

**FIRST**

When **FIRST** is specified and neither the BEFORE nor AFTER phrase is specified, the substitution field replaces the leftmost occurrence of the subject field in the inspected item (**identifier-1**).
When both the TALLYING and REPLACING phrases are specified (format 3), the INSPECT statement is executed as if an INSPECT TALLYING statement (format 1) were specified, immediately followed by an INSPECT REPLACING statement (format 2).

The following replacement rules apply:

• When the subject field is a figurative constant, the one-character substitution field replaces each character in the inspected item that is equivalent to the figurative constant.

• When the substitution field is a figurative constant, the substitution field replaces each nonoverlapping occurrence of the subject field in the inspected item.

• When the subject and substitution fields are character-strings, the character-string specified in the substitution field replaces each nonoverlapping occurrence of the subject field in the inspected item.

• After replacement has occurred in a given character position in the inspected item, no further replacement for that character position is made in this execution of the INSPECT statement.

BEFORE and AFTER phrases (all formats)

This phrase narrows the set of items being tallied or replaced.

No more than one BEFORE phrase and one AFTER phrase can be specified for any one ALL, LEADING, CHARACTERS, FIRST or CONVERTING phrase.

identifier-4 or literal-2

Is the delimiter.

Delimiters are not counted or replaced.

INITIAL

The first occurrence of a specified item.

The BEFORE and AFTER phrases change how counting and replacing are done:

• When BEFORE is specified, counting or replacing of the inspected item (identifier-1) begins at the leftmost character position and continues until the first occurrence of the delimiter is encountered. If no delimiter is present in the inspected item, counting or replacing continues toward the rightmost character position.

• When AFTER is specified, counting or replacing of the inspected item (identifier-1) begins with the first character position to the right of the delimiter and continues toward the rightmost character position in the inspected item. If no delimiter is present in the inspected item, no counting or replacement takes place.

CONVERTING phrase (format 4)

This phrase converts all occurrences of a specific character or string of characters in a data item (identifier-1) to user-supplied replacement characters.

identifier-6 or literal-4

Specifies the character string to be replaced.

The same character must not appear more than once in either literal-4 or identifier-6.

identifier-7 or literal-5

Specifies the replacing character string.

The replacing character string (identifier-7 or literal-5) must be the same size as the replaced character string (identifier-6 or literal-4).

A format-4 INSPECT statement is interpreted and executed as if a format-2 INSPECT statement had been written with a series of ALL phrases (one for each character of literal-4), specifying the same identifier-1. The effect is as if each single character of literal-4 were referenced as literal-1, and the corresponding single character of literal-5 referenced as literal-3. Correspondence between the characters of literal-4 and the characters of literal-5 is by ordinal position within the data item.
If identifier-4, identifier-6, or identifier-7 occupies the same storage area as identifier-1, the result of the execution of this statement is undefined, even if they are defined by the same data description entry.

The following table describes the treatment of data items that can be used as an operand in the INSPECT statement:

<table>
<thead>
<tr>
<th>When referenced by any identifier except identifier-2, the content of each item of category ...</th>
<th>Is treated ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alphanumeric or alphabetic</td>
<td>As an alphanumeric character string</td>
</tr>
<tr>
<td>DBCS</td>
<td>As a DBCS character string</td>
</tr>
<tr>
<td>National</td>
<td>As a national character string</td>
</tr>
<tr>
<td>Alphanumeric-edited, numeric-edited with usage DISPLAY, or numeric with usage DISPLAY (unsigned, external decimal)</td>
<td>As if redefined as category alphanumeric, with the INSPECT statement referring to an alphanumeric character string</td>
</tr>
<tr>
<td>National-edited, numeric-edited with usage NATIONAL or numeric with usage NATIONAL (unsigned, external decimal)</td>
<td>As if redefined as category national, with the INSPECT statement referring to a national character string</td>
</tr>
<tr>
<td>Numeric with usage DISPLAY (signed, external decimal)</td>
<td>As if moved to an unsigned external decimal item of usage DISPLAY with the same length as the identifier and then redefined as category alphanumeric, with the INSPECT statement referring to an alphanumeric character string If the sign is a separate character, the byte containing the sign is not examined and, therefore, not replaced. If the referenced item is identifier-1, the string that results from any replacing or converting action is copied back to identifier-1.</td>
</tr>
<tr>
<td>Numeric with usage NATIONAL (signed, external decimal)</td>
<td>As if moved to an unsigned external decimal item of usage NATIONAL with the same length as the identifier and then redefined as category national, with the INSPECT statement referring to a national character string If the sign is a separate character, the byte containing the sign is not examined and, therefore, not replaced. If the referenced item is identifier-1, the string that results from any replacing or converting action is copied back to identifier-1.</td>
</tr>
<tr>
<td>External floating-point with usage DISPLAY</td>
<td>As if redefined as category alphanumeric, with the INSPECT statement referring to an alphanumeric character-string</td>
</tr>
<tr>
<td>External floating-point with usage NATIONAL</td>
<td>As if redefined as category national, with the INSPECT statement referring to a national character-string</td>
</tr>
</tbody>
</table>
Data flow

Except when the BEFORE or AFTER phrase is specified, inspection begins at the leftmost character position of the inspected item (identifier-1) and proceeds character-by-character to the rightmost position.

The comparands of the following phrases are compared in the left-to-right order in which they are specified in the INSPECT statement:

- TALLYING (literal-1 or identifier-3, ... )
- REPLACING (literal-3 or identifier-5, ... )

If any identifier is subscripted or reference modified, or is a function-identifier, the subscript, reference-modifier, or function is evaluated only once as the first operation in the execution of the INSPECT statement.

For examples of TALLYING and REPLACING, see Tallying and replacing data items (INSPECT) in the Enterprise COBOL Programming Guide.

Comparison cycle

The comparison cycle consists of the actions as described in this topic.

1. The first comparand is compared with an equal number of leftmost contiguous character positions in the inspected item. The comparand matches the inspected characters only if both are equal, character-for-character.

   If the CHARACTERS phrase is specified, an implied one-character comparand is used. The implied character is always considered to match the inspected character in the inspected item.

2. If no match occurs for the first comparand and there are more comparands, the comparison is repeated for each successive comparand until either a match is found or all comparands have been acted upon.

3. Depending on whether a match is found, these actions are taken:
   - If a match is found, tallying or replacing takes place as described in the TALLYING and REPLACING phrase descriptions.
     
     If there are more character positions in the inspected item, the first character position following the rightmost matching character is now considered to be in the leftmost character position. The process described in actions 1 and 2 is then repeated.
   - If no match is found and there are more character positions in the inspected item, the first character position following the leftmost inspected character is now considered to be in the leftmost character position. The process described in actions 1 and 2 is then repeated.

4. Actions 1 through 3 are repeated until the rightmost character position in the inspected item either has been matched or has been considered as being in the leftmost character position.

When the BEFORE or AFTER phrase is specified, the comparison cycle is modified, as described in “BEFORE and AFTER phrases (all formats)” on page 331.
Example of the INSPECT statement

The topic shows an example of INSPECT statement results.

```
INSPECT ID-1 TALLYING ID-2 FOR ALL '***' REPLACING ALL '***' BY ZEROS.

ID-1 before execution: * * * 0 * *

Execution for TALLYING phrase:
1st comparison: * = *  (true)  1
2nd comparison: * 0  = *  (false)  1
3rd comparison: 0 *  = *  (false)  1
4th comparison: **  = *  (true)  2

Execution for REPLACING phrase:
5th comparison: **  = *  (true)  ID-1 changed to
6th comparison: * 0  = *  (false)  ID-1 unchanged
7th comparison: 0 *  = *  (false)  ID-1 unchanged
8th comparison: **  = *  (true)  ID-1 changed to

At the end of inspection:
ID-1 contains: 0 0 * 0 0 0
ID-2 contains: 2
```
The INVOKE statement can create object instances of a COBOL or Java class and can invoke a method defined in a COBOL or Java class.

**Format**

```
INVOKE identifier-1 class-name-1 SELF SUPER
    USING BY VALUE LENGTH OF identifier-3 literal-2
    RETURNING identifier-4
    ON EXCEPTION imperative-statement-1
    NOT ON EXCEPTION imperative-statement-2
END-INVOKE
```

**identifier-1**

Must be defined as USAGE OBJECT REFERENCE. The contents of `identifier-1` specify the object on which a method is invoked.

When `identifier-1` is specified, either `literal-1` or `identifier-2` must be specified, identifying the name of the method to be invoked.

It must not be a dynamic-length elementary item or a dynamic-length group item.

The results of the INVOKE statement are undefined if either:
- `identifier-1` does not contain a valid reference to an object.
- `identifier-1` contains NULL.

**class-name-1**

When `class-name-1` is specified together with `literal-1` or `identifier-2`, the INVOKE statement invokes a static or factory method of the class referenced by `class-name-1`. `literal-1` or `identifier-2` specifies the name of the method that is to be invoked. The method must be a static method if `class-name-1` is a Java class; the method must be a factory method if `class-name-1` is a COBOL class.
When class-name-1 is specified together with NEW, the INVOKE statement creates a new object that is an instance of class class-name-1.

You must specify class-name-1 in the REPOSITORY paragraph of the configuration section of the class or program that contains the INVOKE statement.

**SELF**

An implicit reference to the object used to invoke the currently executing method. When SELF is specified, the INVOKE statement must appear within the PROCEDURE DIVISION of a method.

**SUPER**

An implicit reference to the object that was used to invoke the currently executing method. The resolution of the method to be invoked will ignore any methods declared in the class definition of the currently executing method and methods defined in any class derived from that class; thus the method invoked will be one that is inherited from an ancestor class.

**literal-1**

The value of literal-1 is the name of the method to be invoked. The referenced object must support the method identified by literal-1.

Literal-1 must be an alphanumeric literal or a national literal.

Literal-1 is interpreted in a case-sensitive manner. The method name, the number of arguments, and the data types of the arguments in the USING phrase of the INVOKE statement are used to select the method with matching signature that is supported by the object. The method can be overloaded.

**identifier-2**

A data item of category alphabetic, alphanumeric, or national that at run time contains the name of the method to be invoked. The referenced object must support the method identified by identifier-2.

If identifier-2 is specified, identifier-1 must be defined as USAGE OBJECT REFERENCE without any optional phrases; that is, identifier-1 must be a universal object reference.

It must not be a dynamic-length elementary item or a dynamic-length group item.

The content of identifier-2 is interpreted in a case-sensitive manner. The method name, the number of arguments, and the data types of the arguments in the USING phrase of the INVOKE statement are used to select the method with matching signature that is supported by the object. The method can be overloaded.

**NEW**

The NEW operand specifies that the INVOKE statement is to create a new object instance of the class class-name-1. class-name-1 must be specified.

When class-name-1 is implemented in Java, the USING phrase of the INVOKE statement can be specified. The number of arguments and the data types of the arguments in the USING phrase of the INVOKE statement are used to select the Java constructor with matching signature that is supported by the class. An object instance of class class-name-1 is allocated, the selected constructor (or the default constructor) is executed, and a reference to the created object is returned.

When class-name-1 is implemented in COBOL, the USING phrase of the INVOKE statement must not be specified. An object instance of class class-name-1 is allocated, instance data items are initialized to the values specified in associated VALUE clauses, and a reference to the created object is returned.

When NEW is specified, you must also specify a RETURNING phrase as described in “RETURNING phrase” on page 337.

**USING phrase**

The USING phrase specifies arguments that are passed to the target method. The argument data types and argument linkage conventions are restricted to those supported by Java. See “BY VALUE phrase” on page 336 for details.

**BY VALUE phrase**
Arguments specified in an INVOKE statement must be passed BY VALUE.

The BY VALUE phrase specifies that the value of the argument is passed, not a reference to the sending data item. The invoked method can modify the formal parameter that corresponds to an argument passed by value, but changes do not affect the argument because the invoked method has access only to a temporary copy of the sending data item.

**identifier-3**

Must be an elementary data item in the DATA DIVISION. The data type of **identifier-3** must be one of the types supported for Java interoperation, as listed in “Interoperable data types for COBOL and Java” on page 339. Miscellaneous cases that are also supported as **identifier-3** are listed in “Miscellaneous argument types for COBOL and Java” on page 341, with their corresponding Java type.

See Conformance requirements for arguments for additional requirements that apply to **identifier-3**.

It must not be a dynamic-length elementary item or a dynamic-length group item.

**literal-2**

Must be of a type suitable for Java interoperation and must exactly match the type of the corresponding parameter in the target method. Supported literal forms are listed in “Miscellaneous argument types for COBOL and Java” on page 341, with their corresponding Java type.

**literal-2** must not be a DBCS literal.

**LENGTH OF identifier-3**

Specifies that the length of **identifier-3** is passed as an argument in the LENGTH OF special register. A LENGTH OF special register passed BY VALUE is treated as a PIC 9(9) binary value. For information about the LENGTH OF special register, see “LENGTH OF” on page 19.

**Conformance requirements for arguments**

When **identifier-3** is an object reference, certain rules apply.

The rules are:

- A class-name must be specified in the data description entry for that object reference. That is, **identifier-3** must not be a universal object reference.
- The specified class-name must reference a class that is exactly the class of the corresponding parameter in the invoked method. That is, the class of **identifier-3** must not be a subclass or a superclass of the corresponding parameter's class.

When **identifier-3** is not an object reference, the following rules apply:

- If the target method is implemented in COBOL, the description of **identifier-3** must exactly match the description of the corresponding formal parameter in the target method.
- If the target method is implemented in Java, the description of **identifier-3** must correspond to the Java type of the formal parameter in the target method, as specified in “Interoperable data types for COBOL and Java” on page 339.

**Usage note:** Adherence to conformance requirements for arguments is the responsibility of the programmer. Conformance requirements are not verified by the compiler.

**RETURNING phrase**

The RETURNING phrase specifies a data item that will contain the value returned from the invoked method. You can specify the RETURNING phrase on the INVOKE statement when invoking methods that are written in COBOL or Java.

**identifier-4**

The RETURNING data item. **identifier-4**:

- Must be defined in the DATA DIVISION
Must not be reference-modified
Is not changed if an EXCEPTION occurs

The data type of identifier-4 must be one of the types supported for Java interoperation, as listed in "Interoperable data types for COBOL and Java" on page 339.

See Conformance requirements for the RETURNING item for additional requirements that apply to identifier-4.

It must not be a dynamic-length elementary item or a dynamic-length group item.

If identifier-4 is specified and the target method is written in COBOL, the target method must have a RETURNING phrase in its PROCEDURE DIVISION header. When the target method returns, its return value is assigned to identifier-4 using the rules for the SET statement if identifier-4 is described with USAGE OBJECT REFERENCE; otherwise, the rules for the MOVE statement are used.

The RETURNING data item is an output-only parameter. On entry to the called method, the initial state of the PROCEDURE DIVISION RETURNING data item has an undefined and unpredictable value. You must initialize the PROCEDURE DIVISION RETURNING data item in the invoked method before you reference its value. The value that is passed back to the invoker is the final value of the PROCEDURE DIVISION RETURNING data item when the invoked method returns.

See Managing local and global references in the Enterprise COBOL Programming Guide for discussion of local and global object references as defined in Java. These attributes affect the life-time of object references.

Usage note: The RETURN-CODE special register is not set by execution of INVOKE statements.

Conformance requirements for the RETURNING item

For INVOKE statements that specify class-name-1 NEW, the RETURNING phrase is required.

The returning item must be one of the following ones:

- A universal object reference
- An object reference specifying class-name-1
- An object reference specifying a superclass of class-name-1

For INVOKE statements without the NEW phrase, the RETURNING item specified in the method invocation and in the corresponding target method must satisfy the following requirements:

- The presence or absence of a return value must be the same on the INVOKE statement and in the target method.
- If the RETURNING item is not an object reference, the following rules apply:
  - If the target method is implemented in COBOL, the returning item in the INVOKE statement and the RETURNING item in the target method must have an identical data description entry.
  - If the target method is implemented in Java, the returning item in the INVOKE statement must correspond to the Java type of the method result, as described in “Interoperable data types for COBOL and Java” on page 339.
- If the RETURNING item is an object reference, the RETURNING item specified in the INVOKE statement must be an object reference typed exactly to the class of the returning item specified in the target method. That is, the class of identifier-4 must not be a subclass or a superclass of the class of the returning item in the target method.

Usage note: Adherence to conformance requirements for returning items is the responsibility of the programmer. Conformance requirements are not verified by the compiler.

ON EXCEPTION phrase

An exception condition occurs when the identified object or class does not support a method with a signature that matches the signature of the method specified in the INVOKE statement. When an exception condition occurs, one of the following actions occurs:
• If the ON EXCEPTION phrase is specified, control is transferred to imperative-statement-1.
• If the ON EXCEPTION phrase is not specified, a severity-3 Language Environment condition is raised at run time.

**NOT ON EXCEPTION phrase**

If an exception condition does not occur (that is, the identified method is supported by the specified object), control is transferred to the invoked method. After control is returned from the invoked method, control is then transferred:

1. To imperative-statement-2, if the NOT ON EXCEPTION phrase is specified.
2. To the end of the INVOKE statement if the NOT ON EXCEPTION phrase is not specified.

**END-INVOKE phrase**

This explicit scope terminator serves to delimit the scope of the INVOKE statement. An INVOKE statement that is terminated by END-INVOKE, along with its contained statements, becomes a unit that is treated as though it were an imperative statement. It can be specified as an imperative statement in a conditional statement; for example, in the exception phrase of another statement.

**Interoperable data types for COBOL and Java**

A subset of COBOL data types can be used for interoper操作 between COBOL and Java.

You can specify the interoperable data types as arguments in COBOL INVOKE statements and as the RETURNING item in COBOL INVOKE statements. Similarly, you can pass these types as arguments from a Java method invocation expression and receive them as parameters in the USING phrase or as the RETURNING item in the PROCEDURE DIVISION header of a COBOL method.

The following table lists the primitive Java types and the COBOL data types that are supported for interoper操作 and the correspondence between them.

<table>
<thead>
<tr>
<th>Java data type</th>
<th>COBOL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean¹</td>
<td>Conditional variable and two condition-names of the form:</td>
</tr>
<tr>
<td></td>
<td>level-number</td>
</tr>
<tr>
<td></td>
<td>88</td>
</tr>
<tr>
<td></td>
<td>88</td>
</tr>
<tr>
<td>byte</td>
<td>Single-byte alphanumeric, PIC X or PIC A</td>
</tr>
<tr>
<td>short</td>
<td>USAGE BINARY, COMP, COMP-4, or COMP-5, with a PICTURE clause of the form S9(n), where 1 &lt;= n &lt;= 4</td>
</tr>
<tr>
<td>int</td>
<td>USAGE BINARY, COMP, COMP-4, or COMP-5, with a PICTURE clause of the form S9(n), where 5 &lt;= n &lt;= 9</td>
</tr>
<tr>
<td>long</td>
<td>USAGE BINARY, COMP, COMP-4, or COMP-5, with a PICTURE clause of the form S9(n), where 10 &lt;= n &lt;= 18</td>
</tr>
<tr>
<td>float²</td>
<td>USAGE COMP-1</td>
</tr>
<tr>
<td>double²</td>
<td>USAGE COMP-2</td>
</tr>
<tr>
<td>char</td>
<td>Single-character national: PIC N USAGE NATIONAL (an elementary data item of category national)</td>
</tr>
<tr>
<td>class types</td>
<td>USAGE OBJECT REFERENCE class-name</td>
</tr>
</tbody>
</table>

¹ Conditional variable and two condition-names of the form: level-number data-name.
² The following table lists the primitive Java types and the COBOL data types that are supported for interoper operation and the correspondence between them.
Table 40. Interoperable Java and COBOL data types (continued)

<table>
<thead>
<tr>
<th>Java data type</th>
<th>COBOL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean[]</td>
<td>object reference jbooleleanArray</td>
</tr>
<tr>
<td>byte[]</td>
<td>object reference jbyteArray</td>
</tr>
<tr>
<td>short[]</td>
<td>object reference jshortArray</td>
</tr>
<tr>
<td>int[]</td>
<td>object reference jintArray</td>
</tr>
<tr>
<td>long[]</td>
<td>object reference jlongArray</td>
</tr>
<tr>
<td>char[]</td>
<td>object reference jcharArray</td>
</tr>
<tr>
<td>Object[]</td>
<td>object reference jobjectArray</td>
</tr>
<tr>
<td>String</td>
<td>object reference jstring</td>
</tr>
</tbody>
</table>

The following Java array types are not currently supported:

<table>
<thead>
<tr>
<th>Java data type</th>
<th>COBOL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>float[]</td>
<td>object reference jfloatArray</td>
</tr>
<tr>
<td>double[]</td>
<td>object reference jdoubleArray</td>
</tr>
</tbody>
</table>

In addition to the primitive types, Java Strings and arrays of Java primitive types can interoperate with COBOL. This requires specialized mechanisms provided by the COBOL runtime system and the Java Native Interface (JNI).

In a Java program, to pass array data to COBOL or to receive array data from COBOL, you declare the array types using the usual Java syntax. In the COBOL program, you declare the array as an object reference that contains an instance of one of the special classes provided for array support. Conversion between the Java and COBOL types is automatic at the time of method invocation.

In a Java program, to pass String data to COBOL or to receive String data from COBOL, you declare the array types using the usual Java syntax. In the COBOL program, you declare the String as an object reference that contains an instance of the special jstring class. Conversion between the Java and COBOL types is automatic at the time of method invocation. The following table lists the Java array and String data types and the corresponding special COBOL data types.

Table 41. Interoperable COBOL and Java array and String data types

<table>
<thead>
<tr>
<th>Java data type</th>
<th>COBOL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean[]</td>
<td>object reference jbooleleanArray</td>
</tr>
<tr>
<td>byte[]</td>
<td>object reference jbyteArray</td>
</tr>
<tr>
<td>short[]</td>
<td>object reference jshortArray</td>
</tr>
<tr>
<td>int[]</td>
<td>object reference jintArray</td>
</tr>
<tr>
<td>long[]</td>
<td>object reference jlongArray</td>
</tr>
<tr>
<td>char[]</td>
<td>object reference jcharArray</td>
</tr>
<tr>
<td>Object[]</td>
<td>object reference jobjectArray</td>
</tr>
<tr>
<td>String</td>
<td>object reference jstring</td>
</tr>
</tbody>
</table>

You must code an entry in the repository paragraph for each special class that you want to use, just as you do for other classes. For example, to use jstring, code the following entry:

Configuration Section.
Repository.
Class jstring is "jstring".
Alternatively, for the String type, the COBOL repository entry can specify an external class name of java.lang.String:

<table>
<thead>
<tr>
<th>Repository.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class jstring is &quot;java.lang.String&quot;.</td>
</tr>
</tbody>
</table>

Callable services are provided by the Java Native Interface (JNI) for manipulating the COBOL objects of these types in COBOL. For example, callable services can be used to set COBOL alphanumeric or national data into a jstring object or to extract data from a jstring object. For details on use of JNI callable services for these and other purposes, see Accessing JNI services in the Enterprise COBOL Programming Guide.

For details on repository entries for class definitions, see “REPOSITORY paragraph” on page 116. For examples, see Example: external class-names and Java packages in the Enterprise COBOL Programming Guide.

**Miscellaneous argument types for COBOL and Java**

There are miscellaneous cases of COBOL items that can be used as arguments in an INVOKE statement. The COBOL miscellaneous argument types and the corresponding Java types are listed in the following table.

<table>
<thead>
<tr>
<th>Table 42. COBOL miscellaneous argument types and corresponding Java types</th>
</tr>
</thead>
<tbody>
<tr>
<td>COBOL argument</td>
</tr>
<tr>
<td>Reference-modified item of usage display with length one</td>
</tr>
<tr>
<td>Reference-modified item of usage national with length one (either an elementary data item of usage national or a national group item)</td>
</tr>
<tr>
<td>SHIFT-IN and SHIFT-OUT special registers</td>
</tr>
<tr>
<td>LINAGE-COUNTER special register when its usage is binary</td>
</tr>
<tr>
<td>LENGTH OF special register</td>
</tr>
</tbody>
</table>

The following table lists COBOL literal types that can be used as arguments in an INVOKE statement, with the corresponding Java type.

<table>
<thead>
<tr>
<th>Table 43. COBOL literal argument types and corresponding Java types</th>
</tr>
</thead>
<tbody>
<tr>
<td>COBOL literal argument</td>
</tr>
<tr>
<td>Fixed-point numeric literal with no decimal positions and with nine digits or less</td>
</tr>
<tr>
<td>Floating-point numeric literal</td>
</tr>
<tr>
<td>Figurative constant ZERO</td>
</tr>
<tr>
<td>One-character alphanumeric literal</td>
</tr>
<tr>
<td>One-character national literal</td>
</tr>
<tr>
<td>Symbolic character</td>
</tr>
<tr>
<td>Figurative constants SPACE, QUOTE, HIGH-VALUE, or LOW-VALUE</td>
</tr>
</tbody>
</table>
**JSON GENERATE statement**

The JSON GENERATE statement converts data to JSON format.

**Format**

```
JSON GENERATE — identifier-1 — FROM — identifier-2 —

COUNT IN identifier-3

NAME OF identifier-4 IS literal-1 OMITTED

SUPPRESS identifier-5

CONVERTING phrase-1 ALSO phrase-1

ON EXCEPTION imperative-statement-1

NOT ON EXCEPTION imperative-statement-2

END-JSON
```

**phrase-1 Format**

```
identifier-6 TO JSON BOOLEAN USING condition-name-1 literal-2
```

**identifier-1**

The receiving area for the generated JSON text. `identifier-1` must reference one of the following items:

- An elementary data item of category alphanumeric
• An alphanumeric group item
• An elementary data item of category national
• A national group item

When identifier-1 references a national group item, identifier-1 is processed as an elementary data item of category national. When identifier-1 references an alphanumeric group item, identifier-1 is treated as though it were an elementary data item of category alphanumeric.

identifier-1 must not be defined with the JUSTIFIED clause, and cannot be a function identifier. identifier-1 can be subscripted or reference modified.

identifier-1 must not overlap identifier-2 or identifier-3.

identifier-1 must not be a dynamic-length group item or a dynamic-length elementary item.

The generated JSON text is encoded in UTF-8 (CCSID 1208) unless identifier-1 is of category national, in which case it is encoded in UTF-16 big-Endian (CCSID 1200). Conversion of the data values and NAME phrase literals is done according to the compiler CODEPAGE option in effect for the compilation. Conversion of original data names is always done using CCSID 1140.

identifier-1 must be large enough to contain the generated JSON text. Typically, it should be from 2 to 3 times the size of identifier-2, depending on the lengths of the data-names within identifier-2. If identifier-1 is not large enough, an exception condition exists at the end of the JSON GENERATE statement. If the COUNT phrase is specified, identifier-3 contains the number of character encoding units that were actually generated.

identifier-2

The group or elementary data item to be converted to JSON format.

identifier-2 cannot be a function identifier or be reference modified, but it can be subscripted.

identifier-2 must not overlap identifier-1 or identifier-3.

identifier-2 might not be a dynamic-length group item or a dynamic-length elementary item.

The data description entry for identifier-2 must not contain a RENAMES clause.

The following data items that are specified by identifier-2 are ignored by the JSON GENERATE statement:

• Any subordinate unnamed elementary data items or elementary FILLER data items
• Any slack bytes inserted for SYNCHRONIZED items
• Any data item subordinate to identifier-2 that is defined with the REDEFINES clause or that is subordinate to such a redefining item
• Any data item subordinate to identifier-2 that is defined with the RENAMES clause
• Any group data item whose subordinate data items are all ignored

All data items specified by identifier-2 that are not ignored according to the previous rules must satisfy the following conditions:

• Each elementary data item must have a USAGE other than POINTER, FUNCTION-POINTER, PROCEDURE-POINTER, or OBJECT REFERENCE.
• There must be at least one such elementary data item.
• Each non-FILLER data-name must have a unique identifier within identifier-2.

For example, consider the following data declaration:

```
01 STRUCT.
  02 STAT PIC X(4).
  02 IN-AREA PIC X(100).
  02 OK-AREA REDEFINES IN-AREA.
  03 FLAGS PIC X.
  03 PIC X(3).
  03 COUNTER USAGE COMP-5 PIC S9(9).
```
The following data items from the example can be specified as identifier-2:

- **STRUCT**, of which subordinate data items STAT and IN-AREA would be converted to JSON format. (OK-AREA, NG-AREA1, and NG-AREA2 are ignored because they specify the REDEFINES clause.)

- **OK-AREA**, of which subordinate data items FLAGS, COUNTER, and UNREFERENCED would be converted. (The item whose data description entry specifies 03 PIC X(3) is ignored because it is an elementary FILLER data item. ASFNPTR is ignored because it specifies the REDEFINES clause.)

- Any of the elementary data items that are subordinate to STRUCT except:
  - ASFNPTR or PTR (disallowed usage)
  - UNREFERENCED OF NG-AREA2 (nonunique names for data items that are otherwise eligible)
  - Any FILLER data items

The following data items cannot be specified as identifier-2:

- **NG-AREA1**, because subordinate data item PTR specifies USAGE POINTER but does not specify the REDEFINES clause. (PTR would be ignored if it is defined with the REDEFINES clause.)

- **NG-AREA2**, because subordinate elementary data items have the nonunique name UNREFERENCED.

**COUNT phrase**

If the COUNT phrase is specified, identifier-3 contains (after successful execution of the JSON GENERATE statement) the count of generated JSON character encoding units. If identifier-1 (the receiver) is of category national, the count is in double-bytes. Otherwise, the count is in bytes.

**identifier-3**

The data count field. Must be an integer data item defined without the symbol P in its picture string.

**literal-1** must be an alphanumeric or national literal containing the name to be generated in the JSON text corresponding to identifier-4. With PTF for APAR PH18641 installed, alternatively, you can specify OMITTED to generate an anonymous JSON object, whose top-level parent name is not generated. When you specify OMITTED, identifier-4 must reference identifier-2.

**NAME phrase**

Allows you to override the default JSON names derived from identifier-2 or its subordinate data items.

**identifier-4** must reference identifier-2 or one of its subordinate data items. It cannot be a function identifier and cannot be reference modified or subscripted. It must not specify any data item which is ignored by the JSON GENERATE statement. For more information about identifier-2, see the description of identifier-2. If identifier-4 is specified more than once in the NAME phrase, the last specification is used.

**literal-1** must be an alphanumeric or national literal containing the name to be generated in the JSON text corresponding to identifier-4. With PTF for APAR PH18641 installed, alternatively, you can specify OMITTED to generate an anonymous JSON object, whose top-level parent name is not generated. When you specify OMITTED, identifier-4 must reference identifier-2.

**SUPPRESS phrase**

Allows you to identify and unconditionally exclude items that are subordinate to identifier-2, and thus selectively generate output for the JSON GENERATE statement. If the SUPPRESS phrase is specified, identifier-1 must be large enough to contain the generated JSON document before any suppression.

**identifier-5** must reference a data item that is subordinate to identifier-2 and that is not otherwise ignored by the operation of the JSON GENERATE statement. identifier-5 cannot be a function identifier and cannot be reference modified or subscripted. If identifier-5 specifies a group data item,
that group data item and all data items that are subordinate to the group item are excluded. Duplicate
specifications of identifier-5 are permitted.

When the SUPPRESS phrase is specified, a group item subordinate to identifier-2 is excluded from the
generated JSON text if all the eligible items subordinate to the group item are excluded. The
outermost object that corresponds to identifier-2 itself is always generated, even if all the items
subordinate to identifier-2 are excluded. In this case, the value generated for identifier-2 is an empty
object, as follows:

```
{"identifier-2":{}}
```

For example, consider the following data declaration:

```
1 a.
  2 b.
  3 c occurs 0 to 2 depending j.
  4 d pic x.
  2 e pic x.
```

If the ODO object j contains the value 2 and group a is populated with all ‘_’, the statement JSON
GENERATE x FROM a (without the SUPPRESS phrase) produces the following JSON text:

```
{"a":{"b":{"c":[{"d":"_"},{"d":"_"}]},"e":"_"}}
```

Group item b is eliminated from the output if a SUPPRESS phrase specifies any one of data items b, c
or d, resulting in the following JSON text:

```
{"a":{"e":"_"}}
```

As an example of complete recursive suppression, the statement JSON GENERATE x FROM a
SUPPRESS b e produces:

```
{"a":{}}
```

JSON has an explicit representation of a table with no elements:

```
{"table-name":[]}
```

which is thus retained in the generated JSON text unless explicitly suppressed.

For example if ODO object j contains the value 0 and thus table d has no occurrences, and group a is
populated with all ‘_’, the statement JSON GENERATE x FROM a produces the following JSON text:

```
{"a":{"b":{"c":[]},"e":"_"}}
```

Components of a zero-occurrence group table do not contribute any JSON text to the output. As a
result, a SUPPRESS phrase that specified only d would have no effect on this generated output.

Suppressing data items b or c, and e, which do contribute JSON text, has the same result as for non-
zero occurrences of table c, illustrated above.

CONVERTING phrase

Allows you to specify items that will be generated as JSON BOOLEAN name/value pairs.

**identifier-6** must be a single-byte alphanumeric elementary data item whose data definition entry
contains PICTURE X.

**condition-name-1** and **literal-2** represent values of **identifier-6** that will be generated as a JSON
BOOLEAN true value. All other values of **identifier-6** will be generated as a JSON BOOLEAN false
value. **condition-name-1** must be a level-88 item directly subordinate to **identifier-6** and can be
specified with multiple values or value ranges. **literal-2** must be a single-byte alphanumeric literal.

The CONVERTING phrase can be specified with multiple items to be generated as JSON BOOLEAN
name/value pairs by using the ALSO keyword.
For example, consider the following COBOL structure and statements:

```
01 myrecord.
  02 data-a PIC X.
  02 data-b PIC X.
    88 data-b-flag VALUE 'a' THRU 'z'.
MOVE 'F' TO data-a
MOVE 'b' TO data-b

JSON GENERATE docx FROM myrecord
  CONVERTING data-a TO BOOLEAN USING 'T'
  ALSO data-b TO BOOLEAN USING data-b-flag
```

The output of the JSON GENERATE statement would be the UTF-8 encoded text in `docx` containing:

```
{ "myrecord" : { "data-a" : false, "data-b" : true } }
```

**ON EXCEPTION phrase**

An exception condition exists when an error occurs during generation of the JSON document, for example if `identifier-1` is not large enough to contain the generated JSON document. In this case, JSON generation stops and the content of the receiver, `identifier-1`, is undefined. If the COUNT phrase is specified, `identifier-3` contains the number of character positions that were actually generated.

If the ON EXCEPTION phrase is specified, control is transferred to `imperative-statement-1`. If the ON EXCEPTION phrase is not specified, the NOT ON EXCEPTION phrase, if any, is ignored, and control is transferred to the end of the JSON GENERATE statement. Special register JSON-CODE contains an exception code, as detailed in JSON GENERATE exceptions in the Enterprise COBOL Programming Guide.

**NOT ON EXCEPTION phrase**

If an exception condition does not occur during generation of the JSON document, control is passed to `imperative-statement-2`, if specified, otherwise to the end of the JSON GENERATE statement. The ON EXCEPTION phrase, if specified, is ignored. Special register JSON-CODE contains zero after execution of the JSON GENERATE statement.

**END-JSON phrase**

This explicit scope terminator delimits the scope of the JSON GENERATE or JSON PARSE statements. END-JSON permits a conditional JSON GENERATE or JSON PARSE statement (that is, a JSON GENERATE or JSON PARSE statement that specifies the ON EXCEPTION or NOT ON EXCEPTION phrase) to be nested in another conditional statement.

The scope of a conditional JSON GENERATE or JSON PARSE statement can be terminated by:

- An END-JSON phrase at the same level of nesting
- A separator period

END-JSON can also be used with a JSON GENERATE or JSON PARSE statement that does not specify either the ON EXCEPTION or the NOT ON EXCEPTION phrase.

For more information about explicit scope terminators, see Delimited scope statements on page 270.

**Nested JSON GENERATE or JSON PARSE statements**

When a given JSON GENERATE or JSON PARSE statement appears in imperative-statement-1 or imperative-statement-2 of another JSON GENERATE or JSON PARSE statement, that given JSON GENERATE or JSON PARSE statement is a nested JSON GENERATE or JSON PARSE statement.

Nested JSON GENERATE or JSON PARSE statements are considered to be matched JSON GENERATE and END-JSON combinations, or JSON PARSE and END-JSON combinations, proceeding from left to right. Thus, any END-JSON phrase that is encountered is matched with the nearest preceding JSON GENERATE or JSON PARSE statement that has not been implicitly or explicitly terminated.
**Operation of JSON GENERATE**

The content of each eligible elementary data item within *identifier-2* that has not been excluded from JSON generation according to a SUPPRESS phrase, is converted to character format in the resulting JSON text.

Only the first definition of each storage area is processed. Redefinitions of data items are not included. Data items that are effectively defined by the RENAMES clause are also not included.

A table item that has zero element occurrences is included in the JSON text as an empty array, such as `{"name": [ ]}`.

**Note:** if the zero-occurrence table is a group item, its subordinate items do not appear in the JSON text and thus specifying them in the SUPPRESS phrase has no effect. See the description of the SUPPRESS phrase for more information.

For information about the format conversion of elementary data, see “Format conversion of elementary data” on page 347 and “Trimming of generated JSON data” on page 349.

The JSON names are obtained from the NAME phrase if specified; otherwise by default they are derived from the data-names within *identifier-2* as described in “JSON name formation” on page 349. The names of group items that contain the selected elementary items are retained as the names of parent JSON objects.

No extra white space (new lines, indentation, and so forth) is inserted to make the generated JSON text more readable.

If the receiving area specified by *identifier-1* is not large enough to contain the resulting JSON text, an exception condition exists. See the description of the ON EXCEPTION phrase above for details.

If *identifier-1* is longer than the generated JSON text, only that part of *identifier-1* in which JSON is generated is changed. The rest of *identifier-1* contains the data that was present before this execution of the JSON GENERATE statement.

To avoid referring to that data, either initialize *identifier-1* to spaces before the JSON GENERATE statement or specify the COUNT phrase.

If the COUNT phrase is specified, *identifier-3* contains (after successful execution of the JSON GENERATE statement) the total number of character positions (UTF-16 encoding units or bytes) that were generated. You can use *identifier-3* as a reference modification length field to refer to the part of *identifier-2* that contains the generated JSON text.

After execution of the JSON GENERATE statement, special register JSON-CODE contains either zero, which indicates successful completion, or a nonzero exception code. For details, see JSON GENERATE exceptions in the Enterprise COBOL Programming Guide.

The JSON PARSE statement also uses special register JSON-CODE. Therefore if you code a JSON GENERATE statement in the processing procedure of a JSON PARSE statement, save the value of JSON-CODE before that JSON GENERATE statement executes and restore the saved value after the JSON GENERATE statement terminates.

A byte order mark is not generated for JSON texts.

**Format conversion of elementary data**

Elementary data items within *identifier-2* are converted in the sequence of the following steps. Some of these steps are optional.

**Conversion to character format:**

Elementary data items are converted to character format depending on the type of the data item:

- Data items of category alphabetic, alphanumeric, alphanumeric-edited, DBCS, external floating-point, national, national-edited, and numeric-edited are not converted, except as required to the correct Unicode encoding form.
• Fixed-point numeric data items other than COMPUTATIONAL-5 (COMP-5) binary data items or binary data items compiled with the TRUNC(BIN) compiler option are converted as if they were moved to a numeric-edited item that has:
  – As many integer positions as the numeric item has, but with at least one integer position, possibly zero (0)
  – An explicit decimal point, if the numeric item has at least one decimal position; the decimal point is represented by a period regardless of whether the DECIMAL-POINT IS COMMA clause is specified in the SPECIAL-NAMES paragraph
  – The same number of decimal positions as the numeric item has
  – A leading '-' picture symbol if the data item is signed (has an S in its PICTURE clause)

• COMPUTATIONAL-5 (COMP-5) binary data items or binary data items compiled with the TRUNC(BIN) compiler option are converted in the same way as the other fixed-point numeric items, except for the number of integer positions. The number of integer positions is computed depending on the number of '9' symbols in the picture character string as follows:
  – 5 minus the number of decimal places, if the data item has 1 to 4 '9' picture symbols
  – 10 minus the number of decimal places, if the data item has 5 to 9 '9' picture symbols
  – 20 minus the number of decimal places, if the data item has 10 to 18 '9' picture symbols

• Internal floating-point data items are converted as if they were moved to a data item as follows:
  – For COMP-1: an external floating-point data item with PICTURE -9.9(8)E+99
  – For COMP-2: an external floating-point data item with PICTURE -9.9(17)E+99 (illegal because of the number of digit positions)

• External floating-point data items are converted as if they were moved to another external floating-point data item with the same precision and scale, and with:
  – A - sign for the mantissa.
  – An actual decimal point, indicated by a . PICTURE symbol.
  – A + sign for the exponent.

  For example, an external floating-point item defined with PICTURE -9(3)V9(5)E-99 would be converted as if it were moved to an external floating-point item defined with PICTURE -9(3).9(5)E+99.

• Index data items are converted as if they were declared USAGE COMP-5 PICTURE S9(9).

Trimming:
After any conversion to character format, leading and trailing spaces and leading zeroes are eliminated, as described under “Trimming of generated JSON data” on page 349.

Conversion to the target encoding:
If identifier-1 is a data item of category national, any nonnational values are converted to national format. Otherwise, all values are represented in UTF-8. The conversion is done according to the compiler CODEPAGE option in effect for the compilation.

Escaping characters:
The characters NX'0022' (") and NX'005C' (\) are escaped as \" and \\, respectively.

For compactness and appearance, other common characters are represented by a two-character escape sequence as follows:

\b - backspace
\t - tab
\n - line feed
\f - form feed
\r - carriage return
\x - next line
The remaining characters in the range NX'0000' through NX'001F' are escaped as \\uhhhh, where "h" represents a hexadecimal digit 0 through F.

Representation of out-of-range Unicode characters:

Any remaining Unicode character that has a Unicode scalar value greater than NX'FFFF' is represented by a surrogate pair for national output, or a four-byte sequence for UTF-8 output. For example, the musical symbol G clef (U+1D11E) is represented in UTF-16 by the surrogate pair NX'D834' NX'DD1E', and in UTF-8 by the byte sequence x'F09D849E'.

Trimming of generated JSON data

Trimming is performed on data values after any conversion to character format.

For more information about the conversion, see “Format conversion of elementary data” on page 347.

For values converted from signed numeric values, the leading space is removed if the value is positive.

For values converted from numeric items, leading zeroes (after any initial minus sign) up to but not including the digit immediately before the actual or implied decimal point are eliminated. Trailing zeroes after a decimal point are retained. For example:

- -012.340 becomes -12.340
- 0000.45 becomes 0.45
- 0013 becomes 13
- 0000 becomes 0

Character values from data items of class alphabetic, alphanumeric, DBCS, and national have either trailing or leading spaces removed, depending on whether the corresponding data items have left (default) or right justification, respectively. That is, trailing spaces are removed from values whose corresponding data items do not specify the JUSTIFIED clause. Leading spaces are removed from values whose data items do specify the JUSTIFIED clause. If a character value consists solely of spaces, one space remains as the value after trimming is finished.

JSON name formation

In the JSON text that is generated from identifier-2, the names are obtained from the NAME phrase if specified; otherwise they are derived from the name of the data item specified by identifier-2 and from any eligible data-names that are subordinate to identifier-2. The exact mixed-case spelling of data-names from the data description entry is retained. The spellings from any references to data items (for example, in an OCCURS DEPENDING ON clause) are not used.
The JSON PARSE statement converts JSON text to COBOL data formats.

**Format**

```
JSON PARSE  identifier-1  INTO  identifier-2
            WITH  DETAIL

            NAME  identifier-3
            OF

            SUPPRESS  identifier-4

            CONVERTING  phrase-1
            ALSO  phrase-1

            EXCEPTION  imperative-statement-1

            NOT  ON  EXCEPTION  imperative-statement-2

            END-JSON
```

**phrase-1 Format**

```
identifier-5  FROM  JSON  BOOLEAN  BOOL  using-phrase-1
```

**using-phrase-1 Format**

```
USING
condition-name-1
AND
condition-name-2
AND
condition-name-3
AND
literal-2
AND
literal-3
```
**Note:** To use the JSON PARSE statement, the CODEPAGE compiler option must specify a single-byte EBCDIC CCSID.

**identifier-1**
The data item that contains the JSON text. *identifier-1* must reference one of the following items:

- An elementary data item of category alphanumeric
- An alphanumeric group item

When *identifier-1* references an alphanumeric group item, *identifier-1* is treated as though it were an elementary data item of category alphanumeric.

*identifier-1* must not be defined with the JUSTIFIED clause, and cannot be a function identifier. *identifier-1* can be subscripted or reference modified.

*identifier-1* must not overlap *identifier-2*.

*identifier-1* must not be a dynamic-length group item or a dynamic-length elementary item.

The JSON text is assumed to be encoded in UTF-8 (CCSID 1208).

All the escaped character sequences defined in the JSON specification are accepted. Also accepted is the sequence “\x”, which is generated by JSON GENERATE, and which represents the EBCDIC NL (newline) control character X’15’, equivalent to the Unicode NEXT LINE control character, NX’0085’.

Conversion from Unicode of the JSON names and values is done according to the compiler CODEPAGE option in effect for the compilation.

**identifier-2**
The group or elementary data item to be populated from the JSON text.

*identifier-2* cannot be a function identifier or be reference modified, but it can be subscripted.

*identifier-2* must not overlap *identifier-1*.

*identifier-2* and its subordinate data items must not contain the UNBOUNDED clause.

*identifier-2* might not be a dynamic-length group item or a dynamic-length elementary item.

The data description entry for *identifier-2* must not contain a RENAMES clause.

The following data items that are specified by *identifier-2* are ignored by the JSON PARSE statement:

- Any subordinate unnamed elementary data items or elementary FILLER data items
- Any slack bytes inserted for SYNCHRONIZED items
- Any data item subordinate to *identifier-2* that is defined with the REDEFINES clause or that is subordinate to such a redefining item
- Any data item subordinate to *identifier-2* that is defined with the RENAMES clause
- Any group data item all of whose subordinate data items are ignored

All data items specified by *identifier-2* that are not ignored according to the previous rules must satisfy the following conditions:

- Each elementary data item must have a USAGE other than DISPLAY-1, FUNCTION-POINTER, INDEX, OBJECT REFERENCE, POINTER, or PROCEDURE-POINTER.
- There must be at least one such elementary data item.
- Each non-FILLER data-name must have a unique identifier within *identifier-2*.
- If (the data declaration of) *identifier-2* or any subordinate data item contains the OCCURS DEPENDING ON clause, then the object(s) of the OCCURS DEPENDING ON clause(s) must not be subordinate to *identifier-2*. Thus, any objects of OCCURS DEPENDING ON clauses will not be updated by the JSON PARSE statement.

**WITH DETAIL phrase**
The WITH DETAIL phrase causes the JSON PARSE statement to emit runtime messages for any
nonexception and exception conditions encountered during parsing.
NAME phrase
For the purpose of matching the name of a JSON name/value pair, the NAME phrase allows you to
effectively change the name of a data item to the specified literal during the execution of the JSON
PARSE statement.
OMITTED can be specified to parse an anonymous JSON object, whose top parent name is not
specified.
identifier-3 must reference identifier-2 or one of its subordinate data items. It cannot be a function
identifier and cannot be reference modified or subscripted. It must not specify any data item which is
ignored by the JSON PARSE statement. For more information about identifier-2, see the description of
identifier-2. If identifier-3 is specified more than once in the NAME phrase, the last specification is
used. If OMITTED is specified, identifier-3 must refer to identifier-2.
literal-1 must be an alphanumeric or national literal containing the JSON name to be associated with
identifier-3.
The NAME phrase in aggregate must not result in an ambiguous name specification. For example,
given the following data declarations and JSON text:
01 G.
05 H.
10 A pic x(10).
10 3_ pic 9.
10 C-C pic x(10).
'{"g": {"H": {"A": "Eh?", "3_": 5, "C-C": "See"}}}'.

Then, if it were allowed, specifying the NAME phrase as:
NAME of A is 'C-C'

would result in no data item receiving the value "Eh?", and an ambiguity about which data item should
receive the value "See", effectively defining the declaration of group G as:
01 G.
05 H.
10 C-C pic x(10).
10 3_ pic 9.
10 C-C pic x(10).

which would be illegal if referenced as identifier-2 in a JSON PARSE statement. Specifying the NAME
phrase as:
NAME of A is 'C-C' C-C is 'A'

is not ambiguous, and would simply swap the assignments to data items A and C-C.
Given the following data declaration and JSON text:
01. top
02. A pic x(20).
03. B pic x(20).
'{"A":"value1","B":"value2"}'

This JSON object can be parsed using OMITTED:
NAME top IS OMITTED

If OMITTED is not specified, the JSON object would need to contain a parent name called "top":
'{"top":{"A":"value1","B":"value2"}}'

352 Enterprise COBOL for z/OS: Enterprise COBOL for z/OS, V6.3 Language Reference


SUPPRESS phrase

Allows you to identify and unconditionally exclude items that are subordinate to identifier-2 from assignment by the JSON PARSE statement.

identifier-4 must reference a data item that is subordinate to identifier-2 and that is not otherwise ignored by the operation of the JSON PARSE statement. identifier-4 cannot be a function identifier and cannot be reference modified or subscripted. identifier-4 can reference an entire table.

If identifier-4 specifies a group data item, that group data item and all data items that are subordinate to the group item are excluded.

Duplicate specifications of identifier-4 are permitted.

A data item that is specified in the SUPPRESS phrase, is suppressed even if the same data item is also specified in the NAME phrase.

CONVERTING phrase

Allows you to specify items that will be parsed as JSON BOOLEAN name/value pairs.

identifier-5 must be a single-byte alphanumeric elementary data item whose data definition entry contains PICTURE X.

The USING phrase provides various methods of specifying the values that shall be effectively moved into identifier-5 when a JSON BOOLEAN true or false value is encountered during parsing.

condition-name-1 must be a level-88 item directly subordinate to identifier-5 and must be specified with both the VALUE clause and the WHEN SET TO FALSE phrase. The first VALUE clause literal (of possibly many values and ranges) will be used to populate identifier-5 when parsing a JSON BOOLEAN true value. The FALSE value will be used to populate identifier-5 when parsing a JSON BOOLEAN false value.

condition-name-2 and condition-name-3 must be level-88 items directly subordinate to identifier-5 whose VALUE clauses are used to populate identifier-5 when a JSON BOOLEAN true or false value is parsed respectively. The first VALUE clause literal will be used in both cases.

literal-2 and literal-3 must be single-byte alphanumeric literals. literal-2 and literal-3 are used to populate identifier-5 when a JSON BOOLEAN true or false value is parsed respectively.

The CONVERTING phrase can be specified with multiple items to be parsed as JSON BOOLEAN name/value pairs by using the ALSO keyword.

Example: CONVERTING phrase with all three formats of the USING phrase

Consider the following COBOL structure and statements:

```
01 docx pic x(1000).
01 myrecord.
   02 data-a pic x.
      88 data-a-flag value 'T' false 'F'.
   02 data-b pic x.
      88 data-b-true value '1'.
      88 data-b-false value '0'.
   02 data-c pic x.

JSON PARSE docx INTO myrecord
   CONVERTING data-a FROM BOOLEAN USING data-a-flag
   ALSO data-b FROM BOOLEAN USING data-b-true AND data-b-false
   ALSO data-c FROM BOOLEAN USING 'a' AND 'z'
DISPLAY data-a
DISPLAY data-b
DISPLAY data-c
```

Assume `docx` contains the following UTF-8 encoded JSON text:

```json
{ "myrecord" :
  { "data-a" : true,
```
ON EXCEPTION phrase

An exception condition exists when an error occurs during parsing of the JSON text, for example an ill-formed JSON value, or during assignment of a value to a COBOL data item. In such cases, JSON parsing stops and the receiver, identifier-2, might be partially modified.

Special register JSON-CODE contains an exception code, as detailed in JSON PARSE conditions and associated codes and runtime messages in the Enterprise COBOL Programming Guide. Special register JSON-STATUS might also contain a nonzero status value, representing one or more nonexception conditions that occurred prior to the exception condition. For more details, see Operation of JSON PARSE.

If the ON EXCEPTION phrase is specified, control is transferred to imperative-statement-1. If the ON EXCEPTION phrase is not specified, the NOT ON EXCEPTION phrase, if any, is ignored, and control is transferred to the end of the JSON PARSE statement.

NOT ON EXCEPTION phrase

If an exception condition does not occur during parsing of the JSON text, control is passed to imperative-statement-2, if specified, otherwise to the end of the JSON PARSE statement. The ON EXCEPTION phrase, if specified, is ignored. Special register JSON-CODE contains zero after execution of the JSON PARSE statement.

Nonexception conditions that can occur during execution of the JSON PARSE statement might result in special register JSON-STATUS being set to a nonzero status value, and the receiver identifier-2 being partially modified.

For more details, see “Operation of JSON PARSE” and JSON PARSE conditions and associated codes and runtime messages in the Enterprise COBOL Programming Guide.

END-JSON phrase

This explicit scope terminator delimits the scope of JSON GENERATE or JSON PARSE statements. END-JSON permits a conditional JSON GENERATE or JSON PARSE statement (that is, a JSON GENERATE or JSON PARSE statement that specifies the ON EXCEPTION or NOT ON EXCEPTION phrase) to be nested in another conditional statement.

The scope of a conditional JSON GENERATE or JSON PARSE statement can be terminated by:

• An END-JSON phrase at the same level of nesting
• A separator period

END-JSON can also be used with a JSON GENERATE or JSON PARSE statement that does not specify either the ON EXCEPTION or the NOT ON EXCEPTION phrase.

For more information about explicit scope terminators, see “Delimited scope statements” on page 270.

Video resource

Watch the video to get an overview of the JSON support in Enterprise COBOL V6.
Nested JSON GENERATE or JSON PARSE statements

When a given JSON GENERATE or JSON PARSE statement appears in *imperative-statement-1* or *imperative-statement-2* of another JSON GENERATE or JSON PARSE statement, that given JSON GENERATE or JSON PARSE statement is a nested JSON GENERATE or JSON PARSE statement.

Nested JSON GENERATE or JSON PARSE statements are considered to be matched JSON GENERATE and END-JSON combinations, or JSON PARSE and END-JSON combinations, proceeding from left to right. Thus, any END-JSON phrase that is encountered is matched with the nearest preceding JSON GENERATE or JSON PARSE statement that has not been implicitly or explicitly terminated.

Operation of JSON PARSE

The JSON text must be valid. Otherwise the statement terminates with an exception condition, and *identifier-2* might be partially modified.

Parsing is guided by a name-matching algorithm in which the containment structure of the names must match exactly, except that omissions of complete elementary or "group" levels are tolerated in the JSON text. For both the JSON name and the COBOL data name, each lowercase single-byte alphabetic letter, a through z, is considered to be equivalent to its corresponding single-byte uppercase alphabetic letter, A through Z. In other words, the name-matching between JSON names and COBOL data names is case insensitive. In the case of an applicable NAME phrase specification, the NAME phrase literal, after being converted by the compiler from the codepage in effect by the CODEPAGE compiler option to Unicode UTF-8, must match the JSON name exactly in a case sensitive match.

For each matching JSON name and data item name, the matching JSON value is assigned to the corresponding data item and occurrence in accordance with the table of “Valid and invalid elementary moves” on page 357, and with the same semantics as the equivalent COBOL MOVE statement.

Whitespace characters (SP, HT, LF, and CR) are ignored, except within strings, and are illegal within numbers.

If, for each elementary data item subordinate to *identifier-2*, there is exactly one matching appropriately qualified JSON name/value pair, regardless of the order in the JSON text, then the JSON PARSE statement terminates without an exception, and with special registers JSON-STATUS and JSON-CODE containing zero.

A number of other conditions encountered during execution of the JSON PARSE statement might result in a nonzero JSON-STATUS value, or an exception and a nonzero JSON-CODE value.

For example, if any combination of JSON value and COBOL data type is invalid, the statement terminates with an exception condition. If any matching JSON value results in one or more substitution characters when translated from Unicode to the CCSID specified by the CODEPAGE compiler option, the value is accepted but results in a nonzero JSON-STATUS value and possibly one or more runtime messages, under control of the WITH DETAIL phrase.

Superfluous JSON name/value pairs that do not match any data item names in *identifier-2* are tolerated, but result in setting special register JSON-STATUS to a condition code, and possibly in one or more runtime messages, under control of the WITH DETAIL phrase.

If there are no matching items, the statement terminates with an exception condition, and *identifier-2* is unmodified.

The special value null is interpreted as an instruction to skip assignment of the corresponding data item or occurrence.

This could be useful for tables. For example parsing the JSON text fragment:

```json
{"myTable":[31, null, 37, null, 41]}
```

would result in setting only the first, third, and fifth occurrences of data item “myTable”.

Chapter 20. PROCEDURE DIVISION statements 355
Each JSON array should have the same cardinality as the associated table data item. If the JSON array has fewer elements than the table item, the additional table elements are not modified. If the JSON array has more values than the matching table item, the additional values are ignored. Both kinds of mismatch result in a nonzero JSON-STATUS setting.

The special values true and false are not supported, and result in an exception condition if present as the value of a JSON name that matches a data item in the receiver.

For a complete description of JSON PARSE conditions, see JSON PARSE conditions and associated codes and runtime messages in the Enterprise COBOL Programming Guide.

Examples of matched and mismatched data definitions and JSON text

The following data definitions and JSON text are considered an exact match.

Data definitions:
```
01 G.
  05 h.
    10 a pic x(10).
    10 _3_ pic 9.
    10 C-c pic x(10).
```

JSON text:
```
{"g": {"H": {"A": "Eh?", "3_": 5, "c-C": "See"}}}
```

The JSON text subsequently generated from this structure would also be an exact match for the JSON text input to the JSON PARSE statement:
```
{"G": {"h": {"a": "Eh?", "3_": 5, "C-c": "See"}}}
```

Omission in the JSON text of names corresponding with elementary items or complete substructures is tolerated, but the level (“qualification”) must match. For example, the following data definitions and JSON text are compatible, but data-items “3_” and “etCetera” would not be modified by the corresponding JSON PARSE statement, and would result in a JSON-STATUS value of 1 at statement termination and a runtime message under control of the WITH DETAIL phrase:
```
01 G.
  05 h.
    10 a pic x(10).
    10 _3_ pic 9.
    10 C-c pic x(10).
    05 etCetera.
      10 etCetera pic x(10).

{"g": {"H": {"A": "Eh?", "c-C": "See"}}}
```

Superfluous items in the JSON are tolerated, but result in nonzero JSON-STATUS codes, and runtime messages under control of the WITH DETAIL phrase. For example, the following data definitions and JSON text are compatible, and would result in completely populating group G, and terminating without an exception. However, because JSON name/value pair “B”: “Bee” would not be used, special register JSON-STATUS would be set to a reason code of 2:
```
01 G.
  05 h.
    10 a pic x(10).
    10 _3_ pic 9.
    10 C-c pic x(10).

{"G": {"h": {"A": "Eh?", "B": "Bee", "3_": 5, "c-C": "See"}}}
```

The following data definitions and JSON text are fully compatible, despite the name order mismatch.
The following data definitions and JSON text are not compatible, because of the omitted name level (qualification by "H"), and, because no data items would be changed, would result in an exception condition:

01 G.
05 h.
10 a pic x(10).
10 3_ pic 9.
10 C-c pic x(10).

{"g": {"H": {"3_": 5, "A": "Eh?", "c-C": "See"}}} 

The following data definitions and JSON text are not compatible, because the JSON values are all incompatible with the corresponding data items, and would result in an exception condition:

01 G.
02 h.
10 a pic a(10).
10 3_ pic 9.
10 C-c pic 99.

{"g": {"H": {"A": 42, "3_": "x", "c-C": "abc"}}} 

Count of table elements set by JSON PARSE

JSON texts do not necessarily include a count of values in an array. To determine how many table elements are set by a given JSON PARSE statement, preset all occurrences of the table to a special value. The preset value should be known to be absent from the incoming JSON array values, then after execution of the JSON PARSE statement, search the table for the first occurrence of the special value. The preceding values are those that were set by the JSON PARSE statement.

Valid and invalid elementary moves

The table shows valid and invalid elementary moves for each category.

In the table:

- YES = Move is valid.
- NO = Move is invalid.
- Column headings indicate receiving item categories; row headings indicate JSON values.

<table>
<thead>
<tr>
<th>Table 44. Valid and invalid elementary moves</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid and invalid elementary moves</td>
</tr>
<tr>
<td>------------------------------------------</td>
</tr>
<tr>
<td>String</td>
</tr>
<tr>
<td>Number (fixed-point integer)</td>
</tr>
</tbody>
</table>
Parsing JSON values that result in a loss of significance or information

JSON values may exceed the size, length, or precision of the data items they are being assigned to by the JSON PARSE statement. When such a situation occurs, the JSON PARSE statement will truncate the value similarly to the equivalent MOVE statement and populate the receiving data item with a loss of significance or information, and may set a reason code of 128 or 256 in the JSON-STATUS special register. If the user also specifies the WITH DETAIL phrase then one or more runtime messages may also be emitted. The type of truncation that occurs depends on the type of the receiving data item, and the type of the JSON value. Some types of truncation such as loss of precision due to assignment to a floating-point receiver, or from a floating-point sender, will not be recognized by the JSON PARSE statement.

The assignment of a JSON value to its corresponding data item follows the Alignment Rules.

JSON-STATUS special register values 128 and 256

The JSON-STATUS special register is set to 128 when a loss of significance of a fixed-point integer, fixed-point noninteger, or floating-point JSON value occurs similar to a COMPUTE SIZE ERROR condition. That is, when a JSON value, after decimal point alignment, exceeds the largest value that can be contained in the receiver. This also applies to JSON string values that contain only digits.

The JSON-STATUS special register is set to 256 when a loss of information of a JSON string value occurs.

MERGE statement

The MERGE statement combines two or more identically sequenced files (that is, files that have already been sorted according to an identical set of ascending or descending keys) on one or more keys and makes records available in merged order to an output procedure or output file.

A MERGE statement can appear anywhere in the PROCEDURE DIVISION except in a declarative section.

The MERGE statement is not supported for programs compiled with the THREAD compiler option.
file-name-1
The name given in the SD entry that describes the records to be merged.

No file-name can be repeated in the MERGE statement.

No pair of file-names in a MERGE statement can be specified in the same SAME AREA, SAME SORT AREA, or SAME SORT-MERGE AREA clause. However, any file-names in the MERGE statement can be specified in the same SAME RECORD AREA clause.

When the MERGE statement is executed, all records contained in file-name-2, file-name-3, ..., are accepted by the merge program and then merged according to the keys specified.

ASCENDING/DESCENDING KEY phrase
This phrase specifies that records are to be processed in an ascending or descending sequence (depending on the phrase specified), based on the specified merge keys.

data-name-1
Specifies a KEY data item on which the merge will be based. Each such data-name must identify a data item in a record associated with file-name-1. The data-names following the word KEY are listed from left to right in the MERGE statement in order of decreasing significance without regard to how they are divided into KEY phrases. The leftmost data-name is the major key, the next data-name is the next most significant key, and so forth.

The following rules apply:

• A specific key data item must be physically located in the same position and have the same data format in each input file. However, it need not have the same data-name.

• If file-name-1 has more than one record description, the KEY data items need be described in only one of the record descriptions.

• If file-name-1 contains variable-length records, all of the KEY data-items must be contained within the first n character positions of the record, where n equals the minimum records size specified for file-name-1.
• KEY data items must not contain an OCCURS clause or be subordinate to an item that contains an OCCURS clause.

• KEY data items cannot be:
  – Variably located
  – Group items that contain variable-occurrence data items
  – Category numeric described with usage NATIONAL (national decimal type)
  – Category external floating-point described with usage NATIONAL (national floating-point)
  – Category DBCS
  – Dynamic-length elementary items
  – Dynamic-length group items

• KEY data items can be qualified.

• KEY data items can be any of the following data categories:
  – Alphabetic, alphanumeric, alphanumeric-edited
  – Numeric (except numeric with usage NATIONAL)
  – Numeric-edited (with usage DISPLAY or NATIONAL)
  – Internal floating-point or display floating-point
  – National or national-edited

The direction of the merge operation depends on the specification of the ASCENDING or DESCENDING keywords as follows:

• When ASCENDING is specified, the sequence is from the lowest key value to the highest key value.

• When DESCENDING is specified, the sequence is from the highest key value to the lowest key value.

If the KEY data item is described with usage NATIONAL, the sequence of the KEY values is based on the binary values of the national characters.

When the COLLATING SEQUENCE phrase is not specified, the key comparisons are performed according to the rules for comparison of operands in a relation condition. For details, see “General relation conditions” on page 250.

When the COLLATING SEQUENCE phrase is specified, the indicated collating sequence is used for key data items of alphabetic, alphanumeric, alphanumeric-edited, external floating-point, and numeric-edited categories. For all other key data items, the comparisons are performed according to the rules for comparison of operands in a relation condition.

**COLLATING SEQUENCE phrase**

This phrase specifies the collating sequence to be used in alphanumeric comparisons for the KEY data items in this merge operation.

The COLLATING SEQUENCE phrase has no effect for keys that are not alphabetic or alphanumeric.

**alphabet-name-1**

Must be specified in the ALPHABET clause of the SPECIAL-NAMES paragraph. Any one of the alphabet-name clause phrases can be specified, with the following results:

**STANDARD-1**

The ASCII collating sequence is used for all alphanumeric comparisons. (The ASCII collating sequence is shown in “US English ASCII code page” on page 582.)
STANDARD-2

The 7-bit code defined in the International Reference Version of ISO/IEC 646, 7-bit coded character set for information interchange is used for all alphanumeric comparisons.

NATIVE

The EBCDIC collating sequence is used for all alphanumeric comparisons. (The EBCDIC collating sequence is shown in “EBCDIC collating sequence” on page 579.)

EBCDIC

The EBCDIC collating sequence is used for all alphanumeric comparisons. (The EBCDIC collating sequence is shown in “EBCDIC collating sequence” on page 579.)

literal

The collating sequence established by the specification of literals in the ALPHABET-NAME clause is used for all alphanumeric comparisons.

When the COLLATING SEQUENCE phrase is omitted, the PROGRAM COLLATING SEQUENCE clause (if specified) in the OBJECT-COMPUTER paragraph identifies the collating sequence to be used. When both the COLLATING SEQUENCE phrase of the MERGE statement and the PROGRAM COLLATING SEQUENCE clause of the OBJECT-COMPUTER paragraph are omitted, the EBCDIC collating sequence is used.

USING phrase

file-name-2, file-name-3, ...

Specifies the input files.

During the MERGE operation, all the records on file-name-2, file-name-3, ... (that is, the input files) are transferred to file-name-1. At the time the MERGE statement is executed, these files must not be open. The input files are automatically opened, read, and closed. If DECLARATIVE procedures are specified for these files for input operations, the declaratives will be driven for errors if errors occur.

All input files must specify sequential or dynamic access mode and be described in FD entries in the DATA DIVISION.

If file-name-1 contains variable-length records, the size of the records contained in the input files (file-name-2, file-name-3, ...) must be neither less than the smallest record nor greater than the largest record described for file-name-1. If file-name-1 contains fixed-length records, the size of the records contained in the input files must not be greater than the largest record described for file-name-1. For more information, see Sorting and merging files in the Enterprise COBOL Programming Guide.

GIVING phrase

file-name-4, ...

Specifies the output files.

When the GIVING phrase is specified, all the merged records in file-name-1 are automatically transferred to the output files (file-name-4, ...).

All output files must specify sequential or dynamic access mode and be described in FD entries in the DATA DIVISION.

If the output files (file-name-4, ...) contain variable-length records, the size of the records contained in file-name-1 must be neither less than the smallest record nor greater than the largest record described for the output files. If the output files contain fixed-length records, the size of the records contained in file-name-1 must not be greater than the largest record described for the output files. For more information, see Sorting and merging files in the Enterprise COBOL Programming Guide.

At the time the MERGE statement is executed, the output files (file-name-4, ...) must not be open. The output files are automatically opened, written to, and closed. If DECLARATIVE procedures are specified for these files for output operations, the declaratives will be driven for errors if errors occur.
OUTPUT PROCEDURE phrase
This phrase specifies the name of a procedure that is to select or modify output records from the merge operation.

procedure-name-1
   Specifies the first (or only) section or paragraph in the OUTPUT PROCEDURE.

procedure-name-2
   Identifies the last section or paragraph of the OUTPUT PROCEDURE.

The OUTPUT PROCEDURE can consist of any procedure needed to select, modify, or copy the records that are made available one at time by the RETURN statement in merged order from the file referenced by file-name-1. The range includes all statements that are executed as the result of a transfer of control by CALL, EXIT, GO TO, PERFORM, and XML PARSE statements in the range of the output procedure. The range also includes all statements in declarative procedures that are executed as a result of the execution of statements in the range of the output procedure. The range of the output procedure must not cause the execution of any MERGE, RELEASE, or format 1 SORT statement.

If an output procedure is specified, control passes to it after the file referenced by file-name-1 has been sequenced by the MERGE statement. The compiler inserts a return mechanism at the end of the last statement in the output procedure and when control passes the last statement in the output procedure, the return mechanism provides the termination of the merge and then passes control to the next executable statement after the MERGE statement. Before entering the output procedure, the merge procedure reaches a point at which it can select the next record in merged order when requested. The RETURN statements in the output procedure are the requests for the next record.

The OUTPUT PROCEDURE phrase is similar to a basic PERFORM statement. For example, if you name a procedure in an OUTPUT PROCEDURE, that procedure is executed during the merging operation just as if it were named in a PERFORM statement. As with the PERFORM statement, execution of the procedure is terminated after the last statement completes execution. The last statement in an OUTPUT PROCEDURE can be the EXIT statement (see “EXIT statement” on page 316).

MERGE special registers
The topic describes special registers of the MERGE statement.

SORT-CONTROL special register
   You identify the sort control file (through which you can specify additional options to the sort/merge function) with the SORT-CONTROL special register.

   If you use a sort control file to specify control statements, the values specified in the sort control file take precedence over those in the other SORT special registers.

   For information, see “SORT-CONTROL” on page 22.

SORT-MESSAGE special register
   For information, see “SORT-MESSAGE” on page 23. The special register SORT-MESSAGE is equivalent to an option control statement keyword in the sort control file.

SORT-RETURN special register
   For information, see “SORT-RETURN” on page 23.

Segmentation considerations
   If a MERGE statement is coded in a fixed segment, any output procedure referenced by that MERGE statement must be either totally within a fixed segment or wholly contained in a single independent segment.

   If a MERGE statement is coded in an independent segment, any output procedure referenced by that MERGE statement must be either totally within a fixed segment or wholly contained within the same independent segment as that MERGE statement.
The MOVE statement transfers data from one area of storage to one or more other areas.

**Format 1: MOVE statement**

```
MOVE identifier-1 literal-1 TO identifier-2
```

**Format 2: MOVE statement with CORRESPONDING phrase**

```
MOVE CORRESPONDING CORR identifier-1 TO identifier-2
```

CORR is an abbreviation for, and is equivalent to, CORRESPONDING.

**identifier-1, literal-1**

The sending area.

**identifier-2**

The receiving areas. identifier-2 must not reference an intrinsic function.

When format 1 is specified:

- All identifiers can reference alphanumeric group items, national group items, UTF-8 group items, or elementary items.
- When one of identifier-1 or identifier-2 references a national group item and the other operand references an alphanumeric group item, the national group is processed as a group item; in all other cases, the national group item is processed as an elementary data item of category national.
- When one of identifier-1 or identifier-2 references a UTF-8 group item and the other operand references an alphanumeric group item, the UTF-8 group is processed as a group item; in all other cases, the UTF-8 group item is processed as an elementary data item of category UTF-8.
- The data in the sending area is moved into the data item referenced by each identifier-2 in the order in which the identifier-2 data items are specified in the MOVE statement. See “Elementary moves” on page 364 and “Group moves” on page 368 below.

When format 2 is specified:

- Both identifiers must be group items.
- A national group item is processed as a group item (and not as an elementary data item of category national).
- A UTF-8 group item is processed as a group item (and not as an elementary data item of category UTF-8).
- Selected items in identifier-1 are moved to identifier-2 according to the rules for the “CORRESPONDING phrase” on page 272. The results are the same as if each pair of CORRESPONDING identifiers were referenced in a separate MOVE statement.

Data items described with the following types of usage cannot be specified in a MOVE statement:

- INDEX
- POINTER
- FUNCTION-POINTER
- PROCEDURE-POINTER
• OBJECT REFERENCE

A data item defined with a usage of INDEX, POINTER, FUNCTION-POINTER, PROCEDURE-POINTER, or OBJECT REFERENCE can be part of an alphanumeric group item that is referenced in a MOVE CORRESPONDING statement; however, no movement of data from those data items takes place.

The evaluation of the length of the sending or receiving area can be affected by the DEPENDING ON phrase of the OCCURS clause (see “OCCURS clause” on page 182).

If the sending field (identifier-1) is reference-modified or subscripted, or is an alphanumeric or national function-identifier, the reference-modifier, subscript, or function is evaluated only once, immediately before data is moved to the first of the receiving operands.

Any length evaluation, subscripting, or reference-modification associated with a receiving field (identifier-2) is evaluated immediately before the data is moved into that receiving field.

For example, the result of the statement:

```cobol
MOVE A(B) TO B, C(B).
```

is equivalent to:

```cobol
MOVE A(B) TO TEMP.
MOVE TEMP TO B.
MOVE TEMP TO C(B).
```

where TEMP is defined as an intermediate result item. The subscript B has changed in value between the time that the first move took place and the time that the final move to C(B) is executed.

For further information about intermediate results, see Appendix A. Intermediate results and arithmetic precision in the Enterprise COBOL Programming Guide.

After execution of a MOVE statement, the sending fields contain the same data as before execution.

**Usage note:** Overlapping operands in a MOVE statement can cause unpredictable results.

**Elementary moves**

An elementary move is one in which the receiving item is an elementary data item and the sending item is an elementary data item or a literal.

Valid operands belong to one of the following categories:

- **Alphabetic:** includes data items of category alphabetic and the figurative constant SPACE
- **Alphanumeric:** includes the following items:
  - Data items of category alphanumeric
  - Alphanumeric functions
  - Alphanumeric literals
  - The figurative constant ALL alphanumeric-literal and all other figurative constants (except NULL) when used in a context that requires an alphanumeric sending item
- **Alphanumeric-edited:** includes data items of category alphanumeric-edited
- **DBCS:** includes data items of category DBCS, DBCS literals, and the figurative constant ALL DBCS-literal.
- **External floating-point:** includes data items of category external floating point (described with USAGE DISPLAY or USAGE NATIONAL) and floating-point literals.
- **Internal floating-point:** includes data items of category internal floating-point (defined as USAGE COMP-1 or USAGE COMP-2)
- **National:** includes the following items:
  - National group items (treated as elementary item of category national)
– Data items of category national
– National literals
– National functions
– Figurative constants ZERO, SPACE, QUOTE, and ALL national-literal when used in a context that requires a national sending item

- National-edited: includes data items of category national-edited
- Numeric: includes the following items:
  – Data items of category numeric
  – Numeric literals
  – The figurative constant ZERO (when ZERO is moved to a numeric or numeric-edited item).
- Numeric-edited: includes data items of category numeric-edited.
- UTF-8: includes data items of category UTF-8.

**Elementary move rules**

Any necessary conversion of data from one form of internal representation to another takes place during the move, along with any specified editing in, or de-editing implied by, the receiving item. The code page used for conversion to or from alphanumeric characters is the one in effect for the CODEPAGE compiler option when the source code was compiled.

The following rules outline the execution of valid elementary moves. When the receiving field is:

**Alphabetic:**

- Alignment and any necessary space filling or truncation occur as described under “Alignment rules” on page 158.
- If the size of the sending item is greater than the size of the receiving item, excess characters on the right are truncated after the receiving item is filled.

**Alphanumeric or alphanumeric-edited:**

- If the sending item is a national decimal integer item, the sending data is converted to usage DISPLAY and treated as though it were moved to a temporary data item of category alphanumeric with the same number of character positions as the sending item. The resulting alphanumeric data item is treated as the sending item.
- Alignment and any necessary space filling or truncation take place, as described under “Alignment rules” on page 158.
- If the receiving item is a dynamic-length elementary item, the length of the receiver is set to the minimum of the length of the sender or the specified or implied value of the LIMIT phrase. If the length of the sending item is greater than the length of the receiving item, then excess characters on the right are truncated after the receiving item is filled.
- For items that are not dynamic-length elementary items, if the size of the sending item is greater than the size of the receiving item, excess characters on the right are truncated after the receiving item is filled.
- If the initial sending item has an operational sign, the unsigned value is used. If the operational sign occupies a separate character, that character is not moved, and the size of the sending item is considered to be one less character than the actual size.

**DBCS:**

- If the sending and receiving items are not the same size, the sending data is either truncated on the right or padded with DBCS spaces on the right.

**External floating-point:**

- For a floating-point sending item, the floating-point value is converted to the usage of the receiving external floating-point item (if different from the sending item’s representation).
• For other sending items, the numeric value is treated as though that value were converted to internal floating-point and then converted to the usage of the receiving external floating-point item.

**Internal floating-point:**
• When the category of the sending operand is not internal floating-point, the numeric value of the sending item is converted to internal floating-point format.

**National or national-edited:**
• If the representation of the sending item is not national characters, the sending data is converted to national characters and treated as though it were moved to a temporary data item of category national of a length not to cause truncation or padding. The resulting category national data item is treated as the sending data item.
• If the representation of the sending item is national characters, the sending data is used without conversion.
• Alignment and any necessary space filling or truncation take place as described under “Alignment rules” on page 158. The programmer is responsible for ensuring that multiple encoding units that together form a graphic character are not split by truncation.
• If the sending item has an operational sign, the unsigned value is used. If the operational sign occupies a separate character, that character is not moved, and the size of the sending item is considered to be one less character than the actual size.

**UTF-8:**
• If the representation of the sending item is not UTF-8 characters, the sending data is converted to UTF-8 characters and treated as though it were moved to a temporary data item of category UTF-8 of a length not to cause truncation or padding. The resulting category UTF-8 data item is treated as the sending data item.
• If the representation of the sending item is UTF-8 characters, the sending data is used without conversion.
• Alignment and any necessary space filling or truncation take place as described under “Alignment rules” on page 158. The programmer is responsible for ensuring that multiple encoding units that together form a graphic character are not split by truncation.
• If the sending item has an operational sign, the unsigned value is used. If the operational sign occupies a separate character, that character is not moved, and the size of the sending item is considered to be one less character than the actual size.

**Numeric or numeric-edited:**
• Except when zeros are replaced because of editing requirements, alignment by decimal point and any necessary zero filling take place, as described under “Alignment rules” on page 158.
• If the receiving item is signed, the sign of the sending item is placed in the receiving item, with any necessary sign conversion. If the sending item is unsigned, a positive operational sign is generated for the receiving item.
• If the receiving item is unsigned, no operational sign is generated for the receiving item and the absolute value of the sending item is used in the move.
• When the category of the sending item is alphanumeric, alphanumeric-edited, national, or national-edited, the data is moved as if the sending item were described as an unsigned integer.
• When the sending item is floating-point, the data is first converted to either a binary or internal decimal representation and is then moved.
• When the receiving item is numeric-edited, editing takes place as defined by the picture character string or BLANK WHEN ZERO clause associated with the receiving item.
• When the sending item is numeric-edited, the compiler de-edits the sending data to establish the unedited value of the numeric-edited item (this value can be signed). The unedited numeric value is used in the move to the receiving numeric or numeric-edited data item.

**Usage notes:**
1. If the receiving item is of category alphanumeric, alphanumeric-edited, numeric-edited, national, or national-edited and the sending field is numeric, any digit positions described with picture symbol P in the sending item are considered to have the value zero. Each P is counted in the size of the sending item.

2. If the receiving item is numeric and the sending field is an alphanumeric literal, a national literal, or an ALL literal, all characters of the literal must be numeric characters.

Valid and invalid elementary moves
The table shows valid and invalid elementary moves for each category.

In the table:
- **YES** = Move is valid.
- **NO** = Move is invalid.
- Column headings indicate receiving item categories; row headings indicate sending item categories.

<table>
<thead>
<tr>
<th>Valid and invalid elementary moves</th>
<th>Alphabetic</th>
<th>Alphanumeric</th>
<th>Alphanumeric edited</th>
<th>Numeric</th>
<th>Numeric-edited</th>
<th>Extern al floating-point</th>
<th>Internal floating-point</th>
<th>DBCS</th>
<th>National</th>
<th>National-edited</th>
<th>UTF-8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alphabetic and SPACE sending item</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Alphanumeric sending item²</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes³</td>
<td>Yes³</td>
<td>Yes³</td>
<td>Yes³</td>
<td>Yes³</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Alphanumeric-edited sending item</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Numeric integer and ZERO sending item⁴</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Numeric noninteger sending item⁵</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Numeric-edited sending item</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Floating-point sending item⁶</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>DBCS sending item⁷</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>National sending item⁸</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>National-edited sending item</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 45. Valid and invalid elementary moves (continued)

<table>
<thead>
<tr>
<th>Valid and invalid elementary moves</th>
<th>Alpha-</th>
<th>Alphano-</th>
<th>Alpha-</th>
<th>Numeric</th>
<th>Numeric-</th>
<th>Extern-</th>
<th>Intern-</th>
<th>DBCS</th>
<th>National</th>
<th>National-</th>
<th>UTF-8</th>
</tr>
</thead>
<tbody>
<tr>
<td>UTF-8 sending item</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

1. Includes DBCS data items.
2. Includes alphanumeric literals.
3. Figurative constants and alphanumeric literals must consist only of numeric characters and will be treated as numeric integer fields.
4. Includes integer numeric literals.
5. Includes noninteger numeric literals.
6. Includes floating-point literals, external floating-point data items (USAGE DISPLAY or USAGE NATIONAL), and internal floating-point data items (USAGE COMP-1 or USAGE COMP-2).
7. Includes DBCS data-items, DBCS literals, and figurative constant SPACE.
8. Figurative constants and alphanumeric literals must consist only of numeric characters and will be treated as numeric integer fields. The ALL literal cannot be used as a sending item.
9. Includes national data items, national literals, national functions, and figurative constants ZERO, SPACE, QUOTE, and ALL national literal.

Moves involving file record areas

The successful execution of an OPEN statement for a given file makes the record area for that file available. You can move data to or from the record description entries associated with a file only when the file is in the open status.

Execution of an implicit or explicit CLOSE statement removes a file from open status and makes the record area unavailable.

Group moves

A group move can be any move in which an alphanumeric group item is a sending item or a receiving item, or both.

The group moves are:

- A move to an alphanumeric group item from one of the following items:
  - any elementary data item that is valid as a sending item in the MOVE statement
  - a national group item
  - a UTF-8 group item
  - a literal
  - a figurative constant

- A move from an alphanumeric group item to the following items:
  - any elementary data item that is valid as a receiving item in the MOVE statement
  - a national group item
  - a UTF-8 group item
  - an alphanumeric group item

- A move from a UTF-8 group item to the following items:
A group move is treated as though it were an alphanumeric-to-alphanumeric elementary move, except that there is no conversion of data from one form of internal representation to another. In a group move, the receiving area is filled without consideration for the individual elementary items contained within either the sending area or the receiving area, except as noted in the OCCURS clause. (See “OCCURS clause” on page 182.)

Dynamic-length group items cannot be moved to, or from, another group item.

**MULTIPLY statement**

The MULTIPLY statement multiplies numeric items and sets the values of data items equal to the results.

**Format 1: MULTIPLY statement**

```
MULTIPLY identifier-1
       BY literal-1
       identifier-2
       ROUNDED
```

```
       SIZE ERROR
        ON
        imperative-statement-1
```

```
       NOT
        SIZE ERROR
        ON
        imperative-statement-2
```

```
       END-MULTIPLY
```

In format 1, the value of `identifier-1` or `literal-1` is multiplied by the value of `identifier-2`; the product is then placed in `identifier-2`. For each successive occurrence of `identifier-2`, the multiplication takes place in the left-to-right order in which `identifier-2` is specified.
In format 2, the value of `identifier-1` or `literal-1` is multiplied by the value of `identifier-2` or `literal-2`. The product is then stored in the data items referenced by `identifier-3`.

**For all formats:**

- **`identifier-1`, `identifier-2`**
  
  Must name an elementary numeric item.

- **`literal-1`, `literal-2`**
  
  Must be a numeric literal.

**For format-2:**

- **`identifier-3`**
  
  Must name an elementary numeric or numeric-edited item.

Floating-point data items and literals can be used anywhere a numeric data item or literal can be specified.

When the ARITH(COMPAT) compiler option is in effect, the composite of operands can contain a maximum of 30 digits. When the ARITH(EXTEND) compiler option is in effect, the composite of operands can contain a maximum of 31 digits. For more information, see “Arithmetic statement operands” on page 274 and the details on arithmetic intermediate results, Appendix A. Intermediate results and arithmetic precision in the Enterprise COBOL Programming Guide.

**ROUNDED phrase**

For formats 1 and 2, see “ROUNDED phrase” on page 273.

**SIZE ERROR phrases**

For formats 1 and 2, see “SIZE ERROR phrases” on page 273.

**END-MULTIPLY phrase**

This explicit scope terminator serves to delimit the scope of the MULTIPLY statement. END-MULTIPLY permits a conditional MULTIPLY statement to be nested in another conditional statement. END-MULTIPLY can also be used with an imperative MULTIPLY statement.
OPEN statement

The OPEN statement initiates the processing of files. It also checks or writes labels, or both.

Format 1: OPEN statement for sequential files

```
OPEN
   INPUT file-name-1
   REVERSED 1
   WITH NO REWIND 1
   OUTPUT file-name-2
   WITH NO REWIND
   I-O file-name-3
   EXTEND file-name-4
```

Notes:
1 The REVERSED and WITH NO REWIND phrases are not valid for VSAM files.

Format 2: OPEN statement for indexed and relative files

```
OPEN
   INPUT file-name-1
   OUTPUT file-name-2
   I-O file-name-3
   EXTEND file-name-4
```
The phrases INPUT, OUTPUT, I-O, and EXTEND specify the mode to be used for opening the file. At least one of the phrases INPUT, OUTPUT, I-O, or EXTEND must be specified with the OPEN keyword. The INPUT, OUTPUT, I-O, and EXTEND phrases can appear in any order.

**INPUT**
Permits input operations.

**OUTPUT**
Permits output operations. This phrase can be specified when the file is being created.

Do not specify OUTPUT for files that:
- Contain records. The file will be replaced by new data.
  
  If the OUTPUT phrase is specified for a file that already contains records, the data set must be defined as reusable and cannot have an alternate index. The records in the file will be replaced by the new data and any ALTERNATE RECORD KEY clause in the SELECT statement will be ignored.
- Are defined with a DD dummy card. Unpredictable results can occur.

**I-O**
Permits both input and output operations. The I-O phrase can be specified only for files assigned to direct access devices.

The I-O phrase is not valid for line-sequential files.

**EXTEND**
Permits output operations that append to or create a file.

The EXTEND phrase is allowed for sequential access files only if the new data is written in ascending sequence. The EXTEND phrase is allowed for files that specify the LINAGE clause.

For QSAM files, do not specify the EXTEND phrase for a multiple file reel.

If you want to append to a file, but are unsure if the file exists, use the SELECT OPTIONAL clause before opening the file in EXTEND mode. The file will be created or appended to, depending on whether the file exists.

**file-name-1, file-name-2, file-name-3, file-name-4**
Designate a file upon which the OPEN statement is to operate. If more than one file is specified, the files need not have the same organization or access mode. Each file-name must be defined in an FD entry in the DATA DIVISION and must not name a sort or merge file. The FD entry must be equivalent to the information supplied when the file was defined.

**REVERSED**
Valid only for sequential single-reel files. REVERSED is not valid for VSAM files.

If the concept of reels has no meaning for the storage medium (for example, a direct access device), the REVERSED and NO REWIND phrases do not apply.
**NO REWIND**

Valid only for sequential single-reel files. It is not valid for VSAM files.

For information on file sizes, see Appendix B, “Compiler limits,” on page 573.

**General rules**

The topic shows general rules of the OPEN statement.

- If a file opened with the INPUT phrase is an optional file that is not available, the OPEN statement sets the file position indicator to indicate that an optional input file is not available.
- Execution of an OPEN INPUT or OPEN I-O statement sets the file position indicator:
  - For indexed files, to the characters with the lowest ordinal position in the collating sequence associated with the file.
  - For sequential and relative files, to 1.
- When the EXTEND phrase is specified, the OPEN statement positions the file immediately after the last record written in the file. (The record with the highest prime record key value for indexed files or relative key value for relative files is considered the last record.) Subsequent WRITE statements add records as if the file were opened OUTPUT. The EXTEND phrase can be specified when a file is being created; it can also be specified for a file that contains records, or that has contained records that have been deleted. For more information, see note 1 in the “OPEN statement notes” on page 373 and SELECT OPTIONAL in the “SELECT clause” on page 126.
- For VSAM files, if no records exist in the file, the file position indicator is set so that the first format 1 READ statement executed results in an AT END condition.
- When NO REWIND is specified, the OPEN statement execution does not reposition the file; prior to OPEN statement execution, the file must be positioned at its beginning. When the NO REWIND phrase is specified (or when both the NO REWIND and REVERSE phrases are omitted), file positioning is specified with the LABEL parameter of the DD statement.
- When REVERSED is specified, OPEN statement execution positions the QSAM file at its end. Subsequent READ statements make the data records available in reversed order, starting with the last record.

If the PASSWORD clause is specified in the file-control entry, the password data item must contain a valid password before the OPEN statement is executed. If a valid password is not present, OPEN statement execution is unsuccessful.

**OPEN statement notes**

The topic provides notes for the OPEN statement.

The notes are:

1. The successful execution of an OPEN statement determines the availability of the file and results in that file being open mode. A QSAM file is available if it has a DD allocation and is physically present. A VSAM file is available if it has a DD allocation, has been defined using VSAM access method services, and contains records or has previously contained records. For more information regarding file availability, see Opening a file (ESDS, KSDS, or RRDS) in the Enterprise COBOL Programming Guide. The following table shows the results of opening available and unavailable files.

<table>
<thead>
<tr>
<th>Table 46. Availability of a file</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Opened as</strong></td>
</tr>
<tr>
<td>INPUT</td>
</tr>
<tr>
<td>INPUT (optional file)</td>
</tr>
</tbody>
</table>
Table 46. Availability of a file (continued)

<table>
<thead>
<tr>
<th>Opened as</th>
<th>File is available</th>
<th>File is unavailable</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-O</td>
<td>Normal open</td>
<td>Open is unsuccessful. (file status 35)</td>
</tr>
<tr>
<td>I-O (optional file)</td>
<td>Normal open</td>
<td>Open causes the file to be created. (file status 05)</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>Normal open; the file contains no records</td>
<td>Open causes the file to be created.</td>
</tr>
<tr>
<td>EXTEND</td>
<td>Normal open</td>
<td>Open is unsuccessful. (file status 35)</td>
</tr>
<tr>
<td>EXTEND (optional file)</td>
<td>Normal open</td>
<td>Open causes the file to be created. (file status 05)</td>
</tr>
</tbody>
</table>

2. The successful execution of the OPEN statement places the file in open status and makes the associated record area available to the program.

3. The OPEN statement does not obtain or release the first data record.

4. You can move data to or from the record area only when the file is in open status.

5. An OPEN statement must be successfully executed prior to the execution of any of the permissible input-output statements, except a SORT or MERGE statement with the USING or GIVING phrase. In the following table, an ‘X’ indicates that the specified statement can be used with the open mode given at the top of the column.

6. The VSAMOPENFS option affects the user file status reported from successful OPEN statements on VSAM files.

7. The CBLQDA Language Environment (LE) runtime option affects the user file status reported from OPEN statements on QSAM files that are not found:
   - With CBLQDA(ON), LE would create a temporary data set and the OPEN would be successful.
   - With CBLQDA(OFF), LE would not create a temporary data set and the OPEN would fail.

   For details about the CBLQDA runtime option, see CBLQDA in the z/OS Language Environment Programming Reference.

Table 47. Permissible statements for sequential files

<table>
<thead>
<tr>
<th>Statement</th>
<th>Input open mode</th>
<th>Output open mode</th>
<th>I-O open mode</th>
<th>Extend open mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>WRITE</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>REWRITE</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

In the following table, an ‘X’ indicates that the specified statement, used in the access mode given for that row, can be used with the open mode given at the top of the column.
In the following table, an 'X' indicates that the specified statement can be used with the open mode given at the top of the column.

### Table 48. Permissible statements for indexed and relative files

<table>
<thead>
<tr>
<th>File access mode</th>
<th>Statement</th>
<th>Input open mode</th>
<th>Output open mode</th>
<th>I-O open mode</th>
<th>Extend open mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>READ</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>WRITE</td>
<td>-</td>
<td>X</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>REWRITE</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>START</td>
<td>-</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>DELETE</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>Random</td>
<td>READ</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>WRITE</td>
<td>-</td>
<td>X</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>REWRITE</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>START</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>Dynamic</td>
<td>READ</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>WRITE</td>
<td>-</td>
<td>X</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>REWRITE</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>START</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>DELETE</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
</tbody>
</table>

### Table 49. Permissible statements for line-sequential files

<table>
<thead>
<tr>
<th>Statement</th>
<th>Input open mode</th>
<th>Output open mode</th>
<th>I-O open mode</th>
<th>Extend open mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WRITE</td>
<td>-</td>
<td>X</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>REWRITE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

1. A file can be opened for INPUT, OUTPUT, I-O, or EXTEND (sequential and line-sequential files only) in the same program. After the first OPEN statement execution for a given file, each subsequent OPEN statement execution must be preceded by a successful CLOSE file statement execution without the REEL or UNIT phrase (for QSAM files only), or the LOCK phrase.

2. If the FILE STATUS clause is specified in the file-control entry, the associated file status key is updated when the OPEN statement is executed.

3. If an OPEN statement is executed for a file that is already open, the EXCEPTION/ERROR procedure (if specified) for this file is run.
PERFORM statement

The PERFORM statement transfers control explicitly to one or more procedures and implicitly returns control to the next executable statement after execution of the specified procedures is completed.

**Format 1: Basic PERFORM statement**

```
PERFORM procedure-name-1 THROUGH procedure-name-2
                           THRU
                           imperative-statement-1
                           END-PERFORM
```

*procedure-name-1, procedure-name-2*

Must name a section or paragraph in the procedure division.

When both *procedure-name-1* and *procedure-name-2* are specified, if either is a procedure-name in a declarative procedure, both must be procedure-names in the same declarative procedure.

If *procedure-name-1* is specified, *imperative-statement-1* and the END-PERFORM phrase must not be specified.

If *procedure-name-1* is omitted, *imperative-statement-1* and the END-PERFORM phrase must be specified.

*imperative-statement-1*

The statements to be executed for an in-line PERFORM

**Inline and out-of-line PERFORM statements**

The PERFORM statement is an inline PERFORM statement, when *procedure-name-1* is omitted.

The PERFORM statement is an out-of-line PERFORM statement, when *procedure-name-1* is specified.

An inline PERFORM must be delimited by the END-PERFORM phrase.

The inline and out-of-line formats cannot be combined. For example, if *procedure-name-1* is specified, imperative statements and the END-PERFORM phrase must not be specified.

You can use the EXIT PERFORM statement to exit from an inline PERFORM without using a GO TO statement or a PERFORM ... THROUGH statement. For details, see “Format 5 (inline-perform)” on page 318.

You can use the INLINE directive to decide whether a procedure referenced by PERFORM statements is eligible for inlining. For details, see “INLINE” on page 546.

**END-PERFORM**

Delimits the scope of the in-line PERFORM statement. Execution of an in-line PERFORM is completed after the last statement contained within it has been executed.

**Basic PERFORM statement**

The procedures referenced in the basic PERFORM statement are executed once, and control then passes to the next executable statement following the PERFORM statement.

**Note:** A PERFORM statement must not cause itself to be executed. A recursive PERFORM statement can cause unpredictable results.
An in-line PERFORM statement functions according to the same general rules as an otherwise identical out-of-line PERFORM statement, except that statements contained within the in-line PERFORM are executed in place of the statements contained within the range of \textit{procedure-name-1} (through \textit{procedure-name-2}, if specified). Unless specifically qualified by the word \textit{in-line} or the word \textit{out-of-line}, all the rules that apply to the out-of-line PERFORM statement also apply to the in-line PERFORM.

Whenever an out-of-line PERFORM statement is executed, control is transferred to the first statement of the procedure named \textit{procedure-name-1}. Control is always returned to the statement following the PERFORM statement. The point from which this control is returned is determined as follows:

- If \textit{procedure-name-1} is a paragraph name and \textit{procedure-name-2} is not specified, the return is made after the execution of the last statement of the \textit{procedure-name-1} paragraph.
- If \textit{procedure-name-1} is a section name and \textit{procedure-name-2} is not specified, the return is made after the execution of the last statement of the last paragraph in the \textit{procedure-name-1} section.
- If \textit{procedure-name-2} is specified and it is a paragraph name, the return is made after the execution of the last statement of the \textit{procedure-name-2} paragraph.
- If \textit{procedure-name-2} is specified and it is a section name, the return is made after the execution of the last statement of the last paragraph in the \textit{procedure-name-2} section.

The only necessary relationship between \textit{procedure-name-1} and \textit{procedure-name-2} is that a consecutive sequence of operations is executed, beginning at the procedure named by \textit{procedure-name-1} and ending with the execution of the procedure named by \textit{procedure-name-2}.

PERFORM statements can be specified within the performed procedure. If there are two or more logical paths to the return point, then \textit{procedure-name-2} can name a paragraph that consists only of an EXIT statement; all the paths to the return point must then lead to this paragraph.

When the performed procedures include another PERFORM statement, the sequence of procedures associated with the embedded PERFORM statement must be totally included in or totally excluded from the performed procedures of the first PERFORM statement. That is, an active PERFORM statement whose execution point begins within the range of performed procedures of another active PERFORM statement must not allow control to pass through the exit point of the other active PERFORM statement. However, two or more active PERFORM statements can have a common exit.

When control passes to the sequence of procedures by means other than a PERFORM statement, control passes through the exit point to the next executable statement, as if no PERFORM statement referred to these procedures.

The following figure illustrates valid sequences of execution for PERFORM statements.
This figure illustrates valid PERFORM statement execution sequences. Letters of the alphabet are used to represent procedures. The following examples are shown:

1. PERFORM a THRU m. The sequence of procedures is a, d, f, j, and m. Procedure d contains PERFORM f THRU j. In this sequence, procedures f THRU j are nested within the range of procedures a THRU m.

2. PERFORM a THRU m. The sequence of procedures is a, d, h, m, f, and j. Procedure d contains PERFORM f THRU j. In this sequence, procedures f THRU j are wholly outside the range of procedures a THRU m.

3. PERFORM a THRU m. The sequence of procedures is a, f, m, j, and d. Procedure d contains PERFORM f THRU j. In this sequence, the two PERFORM statements have overlapping range; f thru j overlaps a thru m.

4. PERFORM a THRU m. The sequence of procedures is a, d, f, j, and m. Procedure m terminates with an EXIT statement. Procedure d contains PERFORM d THRU m. In this sequence, both PERFORM statements share the same exit point.

**PERFORM with TIMES phrase**

The procedures referred to in the TIMES phrase of the PERFORM statement are executed the number of times specified by the value in identifier-1 or integer-1, up to a maximum of 999,999,999 times. Control then passes to the next executable statement following the PERFORM statement.
If procedure-name-1 is specified, imperative-statement-1 and the END-PERFORM phrase must not be specified.

**identifier-1**

Must name an integer item.

If identifier-1 is zero or a negative number at the time the PERFORM statement is initiated, control passes to the statement following the PERFORM statement.

After the PERFORM statement has been initiated, any change to identifier-1 has no effect in varying the number of times the procedures are initiated.

**integer-1**

Can be a positive signed integer.

**PERFORM with UNTIL phrase**

In the UNTIL phrase format, the procedures referred to are performed until the condition specified by the UNTIL phrase is true. Control is then passed to the next executable statement following the PERFORM statement.

**Format 3: PERFORM statement with UNTIL phrase**

```
PERFORM procedure-name-1 THROUGH procedure-name-2 phrase 1
phrase 1
imperative-statement-1 END-PERFORM
phrase 1
TEST BEFORE AFTER
UNTIL condition-1
```

If procedure-name-1 is specified, imperative-statement-1 and the END-PERFORM phrase must not be specified.

**condition-1**

Can be any condition described under “Conditional expressions” on page 246. If the condition is true at the time the PERFORM statement is initiated, the specified procedures are not executed.

Any subscripting associated with the operands specified in condition-1 is evaluated each time the condition is tested.

If the TEST BEFORE phrase is specified or assumed, the condition is tested before any statements are executed (corresponds to DO WHILE).

If the TEST AFTER phrase is specified, the statements to be performed are executed at least once before the condition is tested (corresponds to DO UNTIL).

In either case, if the condition is true, control is transferred to the next executable statement following the end of the PERFORM statement. If neither the TEST BEFORE nor the TEST AFTER phrase is specified, the TEST BEFORE phrase is assumed.
PERFORM with VARYING phrase

The VARYING phrase increases or decreases the value of one or more identifiers or index-names, according to certain rules.

For more information, see “Varying phrase rules” on page 384.

The format-4 VARYING phrase PERFORM statement can serially search an entire seven-dimensional table.

Format 4: PERFORM statement with VARYING phrase

```
PERFORM procedure-name-1
     THROUGH procedure-name-2
     THRU phrase 1 phrase 2
     phrase 1
     WITH TEST BEFORE AFTER VARYING identifier-2 index-name-1
     FROM identifier-3 index-name-2 literal-1
     BY identifier-4 literal-2 UNTIL condition-1

phrase 2

AFTER identifier-5 index-name-3 FROM identifier-6 index-name-4 literal-3
     phrase 3

phrase 3

BY identifier-7 literal-4 UNTIL condition-2
```

If `procedure-name-1` is specified, `imperative-statement-1` and the END-PERFORM phrase must not be specified. If `procedure-name-1` is omitted, the AFTER phrase must not be specified.

**identifier-2** through **identifier-7**

Must name a numeric elementary item.

**literal-1** through **literal-4**

Must represent a numeric literal.

**condition-1, condition-2**

Can be any condition described under “Conditional expressions” on page 246. If the condition is true at the time the PERFORM statement is initiated, the specified procedures are not executed.

After the conditions specified in the UNTIL phrase are satisfied, control is passed to the next executable statement following the PERFORM statement.
If any of the operands specified in \textit{condition-1 or condition-2} is subscripted, reference modified, or is a function-identifier, the subscript, reference-modifier, or function is evaluated each time the condition is tested.

Floating-point data items and literals can be used anywhere a numeric data item or literal can be specified.

When \textit{TEST BEFORE} is indicated, all specified conditions are tested before the first execution, and the statements to be performed are executed, if at all, only when all specified tests fail. When \textit{TEST AFTER} is indicated, the statements to be performed are executed at least once, before any condition is tested.

If neither the \textit{TEST BEFORE} nor the \textit{TEST AFTER} phrase is specified, the \textit{TEST BEFORE} phrase is assumed.

\textbf{Varying identifiers}

The way in which operands are increased or decreased depends on the number of variables specified. In the discussion, every reference to \textit{identifier-n} refers equally to \textit{index-name-n} (except when \textit{identifier-n} is the object of the BY phrase).

If \textit{identifier-2} or \textit{identifier-5} is subscripted, the subscripts are evaluated each time the content of the data item referenced by the identifier is set or augmented. If \textit{identifier-3}, \textit{identifier-4}, \textit{identifier-6}, or \textit{identifier-7} is subscripted, the subscripts are evaluated each time the content of the data item referenced by the identifier is used in a setting or an augmenting operation.

The following figure illustrates the logic of the \textit{PERFORM} statement when an identifier is varied with \textit{TEST BEFORE}.

![Logic diagram for \textit{PERFORM} statement with \textit{TEST BEFORE}]

The following figure illustrates the logic of the \textit{PERFORM} statement when an identifier is varied with \textit{TEST AFTER}.

![Logic diagram for \textit{PERFORM} statement with \textit{TEST AFTER}]

Chapter 20. \textbf{PROCEDURE DIVISION} statements 381
**Varying two identifiers**

The topic lists steps of varying two identifiers.

```
PERFORM PROCEDURE-NAME-1 THROUGH PROCEDURE-NAME-2
   VARYING IDENTIFIER-2 FROM IDENTIFIER-3
       BY IDENTIFIER-4 UNTIL CONDITION-1
   AFTER IDENTIFIER-5 FROM IDENTIFIER-6
       BY IDENTIFIER-7 UNTIL CONDITION-2
```

1. `identifier-2` and `identifier-5` are set to their initial values, `identifier-3` and `identifier-6`, respectively.
2. `condition-1` is evaluated as follows:
   a. If it is false, steps 3 through 7 are executed.
   b. If it is true, control passes directly to the statement following the PERFORM statement.
3. `condition-2` is evaluated as follows:
   a. If it is false, steps 4 through 6 are executed.
   b. If it is true, `identifier-2` is augmented by `identifier-4`, `identifier-5` is set to the current value of `identifier-6`, and step 2 is repeated.
4. `procedure-name-1` and `procedure-name-2` are executed once (if specified).
5. `identifier-5` is augmented by `identifier-7`.
6. Steps 3 through 5 are repeated until `condition-2` is true.
7. Steps 2 through 6 are repeated until `condition-1` is true.

At the end of PERFORM statement execution:

- `identifier-5` contains the current value of `identifier-6`.
- `identifier-2` has a value that exceeds the last-used setting by the increment or decrement value (unless `condition-1` was true at the beginning of PERFORM statement execution, in which case, `identifier-2` contains the current value of `identifier-3`).

The following figure illustrates the logic of the PERFORM statement when two identifiers are varied with TEST BEFORE.
The following figure illustrates the logic of the PERFORM statement when two identifiers are varied with TEST AFTER.
Varying three identifiers
The topic lists steps of varying three identifiers.

PERFORM PROCEDURE-NAME-1 THROUGH PROCEDURE-NAME-2
  VARYING IDENTIFIER-2 FROM IDENTIFIER-3
    BY IDENTIFIER-4 UNTIL CONDITION-1
    AFTER IDENTIFIER-5 FROM IDENTIFIER-6
      BY IDENTIFIER-7 UNTIL CONDITION-2
    AFTER IDENTIFIER-8 FROM IDENTIFIER-9
      BY IDENTIFIER-10 UNTIL CONDITION-3

The actions are the same as those for two identifiers, except that IDENTIFIER-8 goes through the complete cycle each time that IDENTIFIER-5 is augmented by IDENTIFIER-7, which, in turn, goes through a complete cycle each time that IDENTIFIER-2 is varied.

At the end of PERFORM statement execution:
  • IDENTIFIER-5 and IDENTIFIER-8 contain the current values of IDENTIFIER-6 and IDENTIFIER-9, respectively.
  • IDENTIFIER-2 has a value exceeding its last-used setting by one increment/decrement value (unless CONDITION-1 was true at the beginning of PERFORM statement execution, in which case IDENTIFIER-2 contains the current value of IDENTIFIER-3).

Varying more than three identifiers
You can produce analogous PERFORM statement actions to the previous example with the addition of up to four AFTER phrases.

Varying phrase rules
There are certain rules that apply to this phrase, no matter how many variables are specified.

The rules are:
  • In the VARYING or AFTER phrases, when an index-name is specified:
The index-name is initialized and incremented or decremented according to the rules under “INDEX phrase” on page 222. (See also “SET statement” on page 401.)

In the associated FROM phrase, an identifier must be described as an integer and have a positive value; a literal must be a positive integer.

In the associated BY phrase, an identifier must be described as an integer; a literal must be a nonzero integer.

- In the FROM phrase, when an index-name is specified:
  - In the associated VARYING or AFTER phrase, an identifier must be described as an integer. It is initialized as described in the SET statement.
  - In the associated BY phrase, an identifier must be described as an integer and have a nonzero value; a literal must be a nonzero integer.

- In the BY phrase, identifiers and literals must have nonzero values.

- Changing the values of identifiers or index-names in the VARYING, FROM, and BY phrases during execution changes the number of times the procedures are executed.

**READ statement**

For sequential access, the READ statement makes the next logical record from a file available to the object program. For random access, the READ statement makes a specified record from a direct-access file available to the object program.

When the READ statement is executed, the associated file must be open in INPUT or I-O mode.

![Format 1: READ statement for sequential retrieval](image)
**Format 2: READ statement for random retrieval**

```plaintext
READ file-name-1
  RECORD INTO identifier-1
    KEY IS data-name-1
    INVALID
    imperative-statement-3
    NOT INVALID
    imperative-statement-4
END-READ
```

**file-name-1**
Must be defined in a DATA DIVISION FD entry.

**NEXT RECORD**
- Reads the next record in the logical sequence of records. NEXT is optional when the access mode is sequential, and has no effect on READ statement execution.
- You must specify the NEXT RECORD phrase to retrieve records sequentially from files in dynamic access mode.

**INTO identifier-1**
- identifier-1 is the receiving field.
- identifier-1 must be a valid receiving field for the selected sending record description entry in accordance with the rules of the MOVE statement.

The record areas associated with file-name-1 and identifier-1 must not be the same storage area.

When there is only one record description associated with file-name-1 or all the records and the data item referenced by identifier-1 describe an elementary alphanumeric item or an alphanumeric group item, the result of the execution of a READ statement with the INTO phrase is equivalent to the application of the following rules in the order specified:

- The execution of the same READ statement without the INTO phrase.
- The current record is moved from the record area to the area specified by identifier-1 according to the rules for the MOVE statement without the CORRESPONDING phrase. The size of the current record is determined by rules specified for the RECORD clause. If the file description entry contains a RECORD IS VARYING clause, the implied move is a group move. The implied MOVE statement does not occur if the execution of the READ statement was unsuccessful. Any subscripting or reference modification associated with identifier-1 is evaluated after the record has been read and immediately before it is moved to the data item. The record is available in both the record area and the data item referenced by identifier-1.

- identifier-1 must not be a dynamic-length group item or a dynamic-length elementary item.

When there are multiple record descriptions associated with file-name-1 and they do not all describe an alphanumeric group item or elementary alphanumeric item, the following rules apply:

1. If the file referenced by file-name-1 is described as containing variable-length records, or as a QSAM file with RECORDING MODE 'S' or 'U', a group move will take place.
2. If the file referenced by file-name-1 is described as containing fixed-length records, a move will take place according to the rules for a MOVE statement using, as a sending field description, the record that specifies the largest number of character positions. If more than one such record
exists, the sending field record selected will be the one among those records that appears first under the description of \textit{file-name-1}.

**KEY IS phrase**

The KEY IS phrase can be specified only for indexed files. \textit{data-name-1} must identify a record key associated with \textit{file-name-1}. \textit{data-name-1} can be qualified; it cannot be subscripted.

**AT END phrases**

For sequential access, both the AT END phrase and an applicable EXCEPTION/ERROR procedure can be omitted.

For information about at-end condition processing, see \textit{AT END condition}.

**INVALID KEY phrases**

Both the INVALID KEY phrase and an applicable EXCEPTION/ERROR procedure can be omitted.

For information about INVALID KEY phrase processing, see “Invalid key condition” on page 280.

**END-READ phrase**

This explicit scope terminator serves to delimit the scope of the READ statement. END-READ permits a conditional READ statement to be nested in another conditional statement. END-READ can also be used with an imperative READ statement. For more information, see “Delimited scope statements” on page 270.

**Processing files with variable-length records or multiple record descriptions**

If more than one record description entry is associated with \textit{file-name-1}, those records automatically share the same storage area; that is, they are implicitly redefined. After a READ statement is executed, only those data items within the range of the current record are replaced; data items stored beyond that range are undefined. The following example illustrates this concept.

If the length of the current record that is read is less than the minimum size specified by the record description entries for \textit{file-name-1}, the portion of the record area which is to the right of the last valid character read is undefined. If the length of the current record exceeds the record description entries for \textit{file-name-1}, the record is truncated on the right to the maximum record definition size.

In either of the previous cases, the READ statement is successful, and the I-O status is set to either 00 (hiding the record length conflict condition) or 04 (indicating that a record length conflict has occurred), depending on the VLR compiler option setting. If compiler option VLR(COMPAT) is in effect, the I-O status would be set to 00.

For more information about the VLR compiler option, see \textit{VLR} in the \textit{Enterprise COBOL Programming Guide}.

The following example shows two record areas of different sizes in an FD. When a shorter record is read, the content of the remaining record area is undefined.

```cobol
FD INPUT-FILE LABEL RECORD OMITTED.
  01   RECORD-1 PICTURE X(30).
  01   RECORD-2 PICTURE X(20).

Content of input area when READ statement is executed:

ABCDEFGHJKLMNOPQRSTUVWXYZ1234
```

Chapter 20. PROCEDURE DIVISION statements 387
Sequential access mode

Format 1 must be used for all files in sequential access mode.

Execution of a format-1 READ statement retrieves the next logical record from the file. The next record accessed is determined by the file organization.

Sequential files

The NEXT RECORD is the next record in a logical sequence of records. The NEXT phrase need not be specified; it has no effect on READ statement execution.

If SELECT OPTIONAL is specified in the file-control entry for this file, and the file is unavailable during this execution of the object program, execution of the first READ statement causes an at-end condition; however, since no file is available, the system-defined end-of-file processing is not performed.

AT END condition

If the file position indicator indicates that no next logical record exists, or that an optional input file is not available, at-end condition processing occurs in a specific order.

The order is:

1. A value derived from the setting of the file position indicator is placed into the I-O status associated with file-name-1 to indicate the at-end condition.
2. If the AT END phrase is specified in the statement causing the condition, control is transferred to imperative-statement-1 in the AT END phrase. Any USE AFTER STANDARD EXCEPTION procedure associated with file-name-1 is not executed.
3. If the AT END phrase is not specified and an applicable USE AFTER STANDARD EXCEPTION procedure exists, the procedure is executed. Return from that procedure is to the next executable statement following the end of the READ statement.

Both the AT END phrase and an applicable EXCEPTION/ERROR procedure can be omitted.

When the at-end condition occurs, execution of the READ statement is unsuccessful. The contents of the associated record area are undefined and the file position indicator is set to indicate that no valid next record has been established.

For QSAM files, attempts to access or move data into the record area after an unsuccessful read can result in a protection exception.

If an at-end condition does not occur during the execution of a READ statement, the AT END phrase is ignored, if specified, and the following actions occur:

1. The file position indicator is set and the I-O status associated with file-name-1 is updated.
2. If an exception condition that is not an at-end condition exists, control is transferred to the end of the READ statement after the execution of any USE AFTER STANDARD EXCEPTION procedure applicable to file-name-1.

If no USE AFTER STANDARD EXCEPTION procedure is specified, control is transferred to the end of the READ statement or to imperative-statement-2, if specified.
3. If no exception condition exists, the record is made available in the record area and any implicit move resulting from the presence of an INTO phrase is executed. Control is transferred to the end of the READ statement or to `imperative-statement-2`, if specified. In the latter case, execution continues according to the rules for each statement specified in `imperative-statement-2`. If a procedure branching or conditional statement which causes explicit transfer of control is executed, control is transferred in accordance with the rules for that statement; otherwise, upon completion of the execution of `imperative-statement-2`, control is transferred to the end of the READ statement.

After the unsuccessful execution of a READ statement, the contents of the associated record area are undefined and the file position indicator is set to indicate that no valid next record has been established. Attempts to access or move data into the record area after an unsuccessful read can result in a protection exception.

**Indexed or relative files**

The NEXT RECORD is the next logical record in the key sequence.

For indexed files, the key sequence is the sequence of ascending values of the current key of reference. For relative files, the key sequence is the sequence of ascending values of relative record numbers for records that exist in the file.

Before the READ statement is executed, the file position indicator must have been set by a successful OPEN, START, or READ statement. When the READ statement is executed, the record indicated by the file position indicator is made available if it is still accessible through the path indicated by the file position indicator.

If the record is no longer accessible (because it has been deleted, for example), the file position indicator is updated to point to the next existing record in the file, and that record is made available.

For files in sequential access mode, the NEXT phrase need not be specified.

For files in dynamic access mode, the NEXT phrase must be specified for sequential record retrieval.

**AT END condition**

This condition exists when the file position indicator indicates that no next logical record exists or that an optional input file is not available. The same procedure occurs as for sequential files (see AT END condition).

If neither an at-end nor an invalid key condition occurs during the execution of a READ statement, the AT END or the INVALID KEY phrase is ignored, if specified. The same actions occur as when the at-end condition does not occur with sequential files (see AT END condition).

**Sequentially accessed indexed files**

When an ALTERNATE RECORD KEY with DUPLICATES is the key of reference, file records with duplicate key values are made available in the order in which they were placed in the file.

**Sequentially accessed relative files**

If the RELATIVE KEY clause is specified for this file, READ statement execution updates the RELATIVE KEY data item to indicate the relative record number of the record being made available.

**Random access mode**

Format 2 must be specified for indexed and relative files in random access mode, and also for files in the dynamic access mode when record retrieval is random.

Execution of the READ statement depends on the file organization, as explained in the following sections.

**Indexed files**

Execution of a format-2 READ statement causes the value of the key of reference to be compared with the value of the corresponding key data item in the file records, until the first record having an equal value is found. The file position indicator is positioned to this record, which is then made available. If no record
can be so identified, an INVALID KEY condition exists, and READ statement execution is unsuccessful. (See “Invalid key condition” on page 280 for details of the invalid key condition.)

If the KEY phrase is not specified, the prime RECORD KEY becomes the key of reference for this request. When dynamic access is specified, the prime RECORD KEY is also used as the key of reference for subsequent executions of sequential READ statements, until a different key of reference is established.

When the KEY phrase is specified, data-name-1 becomes the key of reference for this request. When dynamic access is specified, this key of reference is used for subsequent executions of sequential READ statements, until a different key of reference is established.

**Relative files**

Execution of a format-2 READ statement sets the file position indicator pointer to the record whose relative record number is contained in the RELATIVE KEY data item, and makes that record available.

If the file does not contain such a record, the INVALID KEY condition exists, and READ statement execution is unsuccessful. (See “Invalid key condition” on page 280 for details of the invalid key condition).

The KEY phrase must not be specified for relative files.

**Dynamic access mode**

For files with indexed or relative organization, dynamic access mode can be specified in the file-control entry. In dynamic access mode, either sequential or random record retrieval can be used, depending on the format used.

Format 1 with the NEXT phrase must be specified for sequential retrieval. All other rules for sequential access apply.

**READ statement notes**

This topic provides notes on the READ statement.

- If the FILE-STATUS clause is specified in the file-control entry, the associated file status key is updated when the READ statement is executed.
- After unsuccessful READ statement execution, the contents of the associated record area and the value of the file position indicator are undefined. Attempts to access or move data into the record area after an unsuccessful read can result in a protection exception.
- If the number of character positions in a record is less than the minimum size specified by the record description entries for file-name-1, after a READ statement is executed, the portion of the record area that is to the right of the last valid character read is undefined. If the number of character positions in a record exceed the maximum size specified by the record description entries for file-name-1, after a READ statement is executed, the record is truncated on the right to the maximum size.

In either of these cases, the READ statement is successful and the I-O status is set to either 00 (hiding the record length conflict condition) or 04 (indicating that a record length conflict has occurred), depending on the VLR compiler option setting.

- When the VLR(COMPAT) compiler option is in effect, the status value of 00 is set.
- When the VLR(STANDARD) compiler option is in effect, the status value of 04 is set.

For more information about the VLR compiler option, see VLR in the Enterprise COBOL Programming Guide.
**RELEASE statement**

The RELEASE statement transfers records from an input/output area to the initial phase of a sorting operation.

The RELEASE statement can be used only within the range of an INPUT PROCEDURE associated with a SORT statement.

**Format: RELEASE**

```
RELEASE record-name-1 FROM identifier-1
```

Within an INPUT PROCEDURE, at least one RELEASE statement must be specified.

When the RELEASE statement is executed, the current contents of `record-name-1` are placed in the sort file. This makes the record available to the initial phase of the sorting operation.

**record-name-1**

Must specify the name of a logical record in a sort-merge file description entry (SD). `record-name-1` can be qualified.

**FROM phrase**

The result of the execution of the RELEASE statement with the FROM `identifier-1` phrase is equivalent to the execution of the following statements in the order specified.

```
MOVE identifier-1 to record-name-1.
RELEASE record-name-1.
```

The MOVE is performed according to the rules for the MOVE statement without the CORRESPONDING phrase.

**identifier-1**

`identifier-1` must reference one of the following items:

- An entry in the WORKING-STORAGE SECTION, the LOCAL-STORAGE SECTION, or the LINKAGE SECTION
- A record description for another previously opened file
- An alphanumeric or national function.

`identifier-1` must be a valid sending item with `record-name-1` as the receiving item in accordance with the rules of the MOVE statement.

`identifier-1` and `record-name-1` must not refer to the same storage area.

`identifier-1` must not be a dynamic-length group item or a dynamic-length elementary item.

After the RELEASE statement is executed, the information is still available in `identifier-1`. (See “INTO and FROM phrases” on page 281 under “Common processing facilities”.)

If the RELEASE statement is executed without specifying the SD entry for `file-name-1` in a SAME RECORD AREA clause, the information in `record-name-1` is no longer available.

If the SD entry is specified in a SAME RECORD AREA clause, `record-name-1` is still available as a record of the other files named in that clause.

When FROM `identifier-1` is specified, the information is still available in `identifier-1`.

When control passes from the INPUT PROCEDURE, the sort file consists of all those records placed in it by execution of RELEASE statements.
RETURN statement

The RETURN statement transfers records from the final phase of a sorting or merging operation to an OUTPUT PROCEDURE.

The RETURN statement can be used only within the range of an OUTPUT PROCEDURE associated with a SORT or MERGE statement.

Format: RETURN statement

```
RETURN file-name-1 RECORD INTO identifier-1
  AT END imperative-statement-1

NOT AT END imperative-statement-2 END-RETURN
```

Within an OUTPUT PROCEDURE, at least one RETURN statement must be specified.

When the RETURN statement is executed, the next record from file-name-1 is made available for processing by the OUTPUT PROCEDURE.

file-name-1

Must be described in a DATA DIVISION SD entry.

If more than one record description is associated with file-name-1, those records automatically share the same storage; that is, the area is implicitly redefined. After RETURN statement execution, only the contents of the current record are available. If any data items lie beyond the length of the current record, their contents are undefined.

INTO phrase

When there is only one record description associated with file-name-1 or all the records and the data item referenced by identifier-1 describe an elementary alphanumeric item or an alphanumeric group item, the result of the execution of a RETURN statement with the INTO phrase is equivalent to the application of the following rules in the order specified:

• The execution of the same RETURN statement without the INTO phrase.
• The current record is moved from the record area to the area specified by identifier-1 according to the rules for the MOVE statement without the CORRESPONDING phrase. The size of the current record is determined by rules specified for the RECORD clause. If the file description entry contains a RECORD IS VARYING clause, the implied move is a group move. The implied MOVE statement does not occur if the execution of the RETURN statement was unsuccessful. Any subscripting or reference modification associated with identifier-1 is evaluated after the record has been read and immediately before it is moved to the data item. The record is available in both the record area and the data item referenced by identifier-1.

When there are multiple record descriptions associated with file-name-1 and they do not all describe an alphanumeric group item or elementary alphanumeric item, the following rules apply:

1. If the file referenced by file-name-1 contains variable-length records, a group move takes place.
2. If the file referenced by file-name-1 contains fixed-length records, a move takes place according to the rules for a MOVE statement using, as a sending field description, the record that specifies the largest number of character positions. If more than one such record exists, the sending field record
selected will be the one among those records that appears first under the description of *file-name*-1.

*identifier*-1 must be a valid receiving field for the selected sending record description entry in accordance with the rules of the MOVE statement.

The record areas associated with *file-name*-1 and *identifier*-1 must not be the same storage area.

**AT END phrases**

The imperative-statement specified on the AT END phrase executes after all records have been returned from *file-name*-1. No more RETURN statements can be executed as part of the current output procedure.

If an at-end condition does not occur during the execution of a RETURN statement, then after the record is made available and after executing any implicit move resulting from the presence of an INTO phrase, control is transferred to the imperative statement specified by the NOT AT END phrase. If an at-end condition does occur, control is transferred to the end of the RETURN statement.

**END-RETURN phrase**

This explicit scope terminator serves to delimit the scope of the RETURN statement. END-RETURN permits a conditional RETURN statement to be nested in another conditional statement. END-RETURN can also be used with an imperative RETURN statement.

For more information, see “Delimited scope statements” on page 270.

**REWRITE statement**

The REWRITE statement logically replaces an existing record in a direct-access file. When the REWRITE statement is executed, the associated direct-access file must be open in I-O mode.

The REWRITE statement is not supported for line-sequential files.

```
Format: REWRITE statement

REWRITE record-name-1 FROM identifier-1

INVALID
imperative-statement-1

KEY

NOT INVALID
imperative-statement-2

KEY

END-REWRITE
```

*record-name*-1

Must be the name of a logical record in a DATA DIVISION FD entry. The record-name can be qualified.

*FROM phrase*

The result of the execution of the REWRITE statement with the FROM *identifier*-1 phrase is equivalent to the execution of the following statements in the order specified.

```
MOVE identifier-1 TO record-name-1.
REWRITE record-name-1
```

The MOVE is performed according to the rules for the MOVE statement without the CORRESPONDING phrase.
**identifier-1**

*identifier-1* can reference one of the following items:

- A record description for another previously opened file
- An alphanumeric or national function
- A data item defined in the WORKING-STORAGE SECTION, the LOCAL-STORAGE SECTION, or the LINKAGE SECTION

*identifier-1* must be a valid sending item with *record-name-1* as the receiving item in accordance with the rules of the MOVE statement.

*identifier-1* and *record-name-1* must not refer to the same storage area.

*identifier-1* must not be a dynamic-length group item or a dynamic-length elementary item.

After the REWRITE statement is executed, the information is still available in *identifier-1* ("INTO and FROM phrases" on page 281 under "Common processing facilities").

**INVALID KEY phrases**

An INVALID KEY condition exists when:

- The access mode is sequential, and the value contained in the prime RECORD KEY of the record to be replaced does not equal the value of the prime RECORD KEY data item of the last-retrieved record from the file
- The value contained in the prime RECORD KEY does not equal that of any record in the file
- The value of an ALTERNATE RECORD KEY data item for which DUPLICATES is not specified is equal to that of a record already in the file

For details of invalid key processing, see Invalid key condition.

**END-REWRITE phrase**

This explicit scope terminator serves to delimit the scope of the REWRITE statement. END-REWRITE permits a conditional REWRITE statement to be nested in another conditional statement. END-REWRITE can also be used with an imperative REWRITE statement.

For more information, see “Delimited scope statements” on page 270.

**Reusing a logical record**

After successful execution of a REWRITE statement, the logical record is no longer available in *record-name-1* unless the associated file is named in a SAME RECORD AREA clause (in which case, the record is also available as a record of the other files named in the SAME RECORD AREA clause).

The file position indicator is not affected by execution of the REWRITE statement.

If the FILE STATUS clause is specified in the file-control entry, the associated file status key is updated when the REWRITE statement is executed.

**Sequential files**

For files in the sequential access mode, the last prior input/output statement executed for this file must be a successfully executed READ statement. When the REWRITE statement is executed, the record retrieved by that READ statement is logically replaced.

The number of character positions in *record-name-1* must equal the number of character positions in the record being replaced.

The INVALID KEY phrase must not be specified for a file with sequential organization. An EXCEPTION/ERROR procedure can be specified.
Indexed files

The number of character positions in record-name-1 can be different from the number of character positions in the record being replaced.

When the access mode is sequential, the record to be replaced is specified by the value contained in the prime RECORD KEY. When the REWRITE statement is executed, this value must equal the value of the prime record key data item in the last record read from this file.

Both the INVALID KEY phrase and an applicable EXCEPTION/ERROR procedure can be omitted.

When the access mode is random or dynamic, the record to be replaced is specified by the value contained in the prime RECORD KEY.

Values of ALTERNATE RECORD KEY data items in the rewritten record can differ from those in the record being replaced. The system ensures that later access to the record can be based upon any of the record keys.

If an invalid key condition exists, the execution of the REWRITE statement is unsuccessful, the updating operation does not take place, and the data in record-name-1 is unaffected. (See Invalid key condition under "Common processing facilities").

Relative files

The number of character positions in record-name-1 can be different from the number of character positions in the record being replaced.

For relative files in sequential access mode, the INVALID KEY phrase must not be specified. An EXCEPTION/ERROR procedure can be specified.

For relative files in random or dynamic access mode, the INVALID KEY phrase or an applicable EXCEPTION/ERROR procedure can be specified. Both can be omitted.

When the access mode is random or dynamic, the record to be replaced is specified in the RELATIVE KEY data item. If the file does not contain the record specified, an invalid key condition exists, and, if specified, the INVALID KEY imperative-statement is executed. (See Invalid key condition under "Common processing facilities"). The updating operation does not take place, and the data in record-name is unaffected.

SEARCH statement

The SEARCH statement searches a table for an element that satisfies the specified condition and adjusts the associated index to indicate that element.

Format 1: SEARCH statement for serial search

```
SEARCH  identifier-1
      VARYING  identifier-2
           index-name-1

      AT  imperative-statement-1

      WHEN  condition-1  imperative-statement-2
           NEXT SENTENCE

      END SEARCH
```
Use format 1 (serial search) when the table that you want to search has not been sorted. Use format 1 to search a sorted table when you want to search serially through the table or you want to control subscripts or indexes.

Use format 2 (binary search) when you want to efficiently search across all occurrences in a table. The table must previously have been sorted, and you can sort the table with the format 2 SORT statement.

**AT END and WHEN phrases**

After imperative-statement-1 or imperative-statement-2 is executed, control passes to the end of the SEARCH statement, unless imperative-statement-1 or imperative-statement-2 ends with a GO TO statement.

The function of the AT END phrase is the same for a serial search and a binary search.

**NEXT SENTENCE**

NEXT SENTENCE transfers control to the first statement following the closest separator period.

When NEXT SENTENCE is specified with END-SEARCH, control does not pass to the statement following the END-SEARCH. Instead, control passes to the statement after the closest following period.

For the format-2 SEARCH ALL statement, neither imperative-statement-2 nor NEXT SENTENCE is required. Without them, the SEARCH statement sets the index to the value in the table that matched the condition.

The function of the NEXT SENTENCE phrase is the same for a serial search and a binary search.

**END-SEARCH phrase**

This explicit scope terminator delimits the scope of the SEARCH statement. END-SEARCH permits a conditional SEARCH statement to be nested in another conditional statement.

For more information, see “Delimited scope statements” on page 270.

The function of END-SEARCH is the same for a serial search and a binary search.
Serial search

The topic provides information of using the SEARCH statement for serial search.

**identifier-1 (serial search)**

*identifier-1* identifies the table that is to be searched. *identifier-1* references all occurrences within that table.

The data description entry for *identifier-1* must contain an OCCURS clause.

The data description entry for *identifier-1* should contain an OCCURS clause with the INDEXED BY phrase, but a table can be searched using an index defined for an appropriately described different table.

*identifier-1* can reference a data item that is subordinate to a data item that is described with an OCCURS clause (that is, *identifier-1* can be a subordinate table within a multidimensional table). In this case, the data description entries must specify an INDEXED BY phrase for each dimension of the table.

*identifier-1* must not be subscripted or reference-modified.

*identifier-1* must not be a dynamic-length group item or a dynamic-length elementary item.

**AT END**

The condition that exists when the search operation terminates without satisfying the condition specified in any of the associated WHEN phrases.

Before executing a serial search, you must set the value of the first (or only) index associated with *identifier-1* (the search index) to indicate the starting occurrence for the search.

Before using a serial search on a multidimensional table, you must also set the value of the index for each superordinate dimension.

The SEARCH statement modifies only the value in the search index, and, if the VARYING phrase is specified, the value in *index-name-1* or *identifier-2*. Therefore, to search an entire two-dimensional to seven-dimensional table, you must execute a SEARCH statement for each dimension. In the WHEN phrases, you must specify the indexes for all dimensions. Before the execution of each SEARCH statement, you must initialize the associated indexes with SET statements.

The SEARCH statement executes a serial search beginning at the current setting of the search index.

When the search begins, if the value of the index associated with *identifier-1* is not greater than the highest possible occurrence number, the following actions take place:

- The conditions in the WHEN phrase are evaluated in the order in which they are written.
- If none of the conditions is satisfied, the index for *identifier-1* is increased to correspond to the next table element, and step 1 is repeated.
- If upon evaluation one of the WHEN conditions is satisfied, the search is terminated immediately, and the imperative-statement-2 associated with that condition is executed. The index points to the table element that satisfied the condition. If NEXT SENTENCE is specified, control passes to the statement following the closest period.
- If the end of the table is reached (that is, the value of the incremented index is greater than the highest possible occurrence number) without the WHEN condition being satisfied, the search is terminated.

If, when the search begins, the value of the index-name associated with *identifier-1* is greater than the highest possible occurrence number, the search terminates immediately.

When the search terminates, if the AT END phrase is specified, imperative-statement-1 is executed. If the AT END phrase is omitted, control passes to the next statement after the SEARCH statement.

**Example: multidimensional serial search**
The following code fragment shows a search of the inner dimension (table C) in the third occurrence within the superordinate table (table R):

```
Working-storage section.
1 G.
  2 R occurs 10 indexed by Rindex.
  3 4 X C occurs 10 ascending key X indexed by Cindex.
  4 4 X pic 99.
  1 Arg pic 99 value 34.
Procedure division.
...  
* To search within occurrence 3 of table R, set its index to 3
* To search table C beginning at occurrence 1, set its index to 1
  Set Rindex to 3
  Set Cindex to 1
  * In the SEARCH statement, specify C without indexes
  * Search C
  * Specify indexes for both dimensions in the WHEN phrase
    when X(Rindex Cindex) = Arg
    display "Found " X(Rindex Cindex)
End-search
...  
VARYING phrase

index-name-1
One of the following actions applies:

- If index-name-1 is an index for identifier-1, this index is used for the search. Otherwise, the first (or only) index-name is used.
- If index-name-1 is an index for another table element, then the first (or only) index-name for identifier-1 is used for the search; the occurrence number represented by index-name-1 is increased by the same amount as the search index-name and at the same time.

When the VARYING index-name-1 phrase is omitted, the first (or only) index-name for identifier-1 is used for the search.

If indexing is used to search a table without an INDEXED BY phrase, correct results are ensured only if both the table defined with the index and the table defined without the index have table elements of the same length and with the same number of occurrences.

When the object of the VARYING phrase is an index-name for another table element, one serial SEARCH statement steps through two table elements at once.

identifier-2
Must be either an index data item or an elementary integer item. identifier-2 cannot be subscripted by the first (or only) index-name specified for identifier-1. During the search, one of the following actions applies:

- If identifier-2 is an index data item, then, whenever the search index is increased, the specified index data item is simultaneously increased by the same amount.
- If identifier-2 is an integer data item, then, whenever the search index is increased, the specified data item is simultaneously increased by 1.

WHEN phrase (serial search)

condition-1
Can be any condition described under “Conditional expressions” on page 246.

The following figure illustrates a format-1 SEARCH operation containing two WHEN phrases.
**Binary search**

The topic provides information of using the SEARCH statement for binary search.

*identifier-1* (binary search)

*identifier-1* identifies the table that is to be searched. *identifier-1* references all occurrences within that table.

The data description entry for *identifier-1* must contain an OCCURS clause with the INDEXED BY and KEY IS phrases.

*identifier-1* can reference a data item that is subordinate to a data item that contains an OCCURS clause (that is, *identifier-1* can be a subordinate table within a multidimensional table). In this case, the data description entry must specify an INDEXED BY phrase for each dimension of the table.

*identifier-1* must not be subscripted or reference-modified.

**AT END**

The condition that exists when the search operation terminates without satisfying the conditions specified in the WHEN phrase.

The SEARCH ALL statement executes a binary search. The index associated with *identifier-1* (the search index) need not be initialized by SET statements. The search index is varied during the search operation so that its value is at no time less than the value of the first table element, nor ever greater than the value
of the last table element. The index used is always that associated with the first index-name specified in the OCCURS clause.

Before using a binary search on a multidimensional table, you must execute SET statements to set the value of the index for each superordinate dimension.

The SEARCH statement modifies only the value in the search index. Therefore, to search an entire two-dimensional to seven-dimensional table, you must execute a SEARCH statement for each dimension. In the WHEN phrases, you must specify the indexes for all dimensions.

If the search ends without the WHEN condition being satisfied and the AT END phrase is specified, imperative-statement-1 is executed. If the AT END phrase is omitted, control passes to the next statement after the SEARCH statement.

The results of a SEARCH ALL operation are predictable only when:

- The data in the table is ordered in ASCENDING KEY or DESCENDING KEY order
- The contents of the ASCENDING or DESCENDING keys specified in the WHEN clause provide a unique table reference.

**WHEN phrase (binary search)**

If a relation condition is specified in the WHEN phrase, the evaluation of the relation is based on the USAGE of the data item referenced by data-name-1. The search argument is moved to a temporary data item with the same USAGE as data-name-1, and this temporary data item is used for the compare operations associated with the SEARCH.

If the WHEN phrase cannot be satisfied for any setting of the index within this range, the search is unsuccessful. Control is passed to imperative-statement-1 of the AT END phrase, when specified, or to the next statement after the SEARCH statement. In either case, the final setting of the index is not predictable.

If the WHEN phrase can be satisfied, control passes to imperative-statement-2, if specified, or to the next executable sentence if the NEXT SENTENCE phrase is specified. The index contains the value indicating the occurrence that allowed the WHEN conditions to be satisfied.

After imperative-statement-2 is executed, control passes to the end of the SEARCH statement, unless imperative-statement-2 ends with a GO TO statement.

**condition-name-1, condition-name-2**

Each condition-name specified must have only a single value, and each must be associated with an ASCENDING KEY or DESCENDING KEY data item for this table element.

**data-name-1, data-name-2**

Must specify an ASCENDING KEY or DESCENDING KEY data item in the table element referenced by identifier-1 and must be subscripted by the first index-name associated with identifier-1. Each data-name can be qualified.

- data-name-1 must be a valid operand for comparison with identifier-3, literal-1, or arithmetic-expression-1 according to the rules of comparison.
- data-name-2 must be a valid operand for comparison with identifier-4, literal-2, or arithmetic-expression-2 according to the rules of comparison.

- data-name-1 and data-name-2 cannot reference:
  - Floating-point data items
  - Group items containing variable-occurrence data items

**identifier-3, identifier-4**

Must not be an ASCENDING KEY or DESCENDING KEY data item for identifier-1 or an item that is subscripted by the first index-name for identifier-1.
identifier-3 and identifier-4 cannot be data items defined with any of the usages POINTER, FUNCTION-POINTER, PROCEDURE-POINTER, or OBJECT REFERENCE.

If identifier-3 or literal-1 is of class national, data-name-1 must be of class national.

If identifier-4 or literal-2 is of class national, data-name-2 must be of class national.

literal-1, literal-2

literal-1 or literal-2 must be a valid operand for comparison with data-name-1 or data-name-2, respectively.

arithmetic-expression

Can be any of the expressions defined under “Arithmetic expressions” on page 244, with the following restriction: Any identifier in arithmetic-expression must not be an ASCENDING KEY or DESCENDING KEY data item for identifier-1 or an item that is subscripted by the first index-name for identifier-1.

When an ASCENDING KEY or DESCENDING KEY data item is specified, explicitly or implicitly, in the WHEN phrase, all preceding ASCENDING KEY or DESCENDING KEY data-names for identifier-1 must also be specified.

Search statement considerations

The topic lists considerations of using the SEARCH statement.

Index data items cannot be used as subscripts, because of the restrictions on direct reference to them.

To ensure correct execution of a SEARCH statement for a variable-length table, make sure the object of the OCCURS DEPENDING ON clause (data-name-1) contains a value that specifies the current length of the table.

The scope of a SEARCH statement can be terminated by any of the following items:

• An END-SEARCH phrase at the same level of nesting
• A separator period
• An ELSE or END-IF phrase associated with a previous IF statement

SET statement

The SET statement is used to perform an operation as described in this topic.

The operations are:

• Placing values associated with table elements into indexes associated with index-names
• Incrementing or decrementing an occurrence number
• Setting the status of an external switch to ON or OFF
• Moving data to condition names to make conditions true
• Setting USAGE POINTER data items to a data address
• Setting USAGE PROCEDURE-POINTER data items to an entry address
• Setting USAGE FUNCTION-POINTER data items to an entry address
• Setting USAGE OBJECT REFERENCE data items to refer to an object instance
• Setting the length of dynamic-length elementary items

Index-names are related to a given table through the INDEXED BY phrase of the OCCURS clause; they are not further defined in the program.

When the sending and receiving fields in a SET statement share part of their storage (that is, the operands overlap), the result of the execution of that SET statement is undefined.
Format 1: SET for basic table handling
When this form of the SET statement is executed, the current value of the receiving field is replaced by the value of the sending field (with conversion).

Format 1: SET statement for basic table handling

index-name-1
Receiving field.
Must name an index that is specified in the INDEXED BY phrase of an OCCURS clause.

identifier-1
Receiving field.
Must name either an index data item or an elementary numeric integer item.

index-name-2
Sending field.
Must name an index that is specified in the INDEXED BY phrase of an OCCURS clause. The value of the index before the SET statement is executed must correspond to an occurrence number of its associated table.

identifier-2
Sending field.
Must name either an index data item or an elementary numeric integer item.

integer-1
Sending field.
Must be a positive integer.

The following table shows valid combinations of sending and receiving fields in a format-1 SET statement.

<table>
<thead>
<tr>
<th>Sending field</th>
<th>Index-name receiving field</th>
<th>Index data item receiving field</th>
<th>Integer data item receiving field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index-name*</td>
<td>Valid</td>
<td>Valid**</td>
<td>Valid</td>
</tr>
<tr>
<td>Index data item*</td>
<td>Valid**</td>
<td>Valid**</td>
<td>Invalid</td>
</tr>
<tr>
<td>Integer data item</td>
<td>Valid</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>Integer literal</td>
<td>Valid</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
</tbody>
</table>

*An index-name refers to an index named in the INDEXED BY phrase of an OCCURS clause. An index data item is defined with the USAGE IS INDEX clause.

**No conversion takes place.
Receiving fields are acted upon in the left-to-right order in which they are specified. Any subscripting or indexing associated with identifier-1 is evaluated immediately before that receiving field is acted upon.

The value used for the sending field is the value at the beginning of SET statement execution.

The value of an index after execution of a SEARCH or PERFORM statement can be undefined; therefore, use a format-1 SET statement to reinitialize such indexes before you attempt other table-handling operations.

If index-name-2 is for a table that has a subordinate item that contains an OCCURS DEPENDING ON clause, then undefined values can be received into identifier-1.

For more information about complex OCCURS DEPENDING ON, see Complex OCCURS DEPENDING ON in the Enterprise COBOL Programming Guide.

**Format 2: SET for adjusting indexes**

When this form of the SET statement is executed, the value of the receiving index is increased (UP BY) or decreased (DOWN BY) by a value that corresponds to the value in the sending field.

```
SET index-name-3 UP BY identifier-3 DOWN BY integer-2
```

The receiving field is an index specified by index-name-3. The index value both before and after the SET statement execution must correspond to an occurrence number in an associated table.

The sending field can be specified as identifier-3, which must be an elementary integer data item, or as integer-2, which must be a nonzero integer.

When the format-2 SET statement is executed, the contents of the receiving field are increased (UP BY) or decreased (DOWN BY) by a value that corresponds to the number of occurrences represented by the value of identifier-3 or integer-2. Receiving fields are acted upon in the left-to-right order in which they are specified. The value of the incrementing or decrementing field at the beginning of SET statement execution is used for all receiving fields.

If index-name-3 is for a table that has a subordinate item that contains an OCCURS DEPENDING ON clause, and if the ODO object is changed before executing a format-2 SET Statement, then index-name-3 cannot contain a value that corresponds to an occurrence number of its associated table.

For more information about complex OCCURS DEPENDING ON, see Complex OCCURS DEPENDING ON in the Enterprise COBOL Programming Guide.

**Format 3: SET for external switches**

When this form of the SET statement is executed, the status of each external switch associated with the specified mnemonic-name is turned ON or OFF.

```
SET mnemonic-name-1 TO ON OFF
```

Chapter 20. PROCEDURE DIVISION statements 403
**mnemonic-name-1**
Must be associated with an external switch, the status of which can be altered.

**Format 4: SET for condition-names**
When this form of the SET statement is executed, the value associated with a condition-name is placed in its conditional variable according to the rules of the VALUE clause.

```
Format 4: SET statement for condition-names

SET condition-name-1 TO TRUE
```

**condition-name-1**
Must be associated with a conditional variable.

If more than one literal is specified in the VALUE clause of `condition-name-1`, its associated conditional variable is set equal to the first literal.

If multiple condition-names are specified, the results are the same as if a separate SET statement had been written for each condition-name in the same order in which they are specified in the SET statement.

If SET `condition-name-1` TO FALSE is specified, there must be a corresponding WHEN SET TO FALSE phrase defined for `condition-name-1`.

**Related references**
Format 2 VALUE clause: condition-name value

**Format 5: SET for USAGE IS POINTER data items**
When this form of the SET statement is executed, the current value of the receiving field is replaced by the address value contained in the sending field.

```
Format 5: SET statement for USAGE IS POINTER or USAGE IS POINTER-32 data items

SET identifier-4 ADDRESS OF identifier-5 TO identifier-6 ADDRESS OF identifier-7 NULL
```

** identifier-4**
Receiving field(s).

Must be defined as USAGE IS POINTER or USAGE IS POITNER-32.

**ADDRESS OF identifier-5**
Receiving field(s).

`identifier-5` must be level-01 or level-77 items defined in the LINKAGE SECTION. The addresses of these items are set to the value of the operand specified in the TO phrase.
identifier-5 must not be reference-modified.

identifier-6

Sending field.

Must be defined as USAGE IS POINTER or USAGE IS POINTER-32.

ADDRESS OF identifier-7

Sending field.

identifier-7 must name an item of any level except 66 or 88 in the LINKAGE SECTION, the WORKING-STORAGE SECTION, or the LOCAL-STORAGE SECTION. It cannot name a dynamic-length elementary item or a dynamic-length group item. ADDRESS OF identifier-7 contains the address of the identifier, and not the content of the identifier.

NULL, NULLS

Sending field.

Sets the receiving field to contain the value of an invalid address.

The following table shows valid combinations of sending and receiving fields in a format-5 SET statement.

<table>
<thead>
<tr>
<th>Sending field</th>
<th>USAGE IS POINTER receiving field</th>
<th>USAGE IS POINTER-32 receiving field</th>
<th>ADDRESS OF receiving field</th>
<th>NULL/NULS receiving field</th>
</tr>
</thead>
<tbody>
<tr>
<td>USAGE IS POINTER</td>
<td>Valid</td>
<td>Valid</td>
<td>Valid</td>
<td>Invalid</td>
</tr>
<tr>
<td>USAGE IS POINTER-32</td>
<td>Valid</td>
<td>Valid</td>
<td>Valid</td>
<td>Invalid</td>
</tr>
<tr>
<td>ADDRESS OF</td>
<td>Valid</td>
<td>Valid</td>
<td>Valid</td>
<td>Invalid</td>
</tr>
<tr>
<td>NULL/NULS</td>
<td>Valid</td>
<td>Valid</td>
<td>Valid</td>
<td>Invalid</td>
</tr>
</tbody>
</table>

Format 6: SET for procedure-pointer and function-pointer data items

When this format of the SET statement is executed, the current value of the receiving field is replaced by the address value specified by the sending field.

At run time, function-pointers and procedure-pointers can reference the address of the primary entry point of a COBOL program, an alternate entry point in a COBOL program, or an entry point in a non-COBOL program; or they can be NULL.

COBOL function-pointers are more easily used than procedure-pointers for interoperation with C functions.
Format 6: SET statement for procedure-pointers and function-pointers

```
SET procedure-pointer-data-item-1 TO procedure-pointer-data-item-2
ENTRY identifier-8 literal-1 NULL
NULLS
pointer-data-item-3
```

**procedure-pointer-data-item-1, procedure-pointer-data-item-2**
Must be described as USAGE IS PROCEDURE-POINTER. procedure-pointer-data-item-1 is a receiving field; procedure-pointer-data-item-2 is a sending field.

**function-pointer-data-item-1, function-pointer-data-item-2**
Must be described as USAGE IS FUNCTION-POINTER. function-pointer-data-item-1 is a receiving field; function-pointer-data-item-2 is a sending field.

**identifier-8**
Must be defined as an alphabetic or alphanumeric item such that the value can be a program name. For more information, see “PROGRAM-ID paragraph” on page 96. For entry points in non-COBOL programs, identifier-8 can contain the characters @, #, and $.

**literal-1**
Must be alphanumeric and must conform to the rules for formation of program-names. For details on formation rules, see the discussion of program-name under “PROGRAM-ID paragraph” on page 96.

**identifier-8** or **literal-1** must refer to one of the following types of entry points:

- The primary entry point of a COBOL program as defined by the PROGRAM-ID paragraph. The PROGRAM-ID must reference the outermost program of a compilation unit; it must not reference a nested program.
- An alternate entry point of a COBOL program as defined by a COBOL ENTRY statement.
- An entry point in a non-COBOL program.

The program-name referenced by the SET ... TO ENTRY statement can be affected by the PGMNAME compiler option. For details, see PGMNAME in the Enterprise COBOL Programming Guide.

**NULL, NULLS**
Sets the receiving field to contain the value of an invalid address.

**pointer-data-item-3**
Must be defined with USAGE POINTER. You must set pointer-data-item-3 in a non-COBOL program to point to a valid program entry point.

**Example of COBOL/C interoperability**
The following example demonstrates a COBOL CALL to a C function that returns a function-pointer to a service, followed by a COBOL CALL to the service:

```
IDENTIFICATION DIVISION.
PROGRAM-ID DEMO.
DATA DIVISION.
WORKING-STORAGE SECTION.
```
Format 7: SET for USAGE OBJECT REFERENCE data items

When this format of the SET statement is executed, the value in the receiving item is replaced by the value in the sending item.

```
Format 7: SET statement for object references

<table>
<thead>
<tr>
<th>SET</th>
<th>object-reference-id-1</th>
<th>TO</th>
<th>object-reference-id-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SELF</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

`object-reference-id-1` and `object-reference-id-2` must be defined as USAGE OBJECT REFERENCE. `object-reference-id-1` is the receiving item and `object-reference-id-2` is the sending item. If `object-reference-id-1` is defined as an object reference of a certain class (defined as “USAGE OBJECT REFERENCE class-name”), `object-reference-id-2` must be an object reference of the same class or a class derived from that class.

If the figurative constant NULL is specified, the receiving `object-reference-id-1` is set to the NULL value.

If SELF is specified, the SET statement must appear in the PROCEDURE DIVISION of a method. `object-reference-id-1` is set to reference the object upon which the currently executing method was invoked.

Format 8: SET for length of dynamic-length elementary items

When this format of the SET statement is executed, the length of the dynamic-length elementary item in the receiver is set to the value of the sending item.

```
Format 8: SET for length of dynamic-length elementary items

<table>
<thead>
<tr>
<th>SET</th>
<th>LENGTH OF</th>
<th>identifier-1</th>
<th>TO</th>
<th>identifier-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

`identifier-1` must be a dynamic-length elementary item.

`identifier-2` must be an elementary numeric integer item.

`integer-1` must be an integer greater than or equal to zero.

If `identifier-2` is less than zero then the length of `identifier-1` is set to zero.

If `identifier-2` or `integer-1` is less than the current length of `identifier-1`, then the length of `identifier-1` is set to the specified value and `identifier-1` is truncated on the right.

If `identifier-2` or `integer-1` is longer than the current length of `identifier-1`, then the length of `identifier-1` is set to the specified value, no padding occurs, and the newly available character positions on the right are not initialized.

**SORT statement**

The SORT statement causes a set of records or table elements to be arranged in a user-specified sequence.

For sorting files, the SORT statement accepts records from one or more files, sorts them according to the specified keys, and makes the sorted records available either through an output procedure or in an output file.
For sorting tables, the SORT statement sorts table elements according to specified table keys.

Format 1: SORT statement

- SORT file-name-1
- ON ASCENDING DESCENDING KEY data-name-1
- WITH DUPLICTATES IN ORDER
- COLLATING SEQUENCE IS alphabet-name-1
- USING file-name-2
- INPUT PROCEDURE IS procedure-name-1 THROUGH procedure-name-2
- GIVING file-name-3
- OUTPUT PROCEDURE IS procedure-name-3 THROUGH procedure-name-4

Format 1 SORT statements can appear anywhere in the PROCEDURE DIVISION except in the declarative portion. This format of the SORT statement is not supported for programs that are compiled with the THREAD option. See also “MERGE statement” on page 358.
Format 2: Table SORT statement

Format 2 SORT statements can appear anywhere in the PROCEDURE DIVISION. This format of the SORT statement can be used with programs that are compiled with the THREAD option.

**file-name-1**

The name given in the SD entry that describes the records to be sorted.

No pair of file-names in a SORT statement can be specified in the same SAME SORT AREA clause or the SAME SORT-MERGE AREA clause. File-names associated with the GIVING clause (file-name-3, ...) cannot be specified in the SAME AREA clause; however, they can be associated with the SAME RECORD AREA clause.

**data-name-2**

Specifies a table data-name that is subject to the following rules:

- **data-name-2** must have an OCCURS clause in the data description entry.
- **data-name-2** can be qualified.
- **data-name-2** can be subscripted. The rightmost or only subscript of the table must be omitted or replaced with the word ALL.

The number of occurrences of table elements that are referenced by **data-name-2** is determined by the rules in the OCCURS clause. The sorted table elements are placed in the same table that is referenced by **data-name-2**.

**ASCENDING KEY and DESCENDING KEY phrases (format 1)**

This phrase specifies that records are to be processed in ascending or descending sequence (depending on the phrase specified), based on the specified sort keys.

**data-name-1**

Specifies a KEY data item on which the SORT statement will be based. Each such data-name must identify a data item in a record associated with **file-name-1**. The data-names following the word KEY are listed from left to right in the SORT statement in order of decreasing significance without regard to how they are divided into KEY phrases. The leftmost data-name is the major key, the next data-name is the next most significant key, and so forth. The following rules apply:

- A specific KEY data item must be physically located in the same position and have the same data format in each input file. However, it need not have the same data-name.
- If **file-name-1** has more than one record description, the KEY data items need be described in only one of the record descriptions.
• If `file-name-1` contains variable-length records, all of the KEY data-items must be contained within
  the first $n$ character positions of the record, where $n$ equals the minimum records size specified for
  `file-name-1`.
• KEY data items must not contain an OCCURS clause or be subordinate to an item that contains an
  OCCURS clause.
• KEY data items cannot be:
  – Variably located
  – Group items that contain variable-occurrence data items
  – Category numeric described with usage NATIONAL (national decimal item)
  – Category external floating-point described with usage NATIONAL (national floating-point item)
  – Category DBCS
  – Dynamic-length elementary items
  – Dynamic-length group items
• KEY data items can be qualified.
• KEY data items can belong to any of the following data categories:
  – Alphabetic, alphanumeric, alphanumeric-edited
  – Numeric (except numeric with usage NATIONAL)
  – Numeric-edited (with usage DISPLAY or NATIONAL)
  – Internal floating-point or display floating-point
  – National or national-edited

If `file-name-3` references an indexed file, the first specification of `data-name-1` must be associated with
an ASCENDING phrase and the data item referenced by that `data-name-1` must occupy the same
character positions in this record as the data item associated with the prime record key for that file.

The direction of the sorting operation depends on the specification of the ASCENDING or DESCENDING
keywords as follows:
• When ASCENDING is specified, the sequence is from the lowest key value to the highest key value.
• When DESCENDING is specified, the sequence is from the highest key value to the lowest.
• If the KEY data item is described with usage NATIONAL, the sequence of the KEY values is based on the
  binary values of the national characters.
• If the KEY data item is internal floating point, the sequence of key values will be in numeric order.
• When the COLLATING SEQUENCE phrase is not specified, the key comparisons are performed according
to the rules for comparison of operands in a relation condition. See “General relation conditions” on
  page 250.
• When the COLLATING SEQUENCE phrase is specified, the indicated collating sequence is used for key
data items of alphabetic, alphanumeric, alphanumeric-edited, external floating-point, and numeric-
edited categories. For all other key data items, the comparisons are performed according to the rules
for comparison of operands in a relation condition.

**ASCENDING KEY and DESCENDING KEY phrases (format 2)**

This phrase specifies that table elements are to be processed in ascending or descending sequence,
based on the specified phrase and sort keys.

`data-name-1` Specifies a KEY data name that is subject to the following rules:
• The data item that is identified by a key data-name must be the same as, or subordinate to, the data
  item that is referenced by `data-name-2`.
• KEY data items can be qualified.
• KEY data items can belong to any of the following data categories:
  – Alphabetic, alphanumeric, alphanumeric-edited
  – Numeric (except numeric with usage NATIONAL)
  – Numeric-edited (with usage DISPLAY or NATIONAL)
  – Internal floating-point or display floating-point
  – National or national-edited

• KEY data items cannot be:
  – Variably located
  – Group items that contain variable-occurrence data items
  – Category numeric that is described with usage NATIONAL (national decimal item)
  – Category external floating-point that is described with usage NATIONAL (national floating-point item)
  – Category DBCS
  – Class object or pointer
  – USAGE OBJECT, USAGE POINTER, USAGE PROCEDURE-POINTER, or USAGE FUNCTION-POINTER
  – Subscripted
  – Dynamic-length elementary items
  – Dynamic-length group items

• If the data item that is identified by a KEY data-name is subordinate to data-name-2, the following rules apply:
  – The data item cannot be described with an OCCURS clause.
  – The data item cannot be subordinate to an entry that is also subordinate to data-name-2 and that contains an OCCURS clause.

The KEY phrase can be omitted only if the description of the table that is referenced by data-name-2 contains a KEY phrase.

The words ASCENDING and DESCENDING are transitive across all occurrences of data-name-1 until another word ASCENDING or DESCENDING is encountered.

The data items that are referenced by data-name-1 are key data items, and these data items determine the order in which the sorted table elements are stored. The order of significance of the keys is the order in which data items are specified in the SORT statement, without regard to the association with ASCENDING or DESCENDING phrases.

The SORT statement sorts the table that is referenced by data-name-2 and presents the sorted table in data-name-2. The sorting order is determined by either the ASCENDING and DESCENDING phrases (if specified), or by the KEY phrase that is associated with data-name-2.

The direction of the sorting operation depends on the specification of the ASCENDING or DESCENDING keywords:

• When ASCENDING is specified, the sequence is from the lowest key value to the highest one.
• When DESCENDING is specified, the sequence is from the highest key value to the lowest one.
• If the KEY data item is described with usage NATIONAL, the sequence of the KEY values is based on the binary values of the national characters.
• If the KEY data item is internal floating-point, the sequence of key values is in the numeric order.
• When the COLLATING SEQUENCE phrase is not specified, the EBCDIC sequence is used for key data items of alphabetic, alphanumeric, alphanumeric-edited, external floating-point, and numeric-edited categories. For all the other key data items, the comparisons are performed according to the rules for comparison of operands in a relation condition.
• When the COLLATING SEQUENCE phrase is specified, the indicated collating sequence is used for key data items of alphabetic, alphanumeric, alphanumeric-edited, external floating-point, and numeric-edited categories. For all the other key data items, the comparisons are performed according to the rules for comparison of operands in a relation condition.

To determine the relative order in which table elements are stored, the contents of corresponding key data items are compared according to the rules for comparison of operands in a relation condition. The sorting starts with the most significant key data item with the following rules:

• If the contents of the corresponding key data items are not equal and the key is associated with the ASCENDING phrase, the table element that contains the key data item with the lower value has the lower occurrence number.

• If the contents of the corresponding key data items are not equal and the key is associated with the DESCENDING phrase, the table element that contains the key data item with the higher value has the lower occurrence number.

• If the contents of the corresponding key data items are equal, the determination is based on the contents of the next most significant key data item.

If the KEY phrase is not specified, the sequence is determined by the KEY phrase in the data description entry of the table that is referenced by data-name-2.

If the KEY phrase is specified, it overrides any KEY phrase specified in the data description entry of the table that is referenced by data-name-2.

If data-name-1 is omitted, the data item that is referenced by data-name-2 is the key data item.

DUPLICATES phrase (format 1)

If the DUPLICATES phrase is specified, and the contents of all the key elements associated with one record are equal to the corresponding key elements in one or more other records, the order of return of these records is as follows:

• The order of the associated input files as specified in the SORT statement. Within a given file the order is that in which the records are accessed from that file.

• The order in which these records are released by an input procedure, when an input procedure is specified.

If the DUPLICATES phrase is not specified, the order of these records is undefined.

DUPLICATES phrase (format 2)

When both of the following conditions are met, the contents of table elements are in the relative order that is the same as the order before sorting operation:

• The DUPLICATES phrase is specified.

• The contents of all the key data items that are associated with one table element are equal to the contents of corresponding key data items that are associated with one or more other table elements.

If the DUPLICATES phrase is not specified and the second condition exists, the relative order of the contents of these table elements is undefined.

COLLATING SEQUENCE phrase (both formats)

This phrase specifies the collating sequence to be used in alphanumeric comparisons for the KEY data items in this sorting operation.

The COLLATING SEQUENCE phrase has no effect for keys that are not alphabetic or alphanumeric.
alphabet-name-1
Must be specified in the ALPHABET clause of the SPECIAL-NAMES paragraph. alphabet-name-1 can be associated with any one of the ALPHABET clause phrases, with the following results:

STANDARD-1
The ASCII collating sequence is used for all alphanumeric comparisons. (The ASCII collating sequence is shown in Appendix C, “EBCDIC and ASCII collating sequences,” on page 579.)

STANDARD-2
The International Reference Version of ISO/IEC 646, 7-bit coded character set for information processing interchange is used for all alphanumeric comparisons.

NATIVE
The EBCDIC collating sequence is used for all alphanumeric comparisons. (The EBCDIC collating sequence is shown in Appendix C, “EBCDIC and ASCII collating sequences,” on page 579.)

EBCDIC
The EBCDIC collating sequence is used for all alphanumeric comparisons. (The EBCDIC collating sequence is shown in Appendix C, “EBCDIC and ASCII collating sequences,” on page 579.)

literal
The collating sequence established by the specification of literals in the alphabet-name clause is used for all alphanumeric comparisons.

When the COLLATING SEQUENCE phrase is omitted, the PROGRAM COLLATING SEQUENCE clause (if specified) in the OBJECT-COMPUTER paragraph specifies the collating sequence to be used. When both the COLLATING SEQUENCE phrase and the PROGRAM COLLATING SEQUENCE clause are omitted, the EBCDIC collating sequence is used.

USING phrase

file-name-2, ...
The input files.

When the USING phrase is specified, all the records in file-name-2, ..., (that is, the input files) are transferred automatically to file-name-1. At the time the SORT statement is executed, these files must not be open. The compiler opens, reads, makes records available, and closes these files automatically. If EXCEPTION/ERROR procedures are specified for these files, the compiler makes the necessary linkage to these procedures.

All input files must be described in FD entries in the DATA DIVISION.

If the USING phrase is specified and if file-name-1 contains variable-length records, the size of the records contained in the input files (file-name-2, ...) must be neither less than the smallest record nor greater than the largest record described for file-name-1. If file-name-1 contains fixed-length records, the size of the records contained in the input files must not be greater than the largest record described for file-name-1. For more information, see Describing the input to sorting or merging in the Enterprise COBOL Programming Guide.

INPUT PROCEDURE phrase
This phrase specifies the name of a procedure that is to select or modify input records before the sorting operation begins.

procedure-name-1
Specifies the first (or only) section or paragraph in the input procedure.

procedure-name-2
Identifies the last section or paragraph of the input procedure.

The input procedure can consist of any procedure needed to select, modify, or copy the records that are made available one at a time by the RELEASE statement to the file referenced by file-name-1. The
range includes all statements that are executed as the result of a transfer of control by CALL, EXIT, GO TO, PERFORM, and XML PARSE statements in the range of the input procedure, as well as all statements in declarative procedures that are executed as a result of the execution of statements in the range of the input procedure. The range of the input procedure must not cause the execution of any MERGE, RETURN, or format 1 SORT statement.

If an input procedure is specified, control is passed to the input procedure before the file referenced by file-name-1 is sequenced by the SORT statement. The compiler inserts a return mechanism at the end of the last statement in the input procedure. When control passes the last statement in the input procedure, the records that have been released to the file referenced by file-name-1 are sorted.

**GIVING phrase**

file-name-3, ...

The output files. When the GIVING phrase is specified, all the sorted records in file-name-1 are automatically transferred to the output files (file-name-3, ...).

All output files must be described in FD entries in the DATA DIVISION.

If the output files (file-name-3, ...) contain variable-length records, the size of the records contained in file-name-1 must be neither less than the smallest record nor greater than the largest record described for the output files. If the output files contain fixed-length records, the size of the records contained in file-name-1 must not be greater than the largest record described for the output files. For more information, see Describing the output from sorting or merging in the Enterprise COBOL Programming Guide.

At the time the SORT statement is executed, the output files (file-name-3, ...) must not be open. For each of the output files, the execution of the SORT statement causes the following actions to be taken:

- The processing of the file is initiated. The initiation is performed as if an OPEN statement with the OUTPUT phrase had been executed.
- The sorted logical records are returned and written onto the file. Each record is written as if a WRITE statement without any optional phrases had been executed.

For a relative file, the relative key data item for the first record returned contains the value '1'; for the second record returned, the value '2'. After execution of the SORT statement, the content of the relative key data item indicates the last record returned to the file.

- The processing of the file is terminated. The termination is performed as if a CLOSE statement without optional phrases had been executed.

These implicit functions are performed such that any associated USE AFTER EXCEPTION/ERROR procedures are executed; however, the execution of such a USE procedure must not cause the execution of any statement manipulating the file referenced by, or accessing the record area associated with, file-name-3. On the first attempt to write beyond the externally defined boundaries of the file, any USE AFTER STANDARD EXCEPTION/ERROR procedure specified for the file is executed. If control is returned from that USE procedure or if no such USE procedure is specified, the processing of the file is terminated.

**OUTPUT PROCEDURE phrase**

This phrase specifies the name of a procedure that is to select or modify output records from the sorting operation.

procedure-name-3

Specifies the first (or only) section or paragraph in the output procedure.

procedure-name-4

Identifies the last section or paragraph of the output procedure.
The output procedure can consist of any procedure needed to select, modify, or copy the records that are made available one at a time by the RETURN statement in sorted order from the file referenced by file-name-1. The range includes all statements that are executed as the result of a transfer of control by CALL, EXIT, GOTO, PERFORM, and XML PARSE statements in the range of the output procedure. The range also includes all statements in declarative procedures that are executed as a result of the execution of statements in the range of the output procedure. The range of the output procedure must not cause the execution of any MERGE, RELEASE, or format 1 SORT statement.

If an output procedure is specified, control passes to it after the file referenced by file-name-1 has been sequenced by the SORT statement. The compiler inserts a return mechanism at the end of the last statement in the output procedure and when control passes the last statement in the output procedure, the return mechanism provides the termination of the sort and then passes control to the next executable statement after the SORT statement. Before entering the output procedure, the sort procedure reaches a point at which it can select the next record in sorted order when requested. The RETURN statements in the output procedure are the requests for the next record.

The INPUT PROCEDURE and OUTPUT PROCEDURE phrases are similar to those for a basic PERFORM statement. For example, if you name a procedure in an output procedure, that procedure is executed during the sorting operation just as if it were named in a PERFORM statement. As with the PERFORM statement, execution of the procedure is terminated after the last statement completes execution. The last statement in an input or output procedure can be the EXIT statement (see “EXIT statement” on page 316).

SORT special registers
The special registers, SORT-CORE-SIZE, SORT-MESSAGE, and SORT-MODE-SIZE, are equivalent to option control statement keywords in the sort control file. You define the sort control data set with the SORT-CONTROL special register.

Usage notes:
- The SORT special registers are not applicable to sorting a table by using the format 2 SORT statement.
- If you use a sort control file to specify control statements, the values specified in the sort control file take precedence over those in the special register.

SORT-MESSAGE special register
See “SORT-MESSAGE” on page 23.

SORT-CORE-SIZE special register
See “SORT-CORE-SIZE” on page 22.

SORT-FILE-SIZE special register
See “SORT-FILE-SIZE” on page 22.

SORT-MODE-SIZE special register
See “SORT-MODE-SIZE” on page 23.

SORT-CONTROL special register
See “SORT-CONTROL” on page 22.

SORT-RETURN special register
See “SORT-RETURN” on page 23.

Segmentation considerations
The topic lists considerations of using the SORT statement.

If a SORT statement is coded in a fixed segment, any input or output procedure referenced by that SORT statement must be either totally within a fixed segment or wholly contained in a single independent segment.

If a SORT statement is coded in an independent segment, any input or output procedure referenced by that SORT statement must be either totally within a fixed segment or wholly contained within the same independent segment as that SORT statement.
**START statement**

The START statement provides a means of positioning within an indexed or relative file for subsequent sequential record retrieval.

When the START statement is executed, the associated indexed or relative file must be open in either INPUT or I-O mode.

**Format**

```
  START   file-name-1
```

**file-name-1**

Must name a file with sequential or dynamic access. *file-name-1* must be defined in an FD entry in the DATA DIVISION and must not name a sort file.

**KEY phrase**

When the KEY phrase is specified, the file position indicator is positioned at the logical record in the file whose key field satisfies the comparison.

When the KEY phrase is not specified, KEY IS EQUAL (to the prime record key) is implied.

**data-name-1**

Can be qualified; it cannot be subscripted.

When the START statement is executed, a comparison is made between the current value in the key data-name and the corresponding key field in the file's index.

If the FILE STATUS clause is specified in the file-control entry, the associated file status key is updated when the START statement is executed (See “File status key” on page 276).

---

416 Enterprise COBOL for z/OS: Enterprise COBOL for z/OS, V6.3 Language Reference
**INVALID KEY phrases**

If the comparison is not satisfied by any record in the file, an invalid key condition exists; the position of the file position indicator is undefined, and (if specified) the INVALID KEY imperative-statement is executed. (See “INTO and FROM phrases” on page 281 under “Common processing facilities”.)

Both the INVALID KEY phrase and an applicable EXCEPTION/ERROR procedure can be omitted.

**END-START phrase**

This explicit scope terminator serves to delimit the scope of the START statement. END-START permits a conditional START statement to be nested in another conditional statement. END-START can also be used with an imperative START statement.

For more information, see “Delimited scope statements” on page 270.

**Indexed files**

When the KEY phrase is specified, the key data item used for the comparison is *data-name-1*.

When the KEY phrase is not specified, the key data item used for the EQUAL TO comparison is the prime record key.

When START statement execution is successful, the RECORD KEY or ALTERNATE RECORD KEY with which *data-name-1* is associated becomes the key of reference for subsequent READ statements.

*data-name-1*

Can be any of the following items:

- The prime RECORD KEY.
- Any ALTERNATE RECORD KEY.
- A data item within a record description for a file whose leftmost character position corresponds to the leftmost character position of that record key; it can be qualified. The size of the data item must be less than or equal to the length of the record key for the file.

Regardless of its category, *data-name-1* is treated as an alphanumeric item for purposes of the comparison operation.

**Note:** If your key is numeric, you must specify the EQUAL TO condition, otherwise, unexpected results can happen.

The file position indicator points to the first record in the file whose key field satisfies the comparison. If the operands in the comparison are of unequal lengths, the comparison proceeds as if the longer field were truncated on the right to the length of the shorter field. All other numeric and alphanumeric comparison rules apply, except that the PROGRAM COLLATING SEQUENCE clause, if specified, has no effect.

When START statement execution is successful, the RECORD KEY with which *data-name-1* is associated becomes the key of reference for subsequent READ statements.

When START statement execution is unsuccessful, the key of reference is undefined.

**Relative files**

When the KEY phrase is specified, *data-name-1* must specify the RELATIVE KEY.

Whether or not the KEY phrase is specified, the key data item used in the comparison is the RELATIVE KEY data item. Numeric comparison rules apply.

The file position indicator points to the logical record in the file whose key satisfies the specified comparison.
STOP statement

The STOP statement halts execution of the object program either permanently or temporarily.

**Format**

```
STOP literal
RUN
```

**literal**

Can be a fixed-point numeric literal (signed or unsigned) or an alphanumeric literal. It can be any figurative constant except ALL literal.

When STOP literal is specified, the literal is communicated to the operator, and object program execution is suspended. Program execution is resumed only after operator intervention, and continues at the next executable statement in sequence.

The STOP literal statement is useful for special situations when operator intervention is needed during program execution; for example, when a special tape or disk must be mounted or a specific daily code must be entered. However, the ACCEPT and DISPLAY statements are preferred when operator intervention is needed.

Do not use the STOP literal statement in programs compiled with the THREAD compiler option.

When STOP RUN is specified, execution is terminated and control is returned to the system. When STOP RUN is not the last or only statement in a sequence of imperative statements within a sentence, the statements following STOP RUN are not executed.

The STOP RUN statement closes all files defined in any of the programs in the run unit.

The STOP RUN statement frees all allocated buffers for dynamic-length elementary items in all of the programs in the run unit.

For use of the STOP RUN statement in calling and called programs, see the following table.

<table>
<thead>
<tr>
<th>Termination statement</th>
<th>Main program</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>STOP RUN</td>
<td>Returns to the calling program. (Can be the system, which causes the application to end.)</td>
<td>Returns directly to the program that called the main program. (Can be the system, which causes the application to end.)</td>
</tr>
</tbody>
</table>

STRING statement

The STRING statement strings together the partial or complete contents of two or more data items or literals into one single data item.

One STRING statement can be written instead of a series of MOVE statements.
id1, lit1

Represents the sending fields.

DELIMITED BY phrase
Sets the limits of the string.

id2, lit2

Are delimiters; that is, characters that delimit the data to be transferred.

SIZE
Transfers the complete sending area.

INTO phrase
Identifies the receiving field.

id3

Represents the receiving field.

POINTER phrase
Points to a character position in the receiving field. The pointer field indicates a relative alphanumeric character position, DBCS character position, or national character position when the receiving field is of usage DISPLAY, DISPLAY-1, or NATIONAL, respectively.

id4

Represents the pointer field. id4 must be large enough to contain a value equal to the length of the receiving field plus 1. You must initialize id4 to a nonzero value before execution of the STRING statement begins.

When the POINTER phrase is specified, an explicit pointer field is available to control placement of data in the receiving field. It is required to set the explicit pointer's initial value, which must greater than or equal to 1. For fixed-length data items, the pointer’s initial value must less than or equal to the character position count of the receiving field. For dynamic-length elementary items, the pointer's
initial value must be less than or equal to the specified value of the LIMIT phrase on the item's data description entry, or the default limit if no LIMIT is specified.

When the POINTER phrase is specified and the receiving field is a dynamic-length elementary item, the value of the POINTER field may be greater than the length of the receiver. When the value of the POINTER field is equal to the length of the receiver plus one, the STRING statement will effectively concatenate the sending fields to the receiver. When the value of the POINTER field is equal to the length of the receiver plus two or more, the intermediate character positions between the end of the receiver and the beginning of the POINTER field will be padded with spaces, and the sending fields will be concatenated starting at the POINTER field position.

The following rules apply:

- All identifiers except identifier-4 must reference data items described explicitly or implicitly as usage DISPLAY, DISPLAY-1, or NATIONAL.
- All identifiers cannot be dynamic-length group items.
- literal-1 or literal-2 must be of category alphanumeric, DBCS, or national and can be any figurative constant that does not begin with the word ALL (except NULL).
- If identifier-1 or identifier-2 references a data item of category numeric, each numeric item must be described as an integer without the symbol 'P' in its PICTURE character-string.
- identifier-3 must not reference a data item of category numeric-edited, alphanumeric-edited, or national-edited; an external floating-point data item of usage DISPLAY, or an external floating-point data item of usage NATIONAL.
- identifier-3 must not be described with the JUSTIFIED clause.
- If identifier-3 is of usage DISPLAY, identifier-1 and identifier-2 must be of usage DISPLAY and all literals must be alphanumeric literals. Any figurative constant can be specified except one that begins with the word ALL. Each figurative constant represents a 1-character alphanumeric literal.
- If identifier-3 is of usage DISPLAY-1, identifier-1 and identifier-2 must be of usage DISPLAY-1 and all literals must be DBCS literals. The only figurative constant that can be specified is SPACE, which represents a 1-character DBCS literal. ALL DBCS-literal must not be specified.
- If identifier-3 is of usage NATIONAL, identifier-1 and identifier-2 must be of usage NATIONAL and all literals must be national literals. Any figurative constant can be specified except symbolic-character and one that begins with the word ALL. Each figurative constant represents a 1-character national literal.
- If identifier-1 or identifier-2 references an elementary data item of usage DISPLAY that is described as category numeric, numeric-edited, or alphanumeric-edited, the item is treated as if it were redefined as category alphanumeric.
- If identifier-1 or identifier-2 references an elementary data item of usage NATIONAL that is described as category numeric, numeric-edited, or national-edited item, the item is treated as if it were redefined as category national.
- identifier-4 must not be described with the symbol P in its PICTURE character-string.

Evaluation of subscripts, reference modification, variable-lengths, variable locations, and function-identifiers is performed only once, at the beginning of the execution of the STRING statement. Therefore, if identifier-3 or identifier-4 is used as a subscript, reference-modifier, or function argument in the STRING statement, or affects the length or location of any of the identifiers in the STRING statement, the values calculated for those subscripts, reference-modifiers, variable lengths, variable locations, and functions are not affected by any results of the STRING statement.

If identifier-3 and identifier-4 occupy the same storage area, undefined results will occur, even if the identifiers are defined by the same data description entry.

If identifier-1 or identifier-2 occupies the same storage area as identifier-3 or identifier-4, undefined results will occur, even if the identifiers are defined by the same data description entry.

See “Data flow” on page 421 for details of STRING statement processing.
ON OVERFLOW phrases

**imperative-statement-1**

Executed when the pointer value (explicit or implicit):

- Is less than 1
- Exceeds a value equal to the length of the receiving field

When either of the above conditions occurs, an overflow condition exists, and no more data is transferred. Then the STRING operation is terminated, the NOT ON OVERFLOW phrase, if specified, is ignored, and control is transferred to the end of the STRING statement or, if the ON OVERFLOW phrase is specified, to *imperative-statement-1*.

If control is transferred to *imperative-statement-1*, execution continues according to the rules for each statement specified in *imperative-statement-1*. If a procedure branching or conditional statement that causes explicit transfer of control is executed, control is transferred according to the rules for that statement; otherwise, upon completion of the execution of *imperative-statement-1*, control is transferred to the end of the STRING statement.

If at the time of execution of a STRING statement, conditions that would cause an overflow condition are not encountered, then after completion of the transfer of data, the ON OVERFLOW phrase, if specified, is ignored. Control is then transferred to the end of the STRING statement, or if the NOT ON OVERFLOW phrase is specified, to *imperative-statement-2*.

If control is transferred to *imperative-statement-2*, execution continues according to the rules for each statement specified in *imperative-statement-2*. If a procedure branching or conditional statement that causes explicit transfer of control is executed, control is transferred according to the rules for that statement. Otherwise, upon completion of the execution of *imperative-statement-2*, control is transferred to the end of the STRING statement.

**END-STRING phrase**

This explicit scope terminator serves to delimit the scope of the STRING statement. END-STRING permits a conditional STRING statement to be nested in another conditional statement. END-STRING can also be used with an imperative STRING statement.

For more information, see “Delimited scope statements” on page 270.

**Data flow**

When the STRING statement is executed, characters are transferred from the sending fields to the receiving field. The order in which sending fields are processed is the order in which they are specified.

The following rules apply:

- Characters from the sending fields are transferred to the receiving fields in the following manner:
  - For national sending fields, data is transferred using the rules of the MOVE statement for elementary national-to-national moves, except that no space filling takes place.
  - For DBCS sending fields, data is transferred using the rules of the MOVE statement for elementary DBCS-to-DBCS moves, except that no space filling takes place.
  - Otherwise, data is transferred to the receiving fields using the rules of the MOVE statement for elementary alphanumeric-to-alphanumeric moves, except that no space filling takes place (see “MOVE statement” on page 363).

- When DELIMITED BY *identifier-2* or literal-2 is specified, the contents of each sending item are transferred, character-by-character, beginning with the leftmost character position and continuing until either:
  - A delimiter for this sending field is reached (the delimiter itself is not transferred).
  - The rightmost character of this sending field has been transferred.
• When DELIMITED BY SIZE is specified, each entire sending field is transferred to the receiving field.
• When the receiving field is filled, or when all the sending fields have been processed, the operation is ended.
• When the POINTER phrase is specified, an explicit pointer field is available to the COBOL user to control placement of data in the receiving field. The user must set the explicit pointer's initial value, which must not be less than 1 and not more than the character position count of the receiving field.

Usage note: The pointer field must be defined as a field large enough to contain a value equal to the length of the receiving field plus 1; this precludes arithmetic overflow when the system updates the pointer at the end of the transfer.

• When the POINTER phrase is not specified, no pointer is available to the user. However, a conceptual implicit pointer with an initial value of 1 is used by the system.
• Conceptually, when the STRING statement is executed, the initial pointer value (explicit or implicit) is the first character position within the receiving field into which data is to be transferred. Beginning at that position, data is then positioned, character-by-character, from left to right. After each character is positioned, the explicit or implicit pointer is increased by 1. The value in the pointer field is changed only in this manner. At the end of processing, the pointer value always indicates a value equal to one character position beyond the last character transferred into the receiving field.

After STRING statement execution is completed, only that part of the receiving field into which data was transferred is changed. The rest of the receiving field contains the data that was present before this execution of the STRING statement.

Example of the STRING statement
This topic lists an example for the STRING statement.

When the following STRING statement is executed, the results obtained will be like those illustrated in the figure after the statement.

```
STRING ID-1 ID-2 DELIMITED BY ID-3
   ID-4 ID-5 DELIMITED BY SIZE
   INTO ID-7 WITH POINTER ID-8
END-STRING
```

<table>
<thead>
<tr>
<th>ID-1 at execution</th>
<th>ID-2 at execution</th>
<th>ID-4 at execution</th>
<th>ID-5 at execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 * 4 5</td>
<td>A * B C</td>
<td>6 7 8 9 * 0</td>
<td>D E * F G</td>
</tr>
</tbody>
</table>

First group of characters moved

<table>
<thead>
<tr>
<th>ID-3 (delimiter) at execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
</tr>
</tbody>
</table>

Second group of characters moved

<table>
<thead>
<tr>
<th>ID-7 after execution (initialized to ALL Z before execution)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 A 6 7 8 9 * 0 D E * F G Z Z Z Z Z Z</td>
</tr>
</tbody>
</table>

Third group of characters moved

<table>
<thead>
<tr>
<th>ID-8 (pointer) after execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 6</td>
</tr>
</tbody>
</table>

Fourth group of characters moved

(initialized to 01 before execution)
The SUBTRACT statement subtracts one numeric item, or the sum of two or more numeric items, from one or more numeric items, and stores the result.

Format 1: SUBTRACT statement

All identifiers or literals preceding the keyword FROM are added together and their sum is subtracted from and stored immediately in identifier-2. This process is repeated for each successive occurrence of identifier-2, in the left-to-right order in which identifier-2 is specified.
All identifiers or literals preceding the keyword FROM are added together and their sum is subtracted from identifier-2 or literal-2. The result of the subtraction is stored as the new value of each data item referenced by identifier-3.

Elementary data items within identifier-1 are subtracted from, and the results are stored in, the corresponding elementary data items within identifier-2.

When the ARITH(COMPAT) compiler option is in effect, the composite of operands can contain a maximum of 30 digits. When the ARITH(EXTEND) compiler option is in effect, the composite of operands can contain a maximum of 31 digits. For more information about arithmetic intermediate results, see Appendix A. Intermediate results and arithmetic precision in the Enterprise COBOL Programming Guide.

For all formats:
**identifier**
In format 1, must name an elementary numeric data item.

In format 2, must name an elementary numeric data item, unless the identifier follows the word GIVING. Each identifier following the word GIVING must name a numeric or numeric-edited elementary data item.

In format 3, must name an alphanumeric group item or a national group item.

**literal**
Must be a numeric literal.

Floating-point data items and literals can be used anywhere numeric data items and literals can be specified.

**ROUNDED phrase**
For information about the ROUNDED phrase, and for operand considerations, see “ROUNDED phrase” on page 273.

**SIZE ERROR phrases**
For information about the SIZE ERROR phrases, and for operand considerations, see “SIZE ERROR phrases” on page 273.

**CORRESPONDING phrase (format 3)**
See “CORRESPONDING phrase” on page 272.

**END-SUBTRACT phrase**
This explicit scope terminator serves to delimit the scope of the SUBTRACT statement. END-SUBTRACT permits a conditional SUBTRACT statement to be nested in another conditional statement. END-SUBTRACT can also be used with an imperative SUBTRACT statement.

For more information, see “Delimited scope statements” on page 270.
**UNSTRING statement**

The UNSTRING statement causes contiguous data in a sending field to be separated and placed into multiple receiving fields.

### Format

```
UNSTRING identifier-1
```

**DELIMITED**

```
BY ALL literal-1
```

**OR**

```
ALL identifier-3 literal-2
```

**INTO**

```
identifier-4
```

**DELIMITER**

```
IN identifier-5 COUNT IN identifier-6
```

**WITH**

```
POINTER identifier-7
```

**TALLYING**

```
IN identifier-8
```

**OVERFLOW**

```
imperative-statement-1
```

**OVERFLOW**

```
imperative-statement-2
```

**END-UNSTRING**

---

**identifier-1**

Represents the *sending field*. Data is transferred from this field to the data receiving fields (identifier-4).

*identifier-1* must reference a data item of category alphabetic, alphanumeric, alphanumeric-edited, DBCS, national, or national-edited.

**identifier-2, literal-1, identifier-3, literal-2**

Specifies one or more delimiters.

*identifier-2* and *identifier-3* must reference data items of category alphabetic, alphanumeric, alphanumeric-edited, DBCS, national, or national-edited.

*literal-1* or *literal-2* must be of category alphanumeric, DBCS, or national and must not be a figurative constant that begins with the word ALL.

**identifier-4**

Specifies one or more receiving fields.
identifier-4 must reference a data item of category alphabetic, alphanumeric, numeric, DBCS, or national. If the referenced data item is of category numeric, its picture character-string must not contain the picture symbol P, and its usage must be DISPLAY or NATIONAL.

**identifier-5**

Specifies a field to receive the delimiter associated with identifier-4.

Identifier-5 must reference a data item of category alphabetic, alphanumeric, DBCS, or national.

**identifier-6**

Specifies a field to hold the count of characters that are transferred to identifier-4.

identifier-6 must be an integer data item defined without the symbol P in its PICTURE character-string.

**identifier-7**

Specifies a field to hold a relative character position during UNSTRING processing.

identifier-7 must be an integer data item defined without the symbol P in the PICTURE string.

identifier-7 must be described as a data item of sufficient size to contain a value equal to 1 plus the number of character positions in the data item referenced by identifier-1.

**identifier-8**

Specifies a field that is incremented by the number of delimited fields processed.

identifier-8 must be an integer data item defined without the symbol P in its PICTURE character-string.

The following rules apply:

- If identifier-4 references a data item of usage DISPLAY, identifier-1, identifier-2, identifier-3, and identifier-5 must also reference data items of usage DISPLAY and all literals must be alphanumeric literals. Any figurative constant can be specified except NULL or one that begins with the word ALL. Each figurative constant represents a 1-character alphanumeric literal.

- If identifier-4 references a data item of usage DISPLAY-1, identifier-1, identifier-2, identifier-3, and identifier-5 must also reference data items of usage DISPLAY-1 and all literals must be DBCS literals. Figurative constant SPACE is the only figurative constant that can be specified. Each figurative constant represents a 1-character DBCS literal.

- If identifier-4 references a data item of usage NATIONAL, identifier-1, identifier-2, identifier-3, and identifier-5 must also reference data items of usage NATIONAL and all literals must be national literals. Any figurative constant can be specified except NULL or one that begins with the word ALL. Each figurative constant represents a 1-character national literal.

Count fields (identifier-6) and pointer fields (identifier-7) are incremented by number of character positions (alphanumeric, DBCS, or national), not by number of bytes.

One UNSTRING statement can take the place of a series of MOVE statements, except that evaluation or calculation of certain elements is performed only once, at the beginning of the execution of the UNSTRING statement. For more information, see “Values at the end of execution of the UNSTRING statement” on page 431.

The rules for moving are the same as those for a MOVE statement for an elementary sending item of the category of identifier-1, with the appropriate identifier-4 as the receiving item (see “MOVE statement” on page 363). For example, rules for moving a DBCS item are used when identifier-1 is a DBCS item.

**DELIMITED BY phrase**

This phrase specifies delimiters within the data that control the data transfer.

Each identifier-2, identifier-3, literal-1, or literal-2 represents one delimiter.

If the DELIMITED BY phrase is not specified, the DELIMITER IN and COUNT IN phrases must *not* be specified.
ALL

Multiple contiguous occurrences of any delimiters are treated as if there were only one occurrence; this one occurrence is moved to the delimiter receiving field (identifier-5), if specified. The delimiting characters in the sending field are treated as an elementary item of the same usage and category as identifier-1 and are moved into the current delimiter receiving field according to the rules of the MOVE statement.

When DELIMITED BY ALL is not specified, and two or more contiguous occurrences of any delimiter are encountered, the current data receiving field (identifier-4) is filled with spaces or zeros, according to the description of the data receiving field.

Delimiter with two or more characters

A delimiter that contains two or more characters is recognized as a delimiter only if the delimiting characters are in both of the following cases:

• Contiguous
• In the sequence specified in the sending field

Two or more delimiters

When two or more delimiters are specified, an OR condition exists, and each nonoverlapping occurrence of any one of the delimiters is recognized in the sending field in the sequence specified.

For example:

```
DELIMITED BY "AB" or "BC"
```

An occurrence of either AB or BC in the sending field is considered a delimiter. An occurrence of ABC is considered an occurrence of AB.

INTO phrase

This phrase specifies the fields where the data is to be moved.

identifier-4 represents the data receiving fields.

DELIMITER IN

This phrase specifies the fields where the delimiters are to be moved.

identifier-5 represents the delimiter receiving fields.

The DELIMITER IN phrase must not be specified if the DELIMITED BY phrase is not specified.

COUNT IN

This phrase specifies the field where the count of examined character positions is held.

identifier-6 is the data count field for each data transfer. Each field holds the count of examined character positions in the sending field, terminated by the delimiters or the end of the sending field, for the move to this receiving field. The delimiters are not included in this count.

The COUNT IN phrase must not be specified if the DELIMITED BY phrase is not specified.

POINTER phrase

When the POINTER phrase is specified, the value of the pointer field, identifier-7, behaves as if it were increased by 1 for each examined character position in the sending field. When execution of the UNSTRING statement is completed, the pointer field contains a value equal to its initial value plus the number of character positions examined in the sending field.

When this phrase is specified, the user must initialize the pointer field before execution of the UNSTRING statement begins.
**TALLYING IN phrase**

When the TALLYING phrase is specified, the area count field, identifier-8, contains (at the end of execution of the UNSTRING statement) a value equal to the initial value plus the number of data receiving areas acted upon.

When this phrase is specified, the user must initialize the area count field before execution of the UNSTRING statement begins.

**ON OVERFLOW phrases**

An overflow condition exists when:

- The pointer value (explicit or implicit) is less than 1.
- The pointer value (explicit or implicit) exceeds a value equal to the length of the sending field.
- All data receiving fields have been acted upon and the sending field still contains unexamined character positions.

**When an overflow condition occurs**

An overflow condition results in the following actions:

1. No more data is transferred.
2. The UNSTRING operation is terminated.
3. The NOT ON OVERFLOW phrase, if specified, is ignored.
4. Control is transferred to the end of the UNSTRING statement or, if the ON OVERFLOW phrase is specified, to imperative-statement-1.

**imperative-statement-1**

Statement or statements for dealing with an overflow condition.

If control is transferred to imperative-statement-1, execution continues according to the rules for each statement specified in imperative-statement-1. If a procedure branching or conditional statement that causes explicit transfer of control is executed, control is transferred according to the rules for that statement. Otherwise, upon completion of the execution of imperative-statement-1, control is transferred to the end of the UNSTRING statement.

**When an overflow condition does not occur**

When, during execution of an UNSTRING statement, conditions that would cause an overflow condition are not encountered, then:

1. The transfer of data is completed.
2. The ON OVERFLOW phrase, if specified, is ignored.
3. Control is transferred to the end of the UNSTRING statement or, if the NOT ON OVERFLOW phrase is specified, to imperative-statement-2.

**imperative-statement-2**

Statement or statements for dealing with an overflow condition that does not occur.

If control is transferred to imperative-statement-2, execution continues according to the rules for each statement specified in imperative-statement-2. If a procedure branching or conditional statement that causes explicit transfer of control is executed, control is transferred according to the rules for that statement. Otherwise, upon completion of the execution of imperative-statement-2, control is transferred to the end of the UNSTRING statement.

The ON OVERFLOW phrase can be used for examining the sending field whereas the NOT ON OVERFLOW phrase is for normal execution when an overflow condition does not occur. You must include NOT ON OVERFLOW if you want to specify procedures to be executed only when an overflow condition does not occur. For example:
UNSTRING COLOR-LIST
   ... ON OVERFLOW
     DISPLAY 'Error: The string is too large'
   NOT ON OVERFLOW     *> Execute when the UNSTRING is successful
     PERFORM SORT-COLORS
END-UNSTRING

END-UNSTRING phrase
This explicit scope terminator serves to delimit the scope of the UNSTRING statement. END-UNSTRING permits a conditional UNSTRING statement to be nested in another conditional statement. END-UNSTRING can also be used with an imperative UNSTRING statement.

For more information, see “Delimited scope statements” on page 270.

Data flow
The data flow for the UNSTRING statement is based on certain rules.

When the UNSTRING statement is initiated, data is transferred from the sending field to the current data receiving field, according to the following rules:

Stage 1: Examine
1. If the POINTER phrase is specified, the field is examined, beginning at the relative character position specified by the value in the pointer field.
   
   If the POINTER phrase is not specified, the sending field character-string is examined, beginning with the leftmost character position.

2. If the DELIMITED BY phrase is specified, the examination proceeds from left to right, examining character positions one-by-one until a delimiter is encountered. If the end of the sending field is reached before a delimiter is found, the examination ends with the last character position in the sending field. If there are more receiving fields, the next one is selected; otherwise, an overflow condition occurs.

   If the DELIMITED BY phrase is not specified, the number of character positions examined is equal to the size of the current data receiving field, as described in the table below. The size depends on the category treatment of the receiving field, as shown in Table 39 on page 332.

   If the DELIMITED BY phrase is not specified and the receiver is a dynamic-length elementary item, the number of character positions examined is equal to the length of the sender. All identifiers cannot be dynamic-length group items.

| Table 52. Character positions examined when DELIMITED BY is not specified |
|-----------------------------|--------------------------------------------------------------------------------|
| If the receiving field is ... | The number of character positions examined is ...                            |
| Alphanumeric or alphabetic  | Equal to the number of alphanumeric character positions in the current receiving field |
| DBCS                        | Equal to the number of DBCS character positions in the current receiving field |
| National                    | Equal to the number of national character positions in the current receiving field |
| Numeric                     | Equal to the number of character positions in the integer portion of the current receiving field |
| Described with the SIGN IS SEPARATE clause | 1 less than the size of the current receiving field |
| Described as a variable-length data item | Determined by the size of the current receiving field at the beginning of the UNSTRING operation |
Stage 2: Move

3. The examined character positions (excluding any delimiter characters) are treated as an elementary data item of the same data category as the sending field except for the cases shown in the table below.

<table>
<thead>
<tr>
<th>Category of identifier-1 (sending-field)</th>
<th>Category of elementary data item</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alphanumeric-edited</td>
<td>Alphanumeric</td>
</tr>
<tr>
<td>National-edited</td>
<td>National</td>
</tr>
</tbody>
</table>

That elementary data item is moved to the current data receiving field according to the rules for the MOVE statement for the categories of the sending and receiving fields as described in “MOVE statement” on page 363.

4. If the DELIMITER IN phrase is specified, the delimiting characters in the sending field are treated as an elementary alphanumeric item and are moved to the current delimiter receiving field, according to the rules for the MOVE statement. If the delimiting condition is the end of the sending field, the current delimiter receiving field is filled with spaces.

5. If the COUNT IN phrase is specified, a value equal to the number of examined character positions (excluding any delimiters) is moved into the data count field, according to the rules for an elementary move.

Stage 3: Successive iterations

6. If the DELIMITED BY phrase is specified, the sending field is further examined, beginning with the first character position to the right of the delimiter.

   If the DELIMITED BY phrase is not specified, the sending field is further examined, beginning with the first character position to the right of the last character position examined.

7. For each succeeding data receiving field, this process of examining and moving is repeated until either of the following conditions occurs:
   • All the characters in the sending field have been transferred.
   • There are no more unfilled data receiving fields.

Values at the end of execution of the UNSTRING statement

At the beginning of the execution of the UNSTRING statement, certain operations are performed only once.

The operations are:
• Calculations of subscripts, reference modifications, variable-lengths, variable locations
• Evaluations of functions

Therefore, if identifier-4, identifier-5, identifier-6, identifier-7, or identifier-8 is used as a subscript, reference-modifier, or function argument in the UNSTRING statement, or affects the length or location of any of the identifiers in the UNSTRING statement, these values are determined at the beginning of the UNSTRING statement, and are not affected by any results of the UNSTRING statement.

Example of the UNSTRING statement

This topic lists an example for the UNSTRING statement.

The following figure shows the execution results for an example of the UNSTRING statement.
The WRITE statement releases a logical record to an output or input/output file.

When the WRITE statement is executed:

- The associated sequential file must be open in OUTPUT or EXTEND mode.
- The associated indexed or relative file must be open in OUTPUT, I-O, or EXTEND mode.
Format 1: WRITE statement for sequential files

- WRITE record-name-1
- FROM identifier-1
- BEFORE
- AFTER
- ADVANCING identifier-2
- integer-1
- LINE
- LINES
- mnemonic-name-1
- PAGE

invalid_key
not_invalid_key

END-WRITE

phrase 1

- AT END-OF-PAGE imperative-statement-3
- END-OF-PAGE EOP imperative-statement-3

- NOT AT END-OF-PAGE imperative-statement-4
- END-OF-PAGE EOP imperative-statement-4

invalid_key

- INVALID KEY imperative-statement-1

not_invalid_key

- NOT INVALID KEY imperative-statement-2

Format 2: WRITE statement for indexed and relative files

- WRITE record-name-1
- FROM identifier-1

- INVALID KEY imperative-statement-1

- NOT INVALID KEY imperative-statement-2

END-WRITE
**Format 3: WRITE statement for line-sequential files**

```plaintext
WRITE record-name-1
  FROM identifier-1
  AFTER identifier-2
  ADVANCING integer-1
  LINE lines
  PAGE page
END-WRITE
```

**record-name-1**

Must be defined in a DATA DIVISION FD entry. `record-name-1` can be qualified. It must not be associated with a sort or merge file.

For relative files, the number of character positions in the record being written can be different from the number of character positions in the record being replaced.

**FROM phrase**

The result of the execution of the WRITE statement with the FROM `identifier-1` phrase is equivalent to the execution of the following statements in the order specified:

```plaintext
MOVE identifier-1 TO record-name-1.
WRITE record-name-1.
```

The MOVE is performed according to the rules for a MOVE statement without the CORRESPONDING phrase.

**identifier-1**

`identifier-1` can reference any of the following items:

- A data item defined in the WORKING-STORAGE SECTION, the LOCAL-STORAGE SECTION, or the LINKAGE SECTION
- A record description for another previously opened file
- An alphanumeric function
- A national function

`identifier-1` must be a valid sending item for a MOVE statement with `record-name-1` as the receiving item.

`identifier-1` and `record-name-1` must not refer to the same storage area.

After the WRITE statement is executed, the information is still available in `identifier-1`. (See “INTO and FROM phrases” on page 281 under "Common processing facilities".)

**identifier-2**

Must be an integer data item.

**ADVANCING phrase**

The ADVANCING phrase controls positioning of the output record on the page.
The BEFORE and AFTER phrases are not supported for VSAM files. QSAM files are sequentially organized. The ADVANCING and END-OF-PAGE phrases control the vertical positioning of each line on a printed page.

You can specify the ADVANCING PAGE and END-OF-PAGE phrases in a single WRITE statement.

If the printed page is held on an intermediate device (a disk, for example), the format can appear different from the expected format when the output is edited or browsed.

ADVANCING phrase rules

When the ADVANCING phrase is specified, the following rules apply:

1. When BEFORE ADVANCING is specified, the line is printed before the page is advanced.
2. When AFTER ADVANCING is specified, the page is advanced before the line is printed.
3. When identifier-2 is specified, the page is advanced the number of lines equal to the current value in identifier-2. identifier-2 must name an elementary integer data item.
4. When integer is specified, the page is advanced the number of lines equal to the value of integer.
5. Integer or the value in identifier-2 can be zero.
6. When PAGE is specified, the record is printed on the logical page BEFORE or AFTER (depending on the phrase used) the device is positioned to the next logical page. If PAGE has no meaning for the device used, then BEFORE or AFTER (depending on the phrase specified) ADVANCING 1 LINE is provided.

If the FD entry contains a LINAGE clause, the repositioning is to the first printable line of the next page, as specified in that clause. If the LINAGE clause is omitted, the repositioning is to line 1 of the next succeeding page.

7. When mnemonic-name is specified, a skip to channels 1 through 12, or space suppression, takes place. mnemonic-name must be equated with environment-name-1 in the SPECIAL-NAMES paragraph.

The mnemonic-name phrase can also be specified for stacker selection with a card punch file. When using stacker selection, WRITE AFTER ADVANCING must be used.

The ADVANCING phrase of the WRITE statement, or the presence of a LINAGE clause on the file, causes a carriage control character to be generated in the record that is written. If the corresponding file is described with the EXTERNAL clause, all file connectors within the run unit must be defined such that carriage control characters will be generated for records that are written. That is, if all the files have a LINAGE clause, some of the programs can use the WRITE statement with the ADVANCING phrase and other programs can use the WRITE statement without the ADVANCING phrase. However, if none of the files has a LINAGE clause, then if any of the programs use the WRITE statement with the ADVANCING phrase, all of the programs in the run unit that have a WRITE statement must use the WRITE statement with the ADVANCING phrase.

When the ADVANCING phrase is omitted, automatic line advancing is provided, as if AFTER ADVANCING 1 LINE had been specified.

LINAGE-COUNTER rules

If the LINAGE clause is specified for this file, the associated LINAGE-COUNTER special register is modified during the execution of the WRITE statement, according to the following rules:

1. If ADVANCING PAGE is specified, LINAGE-COUNTER is reset to 1.
2. If ADVANCING identifier-2 or integer is specified, LINAGE-COUNTER is increased by the value in identifier-2 or integer.
3. If the ADVANCING phrase is omitted, LINAGE-COUNTER is increased by 1.
4. When the device is repositioned to the first available line of a new page, LINAGE-COUNTER is reset to 1.

Usage note: If you use the ADV compiler option, the compiler adds 1 byte to the record length in order to allow for the control character. If in your record definition you already reserve the first byte for the control character, you should use the NOADV option. For files defined with the LINAGE clause, the NOADV option has no effect. The compiler processes these files as if the ADV option were specified.
END-OF-PAGE phrases

The AT END-OF-PAGE phrase is not supported for VSAM files.

When END-OF-PAGE is specified, and the logical end of the printed page is reached during execution of the WRITE statement, the END-OF-PAGE imperative-statement is executed. When the END-OF-PAGE phrase is specified, the FD entry for this file must contain a LINAGE clause.

The logical end of the printed page is specified in the associated LINAGE clause.

An END-OF-PAGE condition is reached when execution of a WRITE END-OF-PAGE statement causes printing or spacing within the footing area of a page body. This occurs when execution of such a WRITE statement causes the value in the LINAGE-COUNTER special register to equal or exceed the value specified in the WITH FOOTING phrase of the LINAGE clause. The WRITE statement is executed, and then the END-OF-PAGE imperative-statement is executed.

An automatic page overflow condition is reached whenever the execution of any given WRITE statement (with or without the END-OF-PAGE phrase) cannot be completely executed within the current page body. This occurs when a WRITE statement, if executed, would cause the value in the LINAGE-COUNTER to exceed the number of lines for the page body specified in the LINAGE clause. In this case, the line is printed BEFORE or AFTER (depending on the option specified) the device is repositioned to the first printable line on the next logical page, as specified in the LINAGE clause. If the END-OF-PAGE phrase is specified, the END-OF-PAGE imperative-statement is then executed.

If the WITH FOOTING phrase of the LINAGE clause is not specified, the automatic page overflow condition exists because no end-of-page condition (as distinct from the page overflow condition) can be detected.

If the WITH FOOTING phrase is specified, but the execution of a given WRITE statement would cause the LINAGE-COUNTER to exceed both the footing value and the page body value specified in the LINAGE clause, then both the end-of-page condition and the automatic page overflow condition occur simultaneously.

The keywords END-OF-PAGE and EOP are equivalent.

You can specify both the ADVANCING PAGE phrase and the END-OF-PAGE phrase in a single WRITE statement.

INVALID KEY phrases

The INVALID KEY phrase is not supported for VSAM sequential files.

An invalid key condition is caused by the following cases:

• For sequential files, an attempt is made to write beyond the externally defined boundary of the file.

• For indexed files:
  – An attempt is made to write beyond the externally defined boundary of the file.
  – ACCESS SEQUENTIAL is specified and the file is opened OUTPUT, and the value of the prime record key is not greater than that of the previous record.
  – The file is opened OUTPUT or I-O and the value of the prime record key equals that of an already existing record.

• For relative files:
  – An attempt is made to write beyond the externally defined boundary of the file.
  – When the access mode is random or dynamic and the RELATIVE KEY data item specifies a record that already exists in the file.
  – The number of significant digits in the relative record number is larger than the size of the relative key data item for the file.
When an invalid key condition occurs:

- If the INVALID KEY phrase is specified, imperative-statement-1 is executed. For details of invalid key processing, see Invalid key condition.

- Otherwise, the WRITE statement is unsuccessful and the contents of record-name are unaffected (except for QSAM files) and the following case occurs:
  - For sequential files, the file status key, if specified, is updated and an EXCEPTION/ERROR condition exists.
    - If an explicit or implicit EXCEPTION/ERROR procedure is specified for the file, the procedure is executed. If no such procedure is specified, the results are unpredictable.
  - For relative and indexed files, program execution proceeds according to the rules described by Invalid key condition under "Common processing facilities".
    - The INVALID KEY conditions that apply to a relative file in OPEN OUTPUT mode also apply to one in OPEN EXTEND mode.

- If the NOT INVALID KEY phrase is specified and a valid key condition exists at the end of the execution of the WRITE statement, control is passed to imperative-statement-4.

Both the INVALID KEY phrase and an applicable EXCEPTION/ERROR procedure can be omitted.

**END-WRITE phrase**

This explicit scope terminator serves to delimit the scope of the WRITE statement. END-WRITE permits a conditional WRITE statement to be nested in another conditional statement. END-WRITE can also be used with an imperative WRITE statement.

For more information, see “Delimited scope statements” on page 270.

**WRITE for sequential files**

The maximum record size for sequential files is established at the time the file is created and cannot subsequently be changed.

After the WRITE statement is executed, the logical record is no longer available in record-name-1 unless either:

- The associated file is named in a SAME RECORD AREA clause (in which case, the record is also available as a record of the other files named in the SAME RECORD AREA clause)
- The WRITE statement is unsuccessful because of a boundary violation.

In either of these two cases, the logical record is still available in record-name-1.

The file position indicator is not affected by execution of the WRITE statement.

The number of character positions required to store the record in a file might or might not be the same as the number of character positions defined by the logical description of that record in the COBOL program. (See “PICTURE clause editing” on page 201 and “USAGE clause” on page 218.)

If the FILE STATUS clause is specified in the file-control entry, the associated file status key is updated when the WRITE statement is executed, whether or not execution is successful.

The WRITE statement can only be executed for a sequential file opened in OUTPUT or EXTEND mode for QSAM files.

**Punch function files with the IBM 3525**

When the punch function is used, the next I-O operation after the READ statement must be a WRITE statement for the punch function file.

If you want to punch additional data into some of the cards and not into others, a dummy WRITE statement must be executed for the null cards, first filling the output area with SPACES.
If stacker selection for the punch function file is required, you can specify the appropriate stacker function-names in the SPECIAL-NAMES paragraph, and then code WRITE ADVANCING statements using the associated mnemonic-names.

**Print function files**

After the punch function operations (if specified) are completed, you can issue WRITE statements for the print function file.

If you wish to print additional data on some of the data cards and not on others, the WRITE statement for the null cards can be omitted. Any attempt to write beyond the limits of the card results in abnormal termination of the application, thus, the END-OF-PAGE phrase cannot be specified.

Depending on the capabilities of the specific IBM 3525 model in use, the print file can be either a two-line print file or a multiline print file. Up to 64 characters can be printed on each line.

- For a two-line print file, the lines are printed on line 1 (top edge of card) and line 3 (between rows 11 and 12). Line control cannot be specified. Automatic spacing is provided.
- For a multiline print file, up to 25 lines of characters can be printed. Line control can be specified. If line control is not specified, automatic spacing is provided.

Line control is specified by issuing WRITE AFTER ADVANCING statements for the print function file. If line control is used for one such statement, it must be used for all other WRITE statements for the file. The maximum number of printable characters, including any space characters, is 64. Such WRITE statements must not specify space suppression.

Identifier and integer have the same meanings they have for other WRITE AFTER ADVANCING statements. However, such WRITE statements must not increase the line position on the card beyond the card limit, or abnormal termination results.

The mnemonic-name option of the WRITE AFTER ADVANCING statement can also be specified. In the SPECIAL-NAMES paragraph, the environment-names can be associated with the mnemonic-names, as shown in the following table:

<table>
<thead>
<tr>
<th>environment-name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>C02</td>
<td>Line 3</td>
</tr>
<tr>
<td>C03</td>
<td>Line 5</td>
</tr>
<tr>
<td>C04</td>
<td>Line 7</td>
</tr>
<tr>
<td>C05</td>
<td>Line 9</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>C22</td>
<td>Line 21</td>
</tr>
<tr>
<td>C12</td>
<td>Line 23</td>
</tr>
</tbody>
</table>

**Advanced Function Printing**

When you use the WRITE ADVANCING phrase with a mnemonic-name associated with environment-name AFP-5A, a Print Services Facility (PSF) control character is placed in the control character position of the output record. This control character (\x'5A') allows Advanced Function Printing (AFP) services to be used. For more information, refer to the documentation for the Print Services Facility product: PSF for OS/390® & z/OS (5655-B17).
WRITE for indexed files

Before the WRITE statement is executed for indexed files, you must set the prime record key (the RECORD KEY data item, as defined in the file-control entry) to the required value. Note that RECORD KEY values must be unique within a file.

If the ALTERNATE RECORD KEY clause is also specified in the file-control entry, each alternate record key must be unique, unless the DUPLICATES phrase is specified. If the DUPLICATES phrase is specified, alternate record key values might not be unique. In this case, the system stores the records so that later sequential access to the records allows retrieval in the same order in which they were stored.

When ACCESS IS SEQUENTIAL is specified in the file-control entry, records must be released in ascending order of RECORD KEY values.

When ACCESS IS RANDOM or ACCESS IS DYNAMIC is specified in the file-control entry, records can be released in any programmer-specified order.

WRITE for relative files

For relative record OUTPUT files, the WRITE statement causes actions as described in the topic.

• If ACCESS IS SEQUENTIAL is specified:

  The first record released has relative record number 1, the second record released has relative record number 2, the third number 3, and so on.

  If the RELATIVE KEY is specified in the file-control entry, the relative record number of the record just released is placed in the RELATIVE KEY during execution of the WRITE statement.

• If ACCESS IS RANDOM or ACCESS IS DYNAMIC is specified, the RELATIVE KEY must contain the required relative record number for this record before the WRITE statement is executed. When the WRITE statement is executed, this record is placed at the specified relative record number position in the file.

For I-O files, either ACCESS IS RANDOM or ACCESS IS DYNAMIC must be specified; the WRITE statement inserts new records into the file. The RELATIVE KEY must contain the required relative record number for this record before the WRITE statement is executed. When the WRITE statement is executed, this record is placed at the specified relative record number position in the file.
The XML GENERATE statement converts data to XML format.

**Format**

```plaintext
XML GENERATE  — identifier-1  — FROM — identifier-2 —

  COUNT     — identifier-3

  IN

WITH

  ENCODING — codepage

WITH

  XML-DECLARATION

WITH

  ATTRIBUTES

NAMESPACE — identifier-4

IS

literal-4

NAMESPACE-PREFIX

IS

identifier-5

literal-5

NAME — identifier-6

OF

literal-6

TYPE — identifier-7

OF

ATTRIBUTE

ELEMENT

CONTENT

SUPPRESS — identifier-8

when-phrase

generic-suppression-phrase

ON

EXCEPTION — imperative-statement-1

NOT

ON

EXCEPTION — imperative-statement-2

END-XML
```
**identifier-1**

The receiving area for a generated XML document. *identifier-1* must reference one of the following items:

- An elementary data item of category alphanumeric
- An alphanumeric group item
- An elementary data item of category national
- A national group item

When *identifier-1* references a national group item, *identifier-1* is processed as an elementary data item of category national. When *identifier-1* references an alphanumeric group item, *identifier-1* is treated as though it were an elementary data item of category alphanumeric.

*identifier-1* must not be described with the JUSTIFIED clause, and cannot be a function identifier. *identifier-1* can be subscripted or reference modified.

*identifier-1* must not overlap *identifier-2*, *identifier-3*, codepage (if an identifier), *identifier-4*, or *identifier-5*. 
identifier-1 must not be a dynamic-length group item or a dynamic-length elementary item.
The generated XML output is encoded as described in the documentation of the ENCODING phrase.
identifier-1 must reference a data item of category national, or the ENCODING phrase must specify 1208, if any of the following statements are true:
• The CODEPAGE compiler option specifies an EBCDIC DBCS code page.
• identifier-4 or identifier-5 references a data item of category national.
• literal-4, literal-5, or literal-6 is of category national.
• The generated XML includes data from identifier-2 for:
  – Any data item of class national or class DBCS
  – Any data item with a DBCS name (that is, a data item whose name consists of DBCS characters)
  – Any data item of class alphanumeric that contains DBCS characters
identifier-1 must be large enough to contain the generated XML document. Typically, it must be from 5 to 10 times the size of identifier-2, depending on the length of the data-name or data-names within identifier-2. If identifier-1 is not large enough, an error condition exists at the end of the XML GENERATE statement.

identifier-2
The group or elementary data item to be converted to XML format.
If identifier-2 references a national group item, identifier-2 is processed as a group item. When identifier-2 includes a subordinate national group item, that subordinate item is processed as a group item.
identifier-2 cannot be a function identifier or be reference modified, but it can be subscripted.
identifier-2 must not overlap identifier-1 or identifier-3.

identifier-2 might not be a dynamic-length group item or a dynamic-length elementary item.
The data description entry for identifier-2 must not contain a RENAMES clause.
The following data items that are specified by identifier-2 are ignored by the XML GENERATE statement:
• Any subordinate unnamed elementary data items or elementary FILLER data items
• Any slack bytes inserted for SYNCHRONIZED items
• Any data item subordinate to identifier-2 that is described with the REDEFINES clause or that is subordinate to such a redefining item
• Any data item subordinate to identifier-2 that is described with the RENAMES clause
• Any group data item all of whose subordinate data items are ignored
All data items specified by identifier-2 that are not ignored according to the previous rules must satisfy the following conditions:
• Each elementary data item must either have class alphabetic, alphanumeric, numeric, or national, or be an index data item. (That is, no elementary data item can be described with the USAGE POINTER, USAGE FUNCTION-POINTER, USAGE PROCEDURE-POINTER, or USAGE OBJECT REFERENCE phrase.)
• There must be at least one such elementary data item.
• Each non-FILLER data-name must be unique within any immediately superordinate group data item.
• Any DBCS data-names, when converted to Unicode, must be legal as names in the XML specification, version 1.0. For details about the XML specification, see XML specification.
For example, consider the following data declaration:

```
01 STRUCT.
  02 STAT PIC X(4).
  02 IN-AREA PIC X(100).
  02 OK-AREA REDEFINES IN-AREA.
    03 FLAGS PIC X.
    03 PIC X(3).
    03 COUNTER USAGE COMP-5 PIC S9(9).
    03 ASFNPTR REDEFINES COUNTER USAGE FUNCTION-POINTER.
    03 UNREFERENCED PIC X(92).
  02 NG-AREA1 REDEFINES IN-AREA.
    03 FLAGS PIC X.
    03 PIC X(3).
    03 PTR USAGE POINTER.
    03 ASNUM REDEFINES PTR USAGE COMP-5 PIC S9(9).
    03 PIC X(92)
  02 NG-AREA2 REDEFINES IN-AREA.
    03 FN-CODE PIC X.
    03 UNREFERENCED PIC X(3).
    03 QTYONHAND USAGE BINARY PIC 9(5).
    03 DESC USAGE NATIONAL PIC N(40).
    03 UNREFERENCED PIC X(12).
```

The following data items from the previous example can be specified as `identifier-2`:

- **STRUCT**, of which subordinate data items `STAT` and `IN-AREA` would be converted to XML format. (OK-AREA, NG-AREA1, and NG-AREA2 are ignored because they specify the REDEFINES clause.)

- **OK-AREA**, of which subordinate data items `FLAGS`, `COUNTER`, and `UNREFERENCED` would be converted. (The item whose data description entry specifies `03 PIC X(3)` is ignored because it is an elementary FILLER data item. ASFNPTR is ignored because it specifies the REDEFINES clause.)

- Any of the elementary data items that are subordinate to `STRUCT` except:
  - ASFNPTR or PTR (disallowed usage)
  - UNREFERENCED OF NG-AREA2 (nonunique names for data items that are otherwise eligible)
  - Any FILLER data items

The following data items cannot be specified as `identifier-2`:

- **NG-AREA1**, because subordinate data item `PTR` specifies USAGE POINTER but does not specify the REDEFINES clause. (PTR would be ignored if it specified the REDEFINES clause.)

- **NG-AREA2**, because subordinate elementary data items have the nonunique name UNREFERENCED.

**COUNT IN phrase**

If the `COUNT IN` phrase is specified, `identifier-3` contains (after execution of the XML GENERATE statement) the count of generated XML character encoding units. If `identifier-1` (the receiver) has category national, the count is in UTF-16 character encoding units. For all other encodings (including UTF-8), the count is in bytes.

`identifier-3` must not overlap `identifier-1`, `identifier-2`, `codepage` (if an identifier), `identifier-4`, or `identifier-5`.

**ENCODING phrase**

The `ENCODING` phrase, if specified, determines the encoding of the generated XML document.

`codepage` must be an unsigned integer data item or unsigned integer literal and must represent a valid coded character set identifier (CCSID). Must identify one of the code pages supported for COBOL XML processing as described in *The encoding of XML documents* in the Enterprise COBOL Programming Guide.
If identifier-1 references a data item of category national, codepage must specify 1200, the CCSID for Unicode UTF-16.

If identifier-1 references a data item of category alphanumeric, codepage must specify 1208 or
the CCSID of a supported EBCDIC code page as listed in The encoding of XML documents in the
Enterprise COBOL Programming Guide.

If codepage is an identifier, it must not overlap identifier-1 or identifier-3.

If the ENCODING phrase is omitted and identifier-1 is of category national, the document encoding is
Unicode UTF-16, CCSID 1200.
If the ENCODING phrase is omitted and identifier-1 is of category alphanumeric, the XML document is
encoded using the code page specified by the CODEPAGE compiler option in effect when the source
code was compiled.
If the ENCODING phrase is omitted and identifier-1 is of category alphanumeric, the XML document is
encoded using the code page specified by the EBCDIC_CODEPAGE environment variable in effect
when the source code was compiled.

XML-DECLARATION phrase
If the XML-DECLARATION phrase is specified, the generated XML document starts with an XML
declaration that includes the XML version information and an encoding declaration.

If identifier-1 is of category national, the encoding declaration has the value UTF-16 (encoding="UTF-16").
If identifier-1 is of category alphanumeric, the encoding declaration is derived from the ENCODING
phrase, if specified, or from the CODEPAGE compiler option in effect for the program if the ENCODING
phrase is not specified. See the description of the ENCODING phrase for further details.

For an example of the effect of coding the XML-DECLARATION phrase, see Generating XML output in
the Enterprise COBOL Programming Guide.
If the XML-DECLARATION phrase is omitted, the generated XML document does not include an XML
declaration.

ATTRIBUTES phrase
If the ATTRIBUTES phrase is specified, each eligible item included in the generated XML document is
expressed as an attribute of the XML element that corresponds to the data item immediately
superordinate to that eligible item, rather than as a child element of the XML element. To be eligible, a
data item must be elementary, must have a name other than FILLER, and must not specify an
OCCURS clause in its data description entry.
If the TYPE phrase is specified for particular identifiers, the TYPE phrase takes precedence for those
identifiers over the WITH ATTRIBUTES phrase.
For an example of the effect of the ATTRIBUTES phrase, see Generating XML output in the Enterprise
COBOL Programming Guide.

NAMESPACE and NAMESPACE-PREFIX phrases
Use the NAMESPACE phrase to identify a namespace for the generated XML document. If the
NAMESPACE phrase is not specified, or if identifier-4 has length zero or contains all spaces, the
element names of XML documents produced by the XML GENERATE statement are not in any
namespace.
Use the NAMESPACE-PREFIX phrase to qualify the start and end tag of each element in the generated
XML document with a prefix.
If the NAMESPACE-PREFIX phrase is not specified, or if identifier-5 is of length zero or contains all
spaces, the namespace specified by the NAMESPACE phrase specifies the default namespace for the
document. In this case, the namespace declared on the root element applies by default to each
element name in the document, including that of the root element. (Default namespace declarations
do not apply directly to attribute names.)
If the NAMESPACE-PREFIX phrase is specified, and identifier-5 is not of length zero and does not contain all spaces, then the start and end tag of each element in the generated document is qualified with the specified prefix. The prefix should therefore preferably be short. When the XML GENERATE statement is executed, the prefix must be a valid XML name, but without the colon (\:), as defined in Namespaces in XML 1.0. The prefix can have trailing spaces, which are removed before use.

**identifier-4, literal-4; identifier-5, literal-5**

identifier-4, literal-4: The namespace identifier, which must be a valid Uniform Resource Identifier (URI) as defined in Uniform Resource Identifier (URI): Generic Syntax.

identifier-5, literal-5: The namespace prefix, which serves as an alias for the namespace identifier.

identifier-4 and identifier-5 must reference data items of category alphanumeric or national.

identifier-4 and identifier-5 must not overlap identifier-1 or identifier-3.

literal-4 and literal-5 must be of category alphanumeric or national, and must not be figurative constants.

For full details about namespaces, see Namespaces in XML 1.0.

For examples that show the use of the NAMESPACE and NAMESPACE-PREFIX phrases, see Generating XML output in the Enterprise COBOL Programming Guide.

**NAME phrase**

Allows you to supply element and attribute names.

**identifier-6** must reference identifier-2 or one of its subordinate data items. It cannot be a function identifier and cannot be reference modified or subscripted. It must not specify any data item which is ignored by the XML GENERATE statement. For more information about **identifier-2**, see the description of **identifier-2**. If **identifier-6** is specified more than once in the NAME phrase, the last specification is used.

**literal-6** must be an alphanumeric or national literal containing the attribute or element name to be generated in the XML document corresponding to **identifier-6**. It must be a valid XML local name. If **literal-6** is a national literal, **identifier-1** must reference a data item of category national or the encoding phrase must specify 1208.

**TYPE phrase**

Allows you to control attribute and element generation.

**identifier-7** must reference an elementary data item that is subordinate to **identifier-2**. It cannot be a function identifier and cannot be reference modified or subscripted. It must not specify any data item which is ignored by the XML GENERATE statement. For more information about **identifier-2**, see the description of **identifier-2**. If **identifier-7** is specified more than once in the TYPE phrase, the last specification is used.

- If the XML GENERATE statement also includes a WITH ATTRIBUTES phrase, the TYPE phrase has precedence for **identifier-7**.
- When ATTRIBUTE is specified, **identifier-7** must be eligible to be an XML attribute. **identifier-7** is expressed in the generated XML as an attribute of the XML element immediately superordinate to **identifier-7** rather than as a child element.
- When ELEMENT is specified, **identifier-7** is expressed in the generated XML as an element. The XML element name is derived from **identifier-7** and the element character content is derived from the converted content of **identifier-7** as described in “Operation of XML GENERATE” on page 447.
- When CONTENT is specified, **identifier-7** is expressed in the generated XML as element character content of the XML element that corresponds to the data item immediately superordinate to **identifier-7**. The value of the element character content is derived from the converted content of **identifier-7** as described in “Operation of XML GENERATE” on page 447. When CONTENT is specified for multiple identifiers all corresponding to the same superordinate identifier, the multiple contributions to the element character content are concatenated.
**SUPPRESS phrase**

Allows you to identify and unconditionally suppress items that are subordinate to `identifier-2` and selectively generate output for the XML GENERATE statement. If the SUPPRESS phrase is specified, `identifier-1` must be large enough to contain the generated XML document before any suppression.

With the `generic-suppression-phrase`, elementary items subordinate to `identifier-2` that are not otherwise ignored by XML GENERATE operations are identified generically for potential suppression. Either items of class numeric, if the NUMERIC keyword is specified, or items that are not of class numeric, if the NONNUMERIC keyword is specified, or both, might be suppressed. If the ATTRIBUTE keyword is specified, only items that would be expressed in the generated XML document as an XML attribute are identified for potential suppression. If the ELEMENT keyword is specified, only items that would be expressed in the generated XML document as an XML element are identified for potential suppression. If the CONTENT keyword is specified, only items that would be expressed in the generated XML document as element character content of the XML element corresponding to the data item superordinate to the CONTENT data item are identified for potential suppression.

If multiple `generic-suppression-phrase` are specified, the effect is cumulative.

`identifier-8` explicitly identifies items for potential suppression. If the WHEN phrase is specified, `identifier-8` must reference an elementary data item that is subordinate to `identifier-2` and that is not otherwise ignored by the XML GENERATE operations. `identifier-8` cannot be a function identifier and cannot be reference modified or subscripted. If the WHEN phrase is omitted, `identifier-8` can reference not only an elementary data item but also a group data item. That group data item and all data items that are subordinate to the group item are suppressed. If `identifier-8` is specified more than once in the SUPPRESS phrase, the last specification is used. The explicit suppression specification for `identifier-8` overrides the suppression specification that is implied by any `generic-suppression-phrase`, if `identifier-8` is also one of the identifiers generically identified.

If `identifier-8` is specified, the following rules apply to it:

- If ZERO, ZEROES, or ZEROS is specified in the WHEN phrase, `identifier-8` must not be of USAGE DISPLAY-1.
- If SPACE or SPACES is specified in the WHEN phrase, `identifier-8` must be of USAGE DISPLAY, DISPLAY-1, or NATIONAL. If `identifier-8` is a zoned or national decimal item, it must be an integer.
- If LOW-VALUE, LOW-VALUES, HIGH-VALUE, or HIGH-VALUES is specified in the WHEN phrase, `identifier-8` must be of USAGE DISPLAY or NATIONAL. If `identifier-8` is a zoned or national decimal item, it must be an integer.

If the `generic-suppression-phrase` is specified, data items are selected for potential suppression according to the following rules:

- If ZERO, ZEROES, or ZEROS is specified in the WHEN phrase, all data items except those that are defined with USAGE DISPLAY-1 are selected.
- If SPACE or SPACES is specified in the WHEN phrase, data items of USAGE DISPLAY, DISPLAY-1, or NATIONAL are selected. For zoned or national decimal items, only integers are selected.
- If LOW-VALUE, LOW-VALUES, HIGH-VALUE, or HIGH-VALUES is specified in the WHEN phrase, data items of USAGE DISPLAY or NATIONAL are selected. For zoned or national decimal items, only integers are selected.

The comparison operation that determines whether an item will be suppressed is a relation condition as shown in the table of Comparisons involving figurative constants. That is, the comparison is a numeric comparison if the value specified is ZERO, ZEROS, or ZEROS, and the item is of class numeric. For all other cases, the comparison operation is an alphanumeric, DBCS, or national comparison, depending on whether the item is of usage DISPLAY, DISPLAY-1 or NATIONAL, respectively.

When the SUPPRESS phrase is specified, a group item subordinate to `identifier-2` is suppressed in the generated XML document if all the eligible items subordinate to the group item are suppressed or if, after suppressing any subordinate items, the XML corresponding to the group item would be an empty element with no attributes. The root element is always generated, even if all the items subordinate to `identifier-2` are suppressed.
ON EXCEPTION phrase
An exception condition exists when an error occurs during generation of the XML document, for example if *identifier-1* is not large enough to contain the generated XML document. In this case, XML generation stops and the content of the receiver, *identifier-1*, is undefined. If the COUNT IN phrase is specified, *identifier-3* contains the number of character positions that were generated, which can range from 0 to the length of *identifier-1*.

If the ON EXCEPTION phrase is specified, control is transferred to *imperative-statement-1*. If the ON EXCEPTION phrase is not specified, the NOT ON EXCEPTION phrase, if any, is ignored, and control is transferred to the end of the XML GENERATE statement. Special register XML-CODE contains an exception code, as detailed in Handling XML GENERATE exceptions in the Enterprise COBOL Programming Guide.

NOT ON EXCEPTION phrase
If an exception condition does not occur during generation of the XML document, control is passed to *imperative-statement-2*, if specified, otherwise to the end of the XML GENERATE statement. The ON EXCEPTION phrase, if specified, is ignored. Special register XML-CODE contains zero after execution of the XML GENERATE statement.

END-XML phrase
This explicit scope terminator delimits the scope of XML GENERATE or XML PARSE statements. END-XML permits a conditional XML GENERATE or XML PARSE statement (that is, an XML GENERATE or XML PARSE statement that specifies the ON EXCEPTION or NOT ON EXCEPTION phrase) to be nested in another conditional statement.

The scope of a conditional XML GENERATE or XML PARSE statement can be terminated by:
- An END-XML phrase at the same level of nesting
- A separator period

END-XML can also be used with an XML GENERATE or XML PARSE statement that does not specify either the ON EXCEPTION or the NOT ON EXCEPTION phrase.

For more information about explicit scope terminators, see “Delimited scope statements” on page 270.

Nested XML GENERATE or XML PARSE statements

When a given XML GENERATE or XML PARSE statement appears as *imperative-statement-1* or *imperative-statement-2*, or as part of *imperative-statement-1* or *imperative-statement-2* of another XML GENERATE or XML PARSE statement, that given XML GENERATE or XML PARSE statement is a nested XML GENERATE or XML PARSE statement.

Nested XML GENERATE or XML PARSE statements are considered to be matched XML GENERATE and END-XML combinations, or XML PARSE and END-XML combinations, proceeding from left to right. Thus, any END-XML phrase that is encountered is matched with the nearest preceding XML GENERATE or XML PARSE statement that has not been implicitly or explicitly terminated.

Operation of XML GENERATE
The content of each eligible elementary data item within *identifier-2* that has not been suppressed from XML generation according to a SUPPRESS phrase, is converted to character format.

Only the first definition of each storage area is processed. Redefinitions of data items are not included. Data items that are effectively defined by the RENAMES clause are also not included.

For information about the format conversion of elementary data, see “Format conversion of elementary data” on page 448 and “Trimming of generated XML data” on page 449.

If the TYPE OF phrase is specified, the converted content is then processed as element character content or attribute value, according to the specifications on that phrase. If the TYPE OF phrase is not specified, by default the converted content is inserted as element character content, or, if the WITH ATTRIBUTES phrase is specified and the data item is eligible to be expressed as an attribute, as the value of the attribute, in the generated XML document.
The XML element names and attribute names are obtained from the NAME phrase if specified; otherwise by default they are derived from the data-names within identifier-2 as described in “XML element name and attribute name formation” on page 449. The names of group items that contain the selected elementary items are retained as parent elements. If the NAMESPACE-PREFIX phrase is specified, the prefix value, minus any trailing spaces, is used to qualify the start and end tag of each element.

No extra white space (new lines, indentation, and so forth) is inserted to make the generated XML more readable. An XML declaration is generated if the XML-DECLARATION phrase is specified.

If the receiving area specified by identifier-1 is not large enough to contain the resulting XML document, an error condition exists. See the description of the ON EXCEPTION phrase above for details.

If identifier-1 is longer than the generated XML document, only that part of identifier-1 in which XML is generated is changed. The rest of identifier-1 contains the data that was present before this execution of the XML GENERATE statement. To avoid referring to that data, either initialize identifier-1 to spaces before the XML GENERATE statement or specify the COUNT IN phrase.

If the COUNT IN phrase is specified, identifier-3 contains (after execution of the XML GENERATE statement) the total number of character positions (UTF-16 encoding units or bytes) that were generated. You can use identifier-3 as a reference modification length field to refer to the part of identifier-1 that contains the generated XML document.

After execution of the XML GENERATE statement, special register XML-CODE contains either zero, which indicates successful completion, or a nonzero exception code. For details, see Handling XML GENERATE exceptions in the Enterprise COBOL Programming Guide.

The XML PARSE statement also uses special register XML-CODE. Therefore if you code an XML GENERATE statement in the processing procedure of an XML PARSE statement, save the value of XML-CODE before that XML GENERATE statement executes and restore the saved value after the XML GENERATE statement terminates.

A byte order mark is not generated for XML documents that have Unicode encoding.

Format conversion of elementary data

Elementary data items within identifier-2 are converted in a sequence of several steps, some of them are optional as described below.

Conversion to character format:

Elementary data items are converted to character format depending on the type of the data item:

- Data items of category alphabetic, alphanumeric, alphanumeric-edited, DBCS, external floating-point, national, national-edited, and numeric-edited are not converted.
- Fixed-point numeric data items other than COMPUTATIONAL-5 (COMP-5) binary data items or binary data items compiled with the TRUNC(BIN) compiler option are converted as if they were moved to a numeric-edited item that has:
  - As many integer positions as the numeric item has, but with at least one integer position
  - An explicit decimal point, if the numeric item has at least one decimal position
  - The same number of decimal positions as the numeric item has
  - A leading '-' picture symbol if the data item is signed (has an S in its PICTURE clause)
- COMPUTATIONAL-5 (COMP-5) binary data items or binary data items compiled with the TRUNC(BIN) compiler option are converted in the same way as the other fixed-point numeric items, except for the number of integer positions. The number of integer positions is computed depending on the number of 9 symbols in the picture character string as follows:
  - 5 minus the number of decimal places, if the data item has 1 to 4 9 picture symbols
  - 10 minus the number of decimal places, if the data item has 5 to 9 9 picture symbols
  - 20 minus the number of decimal places, if the data item has 10 to 18 9 picture symbols
- Internal floating-point data items are converted as if they were moved to a data item as follows:
For COMP-1: an external floating-point data item with PICTURE -9.9(8)E+99
For COMP-2: an external floating-point data item with PICTURE -9.9(17)E+99 (illegal because of the number of digit positions)

- Index data items are converted as if they were declared USAGE COMP-5 PICTURE S9(9).

**Trimming:**

After any conversion to character format, leading and trailing spaces and leading zeroes are eliminated, as described under “Trimming of generated XML data” on page 449.

**Conversion to the document encoding:**

If `identifier-1` is a data item of category national, any nonnational values are converted to national format.

**Conversion of special characters to XML references:**

Any remaining instances of the five characters & (ampersand), ' (apostrophe), > (greater-than sign), < (less-than sign), and " (quotation mark) are converted into the equivalent XML references '&', 'apos;', '&gt;', '&lt;', and '&quot;', respectively.

**Replacement of out-of-range Unicode characters:**

Any remaining Unicode character that has a Unicode scalar value greater than x'FFFF' is replaced by an XML character reference. For example, if the document contains a character with Unicode scalar value x'10813', in UTF-16, that value is represented by the surrogate pair (NX'D802', NX'DC13'), which is replaced by the reference '&#x10813;'. For a document encoding of UTF-8, the byte sequence that is equivalent to character reference '&#x10813;' is X'F090A093'.

**Trimming of generated XML data**

Trimming is performed on data values after their conversion to character format.

For more information about the conversion, see “Format conversion of elementary data” on page 448.

For values converted from signed numeric values, the leading space is removed if the value is positive.

For values converted from numeric items, leading zeroes (after any initial minus sign) up to but not including the digit immediately before the actual or implied decimal point are eliminated. Trailing zeroes after a decimal point are retained. For example:

- 0000.45 becomes 0.45.
- 0013 becomes 13.
- 0000 becomes 0.

Character values from data items of class alphabetic, alphanumeric, DBCS, and national have either trailing or leading spaces removed, depending on whether the corresponding data items have left (default) or right justification, respectively. That is, trailing spaces are removed from values whose corresponding data items do not specify the JUSTIFIED clause. Leading spaces are removed from values whose data items do specify the JUSTIFIED clause. If a character value consists solely of spaces, one space remains as the value after trimming is finished.

**XML element name and attribute name formation**

In the XML documents that are generated from `identifier-2`, the XML element names and attribute names are obtained from the NAME phrase if specified; otherwise they are derived from the names of the data item specified by `identifier-2` and from any eligible data-names that are subordinate to `identifier-2`.

The formation of XML element name and attribute name is as follows:

- The exact mixed-case spelling of data-names from the data description entry is retained. The spellings from any references to data items (for example, in an OCCURS DEPENDING ON clause) are not used.
- Data-names that start with a digit are prefixed by an underscore. For example, the data-name '3D' becomes XML tag or attribute name '_3D'.
• Data-names that start with the characters 'xml', in any combination of uppercase and lowercase, are prefixed by an underscore. For example, the data-name 'Xml' becomes XML tag or attribute name '_Xml'.

DBCS data-names, when translated to Unicode, must be legal as names in the XML specification, version 1.0. For details about the XML specification, see XML specification.

**XML PARSE statement**

The XML PARSE statement is the COBOL language interface to either of two high-speed XML parsers, depending on the setting of the XMLPARSE compiler option.

The two high-speed XML parsers are:

• The z/OS XML System Services parser, for enhanced parsing capabilities. This parser is selected by the XMLPARSE(XMLSS) compiler option.

• The XML parser that is provided in the COBOL run time, for compatibility with Enterprise COBOL for z/OS Version 3. The compatible parser is selected by the XMLPARSE(COMPAT) compiler option.

The XML PARSE statement parses an XML document into its individual pieces and passes each piece, one at a time, to a user-written processing procedure.

**Format**

```
XML PARSE identifier-1 WITH ENCODING codepage
    RETURNING NATIONAL
    VALIDATING WITH identifier-2
    FILE xml-schema-name-1
    PROCESSING PROCEDURE IS procedure-name-1
    THROUGH procedure-name-2 ON EXCEPTION imperative-statement-1
    NOT ON EXCEPTION imperative-statement-2
    END-XML
```
**identifier-1**

**identifier-1** must be an elementary data item of category national, a national group, an elementary data item of category alphanumeric, or an alphanumeric group item. **identifier-1** cannot be a function-identifier. **identifier-1** contains the XML document character stream.

If **identifier-1** is a national group item, **identifier-1** is processed as an elementary data item of category national.

If **identifier-1** is of category national, its content must be encoded using Unicode UTF-16BE (CCSID 1200). If the XMLPARSE(COMPAT) compiler option is in effect, **identifier-1** must not contain any character entities that are represented using multiple encoding units. Use a character reference to represent any such characters, for example:

- "&#{x6760};" or
- "&#{x10813};"

The letter x must be lowercase.

**identifier-1** must not be a dynamic-length group item or a dynamic-length elementary item.

If **identifier-1** is of category alphanumeric, its content must be encoded using one of the character sets listed in Coded character sets for XML documents in the Enterprise COBOL Programming Guide. If the XMLPARSE(COMPAT) compiler option is in effect, and **identifier-1** is alphanumeric and contains an XML document that does not specify an encoding declaration, the XML document is parsed with the code page specified by the CODEPAGE compiler option.

If the XMLPARSE(XMLSS) compiler option is in effect, the XML document is parsed with the code page specified in the ENCODING phrase; if the ENCODING phrase is not used, the document is parsed with the code page specified by the CODEPAGE compiler option. Any encoding declaration in the XML document is ignored.

**RETURNING NATIONAL phrase**

The RETURNING NATIONAL phrase can be specified only when the XMLPARSE(XMLSS) compiler option is in effect.

When **identifier-1** references a data item of category alphanumeric and the RETURNING NATIONAL phrase is specified, XML document fragments are automatically converted to Unicode UTF-16 representation and returned to the processing procedure in the national special registers XML-NTEXT, XML-NNAMESPACE, and XML-NNAMESPACE-PREFIX.

When the RETURNING NATIONAL phrase is not specified and **identifier-1** references a data item of category alphanumeric, the XML document fragments are returned to the processing procedure in the alphanumeric special registers XML-TEXT, XML-NAMESPACE, and XML-NAMESPACE-PREFIX except that when XMLPARSE(COMPAT) is in effect, text for the ATTRIBUTE-NATIONAL-CHARACTER and CONTENT-NATIONAL-CHARACTER XML events is always returned in special register XML-NTEXT.

When **identifier-1** references a national data item, XML document fragments are always returned in Unicode UTF-16 representation in the national special registers XML-NTEXT, XML-NNAMESPACE, and XML-NNAMESPACE-PREFIX.

**VALIDATING phrase**

The VALIDATING phrase specifies that the parser should validate the XML document against an XML schema while parsing it. In Enterprise COBOL, the schema used for XML validation is in a preprocessed format known as Optimized Schema Representation or OSR. The VALIDATING phrase can be specified only when the XMLPARSE(XMLSS) compiler option is in effect.


**identifier-2** might not be a dynamic-length group item or a dynamic-length elementary item.

If the FILE keyword is not specified, **identifier-2** must reference a data item that contains the optimized XML schema. **identifier-2** must be of category alphanumeric and cannot be a function-identifier.
If the FILE keyword is specified, xml-schema-name-1 identifies an existing z/OS UNIX file or MVS data set that contains the optimized XML schema. xml-schema-name-1 must be associated with the external file name of the schema by using the XML-SCHEMA clause. For more information about the XML-SCHEMA clause, see “SPECIAL-NAMES paragraph” on page 108.

Restriction: XML validation using the FILE keyword is not supported under CICS.

During parsing with validation, normal XML events are returned as for nonvalidating parsing until an exception occurs due to a validation error or other error in the document.

When an XML document is not valid, the parser signals an XML exception and passes control to the processing procedure with special register XML-EVENT containing ‘EXCEPTION’ and special-register XML-CODE containing return code 24 in the high-order halfword and a reason code in the low-order halfword.

For information about the return code and reason code for exceptions that might occur when parsing XML documents with validation, see XML PARSE exceptions with XMLPARSE(XMLSS) in effect in the Enterprise COBOL Programming Guide.

ENCODING phrase
The ENCODING phrase can be specified only when the XMLPARSE(XMLSS) compiler option is in effect.

The ENCODING phrase specifies an encoding that is assumed for the source XML document in identifier-1. codepage must be an unsigned integer data item or an unsigned integer literal that represents a valid coded character set identifier (CCSID). The ENCODING phrase specification overrides the encoding specified by the CODEPAGE compiler option. The encoding specified in any XML declaration is always ignored.

If identifier-1 references a data item of category national, codepage must specify CCSID 1200, for Unicode UTF-16.

If identifier-1 references a data item of category alphanumeric, codepage must specify CCSID 1208 for UTF-8 or a CCSID for a supported EBCDIC or ASCII codepage. See Coded character sets for XML documents in the Enterprise COBOL Programming Guide for details.

PROCESSING PROCEDURE phrase
Specifies the name of a procedure to handle the various events that the XML parser generates.

procedure-name-1, procedure-name-2
Must name a section or paragraph in the PROCEDURE DIVISION. When both procedure-name-1 and procedure-name-2 are specified, if either is a procedure name in a declarative procedure, both must be procedure names in the same declarative procedure.

procedure-name-1
Specifies the first (or only) section or paragraph in the processing procedure.

procedure-name-2
Specifies the last section or paragraph in the processing procedure.

For each XML event, the parser transfers control to the first statement of the procedure named procedure-name-1. Control is always returned from the processing procedure to the XML parser. The point from which control is returned is determined as follows:

• If procedure-name-1 is a paragraph name and procedure-name-2 is not specified, the return is made after the execution of the last statement of the procedure-name-1 paragraph.

• If procedure-name-1 is a section name and procedure-name-2 is not specified, the return is made after the execution of the last statement of the last paragraph in the procedure-name-1 section.

• If procedure-name-2 is specified and it is a paragraph name, the return is made after the execution of the last statement of the procedure-name-2 paragraph.

• If procedure-name-2 is specified and it is a section name, the return is made after the execution of the last statement of the last paragraph in the procedure-name-2 section.
The only necessary relationship between procedure-name-1 and procedure-name-2 is that they define a consecutive sequence of operations to execute, beginning at the procedure named by procedure-name-1 and ending with the execution of the procedure named by procedure-name-2.

If there are two or more logical paths to the return point, then procedure-name-2 can name a paragraph that consists of only an EXIT statement; all the paths to the return point must then lead to this paragraph.

The processing procedure consists of all the statements at which XML events are handled. The range of the processing procedure includes all statements executed by CALL, EXIT, GO TO, GOBACK, INVOKE, MERGE, PERFORM, and SORT statements that are in the range of the processing procedure, as well as all statements in declarative procedures that are executed as a result of the execution of statements in the range of the processing procedure.

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 range of the processing procedure must not cause the execution of an XML PARSE statement, unless the XML PARSE statement is executed in a method or outermost program to which control was passed by an INVOKE or CALL statement that is executed in the range of the processing procedure.

A program executing on multiple threads can execute the same XML statement or different XML statements simultaneously.

The processing procedure can terminate the run unit with a STOP RUN statement.

For more details about the processing procedure, see “Control flow” on page 454.

ON EXCEPTION

The ON EXCEPTION phrase specifies imperative statements that are executed when the XML PARSE statement raises an exception condition.

An exception condition exists when the XML parser detects an error in processing the XML document. The parser first signals an XML exception by passing control to the processing procedure with special register XML-EVENT containing 'EXCEPTION'. The parser also provides a numeric error code in special register XML-CODE, as detailed in Handling XML PARSE exceptions in the Enterprise COBOL Programming Guide.

An exception condition also exists if the processing procedure sets XML-CODE to -1 before returning to the parser for any normal XML event. In this case, the parser does not signal an EXCEPTION XML event and parsing is terminated.

If the ON EXCEPTION phrase is specified, the parser transfers control to imperative-statement-1. If the ON EXCEPTION phrase is not specified, the NOT ON EXCEPTION phrase, if any, is ignored and control is transferred to the end of the XML PARSE statement.

Special register XML-CODE contains the numeric error code for the XML exception or -1 after execution of the XML PARSE statement.

If the processing procedure handles the XML exception event and sets XML-CODE to zero before returning control to the parser, the exception condition no longer exists. If no other unhandled exceptions occur before termination of the parser, control is transferred to imperative-statement-2 of the NOT ON EXCEPTION phrase, if specified.

NOT ON EXCEPTION

The NOT ON EXCEPTION phrase specifies imperative statements that are executed when no exception condition exists at the termination of XML PARSE processing.

If an exception condition does not exist at termination of XML PARSE processing, control is transferred to imperative-statement-2 of the NOT ON EXCEPTION phrase, if specified. If the NOT ON EXCEPTION phrase is not specified, control is transferred to the end of the XML PARSE statement. The ON EXCEPTION phrase, if specified, is ignored.

Special register XML-CODE contains zero after execution of the XML PARSE statement.
**END-XML phrase**

This explicit scope terminator delimits the scope of XML GENERATE or XML PARSE statements. END-XML permits a conditional XML GENERATE or XML PARSE statement (that is, an XML GENERATE or XML PARSE statement that specifies the ON EXCEPTION or NOT ON EXCEPTION phrase) to be nested in another conditional statement.

The scope of a conditional XML GENERATE or XML PARSE statement can be terminated by:

- An END-XML phrase at the same level of nesting
- A separator period

END-XML can also be used with an XML GENERATE or XML PARSE statement that does not specify either the ON EXCEPTION or NOT ON EXCEPTION phrase.

For more information about explicit scope terminators, see “Delimited scope statements” on page 270.

**Nested XML GENERATE or XML PARSE statements**

When a given XML GENERATE or XML PARSE statement appears as imperative-statement-1 or imperative-statement-2, or as part of imperative-statement-1 or imperative-statement-2 of another XML GENERATE or XML PARSE statement, that given XML GENERATE or XML PARSE statement is a nested XML GENERATE or XML PARSE statement.

Nested XML GENERATE or XML PARSE statements are considered to be matched XML GENERATE and END-XML, or XML PARSE and END-XML combinations proceeding from left to right. Thus, any END-XML phrase that is encountered is matched with the nearest preceding XML GENERATE or XML PARSE statement that has not been implicitly or explicitly terminated.

**Control flow**

When the XML parser receives control from an XML PARSE statement, the parser analyzes the XML document and transfers control at specific points in the process.

The points are:

- The start of the parsing process
- When a document fragment is found
- When the parser detects an error in parsing the XML document
- The end of processing the XML document

Control returns to the XML parser when the end of the processing procedure is reached.

The exchange of control between the parser and the processing procedure continues until either:

- The entire XML document has been parsed, ending with the END-OF-DOCUMENT event.
- The processing procedure terminates parsing deliberately by setting XML-CODE to -1 before returning to the parser.
- When the XMLPARSE(XMLSS) compiler option is in effect: The parser detects an exception of any kind.
- When the XMLPARSE(COMPAT) compiler option is in effect: The parser detects an exception (other than an encoding conflict) and the processing procedure does not reset special register XML-CODE to zero before returning to the parser.
- When the XMLPARSE(COMPAT) compiler option is in effect: The parser detects an encoding conflict exception and the processing procedure does not reset special register XML-CODE to zero or to the CCSID of the document encoding.

In each case, the processing procedure returns control to the parser. Then, the parser terminates and returns control to the XML PARSE statement with the XML-CODE special register containing the most recent value set by the parser or -1 (which might have been set by the parser or by the processing procedure).
For each XML event passed to the processing procedure, the XML-CODE and XML-EVENT special registers contain information about the particular event. Special register XML-EVENT is set to the event name, such as ‘START-OF-DOCUMENT’. For most events, the XML-TEXT or XML-NTEXT special register contains document text. Additionally, when the XMLPARSE(XMLSS) compiler option is in effect, the XML-NAMESPACE and XML-NAMESPACE-PREFIX or the XML-NNAMESPACE and XML-NNAMESPACE-PREFIX special registers contain a namespace identifier and namespace prefix when applicable. See “XML-EVENT” on page 25 for details.

The content of the XML-CODE special register is defined during and after execution of an XML PARSE statement. The contents of all other XML special registers are undefined outside the range of the processing procedure.

For normal XML events, special register XML-CODE contains zero when the processing procedure receives control. For XML exception events, XML-CODE contains an XML exception code as described in XML PARSE exceptions in the Enterprise COBOL Programming Guide.

For more information about the XML special registers, see:

- “XML-CODE” on page 25
- “XML-EVENT” on page 25
- “XML-INFORMATION” on page 30
- “XML-NAMESPACE” on page 31
- “XML-NAMESPACE-PREFIX” on page 32
- “XML-NNAMESPACE” on page 31
- “XML-NNAMESPACE-PREFIX” on page 33
- “XML-NTEXT” on page 33
- “XML-TEXT” on page 34

For an introduction to special registers, see “Special registers” on page 15

For more information about the EXCEPTION event and exception processing, see Handling XML PARSE exceptions in the Enterprise COBOL Programming Guide.
Part 7. Intrinsic functions
Chapter 21. Intrinsic functions

An intrinsic function is a function that performs a mathematical, character, or logical operation. You can use intrinsic functions to make reference to a data item whose value is derived automatically during execution.

Data processing problems often require the use of values that are not directly accessible in the data storage associated with the object program, but instead must be derived through performing operations on other data. An intrinsic function is a function that performs a mathematical, character, or logical operation, and thereby allows you to make reference to a data item whose value is derived automatically during execution.

The intrinsic functions can be grouped into six categories, based on the type of service performed:

- Mathematical
- Statistical
- Date/time
- Financial
- Character-handling
- General

You can reference a function by specifying its name, along with any required arguments, in a PROCEDURE DIVISION statement.

Functions are elementary data items, and return alphanumeric character, national character, numeric, or integer values. Functions cannot serve as receiving operands.

Specifying a function

This topic describes the general format of a function-identifier.

**Format: Function-identifier**

```
FUNCTION function-name-1 (argument-1) reference-modifier
```

- **function-name-1**
  
  *function-name-1* must be one of the intrinsic function names.

- **argument-1**
  
  *argument-1* must be an identifier, a literal (other than a figurative constant), or an arithmetic expression that satisfies the argument requirements for the specified function.

- **reference-modifier**
  
  Can be specified only for functions of type alphanumeric or national.

A function-identifier can be specified wherever a data item of the type of the function is allowed. The argument to a function can be any function or an expression containing a function, including another evaluation of the same function, whose result meets the requirements for the argument.
Within a PROCEDURE DIVISION statement, each function-identifier is evaluated at the same time as any reference modification or subscripting associated with an identifier in that same position would be evaluated.

**Function definition and evaluation**

The class and characteristics of a function, and the number and types of arguments it requires, are determined by its function definition.

These characteristics include:

- For functions of type alphanumeric, national, and UTF-8, the size of the returned value
- For functions of type numeric and integer, the sign of the returned value, and whether the function is integer
- The actual value returned by the function

For some functions, the class and characteristics are determined by the arguments to the function.

The evaluation of any intrinsic function is not affected by the context in which it appears; in other words, function evaluation is not affected by operations or operands outside the function. However, evaluation of a function can be affected by the attributes of its arguments.

Within a PROCEDURE DIVISION statement, each function-identifier is evaluated at the same time as any reference modification or subscripting associated with an identifier in that same position would be evaluated.

**Types of functions**

The topic introduces types of functions in COBOL.

COBOL has the following types of functions:

- Alphanumeric
- National
- UTF-8
- Numeric
- Integer

**Alphanumeric** functions are of class and category alphanumeric. The value returned has an implicit usage of DISPLAY. The number of character positions in the value returned is determined by the function definition.

**National** functions are of class and category national. The value returned has an implicit usage of NATIONAL and is represented in national characters (UTF-16). The number of character positions in the value returned is determined by the function definition.

**UTF-8** functions are of class and category UTF-8. The value returned has an implicit usage of UTF-8 and is represented in UTF-8 characters (UTF-8). The number of character positions in the value returned is determined by the function definition.

**Numeric** functions are of class and category numeric. The returned value is always considered to have an operational sign and is a numeric intermediate result. For more information, see *Using numeric intrinsic functions* in the *Enterprise COBOL Programming Guide*.

**Integer** functions are of class and category numeric. The returned value is always considered to have an operational sign and is an integer intermediate result. The number of digit positions in the value returned is determined by the function definition. For more information, see *Using numeric intrinsic functions* in the *Enterprise COBOL Programming Guide*.
Rules for usage

The topic describes rules of using different types of functions.

Alphanumeric functions

An alphanumeric function can be specified anywhere in the general formats that a data item of class and category alphanumeric is permitted and where the rules associated with the general formats do not specifically prohibit reference to functions, except as noted below.

An alphanumeric function can be used as an argument for any function that allows an alphanumeric argument.

Reference modification of an alphanumeric function is allowed. If reference modification is specified for a function, the evaluation of the reference modification takes place immediately after the evaluation of the function; that is, the function's returned value is reference-modified.

An alphanumeric function cannot be used:

• As a receiving operand of any statement
• Where the rules associated with the general formats require the data item being referenced to have particular characteristics (such as class and category, usage, size, and permissible values) and the evaluation of the function according to its definition and the particular arguments specified would not have those characteristics

National functions

A national function can be specified anywhere in the general formats that a data item of class and category national is permitted and where the rules associated with the general formats do not specifically prohibit reference to functions, except as noted below.

A national function can be used as an argument for any function that allows a national argument.

Reference modification of a national function is allowed. If reference modification is specified for a function, the evaluation of the reference modification takes place immediately after the evaluation of the function; that is, the function's returned value is reference-modified.

A national function cannot be used:

• As a receiving operand of any statement
• Where the rules associated with the general formats require the data item being referenced to have particular characteristics (such as class and category, usage, size, and permissible values) and the evaluation of the function according to its definition and the particular arguments specified would not have those characteristics

UTF-8 functions

A UTF-8 function can be specified anywhere in the general formats that a data item of class and category UTF-8 is permitted and where the rules associated with the general formats do not specifically prohibit reference to functions, except as noted below.

A UTF-8 function can be used as an argument for any function that allows a UTF-8 argument.

Reference modification of a UTF-8 function is allowed. If reference modification is specified for a function, the evaluation of the reference modification takes place immediately after the evaluation of the function; that is, the function's returned value is reference-modified.

A UTF-8 function cannot be used:

• As a receiving operand of any statement
• Where the rules associated with the general formats require the data item being referenced to have particular characteristics (such as class and category, usage, size, and permissible values) and the evaluation of the function according to its definition and the particular arguments specified would not have those characteristics

Numeric functions

A numeric function can be used only where an arithmetic expression can be specified.
A numeric function can be referenced as an argument for a function that allows a numeric argument. A numeric function cannot be used where an integer operand is required, even if the particular reference would yield an integer value. The INTEGER or INTEGER-PART functions can be used to force the type of a numeric argument to be an integer.

**Integer functions**
An integer function can be used only where an arithmetic expression can be specified.

An integer function can be referenced as an argument for a function that allows a numeric argument.

**Usage notes:**
- A function-identifier cannot be used in the BY REFERENCE phrase of a CALL statement (that is, `identifier-2` of the CALL statement must not be a function-identifier).
- The COPY statement accepts function-identifiers of all types in the REPLACING phrase.

**Arguments**
The value returned by some functions is determined by the arguments specified in the function-identifier when the functions are evaluated. Some functions require no arguments; others require a fixed number of arguments, and still others accept a variable number of arguments.

An argument must be one of the following items:
- A data item identifier
- An arithmetic expression
- A function-identifier
- A literal other than a figurative constant
- A special-register

See “Function definitions” on page 465 for function-specific argument specifications.

The types of arguments are:

- **Alphabetic**
  An elementary data item of the class alphabetic or an alphanumeric literal containing only alphabetic characters. The content of the argument is used to determine the value of the function. The length of the argument can be used to determine the value of the function.

- **Alphanumeric**
  A data item of the class alphabetic or alphanumeric or an alphanumeric literal. The content of the argument is used to determine the value of the function. The length of the argument can be used to determine the value of the function.

- **DBCS**
  An elementary data item of class DBCS or a DBCS literal. The content of the argument is used to determine the value of the function. The length of the argument can be used to determine the value of the function. (A DBCS data item or literal can be used as an argument only for the NATIONAL-OF function.)

- **National**
  A data item of class national (category national, national-edited, or numeric-edited). The content of the argument is used to determine the value of the function. The length of the argument can be used to determine the value of the function.

- **UTF-8**
  A data item of class UTF-8 (category UTF-8). The content of the argument is used to determine the value of the function. The length of the argument can be used to determine the value of the function.

- **Integer**
  An arithmetic expression that always results in an integer value. The value of the expression, including its sign, is used to determine the value of the function.
**Numeric**

An arithmetic expression. The expression can include numeric literals and data items of categories numeric, internal floating-point, and external floating-point. The numeric data items can have any usage permitted for the category of the data item (including NATIONAL). The value of the expression, including its sign, is used to determine the value of the function.

**Keyword**

A keyword shall be specified in accordance with the intrinsic function definition. The TRIM intrinsic function is an example of an intrinsic function with a keyword argument. The keywords LEADING and TRAILING may be specified as the second and optional argument of TRIM.

Some functions place constraints on their arguments, such as the acceptable range of values. If the values assigned as arguments for a function do not comply with specified constraints, the returned value is undefined.

If a nested function is used as an argument, the evaluation of its arguments is not affected by the arguments in the outer function.

Only those arguments at the same function level interact with each other. This interaction occurs in two areas:

- The computation of an arithmetic expression that appears as a function argument is affected by other arguments for that function.
- The evaluation of the function takes into consideration the attributes of all of its arguments.

When a function is evaluated, its arguments are evaluated individually in the order specified in the list of arguments, from left to right. The argument being evaluated can be a function-identifier or an expression that includes function-identifiers.

If an arithmetic expression is specified as an argument and if the first operator in the expression is a unary plus or a unary minus, the expression must be immediately preceded by a left parenthesis.

Floating-point literals are allowed wherever a numeric argument is allowed and in arithmetic expressions used in functions that allow a numeric argument.

Internal floating-point items and external floating-point items (both display floating-point and national floating-point) can be used wherever a numeric argument is allowed and in arithmetic expressions as arguments to a function that allows a numeric argument.

Floating-point items and floating-point literals cannot be used where an integer argument is required or where an argument of class alphanumeric or national is required (such as in the LOWER-CASE, REVERSE, UPPER-CASE, NUMVAL, and NUMVAL-C functions).

Where a function allows an alphanumeric, a national, or a UTF-8 argument to be specified, an alphanumeric group, a national or a UTF-8 group, respectively, can also be specified. However, an unbounded group cannot be specified as a function argument, except when it is used in the LENGTH intrinsic function.

**Examples**

See examples of using different types of intrinsic functions.

The following statement illustrates the use of intrinsic function UPPER-CASE to replace each lowercase letter in an alphanumeric argument with the corresponding uppercase letter.

```plaintext
MOVE FUNCTION UPPER-CASE('hello') TO DATA-NAME.
```

This statement moves HELLO into DATA-NAME.

The following statement illustrates the use of intrinsic function LOWER-CASE to replace each uppercase letter in a national argument with the corresponding lowercase letter.

```plaintext
MOVE FUNCTION LOWER-CASE(N'HELLO') TO N-DATA-NAME.
```
This statement moves national characters hello into N-DATA-NAME.

The following statement illustrates the use of a numeric intrinsic function:

```
COMPUTE NUM-ITEM = FUNCTION SUM(A B C)
```

This statement uses the numeric function SUM to add the values of A, B, and C and places the result in NUM-ITEM.

**ALL subscripting**

When a function allows an argument to be repeated a variable number of times, you can refer to a table by specifying the data-name and any qualifiers that identify the table. This can be followed immediately by subscripting where one or more of the subscripts is the word ALL.

**Tip:** The evaluation of an ALL subscript must result in at least one argument or the value returned by the function will be undefined; however, the situation can be diagnosed at run time by specifying the SSRANGE compiler option.

Specifying ALL as a subscript is equivalent to specifying all table elements possible using every valid subscript in that subscript position.

For a table argument specified as Table-name(ALL), the order of the implicit specification of each table element as an argument is from left to right, where the first (or leftmost) argument is Table-name(1) and ALL has been replaced by 1. The next argument is Table-name(2), where the subscript has been incremented by 1. This process continues, with the subscript being incremented by 1 to produce an implicit argument, until the ALL subscript has been incremented through its range of values.

For example,

```
FUNCTION MAX(Table(ALL))
```

is equivalent to

```
FUNCTION MAX(Table(1) Table(2) Table(3) ... Table(n))
```

where \( n \) is the number of elements in Table.

If there are multiple ALL subscripts, Table-name(ALL, ALL, ALL), the first implicit argument is Table-name(1, 1, 1), where each ALL has been replaced by 1. The next argument is Table-name(1, 1, 2), where the rightmost subscript has been incremented by 1. The subscript represented by the rightmost ALL is incremented through its range of values to produce an implicit argument for each value.

Once a subscript specified as ALL has been incremented through its range of values, the next subscript to the left that is specified as ALL is incremented by 1. Each subscript specified as ALL to the right of the newly incremented subscript is set to 1 to produce an implicit argument. Once again, the subscript represented by the rightmost ALL is incremented through its range of values to produce an implicit argument for each value. This process is repeated until each subscript specified as ALL has been incremented through its range of values.

For example,

```
FUNCTION MAX(Table(ALL, ALL))
```

is equivalent to

```
FUNCTION MAX(Table(1, 1) Table(1, 2) Table(1, 3) ... Table(1, n) Table(2, 1) Table(2, 2) Table(2, 3) ... Table(2, n) Table(3, 1) Table(3, 2) Table(3, 3) ... Table(3, n))
```
where \( n \) is the number of elements in the column dimension of \( \text{Table} \), and \( m \) is the number of elements in the row dimension of \( \text{Table} \).

ALL subscripts can be combined with literal, data-name, or index-name subscripts to reference multidimensional tables.

For example,

\[
\text{FUNCTION MAX(Table(ALL, 2))}
\]

is equivalent to

\[
\text{FUNCTION MAX(Table(1, 2)) Table(2, 2) Table(3, 2) \ldots Table(m, 2))}
\]

where \( m \) is the number of elements in the row dimension of \( \text{Table} \).

If an ALL subscript is specified for an argument and the argument is reference-modified, then the reference-modifier is applied to each of the implicitly specified elements of the table.

If an ALL subscript is specified for an operand that is reference-modified, the reference-modifier is applied to each of the implicitly specified elements of the table.

If the ALL subscript is associated with an OCCURS DEPENDING ON clause, the range of values is determined by the object of the OCCURS DEPENDING ON clause.

For example, given a payroll record definition such as:

01 PAYROLL.
  02 PAYROLL-WEEK PIC 99.
  02 PAYROLL-HOURS PIC 999 OCCURS 1 TO 52 DEPENDING ON PAYROLL-WEEK.

The following COMPUTE statements could be used to identify total year-to-date hours, the maximum hours worked in any week, and the specific week corresponding to the maximum hours:

\[
\begin{align*}
\text{COMPUTE YTD-HOURS} &= \text{FUNCTION SUM (PAYROLL-HOURS(ALL))} \\
\text{COMPUTE MAX-HOURS} &= \text{FUNCTION MAX (PAYROLL-HOURS(ALL))} \\
\text{COMPUTE MAX-WEEK} &= \text{FUNCTION ORD-MAX (PAYROLL-HOURS(ALL))}
\end{align*}
\]

In these function invocations, the subscript ALL is used to reference all elements of the PAYROLL-HOURS array (depending on the execution time value of the PAYROLL-WEEK field).

## Function definitions

This section provides an overview of the argument type, function type, and value returned for each of the intrinsic functions.

For more information about the intrinsic functions, see Table 54 on page 466.

Argument types and function types are abbreviated as follows:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Alphabetic</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Meaning</td>
</tr>
<tr>
<td>--------------</td>
<td>---------</td>
</tr>
<tr>
<td>D</td>
<td>DBCS</td>
</tr>
<tr>
<td>I</td>
<td>Integer</td>
</tr>
<tr>
<td>K</td>
<td>Keyword</td>
</tr>
<tr>
<td>N</td>
<td>Numeric</td>
</tr>
<tr>
<td>O</td>
<td>Other, as specified in the function definition (pointer, function-pointer, procedure-pointer, or object reference)</td>
</tr>
<tr>
<td>U</td>
<td>National</td>
</tr>
<tr>
<td>X</td>
<td>Alphanumeric</td>
</tr>
<tr>
<td>UTF8</td>
<td>UTF-8</td>
</tr>
</tbody>
</table>

Each intrinsic function is described in detail in the topics that follow the table below.

### Table 54. Table of functions

<table>
<thead>
<tr>
<th>Function name</th>
<th>Arguments</th>
<th>Function type</th>
<th>Value returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>N1</td>
<td>I or N</td>
<td>Absolute value of N1</td>
</tr>
<tr>
<td>ACOS</td>
<td>N1</td>
<td>N</td>
<td>Arccosine of N1</td>
</tr>
<tr>
<td>ANNUITY</td>
<td>N1, I2</td>
<td>N</td>
<td>Ratio of annuity paid for I2 periods at interest of N1 to initial investment of one</td>
</tr>
<tr>
<td>ASIN</td>
<td>N1</td>
<td>N</td>
<td>Arcsine of N1</td>
</tr>
<tr>
<td>ATAN</td>
<td>N1</td>
<td>N</td>
<td>Arctangent of N1</td>
</tr>
<tr>
<td>BIT-OF</td>
<td>A1, D1, I1, N1, X1, U1, or O1</td>
<td>X</td>
<td>Alphanumeric character string consisting of characters &quot;1&quot; and &quot;0&quot; that correspond to the binary value of each byte in the argument</td>
</tr>
<tr>
<td>BIT-TO-CHAR</td>
<td>X1</td>
<td>X</td>
<td>Character string consisting of bytes that correspond to the bit pattern indicated by the sequence of &quot;0&quot; and &quot;1&quot; characters in the argument</td>
</tr>
<tr>
<td>BYTE-LENGTH</td>
<td>A1, D1, N1, X1, U1, or O1</td>
<td>I</td>
<td>Integer that is equal to the length of the argument in bytes</td>
</tr>
<tr>
<td>CHAR</td>
<td>I1</td>
<td>X</td>
<td>Character in position I1 of program collating sequence</td>
</tr>
<tr>
<td>COS</td>
<td>N1</td>
<td>N</td>
<td>Cosine of N1</td>
</tr>
<tr>
<td>CURRENT-DATE</td>
<td>None</td>
<td>X</td>
<td>Current date and time and difference from Greenwich mean time</td>
</tr>
<tr>
<td>DATE-OF-INTEGER</td>
<td>I1</td>
<td>I</td>
<td>Standard date equivalent (YYYYMMDD) of integer date</td>
</tr>
<tr>
<td>DATE-TO-YYYYMMDD</td>
<td>I1, I2</td>
<td>I</td>
<td>Standard date equivalent (YYYYMMDD) of I1 (standard date with a windowed year, YYMMDD), according to the 100-year interval whose ending year is specified by the sum of I2 and the year at execution time</td>
</tr>
<tr>
<td>Function name</td>
<td>Arguments</td>
<td>Function type</td>
<td>Value returned</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------</td>
<td>---------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DAY-OF-INTEGER</td>
<td>I1</td>
<td>I</td>
<td>Julian date equivalent (YYYYDDD) of integer date</td>
</tr>
<tr>
<td>DAY-TO-YYYYYDDD</td>
<td>I1, I2</td>
<td>I</td>
<td>Julian date equivalent (YYYYYDDD) of I1 (Julian date with a windowed year, YYDDD), according to the 100-year interval whose ending year is specified by the sum of I2 and the year at execution time</td>
</tr>
<tr>
<td>DISPLAY-OF</td>
<td>U1 or U1, I2</td>
<td>X</td>
<td>Each character in U1 converted to a corresponding character representation using a code page identified by I2, if specified, or a default code page selected at compile time if I2 is unspecified</td>
</tr>
<tr>
<td>E</td>
<td>None</td>
<td>N</td>
<td>Approximation of $e$, the base of natural logarithms</td>
</tr>
<tr>
<td>EXP</td>
<td>N1</td>
<td>N</td>
<td>Approximation of the value of $e$ raised to the power of N1</td>
</tr>
<tr>
<td>EXP10</td>
<td>N1</td>
<td>N</td>
<td>Approximation of the value of 10 raised to the power of N1</td>
</tr>
<tr>
<td>FACTORIAL</td>
<td>I1</td>
<td>I</td>
<td>Factorial of I1</td>
</tr>
<tr>
<td>HEX-OF</td>
<td>A1, D1, I1, N1, X1, U1, or O1</td>
<td>X</td>
<td>Alphanumeric character string consisting of the bytes of the argument converted to a hexadecimal representation</td>
</tr>
<tr>
<td>HEX-TO-CHAR</td>
<td>X1</td>
<td>X</td>
<td>Character string consisting of bytes that correspond to the hexadecimal digit characters in the argument</td>
</tr>
<tr>
<td>INTEGER</td>
<td>N1</td>
<td>I</td>
<td>The greatest integer not greater than N1</td>
</tr>
<tr>
<td>INTEGER-OF-DATE</td>
<td>I1</td>
<td>I</td>
<td>Integer date equivalent of standard date (YYYYMMDD)</td>
</tr>
<tr>
<td>INTEGER-OF-DAY</td>
<td>I1</td>
<td>I</td>
<td>Integer date equivalent of Julian date (YYYYYDDD)</td>
</tr>
<tr>
<td>INTEGER-PART</td>
<td>N1</td>
<td>I</td>
<td>Integer part of N1</td>
</tr>
<tr>
<td>LENGTH</td>
<td>A1, N1, O1, X1, or U1</td>
<td>I</td>
<td>Length of argument in national character positions or in alphanumeric character positions or bytes, depending on the argument type</td>
</tr>
<tr>
<td>LOG</td>
<td>N1</td>
<td>N</td>
<td>Natural logarithm of N1</td>
</tr>
<tr>
<td>LOG10</td>
<td>N1</td>
<td>N</td>
<td>Logarithm to base 10 of N1</td>
</tr>
<tr>
<td>LOWER-CASE</td>
<td>A1 or X1</td>
<td>X</td>
<td>All letters in the argument set to lowercase</td>
</tr>
<tr>
<td></td>
<td>U1</td>
<td>U</td>
<td>All letters in the argument set to lowercase</td>
</tr>
<tr>
<td>Function name</td>
<td>Arguments</td>
<td>Function type</td>
<td>Value returned</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------</td>
<td>---------------</td>
<td>----------------</td>
</tr>
<tr>
<td>MAX</td>
<td>A1...</td>
<td>X</td>
<td>Value of maximum argument; note that the type of function depends on the arguments</td>
</tr>
<tr>
<td></td>
<td>I1...</td>
<td>I</td>
<td>Value of maximum argument; note that the type of function depends on the arguments</td>
</tr>
<tr>
<td></td>
<td>N1...</td>
<td>N</td>
<td>Value of maximum argument; note that the type of function depends on the arguments</td>
</tr>
<tr>
<td></td>
<td>X1...</td>
<td>X</td>
<td>Value of maximum argument; note that the type of function depends on the arguments</td>
</tr>
<tr>
<td></td>
<td>U1...</td>
<td>U</td>
<td>Value of maximum argument; note that the type of function depends on the arguments</td>
</tr>
<tr>
<td>MEAN</td>
<td>N1...</td>
<td>N</td>
<td>Arithmetic mean of arguments</td>
</tr>
<tr>
<td>MEDIAN</td>
<td>N1...</td>
<td>N</td>
<td>Median of arguments</td>
</tr>
<tr>
<td>MIDRANGE</td>
<td>N1...</td>
<td>N</td>
<td>Mean of minimum and maximum arguments</td>
</tr>
<tr>
<td>MIN</td>
<td>A1...</td>
<td>X</td>
<td>Value of minimum argument; note that the type of function depends on the arguments</td>
</tr>
<tr>
<td></td>
<td>I1...</td>
<td>I</td>
<td>Value of minimum argument; note that the type of function depends on the arguments</td>
</tr>
<tr>
<td></td>
<td>N1...</td>
<td>N</td>
<td>Value of minimum argument; note that the type of function depends on the arguments</td>
</tr>
<tr>
<td></td>
<td>X1...</td>
<td>X</td>
<td>Value of minimum argument; note that the type of function depends on the arguments</td>
</tr>
<tr>
<td></td>
<td>U1...</td>
<td>U</td>
<td>Value of minimum argument; note that the type of function depends on the arguments</td>
</tr>
<tr>
<td>MOD</td>
<td>I1, I2</td>
<td>I</td>
<td>I1 modulo I2</td>
</tr>
<tr>
<td>NATIONAL-OF</td>
<td>A1, X1, or D1</td>
<td>U</td>
<td>The characters in the argument converted to national characters, using the code page identified by I2, if specified, or a default code page selected at compile time if I2 is unspecified</td>
</tr>
<tr>
<td></td>
<td>A1, X1, or D1; I2</td>
<td>U</td>
<td>The characters in the argument converted to national characters, using the code page identified by I2, if specified, or a default code page selected at compile time if I2 is unspecified</td>
</tr>
<tr>
<td>NUMVAL</td>
<td>X1 or U1</td>
<td>N</td>
<td>Numeric value of simple numeric string</td>
</tr>
<tr>
<td>NUMVAL-C</td>
<td>X1 or U1; X1, X2; U1, U2</td>
<td>N</td>
<td>Numeric value of numeric string with optional commas and currency sign</td>
</tr>
<tr>
<td>Function name</td>
<td>Arguments</td>
<td>Function type</td>
<td>Value returned</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>NUMVAL-F</td>
<td>X1 or U1</td>
<td>N</td>
<td>Numeric value or approximation of the numeric value represented by the alphanumeric character string or national character string specified as the argument</td>
</tr>
<tr>
<td>ORD</td>
<td>A1 or X1</td>
<td>I</td>
<td>Ordinal position of the argument in collating sequence</td>
</tr>
<tr>
<td>ORD-MAX</td>
<td>A1..., N1..., X1..., or U1...</td>
<td>I</td>
<td>Ordinal position of maximum argument</td>
</tr>
<tr>
<td>ORD-MIN</td>
<td>A1..., N1..., X1..., or U1...</td>
<td>I</td>
<td>Ordinal position of minimum argument</td>
</tr>
<tr>
<td>PI</td>
<td>None</td>
<td>N</td>
<td>Value that is an approximation of (\pi), the ratio of the circumference of a circle to its diameter.</td>
</tr>
<tr>
<td>PRESENT-VALUE</td>
<td>N1, N2...</td>
<td>N</td>
<td>Present value of a series of future period-end amounts, N2, at a discount rate of N1</td>
</tr>
<tr>
<td>RANDOM</td>
<td>I1, none</td>
<td>N</td>
<td>Random number</td>
</tr>
<tr>
<td>RANGE</td>
<td>I1...</td>
<td>I</td>
<td>Value of maximum argument minus value of minimum argument; note that the type of function depends on the arguments.</td>
</tr>
<tr>
<td></td>
<td>N1...</td>
<td>N</td>
<td>Value of maximum argument minus value of minimum argument; note that the type of function depends on the arguments.</td>
</tr>
<tr>
<td>REM</td>
<td>N1, N2</td>
<td>N</td>
<td>Remainder of N1/N2</td>
</tr>
<tr>
<td>REVERSE</td>
<td>A1 or X1</td>
<td>X</td>
<td>Reverse order of the characters of the argument</td>
</tr>
<tr>
<td></td>
<td>U1</td>
<td>U</td>
<td>Reverse order of the characters of the argument</td>
</tr>
<tr>
<td>SIGN</td>
<td>N1</td>
<td>I</td>
<td>+1, 0, or -1 depending on the sign of the argument</td>
</tr>
<tr>
<td>SIN</td>
<td>N1</td>
<td>N</td>
<td>Sine of N1</td>
</tr>
<tr>
<td>SQRT</td>
<td>N1</td>
<td>N</td>
<td>Square root of N1</td>
</tr>
<tr>
<td>STANDARD-DEVIAITION</td>
<td>N1...</td>
<td>N</td>
<td>Standard deviation of arguments</td>
</tr>
<tr>
<td>SUM</td>
<td>I1...</td>
<td>I</td>
<td>Sum of arguments; note that the type of function depends on the arguments.</td>
</tr>
<tr>
<td></td>
<td>N1...</td>
<td>N</td>
<td>Sum of arguments; note that the type of function depends on the arguments.</td>
</tr>
<tr>
<td>Function name</td>
<td>Arguments</td>
<td>Function type</td>
<td>Value returned</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------</td>
<td>---------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| TEST-NUMVAL      | X1 or U1      | I             | • If the content of argument-1 conforms to the argument rules for the NUMVAL function, the returned value is 0.  
|                  |               |               | • If one or more characters are in error, the returned value is the position of the first character in error.  
|                  |               |               | • Otherwise, the returned value is (FUNCTION LENGTH (argument-1) + 1).               |
| TEST-NUMVAL-C    | X1 or U1; X1, X2; U1, U2 | I             | • If the content of argument-1 conforms to the argument rules for the NUMVAL-C function, the returned value is 0.  
|                  |               |               | • If one or more characters are in error, the returned value is the position of the first character in error.  
|                  |               |               | • Otherwise, the returned value is (FUNCTION LENGTH (argument-1) + 1).               |
| TEST-NUMVAL-F    | X1 or U1      | I             | • If the content of argument-1 conforms to the argument rules for the NUMVAL-F function, the returned value is 0.  
|                  |               |               | • If one or more characters are in error, the returned value is the position of the first character in error.  
|                  |               |               | • Otherwise, the returned value is (FUNCTION LENGTH (argument-1) + 1).               |
| TRIM             | A1, X1, or U1; K1 | X or U        | Character string that contains the characters in A1, X1, or U1 with leading spaces or trailing spaces deleted (if K1 is specified) |
|                  | A1, X1, or U1 | X or U        | Character string that contains the characters in A1, X1, or U1 with both leading spaces and trailing spaces deleted (if K1 is unspecified) |
| ULENGTH          | A1, X1, or U1 | I             | Length of A1, X1, or U1 in UTF-8 or UTF-16 characters                         |
| UPOS             | A1, X1, or U1, I2 | I             | Index of the I2th UTF-8 or UTF-16 character of A1, X1, or U1                 |
| UPPER-CASE       | A1 or X1      | X             | All letters in the argument set to uppercase                                |
|                  | U1            | U             | All letters in the argument set to uppercase                                |
| USUBSTR          | A1, X1, or U1, I2, I3 | X or U   | Alphanumeric character string that contains the I3 UTF-8 or UTF-16 characters of A1, X1, or U1, starting at the I2th character position |
| USUPPLEMENTARY   | A1, X1 or U1  | I             | Index of the first Unicode supplementary character of A1, X1 or U1           |
| UUID4            | None          | X             | 36-character alphanumeric string that is a version 4 universally unique identifier (UUID) |
### ABS

The ABS function returns the absolute value of the argument.

The function type depends on the argument type as follows:

<table>
<thead>
<tr>
<th>Argument type</th>
<th>Function type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Integer</td>
</tr>
<tr>
<td>Numeric</td>
<td>Numeric</td>
</tr>
</tbody>
</table>

**Format**

```
FUNCTION ABS (argument-1)
```

**argument-1**

Must be of class numeric.

The equivalent arithmetic expression is as follows:

- When the value of `argument-1` is zero or positive, `(argument-1)` is returned.
- When the value of `argument-1` is negative, `(- (argument-1))` is returned.

### ACOS

The ACOS function returns a numeric value in radians that approximates the arccosine of the argument specified.

The function type is numeric.

**Format**

```
FUNCTION ACOS (argument-1)
```
**ANNUITY**

The ANNUITY function returns a numeric value that approximates the ratio of an annuity paid at the end of each period, for a given number of periods, at a given interest rate, to an initial value of one.

The number of periods is specified by argument-2; the rate of interest is specified by argument-1. For example, if argument-1 is zero and argument-2 is four, the value returned is the approximation of the ratio \( \frac{1}{4} \).

The function type is numeric.

Format

```
FUNCTION ANNUITY (argument-1 argument-2)
```

**argument-1**

Must be class numeric. The value of argument-1 must be greater than or equal to zero.

**argument-2**

Must be a positive integer.

When the value of argument-1 is zero, the value returned by the function is the approximation of: \( \frac{1}{\text{argument-2}} \)

When the value of argument-1 is not zero, the value of the function is the approximation of: \( \frac{\text{argument-1}}{(1 - (1 + \text{argument-1})^{-\text{argument-2}})} \)

**ASIN**

The ASIN function returns a numeric value in radians that approximates the arcsine of the argument specified.

The function type is numeric.

Format

```
FUNCTION ASIN (argument-1)
```

**argument-1**

Must be class numeric. The value of argument-1 must be greater than or equal to -1 and less than or equal to +1.

The returned value is the approximation of the arcsine of argument-1 and is greater than or equal to -\( \frac{\text{Pi}}{2} \) and less than or equal to +\( \frac{\text{Pi}}{2} \).
**ATAN**

The ATAN function returns a numeric value in radians that approximates the arctangent of the argument specified.

The function type is numeric.

**Format**

```
FUNCTION ATAN (argument-1)
```

**argument-1**

Must be class numeric.

The returned value is the approximation of the arctangent of `argument-1` and is greater than -Pi/2 and less than +Pi/2.

**BIT-OF**

The BIT-OF function returns an alphanumeric character string consisting of characters "1" and "0" that correspond to the binary value of each byte in the input argument.

The type of the function is alphanumeric.

**Format**

```
FUNCTION BIT-OF (argument-1)
```

**argument-1**

Can be a data item, literal, or intrinsic function result of any data class. `argument-1` identifies the source character string for the conversion.

The returned value is an alphanumeric character string consisting of the bytes of `argument-1` converted to the bit pattern corresponding to the binary value of each byte in `argument-1`. The length of the output character string in bytes is eight times the length of `argument-1` in bytes.

**Note:** If `argument-1` is invalid, the behavior is undefined.

**Examples**

- `FUNCTION BIT-OF('Hello, world!')` returns `11001000100001011001001110010011100101100110101101000000101001101001011010010110110011001100100111000010001011010`
- `FUNCTION BIT-OF(BIN) returns '00000000000000000000000000001100'`
- `FUNCTION BIT-OF(PAC) returns '000100100011010001011111'`
- `FUNCTION BIT-OF(ZON) returns '000100100011010001011111'`
FUNCTION BIT-OF(ZON) returns '1111000111110010111100111111010011110101'

• FUNCTION BIT-OF(NATIONAL-OF(' ')) returns '000000000100000'

BIT-TO-CHAR

The BIT-TO-CHAR function returns a character string consisting of bytes that correspond to the bit pattern indicated by the sequence of "0" and "1" characters in the input argument.

The function type is alphanumeric.

Format

FUNCTION BIT-TO-CHAR (argument-1)

argument-1

Must be an alphanumeric literal, alphanumeric data item, or alphanumeric group item. argument-1 must consist only of the characters "0" and "1". The length of argument-1 must be a multiple of 8 bytes.

The returned value is a character string consisting of bytes that correspond to the bit pattern indicated by the sequence of "0" and "1" characters in argument-1. The length of the result string is equal to the length of the input string divided by 8.

Example

MOVE '1111110010001000' TO MY-BIT-DATA

FUNCTION BIT-TO-CHAR(MY-BIT-DATA) returns a character string with value x'FC88'.

BYTE-LENGTH

The BYTE-LENGTH function returns an integer that is equal to the length of the argument in bytes.

The function type is integer.

Format

FUNCTION BYTE-LENGTH (argument-1)

argument-1

Can be:

• An alphanumeric, national, UTF-8, or DBCS literal
• A group item (including unbounded groups) or an elementary data item of any class, including DBCS
• A data item described with USAGE POINTER, PROCEDURE-POINTER, FUNCTION-POINTER, or OBJECT REFERENCE
• The ADDRESS OF special register
• The LENGTH OF special register
• The XML-NTEXT special register
• The XML-TEXT special register

The returned value is a nine-digit integer determined as follows:

• The returned value is an integer that is the length of argument-1 in number of bytes.
If \textit{argument-1} is an alphanumeric, national or UTF-8 group item, the value returned is equal to the length of \textit{argument-1} in bytes. If any data item subordinate to \textit{argument-1} is described with the DEPENDING phrase of the OCCURS clause, the length of \textit{argument-1} is determined using the contents of the data item specified in the DEPENDING phrase. This evaluation is accomplished according to the rules of the OCCURS clause for a sending data item. For more information, see the discussions of the “OCCURS clause” on page 182 and the “USAGE clause” on page 218.

The returned value includes implicit FILLER positions, if any.

The only difference between the BYTE-LENGTH and LENGTH functions is that BYTE-LENGTH always returns the byte length of \textit{argument-1}, even when \textit{argument-1} is of class national or UTF-8. The BYTE-LENGTH function also accepts DBCS arguments.

Function BYTE-LENGTH is similar to the LENGTH OF special register, which also always returns the byte length of its argument, but the LENGTH OF special register can be used in more contexts. For more information, see Finding the length of data items in the Enterprise COBOL Programming Guide.

**Related references**

“LENGTH” on page 484
“LENGTH OF” on page 19

### CHAR

The CHAR function returns a one-character alphanumeric value that is a character in the program collating sequence having the ordinal position equal to the value of the argument specified.

The function type is alphanumeric.

**Format**

\[
\text{FUNCTION CHAR (argument-1)}
\]

\textit{argument-1}

Must be an integer. The value must be greater than zero and less than or equal to the number of positions in the collating sequence associated with alphanumeric data items (a maximum of 256).

If more than one character has the same position in the program collating sequence, the character returned as the function value is that of the first literal specified for that character position in the ALPHABET clause.

If the current program collating sequence was not specified by an ALPHABET clause, the single-byte EBCDIC collating sequence is used. (See “Conditional expressions” on page 246.)

### COS

The COS function returns a numeric value that approximates the cosine of the angle or arc specified by the argument in radians.

The function type is numeric.

**Format**

\[
\text{FUNCTION COS (argument-1)}
\]

\textit{argument-1}

Must be class numeric.

The returned value is the approximation of the cosine of the argument and is greater than or equal to -1 and less than or equal to +1.
The CURRENT-DATE function returns a 21-character alphanumeric value that represents the calendar date, time of day, and time differential from Greenwich mean time provided by the system on which the function is evaluated.

The function type is alphanumeric.

### Format

```
FUNCTION CURRENT-DATE
```

Reading from left to right, the 21 character positions of the returned value are as follows:

<table>
<thead>
<tr>
<th>Character positions</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-4</td>
<td>Four numeric digits of the year in the Gregorian calendar</td>
</tr>
<tr>
<td>5-6</td>
<td>Two numeric digits of the month of the year, in the range 01 through 12</td>
</tr>
<tr>
<td>7-8</td>
<td>Two numeric digits of the day of the month, in the range 01 through 31</td>
</tr>
<tr>
<td>9-10</td>
<td>Two numeric digits of the hours past midnight, in the range 00 through 23</td>
</tr>
<tr>
<td>11-12</td>
<td>Two numeric digits of the minutes past the hour, in the range 00 through 59</td>
</tr>
<tr>
<td>13-14</td>
<td>Two numeric digits of the seconds past the minute, in the range 00 through 59</td>
</tr>
<tr>
<td>15-16</td>
<td>Two numeric digits of the hundredths of a second past the second, in the range 00 through 99. The value 00 is returned if the system on which the function is evaluated does not have the facility to provide the fractional part of a second.</td>
</tr>
<tr>
<td>17</td>
<td>Either the character ‘-‘ or the character ‘+‘. The character ‘-‘ is returned if the local time indicated in the previous character positions is behind Greenwich mean time. The character ‘+‘ is returned if the local time indicated is the same as or ahead of Greenwich mean time. The character ‘0‘ is returned if the system on which this function is evaluated does not have the facility to provide the local time differential factor.</td>
</tr>
<tr>
<td>18-19</td>
<td>If character position 17 is ‘-‘, two numeric digits are returned in the range 00 through 12 indicating the number of hours that the reported time is behind Greenwich mean time. If character position 17 is ‘+‘, two numeric digits are returned in the range 00 through 13 indicating the number of hours that the reported time is ahead of Greenwich mean time. If character position 17 is ‘0‘, the value 00 is returned.</td>
</tr>
<tr>
<td>20-21</td>
<td>Two numeric digits are returned in the range 00 through 59 indicating the number of additional minutes that the reported time is ahead of or behind Greenwich mean time, depending on whether character position 17 is ‘+‘ or ‘-‘, respectively. If character position 17 is ‘0‘, the value 00 is returned.</td>
</tr>
</tbody>
</table>

For more information, see *Examples: numeric intrinsic functions* in the *Enterprise COBOL Programming Guide*.
DATE-OF-INTEGER

The DATE-OF-INTEGER function converts a date in the Gregorian calendar from integer date form to standard date form (YYYYMMDD).

The function type is integer.

The function result is an eight-digit integer.

Format

FUNCTION DATE-OF-INTEGER
   (argument-1)

argument-1

A positive integer that represents a number of days succeeding December 31, 1600, in the Gregorian calendar. The valid range is 1 to 3,067,671, which corresponds to dates ranging from January 1, 1601 thru December 31, 9999.

The INTDATE compiler option affects the starting date for the integer date functions. For details, see INTDATE in the Enterprise COBOL Programming Guide.

The returned value represents the International Standards Organization (ISO) standard date equivalent to the integer specified as argument-1.

The returned value is an integer of the form YYYYMMDD where YYYY represents a year in the Gregorian calendar; MM represents the month of that year; and DD represents the day of that month.

DATE-TO-YYYYMMDD

The DATE-TO-YYYYMMDD function converts argument-1 from a date with a two-digit year (YYnnnn) to a date with a four-digit year (YYYYnnnn). argument-2, when added to the year at the time of execution, defines the ending year of a 100-year interval, or sliding century window, into which the year of argument-1 falls.

The function type is integer.

Format

FUNCTION DATE-TO-YYYYMMDD
   (argument-1, argument-2)

argument-1

Must be zero or a positive integer less than 991232.

Note: The COBOL run time does not verify that the value is a valid date.

argument-2

Must be an integer. If argument-2 is omitted, the function is evaluated assuming the value 50 was specified.

The sum of the year at the time of execution and the value of argument-2 must be less than 10,000 and greater than 1,699.

See the following examples with returned values from the DATE-TO-YYYYMMDD function:

<table>
<thead>
<tr>
<th>Current year</th>
<th>argument-1 value</th>
<th>argument-2 value</th>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2002</td>
<td>851003</td>
<td>120</td>
<td>20851003</td>
</tr>
</tbody>
</table>
DAY-OF-INTEGER

The DAY-OF-INTEGER function converts a date in the Gregorian calendar from integer date form to Julian date form (YYYYDDD).

The function type is integer.

The function result is a seven-digit integer.

<table>
<thead>
<tr>
<th>Current year</th>
<th>argument-1 value</th>
<th>argument-2 value</th>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2002</td>
<td>851003</td>
<td>-20</td>
<td>18851003</td>
</tr>
<tr>
<td>2002</td>
<td>851003</td>
<td>10</td>
<td>19851003</td>
</tr>
<tr>
<td>1994</td>
<td>981002</td>
<td>-10</td>
<td>18981002</td>
</tr>
</tbody>
</table>

**Format**

```cobol
FUNCTION DAY-OF-INTEGER (<argument-1>)
```

**argument-1**

A positive integer that represents a number of days succeeding December 31, 1600, in the Gregorian calendar. The valid range is 1 to 3,067,671, which corresponds to dates ranging from January 1, 1601 thru December 31, 9999.

The INTDATE compiler option affects the starting date for the integer date functions. For details, see INTDATE in the Enterprise COBOL Programming Guide.

The returned value represents the Julian equivalent of the integer specified as argument-1. The returned value is an integer of the form YYYYDDD where YYYY represents a year in the Gregorian calendar and DDD represents the day of that year.

DAY-TO-YYYYDDD

The DAY-TO-YYYYDDD function converts argument-1 from a date with a two-digit year (YYnnn) to a date with a four-digit year (YYYYnnn). argument-2, when added to the year at the time of execution, defines the ending year of a 100-year interval, or sliding century window, into which the year of argument-1 falls.

The function type is integer.

**Format**

```cobol
FUNCTION DAY-TO-YYYYDDD (<argument-1>, <argument-2>)
```

**argument-1**

Must be zero or a positive integer less than 99367.

The COBOL run time does not verify that the value is a valid date.

**argument-2**

Must be an integer. If argument-2 is omitted, the function is evaluated assuming the value 50 was specified.

The sum of the year at the time of execution and the value of argument-2 must be less than 10,000 and greater than 1,699.

Some examples of returned values from the DAY-TO-YYYYDDD function follow:
**DISPLAY-OF**

The DISPLAY-OF function returns an alphanumeric character string consisting of the content of `argument-1` converted to a specific code page representation.

The type of the function is alphanumeric.

**Format**

```plaintext
FUNCTION DISPLAY-OF (argument-1, argument-2)
```

- **argument-1**
  - Must be of class national (categories national, national-edited, and numeric-edited described with usage NATIONAL) or class UTF-8. `argument-1` identifies the source string for the conversion.

- **argument-2**
  - Must be an integer. `argument-2` identifies the output code page for the conversion.

`argument-2` must be a valid CCSID number and must identify an EBCDIC, ASCII, UTF-8, or EUC code page. An EBCDIC or ASCII code page can contain both single-byte and double-byte characters.

If `argument-2` is omitted, the output code page is the one that was in effect for the CODEPAGE compiler option when the source code was compiled.

The returned value is an alphanumeric character string consisting of the characters of `argument-1` converted to the output code page representation. When a source character cannot be converted to a character in the output code page, the source character is replaced with a substitution character. The following table shows substitution characters for some widely-used code pages:

<table>
<thead>
<tr>
<th>Output code page</th>
<th>Substitution character</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBCS ASCII PC Windows SBCS</td>
<td>X'7F'</td>
</tr>
<tr>
<td>EBCDIC SBCS</td>
<td>X'3F'</td>
</tr>
<tr>
<td>ASCII DBCS</td>
<td>X'FCFC'</td>
</tr>
<tr>
<td>EBCDIC DBCS (except for Thai)</td>
<td>X'FEFE'</td>
</tr>
<tr>
<td>EBCDIC DBCS (Thai)</td>
<td>X'41B8'</td>
</tr>
<tr>
<td>PC DBCS (Japanese or Chinese)</td>
<td>X'FCFC'</td>
</tr>
<tr>
<td>PC DBCS (Korean)</td>
<td>X'BFFC'</td>
</tr>
<tr>
<td>EUC (Korean)</td>
<td>X'AFFE'</td>
</tr>
<tr>
<td>EUC (Japanese)</td>
<td>X'747E'</td>
</tr>
</tbody>
</table>
No exception condition is raised.

The length of the returned value depends on the content of argument-1 and the characteristics of the output code page.

**Usage notes**

- The CCSID for UTF-8 is 1208.
- If the output code page includes DBCS characters, the returned value can be a mixed SBCS and DBCS string.
- The DISPLAY-OF function, with argument-2 specified, can be used to generate character data represented in a code page that differs from that specified in the CODEPAGE compiler option. Subsequent COBOL operations on that data can involve implicit conversions that assume the data is represented in the EBCDIC code page specified in the CODEPAGE compiler option. See *Converting to or from national (Unicode) representation* in the Enterprise COBOL Programming Guide for examples and programming techniques for processing data represented using more than one code page within a single program.

**Exception**: If the conversion fails, a severe runtime error occurs. Verify that the z/OS Unicode conversion services are installed and are configured to include the table for converting from CCSID 1200 to the output code page. See the Customization Guide for installation requirements to support the conversion.

---

**E**

The E function returns an approximation of e, the base of natural logarithms.

The function type is numeric.

**Format**

```
FUNCTION E
```

When ARITH(COMPAT) is in effect, FUNCTION E returns the long precision (64-bit) floating-point approximation of 2.718281828459045235360287471352662.

When ARITH(EXTEND) is in effect, FUNCTION E returns the extended precision (128-bit) floating-point approximation of 2.718281828459045235360287471352662.

---

**EXP**

The EXP function returns an approximation of the value of e raised to the power of the argument.

The function type is numeric.
FUNCTION EXP

\[ \text{ FORMAT } \]
\[
\text{ FUNCTION EXP } \quad (\text{argument-1})
\]

**argument-1**

Must be of class numeric.

The EXP function produces return values that are identical to the Language Environment callable service CEESDEXP when ARITH(COMPAT) is in effect and CEESQEXP when ARITH(EXTEND) is in effect. The COBOL expression, \( \text{FUNCTION E}^{\text{**}} (\text{argument-1}) \), is an approximation of this value.

---

**EXP10**

The EXP10 function returns an approximation of the value of 10 raised to the power of the argument.

The function type is numeric.

\[ \text{ FORMAT } \]
\[
\text{ FUNCTION EXP10 } \quad (\text{argument-1})
\]

**argument-1**

Must be of class numeric.

The EXP10 function produces return values that are identical to the Language Environment callable service CEESDXPD with \( \text{parm1} \) set to 10.0 when ARITH(COMPAT) is in effect and CEESQXPQ with \( \text{parm1} \) set to 10.0 when ARITH(EXTEND) is in effect. The COBOL expression, \( 10^{\text{**}} (\text{argument-1}) \), is an approximation of this value.

---

**FACTORIAL**

The FACTORIAL function returns an integer that is the factorial of the argument specified.

The function type is integer.

\[ \text{ FORMAT } \]
\[
\text{ FUNCTION FACTORIAL } \quad (\text{argument-1})
\]

**argument-1**

If the ARITH(COMPAT) compiler option is in effect, \( \text{argument-1} \) must be an integer greater than or equal to zero and less than or equal to 28. If the ARITH(EXTEND) compiler option is in effect, \( \text{argument-1} \) must be an integer greater than or equal to zero and less than or equal to 29.

If the value of \( \text{argument-1} \) is zero, the value 1 is returned; otherwise, the factorial of \( \text{argument-1} \) is returned.

---

**HEX-OF**

The HEX-OF function returns an alphanumeric character string consisting of the bytes of the input argument converted to a hexadecimal representation.

The type of the function is alphanumeric.
**FUNCTION HEX-OF**

- **Format**

  ```cobol
  FUNCTION HEX-OF (argument-1)
  ```

- **argument-1**
  Can be a data item, literal, or intrinsic function result of any data class. `argument-1` identifies the source character string for the conversion.

  The returned value is an alphanumeric character string consisting of the bytes of `argument-1` converted to a hexadecimal representation. The length of the output character string in bytes is two times the length of `argument-1` in bytes.

  **Note:** If `argument-1` is invalid, the behavior is undefined.

- **Examples**

  - FUNCTION HEX-OF('Hello, world!') returns 'C8859393966B40A6969993845A'
  - FUNCTION HEX-OF(BIN) returns '0000000C'
  - FUNCTION HEX-OF(PAC) returns '12345F'
  - FUNCTION HEX-OF(ZON) returns 'F1F2F3F4F5'
  - FUNCTION HEX-OF(FUNCTION NATIONAL-OF(' ')) returns '0020'

**HEX-TO-CHAR**

The HEX-TO-CHAR function returns a character string consisting of bytes that correspond to the hexadecimal digit characters in the input argument.

The function type is alphanumeric.

- **Format**

  ```cobol
  FUNCTION HEX-TO-CHAR (argument-1)
  ```

- **argument-1**
  Must be an alphanumeric literal, alphanumeric data item, or alphanumeric group item. `argument-1` must consist only of the characters "0" through "9", "A" through "F", and "a" through "f". The length of `argument-1` must be a multiple of 2 bytes.

  The returned value is a character string consisting of bytes that correspond to the hexadecimal digit characters in `argument-1`. The length of the result string is equal to the length of the input string divided by 2.

- **Example**

  ```cobol
  MOVE 'FFAABB' TO MY-HEX-DATA
  ```
FUNCTION HEX-TO-CHAR(MY-HEX-DATA) returns a character string with value x'FFAABB'.

**INTEGER**

The INTEGER function returns the greatest integer value that is less than or equal to the argument specified.

The function type is integer.

**Format**

```
FUNCTION INTEGER (argument-1)
```

*argument-1*

Must be class numeric.

The returned value is the greatest integer less than or equal to the value of *argument-1*. For example, FUNCTION INTEGER (2.5) returns a value of 2 and FUNCTION INTEGER (-2.5) returns a value of -3.

**INTEGER-OF-DATE**

The INTEGER-OF-DATE function converts a date in the Gregorian calendar from standard date form (YYYYMMDD) to integer date form.

The function type is integer.

The function result is a seven-digit integer with a range from 1 to 3,067,671.

**Format**

```
FUNCTION INTEGER-OF-DATE (argument-1)
```

*argument-1*

Must be an integer of the form YYYYMMDD, whose value is obtained from the calculation (YYYY * 10,000) + (MM * 100) + DD, where:

- YYYY represents the year in the Gregorian calendar. It must be an integer greater than 1600, but not greater than 9999.
- MM represents a month and must be a positive integer less than 13.
- DD represents a day and must be a positive integer less than 32, provided that it is valid for the specified month and year combination.

The returned value is an integer that is the number of days that the date represented by *argument-1* succeeds December 31, 1600 in the Gregorian calendar.

The INTDATE compiler option affects the starting date for the integer date functions. For details, see **INTDATE** in the *Enterprise COBOL Programming Guide*.

**INTEGER-OF-DAY**

The INTEGER-OF-DAY function converts a date in the Gregorian calendar from Julian date form (YYYYDDD) to integer date form.

The function type is integer.

The function result is a seven-digit integer.
FUNCTION INTEGER-OF-DAY (argument-1)

*argument-1*

Must be an integer of the form YYYYDDD whose value is obtained from the calculation (YYYY * 1000) + DDD, where:

- YYYY represents the year in the Gregorian calendar. It must be an integer greater than 1600, but not greater than 9999.
- DDD represents the day of the year. It must be a positive integer less than 367, provided that it is valid for the year specified.

The returned value is an integer that is the number of days that the date represented by *argument-1* succeeds December 31, 1600 in the Gregorian calendar.

The INTDATE compiler option affects the starting date for the integer date functions. For details, see INTDATE in the *Enterprise COBOL Programming Guide*.

FUNCTION INTEGER-PART (argument-1)

*argument-1*

Must be class numeric.

If the value of *argument-1* is zero, the returned value is zero. If the value of *argument-1* is positive, the returned value is the greatest integer less than or equal to the value of *argument-1*. If the value of *argument-1* is negative, the returned value is the least integer greater than or equal to the value of *argument-1*.

FUNCTION LENGTH (argument-1)

*argument-1*

Can be:

- An alphanumeric literal, a national literal, or a UTF-8 literal
- A group item (including unbounded groups) or an elementary data item of any class except DBCS

The returned value is an integer equal to the length of the argument in national character positions for arguments of usage NATIONAL and in alphanumeric character positions or bytes for all other arguments. An alphanumeric character position and a byte are equivalent.

The type of the function is integer.
• A data item described with usage POINTER, PROCEDURE-POINTER, FUNCTION-POINTER, or OBJECT REFERENCE
• The ADDRESS OF special register
• The LENGTH OF special register
• The XML-NTEXT special register
• The XML-TEXT special register

The returned value is a 9-digit integer if LP(32) is in effect or an 18-digit integer if LP(64) is in effect and is determined as follows:

• If argument-1 is an alphanumeric literal or an elementary data item of class alphabetic or alphanumeric, the value returned is equal to the number of alphanumeric character positions in the argument.

If argument-1 is a null-terminated alphanumeric literal, the returned value is equal to the number of alphanumeric character positions in the literal excluding the null character at the end of the literal.

The length of an alphanumeric data item or literal containing a mix of single-byte and double-byte characters is counted as though each byte were a single-byte character.

• If argument-1 is an alphanumeric group item, the value returned is equal to the length of argument-1 in alphanumeric character positions regardless of the content of the group. If any data item subordinate to argument-1 is described with the DEPENDING phrase of the OCCURS clause, the length of argument-1 is determined using the contents of the data item specified in the DEPENDING phrase. This evaluation is accomplished according to the rules of the OCCURS clause for a sending data item. For more information, see the discussions of the “OCCURS clause” on page 182 and the “USAGE clause” on page 218.

The returned value includes implicit FILLER positions, if any.

• If argument-1 is a national literal or an elementary data item described with usage NATIONAL, the value returned is equal to the length of argument-1 in national character positions.

For example, if argument-1 is defined as PIC 9(3) with usage NATIONAL, the returned value is 3, although the storage size of the argument is 6 bytes.

• If argument-1 is a national group item, the value returned is equal to the length of argument-1 in national character positions. If any data item subordinate to argument-1 is described with the DEPENDING phrase of the OCCURS clause, the length of argument-1 is determined using the contents of the data item specified in the DEPENDING phrase. This evaluation is accomplished according to the rules of the OCCURS clause for a sending data item. For more information, see the discussions of the “OCCURS clause” on page 182 and the “USAGE clause” on page 218.

The returned value includes implicit FILLER positions, if any.

• If argument-1 is a UTF-8 literal or an elementary data item described with usage UTF-8, the value returned is equal to the length of argument-1 in UTF-8 character positions.

For example, if argument-1 is defined as PIC U(n), the returned value is always n, even though the storage size of the argument is 4*n bytes and the actual number of bytes used by the item varies depending on the actual data in the item. This character length is known at compile time as this is a fixed character-length UTF-8 item.

• If argument-1 is defined with the BYTE-LENGTH phrase of the PICTURE clause or the DYNAMIC LENGTH clause, then the character length is computed at runtime by examining all bytes occupied by the data in argument-1 and counting the number of characters represented.

• If argument-1 is a UTF-8 group item, the value returned is equal to the length of argument-1 in UTF-8 character positions. In this case, like in the BYTE-LENGTH and DYNAMIC LENGTH case of UTF-8 data items, the character length is computed at run time by examining the data in the group item. If any data item subordinate to argument-1 is described with the DEPENDING phrase of the OCCURS clause, the length of argument-1 is determined using the contents of the data item specified in the DEPENDING phrase. This evaluation is accomplished according to the rules of the OCCURS clause for a sending data item. For more information, see the discussions of the “OCCURS clause” on page 182 and the “USAGE clause” on page 218.
The returned value includes implicit FILLER positions, if any.

- Otherwise, the returned value is the number of bytes of storage occupied by argument-1.

**LOG**

The LOG function returns a numeric value that approximates the logarithm to the base e (natural log) of the argument specified.

The function type is numeric.

**Format**

```
FUNCTION LOG — (argument-1 )
```

**argument-1**

Must be class numeric. The value of argument-1 must be greater than zero.

The returned value is the approximation of the logarithm to the base e of argument-1.

**LOG10**

The LOG10 function returns a numeric value that approximates the logarithm to the base 10 of the argument specified.

The function type is numeric.

**Format**

```
FUNCTION LOG10 — (argument-1 )
```

**argument-1**

Must be class numeric. The value of argument-1 must be greater than zero.

The returned value is the approximation of the logarithm to the base 10 of argument-1.

**LOWER-CASE**

The LOWER-CASE function returns a character string that contains the characters in the argument with each uppercase letter replaced by the corresponding lowercase letter.

The function type depends on the type of the argument, as follows:

<table>
<thead>
<tr>
<th>Argument type</th>
<th>Function type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alphabetic</td>
<td>Alphanumeric</td>
</tr>
<tr>
<td>Alphanumeric</td>
<td>Alphanumeric</td>
</tr>
<tr>
<td>National</td>
<td>National</td>
</tr>
<tr>
<td>UTF-8</td>
<td>UTF-8</td>
</tr>
</tbody>
</table>

**Format**

```
FUNCTION LOWER-CASE — (argument-1 )
```
**argument-1**

Must be class alphabetic, alphanumeric, national, or UTF-8 and must be at least one character position in length.

**Note:** If *argument-1* is of the alphanumeric class, it must not contain UTF-8 encoded data.

The same character string as *argument-1* is returned, except that each uppercase letter is replaced by the corresponding lowercase letter.

If *argument-1* is of class alphabetic or alphanumeric, the uppercase letters 'A' through 'Z' are replaced by the corresponding lowercase letters 'a' through 'z', where the range of 'A' through 'Z' and the range of 'a' through 'z' are as shown in “EBCDIC collating sequence” on page 579, regardless of the code page in effect.

If *argument-1* is of class national or UTF-8, each uppercase letter is replaced by its corresponding lowercase letter based on the specification given in the Unicode database UnicodeData.txt, available from the Unicode Consortium at http://www.unicode.org/.

If *argument-1* is not of class UTF-8, the character string returned has the same length as *argument-1*. For UTF-8 arguments, the returned string may have a different byte length than the byte length of *argument-1*.

The character string returned has the same length as *argument-1*.

**MAX**

The MAX function returns the content of the argument that contains the maximum value.

The function type depends on the argument type, as follows:

<table>
<thead>
<tr>
<th>Argument type</th>
<th>Function type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alphabetic</td>
<td>Alphanumeric</td>
</tr>
<tr>
<td>Alphanumeric</td>
<td>Alphanumeric</td>
</tr>
<tr>
<td>National</td>
<td>National</td>
</tr>
<tr>
<td>All arguments integer (includes integer arguments of usage NATIONAL)</td>
<td>Integer</td>
</tr>
<tr>
<td>Numeric (some arguments can be integer) (includes numeric arguments of usage NATIONAL)</td>
<td>Numeric</td>
</tr>
</tbody>
</table>

**Format**

```
FUNCTION MAX (argument-1)
```

*argument-1*

Must be class alphabetic, alphanumeric, national, or numeric.

All arguments must be of the same class, except that a combination of alphabetic and alphanumeric arguments is allowed.

The returned value is the content of *argument-1* having the greatest value. The comparisons used to determine the greatest value are made according to the rules for simple conditions. For more information, see “Conditional expressions” on page 246.
If more than one argument-1 has the same greatest value, the leftmost argument-1 having that value is returned.

If the type of the function is alphanumeric or national, the size of the returned value is the size of the selected argument-1.

### MEAN

The MEAN function returns a numeric value that approximates the arithmetic average of its arguments. The function type is numeric.

**Format**

```
FUNCTION MEAN (argument-1)
```

*argument-1*

Must be class numeric.

The returned value is the arithmetic mean of the argument-1 series. The returned value is defined as the sum of the argument-1 series divided by the number of occurrences referenced by argument-1.

### MEDIAN

The MEDIAN function returns the content of the argument whose value is the middle value in the list formed by arranging the arguments in sorted order.

The function type is numeric.

**Format**

```
FUNCTION MEDIAN (argument-1)
```

*argument-1*

Must be class numeric.

The returned value is the content of argument-1 having the middle value in the list formed by arranging all argument-1 values in sorted order.

If the number of occurrences referenced by argument-1 is odd, the returned value is such that at least half of the occurrences referenced by argument-1 are greater than or equal to the returned value and at least half are less than or equal. If the number of occurrences referenced by argument-1 is even, the returned value is the arithmetic mean of the values referenced by the two middle occurrences.

The comparisons used to arrange the argument values in sorted order are made according to the rules for simple conditions. For more information, see “Conditional expressions” on page 246.

### MIDRANGE

The MIDRANGE function returns a numeric value that approximates the arithmetic average of the values of the minimum argument and the maximum argument.

The function type is numeric.
**FUNCTION MIDRANGE**

\[
\text{FUNCTION MIDRANGE} \quad (\quad \text{argument-1} \quad )
\]

**argument-1**

Must be class numeric.

The returned value is the arithmetic mean of the value of the greatest \textit{argument-1} and the value of the least \textit{argument-1}. The comparisons used to determine the greatest and least values are made according to the rules for simple conditions. For more information, see “Conditional expressions” on page 246.

---

**MIN**

The MIN function returns the content of the argument that contains the minimum value.

The function type depends on the argument type, as follows:

<table>
<thead>
<tr>
<th>Argument type</th>
<th>Function type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alphabetic</td>
<td>Alphanumeric</td>
</tr>
<tr>
<td>Alphanumeric</td>
<td>Alphanumeric</td>
</tr>
<tr>
<td>National</td>
<td>National</td>
</tr>
<tr>
<td>All arguments integer (includes integer arguments of usage NATIONAL)</td>
<td>Integer</td>
</tr>
<tr>
<td>Numeric (some arguments can be integer) (includes numeric arguments of usage NATIONAL)</td>
<td>Numeric</td>
</tr>
</tbody>
</table>

**Format**

\[
\text{FUNCTION MIN} \quad (\quad \text{argument-1} \quad )
\]

**argument-1**

Must be class alphabetic, alphanumerical, national, or numeric.

All arguments must be of the same class, except that a combination of alphabetic and alphanumerical arguments is allowed.

The returned value is the content of \textit{argument-1} having the least value. The comparisons used to determine the least value are made according to the rules for simple conditions. For more information, see “Conditional expressions” on page 246.

If more than one \textit{argument-1} has the same least value, the leftmost \textit{argument-1} having that value is returned.

If the type of the function is alphanumerical or national, the size of the returned value is the size of the selected \textit{argument-1}.
The MOD function returns an integer value that is argument-1 modulo argument-2.

The function type is integer.

The function result is an integer with as many digits as the shorter of argument-1 and argument-2.

**Format**

```cobol
FUNCTION MOD (argument-1 argument-2)
```

**argument-1**

Must be an integer.

**argument-2**

Must be an integer. Must not be zero.

The returned value is argument-1 modulo argument-2. The returned value is defined as:

\[
\text{argument-1} - (\text{argument-2} \times \text{FUNCTION INTEGER (argument-1 / argument-2)})
\]

The following table lists expected results for some values of argument-1 and argument-2.

<table>
<thead>
<tr>
<th>argument-1</th>
<th>argument-2</th>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>-11</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>-5</td>
<td>-4</td>
</tr>
<tr>
<td>-11</td>
<td>-5</td>
<td>-1</td>
</tr>
</tbody>
</table>

The NATIONAL-OF function returns a national character string consisting of the national character representation of the characters in argument-1.

The type of the function is national.

**Format**

```cobol
FUNCTION NATIONAL-OF (argument-1 argument-2)
```

**argument-1**

Must be of class alphabetic, alphanumeric, UTF-8, or DBCS. argument-1 specifies the source string for the conversion.

**argument-2**

Must be an integer. argument-2 identifies the source code page for the conversion.

- argument-2 must be a valid CCSID number and must identify an EBCDIC, ASCII, UTF-8, or EUC code page. An EBCDIC or ASCII code page can contain both single-byte and double-byte characters.
- If argument-2 is omitted and argument-1 is of class UTF-8, then the source code page is 1208. Otherwise, the source code page is the one that was in effect for the CODEPAGE compiler option when the source code was compiled.
The returned value is a national character string consisting of the characters of argument-1 converted to national character representation. When a source character cannot be converted to a national character, the source character is converted to a substitution character. The substitution character is:

- X'001A' if converting a single-byte character
- X'FFFD' if converting a multi-byte character

No exception condition is raised.

The length of the returned value depends on the content of argument-1 and the characteristics of the source code page.

Usage note: The CCSID for UTF-8 is 1208.

Exception: If the conversion fails, a severe runtime error occurs. Verify that the z/OS Unicode conversion services are installed and are configured to include the table for converting from the source code page to CCSID 1200. See the Customization Guide for installation requirements to support the conversion.

### NUMVAL

The NUMVAL function returns the numeric value represented by the alphanumeric character string or national character string specified as the argument. The function removes any leading or trailing spaces in the string to produce a numeric value.

The function type is numeric.

#### Format

```
FUNCTION NUMVAL (argument-1)
```

#### argument-1

Must be an alphanumeric literal, a national literal, or a data item of class national or class alphanumeric that contains a character string in either of the following formats:

#### Format 1: argument-1

```
space + space digit . digit . digit
```

```
space
```
Format 2: argument-1, monetary format

\[
\text{digit} \quad . \quad \text{digit} \quad \text{space} \quad + \quad -. \quad \text{space} \quad \text{CR} \quad \text{DB}
\]

space
A string of one or more spaces.
digit
A string of one or more digits. If the ARITH(COMPAT) compiler option is in effect, the total number of
digits must not exceed 18. If the ARITH(EXTEND) compiler option is in effect, the total number of
digits must not exceed 31.

If the DECIMAL-POINT IS COMMA clause is specified in the SPECIAL-NAMES paragraph, a comma must
be used in argument-1 rather than a decimal point.

The returned value is a floating-point approximation of the numeric value represented by argument-1. The
precision of the returned value depends on the setting of the ARITH compiler option. For details, see
Converting to numbers (NUMVAL, NUMVAL-C, NUMVAL-F) in the Enterprise COBOL Programming Guide.

NUMVAL-C

The NUMVAL-C function returns the numeric value represented by the alphanumeric character string or
national character string specified as argument-1. The function removes the currency string, if any, and
any grouping separators (commas or periods) to produce a numeric value.

The function type is numeric.

\[
\text{Format}
\begin{align*}
\text{FUNCTION NUMVAL-C} & \quad ( \quad \text{argument-1} \quad \text{argument-2} \quad ) \quad
\end{align*}
\]

argument-1
Must be an alphanumeric literal, a national literal, or a data item of class alphanumeric or class
national that contains a character string in either of the following formats:
space
A string of one or more spaces.

cs
The string of one or more characters that form the currency sign. At most one copy of the characters specified by cs can occur in argument-1.

digit
A string of one or more digits. If the ARITH(COMPAT) compiler option is in effect, the total number of digits must not exceed 18. If the ARITH(EXTEND) compiler option is in effect, the total number of digits must not exceed 31.

If the DECIMAL-POINT IS COMMA clause is specified in the SPECIAL-NAMES paragraph, the functions of the comma and decimal point in argument-1 are reversed.

argument-2
Specifies the currency string value.

The following rules apply:
• argument-2 must be specified if the program contains more than one CURRENCY SIGN clause.
• argument-2, if specified, must be of the same class as argument-1.
• argument-2 must not contain any of the digits 0 through 9, any leading or trailing spaces, or any of the special characters '+', ',', ';', or '.'.
• argument-2 can be of any length valid for an elementary or group data item of the class of argument-2, including zero.
Matching of argument-2 is case sensitive. For example, if you specify argument-2 as 'CHF', it will not match 'ChF', 'chf' or 'chF'.

If argument-2 is not specified, the character used for cs is the currency symbol specified for the program.

The returned value is a floating-point approximation of the numeric value represented by argument-1. The precision of the returned value depends on the setting of the ARITH compiler option. For details, see Converting to numbers (NUMVAL, NUMVAL-C, NUMVAL-F) in the Enterprise COBOL Programming Guide.

**NUMVAL-F**

The NUMVAL-F function returns the numeric value represented by the alphanumeric character string or national character string specified as the argument. The function removes any leading or trailing spaces in the string to produce a numeric value.

The function type is numeric.

**Format**

```
FUNCTION NUMVAL-F (argument-1)
```

**argument-1**

Must be an alphanumeric literal, a national literal, or a data item of class national or class alphanumeric that contains a character string in the following format:

**Format: argument-1**

```
 space + space digit . digit space E space + space-string n space
```

- **space**
  A string of one or more spaces.

- **digit**
  A string of one or more digits.
  If the ARITH(COMPAT) compiler option is in effect, the total number of digits must not exceed 18.
  If the ARITH(EXTEND) compiler option is in effect, the total number of digits must not exceed 31.
  If the exponent clause is specified, the mantissa must not exceed 16 digits.

- **n**
  A string of one to four digits representing the exponent value.

- **E**
  If argument-1 is alphanumeric, E must be either an uppercase or lowercase E character.
  If argument-1 is national, E must be either an uppercase or lowercase E national character.
If the DECIMAL-POINT IS COMMA clause is specified in the SPECIAL-NAMES paragraph, a comma must be used in argument-1 rather than a decimal point.

The returned value is a floating-point approximation of the numeric value represented by argument-1. The precision of the returned value depends on the setting of the ARITH compiler option. For details, see Converting to numbers (NUMVAL, NUMVAL-C, NUMVAL-F) in the Enterprise COBOL Programming Guide.

**ORD**

The ORD function returns an integer value that is the ordinal position of its argument in the collating sequence for the program. The lowest ordinal position is 1.

The function type is integer.

The function result is a three-digit integer.

```
FUNCTION ORD (argument-1)
```

*argument-1*

Must be one character in length and must be class alphabetic or alphanumeric.

The returned value is the ordinal position of argument-1 in the collating sequence for the program; it ranges from 1 to 256 depending on the collating sequence.

**ORD-MAX**

The ORD-MAX function returns a value that is the ordinal position in the argument list of the argument that contains the maximum value.

The function type is integer.

```
FUNCTION ORD-MAX (argument-1)
```

*argument-1*

Must be class alphabetic, alphanumeric, national, or numeric.

All arguments must be of the same class, except that a combination of alphabetic and alphanumeric arguments is allowed.

The returned value is the ordinal number that corresponds to the position of the argument having the greatest value in the argument series.

The comparisons used to determine the greatest-valued argument-1 are made according to the rules for simple conditions. For more information, see “Conditional expressions” on page 246.

If more than one argument-1 has the same greatest value, the number returned corresponds to the position of the leftmost argument-1 having that value.
**ORD-MIN**

The ORD-MIN function returns a value that is the ordinal position in the argument list of the argument that contains the minimum value.

The function type is integer.

**Format**

```
FUNCTION ORD-MIN (argument-1)
```

**argument-1**

Must be class alphabetic, alphanumeric, national, or numeric.

All arguments must be of the same class, except that a combination of alphabetic and alphanumeric arguments is allowed.

The returned value is the ordinal number that corresponds to the position of the argument-1 having the least value in the argument-1 series.

The comparisons used to determine the least-valued argument-1 are made according to the rules for simple conditions. For more information, see “Conditional expressions” on page 246.

If more than one argument-1 has the same least value, the number returned corresponds to the position of the leftmost argument-1 having that value.

**PI**

The PI function returns a value that is an approximation of *pi*, the ratio of the circumference of a circle to its diameter.

The function type is numeric.

**Format**

```
FUNCTION PI
```

When ARITH(COMPAT) is in effect, FUNCTION PI returns the long precision (64-bit) floating-point approximation of 3.141592653589793238462643383279503.

When ARITH(EXTEND) is in effect, FUNCTION PI returns the extended precision (128-bit) floating-point approximation of 3.141592653589793238462643383279503.

**PRESENT-VALUE**

The PRESENT-VALUE function returns a value that approximates the present value of a series of future period-end amounts specified by argument-2 at a discount rate specified by argument-1.

The function type is numeric.
**FUNCTION PRESENT-VALUE**

The returned value is an approximation of the summation of a series of calculations with each term in the following form:

\[
\text{argument-2} / (1 + \text{argument-1})^n
\]

There is one term for each occurrence of **argument-2**. The exponent \( n \) is incremented from 1 by 1 for each term in the series.

**RANDOM**

The RANDOM function returns a numeric value that is a pseudorandom number from a rectangular distribution.

The function type is numeric.

**RANGE**

The RANGE function returns a value that is equal to the value of the maximum argument minus the value of the minimum argument.

The function type depends on the argument types, as follows:
**FUNCTION RANGE**

\[
\text{FUNCTION RANGE} \, \left( \text{argument-1} \right)
\]

**Argument-1**

Must be class numeric.

The returned value is equal to \(\text{argument-1}\) with the greatest value minus the \(\text{argument-1}\) with the least value. The comparisons used to determine the greatest and least values are made according to the rules for simple conditions. For more information, see “Conditional expressions” on page 246.

**REM**

The REM function returns a numeric value that is the remainder of \(\text{argument-1}\) divided by \(\text{argument-2}\).

The function type is numeric.

**REVERSE**

The REVERSE function returns a character value of the same length as the argument, whose characters are the same as those specified in the argument except that they are in reverse order. For arguments of type national, character positions are reversed; UTF-16 characters that are surrogate pairs are treated as one character and UTF-16 characters that are not surrogate pairs are treated as one character.

The function type depends on the type of the argument, as follows:

<table>
<thead>
<tr>
<th>Argument type</th>
<th>Function type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alphabetic</td>
<td>Alphanumeric</td>
</tr>
<tr>
<td>Alphanumeric</td>
<td>Alphanumeric</td>
</tr>
<tr>
<td>National</td>
<td>National</td>
</tr>
</tbody>
</table>
FUNCTION REVERSE

\[
\text{FUNCTION REVERSE} \quad (\text{--- argument-1 ---}) \quad \rightarrow
\]

**argument-1**
- Must be class alphabetic, alphanumeric, or national and must be at least one character in length.
  - If `argument-1` is of class alphabetic or alphanumeric, it must contain valid UTF-8 data.
  - If `argument-1` is of class national, it must contain valid UTF-16 data.

The returned value is a character string of the same length as `argument-1`, with the characters of `argument-1` in reversed order. For example, if `argument-1` contains ABC, the returned value is CBA.

**Example 1**
If `argument-1` is an alphanumeric data item that contains the UTF-8 value x'4BC3A4666572' ('Käfer'), the returned value is x'726566C3A44B' ('refsäK').

**Example 2**
If `argument-1` is a national data item that contains the UTF-16 value x'0054 00F6 D847DDF3 0062 0075 0072 D858DC6B 0073' ('Ööber'), the returned value is x'0073 D858DC6B 0072 0075 0062 D847DDF3 00F6 0054' ('ööber').

**Example 3**
If `argument-1` is a UTF-8 encoded item and the UTF-8 argument contains composed characters, the combining characters are counted individually. For example, when encoded in UTF-8, the Unicode character ä can be x'C3A4' or x'61CC88'. With either of the UTF-8 characters in `argument-1`, the returned values of the REVERSE function are different. See the following table for details.

<table>
<thead>
<tr>
<th>Character</th>
<th>Unicode encoding</th>
<th>UTF-8 encoding</th>
<th>Returned values of the REVERSE function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kä</td>
<td>U+004B + U+00E4</td>
<td>x'4BC3A4' (Kä)</td>
<td>x'C3A44B' (âK)</td>
</tr>
<tr>
<td></td>
<td>(precomposed form, latin capital letter K + latin small letter a with diaeresis)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>U+004B + U+0061 + U+0308</td>
<td>x'4B61CC88' (Kä)</td>
<td>x'CC88614B' (âK)</td>
</tr>
<tr>
<td></td>
<td>(canonical decomposition, latin capital letter K + latin small letter a + combining diaeresis)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SIGN**

The SIGN function returns +1, 0, or –1 depending on the sign of the argument. The function type is integer.

\[
\text{FUNCTION SIGN} \quad (\text{--- argument-1 ---}) \quad \rightarrow
\]

**argument-1**
- Must be of class numeric.
The returned value is as follows:

- If the value of `argument-1` is greater than zero, the returned value is 1.
- If the value of `argument-1` is zero, the returned value is 0.
- If the value of `argument-1` is less than zero, the returned value is -1.

**SIN**

The SIN function returns a numeric value that approximates the sine of the angle or arc specified by the argument in radians.

The function type is numeric.

```cobol
FUNCTION SIN
   (argument-1)
```

**argument-1**

Must be class numeric.

The returned value is the approximation of the sine of `argument-1` and is greater than or equal to -1 and less than or equal to +1.

**SQRT**

The SQRT function returns a numeric value that approximates the square root of the argument specified.

The function type is numeric.

```cobol
FUNCTION SQRT
   (argument-1)
```

**argument-1**

Must be class numeric. The value of `argument-1` must be zero or positive.

The returned value is the absolute value of the approximation of the square root of `argument-1`.

**STANDARD-DEVIATION**

The STANDARD-DEVIATION function returns a numeric value that approximates the standard deviation of its arguments.

The function type is numeric.

```cobol
FUNCTION STANDARD-DEVIATION
   (argument-1)
```

**argument-1**

Must be class numeric.

The returned value is the approximation of the standard deviation of the `argument-1` series. The returned value is calculated as follows:
1. The difference between each argument-1 and the arithmetic mean of the argument-1 series is calculated and squared.

2. The values obtained are then added together. This quantity is divided by the number of values in the argument-1 series.

3. The square root of the quotient obtained is then calculated. The returned value is the absolute value of this square root.

If the argument-1 series consists of only one value, or if the argument-1 series consists of all variable-occurrence data items and the total number of occurrences for all of them is one, the returned value is zero.

**SUM**

The SUM function returns a value that is the sum of the arguments.

The function type depends on the argument types, as follows:

<table>
<thead>
<tr>
<th>Argument type</th>
<th>Function type</th>
</tr>
</thead>
<tbody>
<tr>
<td>All arguments integer</td>
<td>Integer</td>
</tr>
<tr>
<td>Numeric (some arguments can be integer)</td>
<td>Numeric</td>
</tr>
</tbody>
</table>

**Format**

```plaintext
FUNCTION SUM (argument-1)
```

**argument-1**

Must be class numeric.

The returned value is the sum of the arguments. If the argument-1 series are all integers, the value returned is an integer. If the argument-1 series are not all integers, a numeric value is returned.

**TAN**

The TAN function returns a numeric value that approximates the tangent of the angle or arc that is specified by the argument in radians.

The function type is numeric.

**Format**

```plaintext
FUNCTION TAN (argument-1)
```

**argument-1**

Must be class numeric.

The returned value is the approximation of the tangent of argument-1.
The TEST-NUMVAL function verifies that the contents of **argument-1** conform to the specification for **argument-1** of the NUMVAL function.

The function type is integer.

**Format**

```
FUNCTION TEST-NUMVAL (argument-1)
```

**argument-1**

Must be an alphanumeric literal, a national literal, or a data item of class alphanumeric or class national.

The returned value is as follows:

- If the content of **argument-1** conforms to the argument rules for the NUMVAL function, the returned value is 0.
- If one or more characters are in error, the returned value is the position of the first character in error.

**Notes:**

- If one or more spaces are embedded within a string of numeric characters, the returned value is the position of the first non-space character following the spaces, because one or more spaces following one or more digits is valid. For example, if **argument-1** is ‘0 1’, the returned value will be 3.
- If the ARITH(COMPAT) compiler option is in effect, the returned value is the position of the 19th digit if no prior error is found, because the character in error for an argument that is greater than 18 digits is the 19th digit.
- If the ARITH(EXTEND) compiler option is in effect, the returned value is the position of the 32nd digit if no prior error is found, because the character in error for an argument that is greater than 31 digits is the 32nd digit.
- Otherwise, the returned value is \( \text{FUNCTION LENGTH} (\text{argument-1}) + 1 \).

These errors include, but are not limited to:

- **argument-1** is zero-length.
- **argument-1** contains only spaces.
- **argument-1** contains valid characters but is incomplete, such as the string ‘ +.’.

The TEST-NUMVAL-C function verifies that the contents of **argument-1** conform to the specification for **argument-1** of the NUMVAL-C function.

The function type is integer.

**Format**

```
FUNCTION TEST-NUMVAL-C (argument-1 argument-2)
```

**argument-1**

Must be an alphanumeric literal, a national literal, or a data item of class alphanumeric or class national that contains a character string in either of the following formats:
A string of one or more spaces.

cs

The string of one or more characters that form the currency sign. At most one copy of the characters specified by cs can occur in argument-1.

digit

A string of one or more digits.

If the ARITH(COMPAT) compiler option is in effect, the total number of digits must not exceed 18.

If the ARITH(EXTEND) compiler option is in effect, the total number of digits must not exceed 31.

If the DECIMAL-POINT IS COMMA clause is specified in the SPECIAL-NAMES paragraph, the functions of the comma and decimal point in argument-1 are reversed.

argument-2

Specifies the currency string value.

The following rules apply:

• argument-2 must be specified if the program contains more than one CURRENCY SIGN clause.

• argument-2, if specified, must be of the same class as argument-1.

• argument-2 must not contain any of the digits 0 through 9, any leading or trailing spaces, or any of the special characters '+', '-', ',', or '.

• argument-2 can be of any length valid for an elementary or group data item of the class of argument-2, including zero.
• Matching of argument-2 is case sensitive. For example, if you specify argument-2 as 'CHF', it will not match 'ChF', 'chf' or 'chF'.

If argument-2 is not specified, the character used for cs is the currency symbol specified for the program.

The returned value is as follows:
• If the content of argument-1 conforms to the argument rules for the NUMVAL-C function, the returned value is 0.
• If one or more characters are in error, the returned value is the position of the first character in error.

Notes:
– If one or more spaces are embedded within a string of numeric characters, the returned value is the position of the first non-space character following the spaces, because one or more spaces following one or more digits is valid. For example, if argument-1 is '0 1', the returned value will be 3.
– If the ARITH(COMPAT) compiler option is in effect, the returned value is the position of the 19th digit if no prior error is found, because the character in error for an argument that is greater than 18 digits is the 19th digit.
– If the ARITH(EXTEND) compiler option is in effect, the returned value is the position of the 32nd digit if no prior error is found, because the character in error for an argument that is greater than 31 digits is the 32nd digit.
• Otherwise, the returned value is (FUNCTION LENGTH (argument-1) + 1).

These errors include, but are not limited to:
– argument-1 is zero-length.
– argument-1 contains only spaces.
– argument-1 contains valid characters but is incomplete, such as the string '+.1'.

TEST-NUMVAL-F

The TEST-NUMVAL-F function verifies that the contents of argument-1 conform to the specification for argument-1 of the NUMVAL-F function.

The function type is integer.

Format

FUNCTION TEST-NUMVAL-F (argument-1)

argument-1

Must be an alphanumeric literal, a national literal, or a data item of class alphanumeric or class national.

The returned value is as follows:
• If the content of argument-1 conforms to the argument rules for the NUMVAL-F function, the returned value is 0.
• If one or more characters are in error, the returned value is the position of the first character in error.

Notes:
– If one or more spaces are embedded within a string of numeric characters, the returned value is the position of the first non-space character following the spaces, because one or more spaces following one or more digits is valid. For example, if argument-1 is '0 1', the returned value will be 3.
– If the ARITH(COMPAT) compiler option is in effect, the returned value is the position of the 19th digit if no prior error is found, because the character in error for an argument that is greater than 18 digits is the 19th digit.
– If the ARITH(EXTEND) compiler option is in effect, the returned value is the position of the 32nd digit if no prior error is found, because the character in error for an argument that is greater than 31 digits is the 32nd digit.
– If the exponent value in the argument contains more than four significant digits, the returned value is the position of the fifth digit of the exponent.
• Otherwise, the returned value is \((FUNCTION\ LENGTH\ (argument-1) + 1)\).

These errors include, but are not limited to:
– \(argument-1\) is zero-length.
– \(argument-1\) contains only spaces.
– \(argument-1\) contains valid characters but is incomplete, such as the string ‘+.’.

**TRIM**

The TRIM function returns a character string that contains the characters in the argument with leading spaces, trailing spaces, or both, removed.

The function type depends on the argument type as follows:

<table>
<thead>
<tr>
<th>Argument type</th>
<th>Function type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alphabetic</td>
<td>Alphanumeric</td>
</tr>
<tr>
<td>Alphabetic</td>
<td>Alphanumeric</td>
</tr>
<tr>
<td>National</td>
<td>National</td>
</tr>
<tr>
<td>UTF-8</td>
<td>UTF-8</td>
</tr>
</tbody>
</table>

**Format**

\[
\text{FUNCTION TRIM} \quad ( \quad \text{argument-1} \quad )
\]

**argument-1**

Must be a data item of class alphabetic, alphanumeric, national, or UTF-8.

The returned value is:
• If LEADING is specified, the returned value is a character string that consists of the characters in \(argument-1\) beginning from the leftmost character position that does not contain a space character through the rightmost character position.
• If TRAILING is specified, the returned value is a character string that consists of the characters in \(argument-1\) beginning from the leftmost character position through the rightmost character position that does not contain a space character.
• If neither LEADING nor TRAILING is specified, the returned value is a character string that consists of the characters in \(argument-1\) beginning from the leftmost character position that does not contain a space character through the rightmost character position that does not contain a space character.
• If \(argument-1\) contains all spaces or \(argument-1\) is of length zero, the returned value is of length zero.
Examples

- FUNCTION TRIM(" Hello, world! ", LEADING) returns "Hello, world!"
- FUNCTION TRIM(" Hello, world! ", TRAILING) returns " Hello, world!"
- FUNCTION TRIM(" Hello, world! ") returns "Hello, world!"
- FUNCTION TRIM(" ") returns ""
- FUNCTION TRIM("" ) returns ""

ULENGTH

The ULENGTH function returns an integer value that is equal to the number of UTF-8 or UTF-16 characters in a character data item argument that contains UTF-8 or UTF-16 data.

The function type is integer.

Format

```
FUNCTION ULENGTH (argument-1)
```

- **argument-1**
  - Must be of class alphabetic, alphanumeric, national or UTF-8. argument-1 must contain valid UTF-8 or UTF-16 encoded characters:
    - If argument-1 is of class alphabetic, alphanumeric or UTF-8, it must contain valid UTF-8 data.
    - If argument-1 is of class national, it must contain valid UTF-16 data.

  The returned value is the number of UTF-8 or UTF-16 characters in argument-1. If LP(32) is in effect, the returned value is a 9-digit integer; if LP(64) is in effect, the returned value is an 18-digit integer.

Example 1

If argument-1 is a UTF-8 encoded item and the UTF-8 argument contains composed characters, the combining characters are counted individually in determining the length. For example, when encoded in UTF-8, the Unicode character ä can be x'C3A4' or x'61CC88'. With either of the UTF-8 characters as argument-1, the returned values of the ULENGTH function are different. See the following table for details.

<table>
<thead>
<tr>
<th>Character</th>
<th>Unicode encoding</th>
<th>UTF-8 encoding</th>
<th>Returned value of the ULENGTH function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ä</td>
<td>U+00E4 (precomposed form, latin small letter a with diaeresis)</td>
<td>x'C3A4'</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>U+0061 + U+0308 (canonical decomposition, latin small letter a + combining diaeresis)</td>
<td>x'61CC88'</td>
<td>2</td>
</tr>
</tbody>
</table>

Example 2

If argument-1 is a national data item that contains UTF-16 data and argument-1 contains surrogate pairs, each pair of low and high surrogates will be counted as one UTF-16 character. For example, if B is a national item that contains the UTF-16 value x'005400F6006200750072D858DC6B0073' (‘Töber”), the returned value from ULENGTH(B) will be 7. Character ‘ö’ = X'D858DC6B’ is counted as one UTF-16 character.
The UPOS function returns an integer value that is equal to the index of the \( n \)th UTF-8 or UTF-16 character in a character data item argument that contains UTF-8 or UTF-16.

The function type is integer.

**Format**

```
FUNCTION UPOS (argument-1 argument-2)
```

**argument-1**

Must be of class alphabetic, alphanumeric, national or UTF-8. \( \text{argument-1} \) must contain valid UTF-8 or UTF-16 encoded characters:

- If \( \text{argument-1} \) is of class alphabetic, alphanumeric or UTF-8, it must contain valid UTF-8 data.
- If \( \text{argument-1} \) is of class national, it must contain valid UTF-16 data.

**argument-2**

Must be an integer.

Suppose \( \text{argument-1} \) is alphabetic or alphanumeric and \( \text{argument-2} = n \), the returned value is the byte position of the \( n \)th UTF-8 character in \( \text{argument-1} \). Suppose \( \text{argument-1} \) is a national data item and \( \text{argument-2} = n \), the returned value is the byte position of the \( n \)th UTF-16 character in \( \text{argument-1} \).

If \( \text{argument-2} \) is not positive or if \( \text{argument-2} \) is larger than ULENGTH(\( \text{argument-1} \)), zero is returned. Otherwise, if \( \text{argument-2} = n \), the returned value is the byte position in \( \text{argument-1} \) where the \( n \)th UTF-8 or UTF-16 character starts.

The returned value of UPOS is a 9-digit integer if LP(32) is in effect or an 18-digit integer if LP(64) is in effect.

**Example 1**

If A is an alphanumeric item that contains the UTF-8 value x'4BC3A4666572' ('Käfer'), the returned values are as follows:

- UPOS(A 1) returns 1
- UPOS(A 2) returns 2
- UPOS(A 3) returns 4
- UPOS(A 4) returns 5
- UPOS(A 5) returns 6

**Example 2**

If B is a national item that contains the UTF-16 value x'005400F6006200750072D858DC6B0073' ('Töber'), the returned values are as follows:

- UPOS (B 1) returns 1
- UPOS (B 2) returns 3
- UPOS (B 3) returns 5
- UPOS (B 4) returns 7
- UPOS (B 5) returns 9
- UPOS (B 6) returns 11
- UPOS (B 7) returns 15
Example 3
If argument-1 is a UTF-8 encoded item and the UTF-8 argument contains composed characters, the combining characters are counted individually. For example, when encoded in UTF-8, the Unicode character ä can be x'C3A4' or x'61CC88'. With either of the UTF-8 characters in argument-1, the returned values of the UPOS function are different. See the following table for details.

<table>
<thead>
<tr>
<th>argument-1</th>
<th>Unicode encoding</th>
<th>UTF-8 encoding</th>
<th>Returned values of the UPOS function</th>
</tr>
</thead>
<tbody>
<tr>
<td>C = äK</td>
<td>U+00E4 + U+004B (precomposed form, latin small letter a with diaeresis + latin capital letter K)</td>
<td>x'C3A44B' (äK)</td>
<td>UPOS(C 1) returns 1, UPOS(C 2) returns 3, UPOS(C 3) returns 0</td>
</tr>
<tr>
<td></td>
<td>U+0061 + U+0308 + U+004B (canonical decomposition, latin small letter a + combining diaeresis + latin capital letter K)</td>
<td>x'61CC884B' (äK)</td>
<td>UPOS(C 1) returns 1, UPOS(C 2) returns 2, UPOS(C 3) returns 4</td>
</tr>
</tbody>
</table>

UPPER-CASE
The UPPER-CASE function returns a character string that contains the characters in the argument with each lowercase letter replaced by the corresponding uppercase letter.

The function type depends on the type of the argument, as follows:

<table>
<thead>
<tr>
<th>Argument type</th>
<th>Function type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alphabetic</td>
<td>Alphanumeric</td>
</tr>
<tr>
<td>Alphanumeric</td>
<td>Alphanumeric</td>
</tr>
<tr>
<td>National</td>
<td>National</td>
</tr>
<tr>
<td>UTF-8</td>
<td>UTF-8</td>
</tr>
</tbody>
</table>

Format

FUNCTION UPPER-CASE (argument-1)

argument-1
Must be of class alphabetic, alphanumeric, national, or UTF-8 and must be at least one character position in length.

Note: If argument-1 is of the alphanumeric class, it must not contain UTF-8 encoded data.

The same character string as argument-1 is returned, except that each lowercase letter is replaced by the corresponding uppercase letter.

If argument-1 is alphabetic or alphanumeric, the lowercase letters 'a' through 'z' are replaced by the corresponding uppercase letters 'A' through 'Z', where the range of 'a' through 'z' and the range of 'A' through 'Z' are as shown in "EBCDIC collating sequence" on page 579, regardless of the code page in effect.

If argument-1 is national or UTF-8, each lowercase letter is replaced by its corresponding uppercase letter based on the specification given in the Unicode database UnicodeData.txt, available from the Unicode Consortium at www.unicode.org/.
If argument-1 is not of class UTF-8, the character string returned has the same length as argument-1. For UTF-8 arguments, the returned string may have a different byte length than the byte length of argument-1.

**USUBSTR**

The USUBSTR function returns a substring of the data in a character data item argument that contains UTF-8 or UTF-16 data.

The function type is alphanumericnational or UTF-8, depending on the class of argument-1.

**Format**

```
FUNCTION USUBSTR — ( argument-1 argument-2 argument-3 )
```

**argument-1**

Must be of class alphabetic, alphanumeric, national or UTF-8. argument-1 must contain valid UTF-8 or UTF-16 encoded characters:

- If argument-1 is of class alphabetic, alphanumeric or UTF-8, it must contain valid UTF-8 data.
- If argument-1 is of class national, it must contain valid UTF-16 data.

**argument-2**

Must be an integer that is greater than zero. It represents the starting position of a substring in argument-1.

**argument-3**

Must be an integer that is greater than or equal to zero. It represents the length of a substring in argument-1.

**Note:** The sum of argument-2 and argument-3 minus one must be less than or equal to ULENGTH(argument-1).

Suppose argument-1 is alphabetic or alphanumeric, argument-2 = n and argument-3 = m, the returned value is an alphanumeric item that contains m UTF-8 characters from argument-1, starting with the n-th UTF-8 character. Suppose argument-1 is a national data item, argument-2 = n and argument-3 = m, the returned value is a national item that contains m UTF-16 characters from argument-1, starting with the n-th UTF-16 character.

**Example 1**

If A is an alphanumeric item that contains the UTF-8 value x'4BC3A4666572' (‘Käfer’), the returned values are as follows:

- USUBSTR(A 1 2) returns x'4BC3A4' (‘Kä’)
- USUBSTR(A 2 1) returns x'C3A4' (‘ä’)
- USUBSTR(A 2 2) returns x'C3A466' (‘äf’)
- USUBSTR(A 3 2) returns x'6665' (‘fe’)

**Example 2**

If B is a national item that contains the UTF-16 value x'005400F6006200750072D858DC6B0073' (‘Töber’), the returned values are as follows:

- USUBSTR(B 1 2) returns x'005400F6' (‘Tö’)
- USUBSTR(B 2 1) returns x'00F6' (‘ö’)
- USUBSTR(B 2 2) returns x'00F60062' (‘öb’)
- USUBSTR(B 3 2) returns x'00620075' (‘be’)

Chapter 21. Intrinsic functions 509
- USUBSTR(B 5 2) returns x'0072D858DC6B' (r$%^a)
- USUBSTR(B 6 2) returns x'D858DC6B0073' (r$%^b)

**Example 3**

If argument-1 is a UTF-8 encoded item and the UTF-8 argument contains composed characters, the combining characters are counted individually. For example, when encoded in UTF-8, the Unicode character ä can be x'C3A4' or x'61CC88'. With either of the UTF-8 characters in argument-1, the returned values of the USUBSTR function are different. See the following table for details.

<table>
<thead>
<tr>
<th>argument-1</th>
<th>Unicode encoding</th>
<th>UTF-8 encoding</th>
<th>Returned values of the USUBSTR function</th>
</tr>
</thead>
<tbody>
<tr>
<td>C = äK</td>
<td>U+00E4 + U+004B</td>
<td>x'C3A44B' (äK)</td>
<td>USUBSTR (C 1 1) returns x'C3A4' (ä)</td>
</tr>
<tr>
<td></td>
<td>(precomposed form, latin small letter a with diaeresis + latin capital letter K)</td>
<td></td>
<td>USUBSTR (C 2 1) returns x'4B' (K)</td>
</tr>
<tr>
<td></td>
<td>U+0061 + U+0308 + U+004B</td>
<td>x'61CC884B' (äK)</td>
<td>USUBSTR (C 1 2) returns x'C3A44B' (äK)</td>
</tr>
<tr>
<td></td>
<td>(canonical decomposition, latin small letter a + combining diaeresis + latin capital letter K)</td>
<td></td>
<td>USUBSTR (C 1 3) returns x'61CC884B' (äK)</td>
</tr>
</tbody>
</table>

**USUPPLEMENTARY**

The USUPPLEMENTARY function returns an integer value that is equal to the index of the first Unicode supplementary character in a character data item argument that is encoded in UTF-8 or UTF-16.

A Unicode supplementary character is a character above U+FFFF, that is, a character outside of the Basic Multilingual Plane (BMP). These characters are encoded in UTF-16 with a surrogate pair (two 16-bit code units), or are encoded in UTF-8 with a 4-byte representation.

The function type is integer.

**Format**

```cobol
FUNCTION USUPPLEMENTARY (argument-1)
```

**argument-1**

Must be of class alphabetic, alphanumeric, or national. argument-1 must contain valid UTF-8 or UTF-16 data based on its class:

- If argument-1 is of class alphabetic or alphanumeric, it must contain valid UTF-8 data.
- If argument-1 is of class national, it must contain valid UTF-16 data.

The returned value is an integer, which differs based on the argument-1 value, and is 9-digit if LP(32) is in effect or 18-digit if LP(64) is in effect:

- If the contents of argument-1 are not valid Unicode (UTF-8 or UTF-16, depending on class), the returned result is unpredictable.
- If argument-1 contains no supplementary characters, the returned value is zero.
- If argument-1 is of class alphabetic or alphanumeric, the returned value is the byte position of the first UTF-8 supplementary character in argument-1.
- If argument-1 is of class national, the returned value is the index, in UTF-16 encoding units, of the first UTF-16 supplementary character in argument-1.
**Example 1**

For example, the musical G-clef symbol is represented in UTF-16 Unicode by the surrogate pair \texttt{nx'D834DD1E'}, or in UTF-8 Unicode by \texttt{x'F09D849E'}. Thus, for the following COBOL program fragment, the output of both DISPLAY statements is value 3.

```cobol
01 N pic N(4) value nx'00200020D834DD1E'.
01 X pic X(6) value x'2020F09D849E'.
01 I pic 9.
...
Compute I = function Usupplementary(N)
Display I
Compute I = function Usupplementary(X)
Display I
```

**Example 2**

If \texttt{argument-1} is a UTF-8 encoded item and the UTF-8 argument contains composed characters, the combining characters are counted individually. For example, when encoded in UTF-8, the Unicode character ä can be \texttt{x'C3A4'} or \texttt{x'61CC88'}. With either of the UTF-8 characters in \texttt{argument-1}, the returned values of the USUPPLEMENTARY function are different. See the following table for details.

<table>
<thead>
<tr>
<th>argument-1</th>
<th>Unicode encoding</th>
<th>UTF-8 encoding</th>
<th>Returned values of the USUPPLEMENTARY function</th>
</tr>
</thead>
<tbody>
<tr>
<td>B = äôK</td>
<td>U+00E4 + U21DE4 + U+004B (precomposed form, latin small letter a with diaeresis + latin capital letter K)</td>
<td>x'C3A4F0A1B7A44B' (äôK)</td>
<td>USUPPLEMENTARY (B) returns 3</td>
</tr>
<tr>
<td></td>
<td>U+0061 + U+0308 + U21DE4 + U+004B (canonical decomposition, latin small letter a + combining diaeresis + latin capital letter K)</td>
<td>x'61CC88F0A1B7A44B' (äôK)</td>
<td>USUPPLEMENTARY (B) returns 4</td>
</tr>
</tbody>
</table>

**UUID4**

The UUID4 function returns a 36-character alphanumeric string that is a version 4 universally unique identifier (UUID).

This intrinsic function is introduced in Enterprise COBOL Version 6 release 3 with PTF for APAR PH20997. The function type is alphanumeric.

**Format**

```
Function UUID4
```

**Example**

The following COBOL program fragment produces output like ‘UUID4: 4ed161b5-0d3c-4f06-8381-5f14678e13da’, with subsequent executions producing different UUID values.

```cobol
DISPLAY "UUID4: " FUNCTION UUID4
```
If a character data item contains valid UTF-8 or UTF-16 data, the UVALID function returns the value zero.
If a character data item contains invalid UTF-8 or UTF-16 data, the UVALID function returns the index of
the first invalid element.

The function type is integer.

**Format**

```
FUNCTION UVALID — ( — argument-1 — )
```

**argument-1**

Must be of class alphabetic, alphanumeric, national or UTF-8.
Must be of class alphabetic, alphanumeric, or national.

The returned value is an integer, which differs based on **argument-1**, and is 9-digit if LP(32) is in effect or
18-digit if LP(64) is in effect:

- If **argument-1** is of class alphabetic, alphanumeric or UTF-8, and it consists of valid UTF-8 encoded
  Unicode data, the returned value is zero.
- If **argument-1** is of class alphabetic, or alphanumeric, and it consists of valid UTF-8 encoded Unicode
data, the returned value is zero.
- If **argument-1** is of class alphabetic, or alphanumeric, and it contains invalid UTF-8 encoded Unicode
  data, the returned value is the position of the first byte where the invalid UTF-8 data starts.
- If **argument-1** is of class alphabetic, or alphanumeric, and it contains invalid UTF-8 encoded Unicode
data, the returned value is the position of the first byte where the invalid UTF-8 data starts.
- If **argument-1** is of class national, and it consists of valid UTF-16 encoded Unicode data, the returned
  value is zero.
- If **argument-1** is of class national, and it contains invalid UTF-16 encoded Unicode data, the returned
  value is the position of the first UTF-16 encoding unit where the invalid UTF-16 data starts. This position
  is one plus the number of well-formed UTF-16 encoding units that precede the invalid data.

**Note:** The UVALID function indicates whether the character string contains well-formed Unicode UTF-8 or
UTF-16 data. It does not indicate whether any or all of the Unicode code points represented by the
character string are assigned to characters.

For UTF-8 data, the validity of a byte varies according to its range as listed in the table:

<table>
<thead>
<tr>
<th>Value Range</th>
<th>Dependency</th>
<th>Validity</th>
</tr>
</thead>
<tbody>
<tr>
<td>x'00' - x'7F'</td>
<td>None</td>
<td>Valid</td>
</tr>
<tr>
<td>x'80' - x'C1'</td>
<td>None</td>
<td>Invalid</td>
</tr>
<tr>
<td>x'C2' - x'DF'</td>
<td>Followed by another byte that is in the range x'80' to x'BF'</td>
<td>Valid</td>
</tr>
</tbody>
</table>
### Table 61. Byte validity for UTF-8 data (continued)

<table>
<thead>
<tr>
<th>Value Range</th>
<th>Dependency</th>
<th>Validity</th>
</tr>
</thead>
</table>
| x'E0' - x'EF' | If the first byte is x'E0', followed by two more bytes that meet the following requirements:  
• The second byte is in the range x'A0' to x'BF'  
• The third byte is in the range x'80' to x'BF' | Valid |
| If the first byte is in the range x'E1' to x'EC', both the second and third bytes are in the range x'80' to x'BF' | Valid |
| If the first byte is x'ED', followed by two more bytes that meet the following requirements:  
• The second byte is in the range x'80' to x'9F'  
• The third byte is in the range x'80' to x'BF' | Valid |
| If the first byte is in the range x'EE' to x'EF', both the second and third bytes are in the range x'80' to x'BF' | Valid |
| x'F0' - x'F4' | If the first byte is x'F0', followed by three more bytes that meet the following requirements:  
• The second byte is in the range x'90' to x'BF'  
• The third byte is in the range x'80' to x'BF'  
• The fourth byte is in the range x'80' to x'BF' | Valid |
| If the first byte is in the range x'F1' to x'F3', all the second, third, and fourth bytes are in the range x'80' to x'BF' | Valid |
| If the first byte is x'F4', followed by three more bytes that meet the following requirements:  
• The second byte is in the range x'80' to x'8F'  
• The third byte is in the range x'80' to x'BF'  
• The fourth byte is in the range x'80' to x'BF' | Valid |
| x'F5' - x'FF' | None | Invalid |

For UTF-16 data, the validity of an encoding unit varies according to its range as listed in the table:

### Table 62. Encoding unit validity for UTF-16 data

<table>
<thead>
<tr>
<th>Value Range</th>
<th>Dependency</th>
<th>Validity</th>
<th>Number of bytes if converted to UTF-8</th>
</tr>
</thead>
<tbody>
<tr>
<td>nx'0000' - nx'007F'</td>
<td>None</td>
<td>Valid</td>
<td>1</td>
</tr>
<tr>
<td>nx'0080' - nx'07FF'</td>
<td>None</td>
<td>Valid</td>
<td>2</td>
</tr>
<tr>
<td>nx'0800' - nx'D7FF'</td>
<td>None</td>
<td>Valid</td>
<td>3</td>
</tr>
</tbody>
</table>
| nx'D800' - nx'DFFF' | Must be followed by a second encoding unit with a value in the range nx'DC00' to nx'DFFF' | Valid | 4  
(A Unicode surrogate pair) |
| | Other cases | Invalid | Not applicable |
| nx'E000' - nx'FFFF' | None | Valid | 3 |

**Example 1**

If A is an alphabetic or alphanumeric data item that contains value x'4BC3A4666572' ('Käfer') in UTF-8 encoding, the returned value from UVALID(A) is 0.
Example 2
If B is a national data item that contains value x'005400F6006200750072D858DC6B0073' ('Töber') in UTF-16 encoding, the returned value from UVALID(B) is 0.

Example 3
If C is a national data item that contains value x'0054D9C3006200750072D858DC6B0073' in UTF-16 encoding, the returned value from UVALID(C) is 2 because x'D9C3' does not have a low surrogate pair.

Example 4
If D is a national data item that contains value x'005400F60062DC010072D858DC6B0073' in UTF-16 encoding, the returned value from UVALID(D) is 4 because x'DC01' does not have a corresponding high surrogate pair.

UWIDTH

The UWIDTH function returns an integer value that is equal to the width in bytes of the nth UTF-8 or UTF-16 character in a character data item argument that is encoded in UTF-8 or UTF-16.

The function type is integer.

Format

FUNCTION UWIDTH (argument-1 argument-2)

argument-1
Must be of class alphabetic, alphanumeric, national or UTF-8. argument-1 must contain valid UTF-8 or UTF-16 encoded characters:

• If argument-1 is of class alphabetic, alphanumeric or UTF-8, it must contain valid UTF-8 data.
• If argument-1 is of class national, it must contain valid UTF-16 data.

argument-2
Must be an integer.

The returned value is an integer.

If argument-2 is not positive or if argument-2 is larger than ULENGTH(argument-1), zero is returned. Otherwise, if argument-2=n, the returned value is the width in bytes of the nth UTF-8 or UTF-16 character in argument-1.

Example 1
If A is an alphanumeric item that contains the UTF-8 value x'4BC3A4666572' ('Käfer'), the returned values are as follows:

• UWIDTH(A 1) returns 1
• UWIDTH(A 2) returns 2
• UWIDTH(A 3) returns 1
• UWIDTH(A 4) returns 1
• UWIDTH(A 5) returns 1

Example 2
If B is a national item that contains the UTF-16 value x'005400F6006200750072D858DC6B0073' ('Töber'), the returned values are as follows:
Example 3

If argument-1 is a UTF-8 encoded item and the UTF-8 argument contains composed characters, the combining characters are counted individually. For example, when encoded in UTF-8, the Unicode character ä can be x'C3A4' or x'61CC88'. With either of the UTF-8 characters in argument-1, the returned values of the UWIDTH function are different. See the following table for details.

<table>
<thead>
<tr>
<th>argument-1</th>
<th>Unicode encoding</th>
<th>UTF-8 encoding</th>
<th>Returned values of the UWIDTH function</th>
</tr>
</thead>
<tbody>
<tr>
<td>C = äK</td>
<td>U+00E4 + U+004B</td>
<td>x'C3A44B' (äK)</td>
<td>UWIDTH (C 1) returns 2</td>
</tr>
<tr>
<td></td>
<td>(precomposed form, latin small letter a with diaeresis + latin capital letter K)</td>
<td></td>
<td>UWIDTH (C 2) returns 1</td>
</tr>
<tr>
<td></td>
<td>U+0061 + U+0308 + U+004B</td>
<td>x'61CC884B' (äK)</td>
<td>UWIDTH (C 3) returns 0</td>
</tr>
<tr>
<td></td>
<td>(canonical decomposition, latin small letter a + combining diaeresis + latin capital letter K)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

VARIANCE

The VARIANCE function returns a numeric value that approximates the variance of its arguments. The function type is numeric.

**Format**

```
FUNCTION VARIANCE (argument-1)
```

**argument-1**

Must be class numeric.

The returned value is the approximation of the variance of the argument-1 series.

The returned value is defined as the square of the standard deviation of the argument-1 series. This value is calculated as follows:

1. The difference between each argument-1 value and the arithmetic mean of the argument-1 series is calculated and squared.
2. The values obtained are then added together. This quantity is divided by the number of values in the argument series.

If the argument-1 series consists of only one value, or if the argument-1 series consists of all variable-occurrence data items and the total number of occurrences for all of them is one, the returned value is zero.
The WHEN-COMPILED function returns the date and time that the program was compiled as provided by the system on which the program was compiled.

The function type is alphanumeric.

**Format**

```
FUNCTION WHEN-COMPILED
```

Reading from left to right, the 21 character positions of the returned value are as follows:

<table>
<thead>
<tr>
<th>Character positions</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-4</td>
<td>Four numeric digits of the year in the Gregorian calendar</td>
</tr>
<tr>
<td>5-6</td>
<td>Two numeric digits of the month of the year, in the range 01 through 12</td>
</tr>
<tr>
<td>7-8</td>
<td>Two numeric digits of the day of the month, in the range 01 through 31</td>
</tr>
<tr>
<td>9-10</td>
<td>Two numeric digits of the hours past midnight, in the range 00 through 23</td>
</tr>
<tr>
<td>11-12</td>
<td>Two numeric digits of the minutes past the hour, in the range 00 through 59</td>
</tr>
<tr>
<td>13-14</td>
<td>Two numeric digits of the seconds past the minute, in the range 00 through 59</td>
</tr>
<tr>
<td>15-16</td>
<td>Two numeric digits of the hundredths of a second past the second, in the range 00 through 99. The value 00 is returned if the system on which the function is evaluated does not have the facility to provide the fractional part of a second.</td>
</tr>
<tr>
<td>17</td>
<td>Either the character '-' or the character '+'. The character '-' is returned if the local time indicated in the previous character positions is behind Greenwich mean time. The character '+' is returned if the local time indicated is the same as or ahead of Greenwich mean time. The character '0' is returned if the system on which this function is evaluated does not have the facility to provide the local time differential factor.</td>
</tr>
<tr>
<td>18-19</td>
<td>If character position 17 is '-', two numeric digits are returned in the range 00 through 12 indicating the number of hours that the reported time is behind Greenwich mean time. If character position 17 is '+', two numeric digits are returned in the range 00 through 13 indicating the number of hours that the reported time is ahead of Greenwich mean time. If character position 17 is '0', the value 00 is returned.</td>
</tr>
<tr>
<td>20-21</td>
<td>Two numeric digits are returned in the range 00 through 59 indicating the number of additional minutes that the reported time is ahead of or behind Greenwich mean time, depending on whether character position 17 is '+' or '-', respectively. If character position 17 is '0', the value 00 is returned.</td>
</tr>
</tbody>
</table>

The returned value is the date and time of compilation of the source unit that contains this function. If a program is a contained program, the returned value is the compilation date and time associated with the containing program.
YEAR-TO-YYYY

The YEAR-TO-YYYY function converts argument-1, a two-digit year, to a four-digit year. argument-2, when added to the year at the time of execution, defines the ending year of a 100-year interval, or sliding century window, into which the year of argument-1 falls.

The function type is integer.

<table>
<thead>
<tr>
<th>Current year</th>
<th>argument-1 value</th>
<th>argument-2 value</th>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1995</td>
<td>4</td>
<td>23</td>
<td>2004</td>
</tr>
<tr>
<td>1995</td>
<td>4</td>
<td>-15</td>
<td>1904</td>
</tr>
<tr>
<td>2008</td>
<td>98</td>
<td>23</td>
<td>1998</td>
</tr>
<tr>
<td>2008</td>
<td>98</td>
<td>-15</td>
<td>1898</td>
</tr>
</tbody>
</table>

**argument-1**

Must be a non-negative integer that is less than 100.

**argument-2**

Must be an integer. If argument-2 is omitted, the function is evaluated assuming the value 50 was specified.

The sum of the year at the time of execution and the value of argument-2 must be less than 10,000 and greater than 1,699.

Examples of return values from the YEAR-TO-YYYY function are shown in the following table.
Part 8. Compiler-directing statements and compiler directives
Chapter 22. Compiler-directing statements

A compiler-directing statement is a statement that causes the compiler to take a specific action during compilation.

You can use compiler-directing statements for the following purposes:

- Extended source library control (BASIS, DELETE, and INSERT statements)
- Source text manipulation (COPY and REPLACE statements)
- Exception handling (USE statement)
- Controlling compiler listings (*CONTROL, *CBL, EJECT, TITLE, SKIP1, SKIP2, and SKIP3 statements)
- Specifying compiler options (CBL and PROCESS statements)
- Specifying COBOL exception handling procedures (USE statements)

The SERVICE LABEL statement is used with Language Environment condition handling. It is also generated by the CICS integrated translator (and the separate CICS translator).

The following compiler directing statements have no effect: ENTER, READY or RESET TRACE, and SERVICE RELOAD.

BASIS statement

The BASIS statement is an extended source text library statement. It provides a complete COBOL program as the source for a compilation.

A complete program can be stored as an entry in a user-defined library and can be used as the source for a compilation. Compiler input is a BASIS statement, optionally followed by any number of INSERT and DELETE statements.

Format

```
sequence-number BASIS basis-name literal-1
```

- **sequence-number**
  
  Can optionally appear in columns 1 through 6, followed by a space. The content of this field is ignored.

- **BASIS**
  
  Can appear anywhere in columns 1 through 72, followed by `basis-name`. There must be no other text in the statement.

- **basis-name, literal-1**
  
  Is the name by which the library entry is known to the system environment.

  For rules of formation and processing rules, see the description under `literal-1` and `text-name` of the “COPY statement” on page 524.

The source file remains unchanged after execution of the BASIS statement.

**Usage note:** If INSERT or DELETE statements are used to modify the COBOL source text provided by a BASIS statement, the sequence field of the COBOL source text must contain numeric sequence numbers in ascending order.
**PROCESS(CBL) statement**

With the PROCESS(CBL) statement, you can specify compiler options to be used in the compilation of the program. The PROCESS(CBL) statement is placed before the IDENTIFICATION DIVISION header of an outermost program.

**Format**

```
PROCESS CBL options-list
```

**options-list**

A series of one or more compiler options, each one separated by a comma or a space.

For more information about compiler options, see Compiler options in the Enterprise COBOL Programming Guide.

The PROCESS(CBL) statement can be preceded by a sequence number in columns 1 through 6. The first character of the sequence number must be numeric, and PROCESS or CBL can begin in column 8 or after; if a sequence number is not specified, PROCESS or CBL can begin in column 1 or after.

The PROCESS(CBL) statement must end before or at column 72, and options cannot be continued across multiple PROCESS(CBL) statements. However, you can use more than one PROCESS(CBL) statement. Multiple PROCESS(CBL) statements must follow one another with no intervening statements of any other type.

The PROCESS(CBL) statement must be placed before any comment lines or other compiler-directing statements.

***CONTROL (*CBL) statement**

With the *CONTROL (or *CBL) statement, you can selectively display or suppress the listing of source code, object code, and storage maps throughout the source text.

**Format**

```
*CONTROL *CBL SOURCE NOSOURCE LIST NOLIST MAP NOMAP
```

For a complete discussion of the output produced by these options, see Getting listings in the Enterprise COBOL Programming Guide.

The *CONTROL and *CBL statements are synonymous. *CONTROL is accepted anywhere that *CBL is accepted.

The characters *CONTROL or *CBL can start in any column beginning with column 7, followed by at least one space or comma and one or more option keywords. The option keywords must be separated by one
or more spaces or commas. This statement must be the only statement on the line, and continuation is not allowed. The statement can be terminated with a period.

The *CONTROL and *CBL statements must be embedded in a program source. For example, in the case of batch applications, the *CONTROL and *CBL statements must be placed between the PROCESS (CBL) statement and the end of the program (or END PROGRAM marker, if specified).

The source line containing the *CONTROL (*CBL) statement will not appear in the source listing.

If an option is defined at installation as a fixed option, that fixed option takes precedence over all of the following parameter and statements:

- PARM (if available)
- CBL statement
- *CONTROL (*CBL) statement

The requested options are handled in the following manner:

1. If an option or its negation appears more than once in a *CONTROL statement, the last occurrence of the option word is used.
2. If the corresponding option has been requested as a parameter to the compiler, then a *CONTROL statement with the negation of the option word must precede the portions of the source text for which listing output is to be inhibited. Listing output then resumes when a *CONTROL statement with the affirmative option word is encountered.
3. If the negation of the corresponding option has been requested as a parameter to the compiler, then that listing is always inhibited.
4. The *CONTROL statement is in effect only within the source program in which it is written, including any contained programs. It does not remain in effect across batch compiles of two or more COBOL source programs.

**Source code listing**

The topic lists statements that control the listing of the input source text lines.

The statement can be any of the following one:

- *CONTROL SOURCE [*CBL SOURCE]
- *CONTROL NOSOURCE [*CBL NOSOURCE]

If a *CONTROL NOSOURCE statement is encountered and SOURCE has been requested as a compilation option, printing of the source listing is suppressed from this point on. An informational (I-level) message is issued stating that printing of the source has been suppressed.

**Object code listing**

The topic lists statements that control the listing of generated object code in the PROCEDURE DIVISION.

The statement can be any of the following one:

- *CONTROL LIST [*CBL LIST]
- *CONTROL NOLIST [*CBL NOLIST]

If a *CONTROL NOLIST statement is encountered, and LIST has been requested as a compilation option, listing of generated object code is suppressed from this point on.

**Storage map listing**

The topic lists statements that control the listing of storage map entries occurring in the DATA DIVISION.

The statement can be any of the following one:
If a *CONTROL NOMAP statement is encountered, and MAP has been requested as a compilation option, listing of storage map entries is suppressed from this point on.

For example, either of the following sets of statements produces a storage map listing in which A and B will not appear:

```
*CONTROL NOMAP         *CBL NOMAP
  01  A                  01  A
  02  B                  02  B
*CONTROL MAP           *CBL MAP
```

## COPY statement

The COPY statement is a library statement that places prewritten text in a COBOL compilation unit.

Prewritten source code entries can be included in a compilation unit at compile time. Thus, an installation can use standard file descriptions, record descriptions, or procedures without recoding them. These entries and procedures can then be saved in user-created libraries; they can then be included in programs and class definitions by means of the COPY statement.

Compilation of the source code containing COPY statements is logically equivalent to processing all COPY statements before processing the resulting source text.

The effect of processing a COPY statement is that the library text associated with `text-name` is copied into the compilation unit, logically replacing the entire COPY statement, beginning with the word COPY and ending with the period, inclusive. When the REPLACING phrase is not specified, the library text is copied unchanged.

### Format

```
COPY text-name literal-1 OF library-name literal-2
SUPPRESS
REPLACING operand-1 BY operand-2
LEADING partial-word-1 == BY == partial-word-2 ==
TRAILING
```

### text-name, library-name

text-name identifies the copy text. library-name identifies where the copy text exists.

- Can be from 1-30 characters in length
- Can contain the following characters: Latin uppercase letters A-Z, Latin lowercase letters a-z, digits 0-9, and hyphen
- The first or last character must not be a hyphen
- Cannot contain an underscore
Neither text-name nor library-name need to be unique within a program. They can be identical to other user-defined words in the program.

text-name need not be qualified. If text-name is not qualified, a library-name of SYSLIB is assumed.

When the compiler searches for COPY members in PDS or PDSE datasets, including those specified in the COPYLOC option with the DSN argument, only the first eight characters of text-name are used as the identifying name. When the compiler searches for COPY text in z/OS Unix directories, such as those directories specified via the -I option of the cob2 command or those directories specified via the SYSLIB environment variable (when the compiler is invoked from cob2) or those directories specified via the COPYLOC option with the PATH argument, all characters are significant.

For details, see “Copy member search order” on page 533.

\[\text{literal-1, literal-2}\]

Must be alphanumeric literals. literal-1 identifies the copy text. literal-2 identifies where the copy text exists.

When compiling from JCL or TSO:

- Literals can be from 1-30 characters in length.
- Literals can contain characters: A-Z, a-z, 0-9, hyphen, @, #, or $.
- The first or last character must not be a hyphen.
- Literals cannot contain an underscore.
- Only the first eight characters are used as the identifying name.

When compiling with the cob2 command and processing COPY text residing in the z/OS UNIX file system, the literal can be from 1 to 160 characters in length.

The uniqueness of text-name and library-name is determined after the formation and conversion rules for a system-dependent name have been applied.

For information about the mapping of characters in the text-name, library-name, and literals, see Compiler-directing statements in the Enterprise COBOL Programming Guide.

\[\text{operand-1, operand-2}\]

Can be pseudo-text, an identifier, a function-identifier, a literal, or a COBOL word (except the word COPY). For details, see “REPLACING phrase” on page 526.

\[\text{partial-word-1, partial-word-2}\]

Can be a partial-word. For details, see “REPLACING phrase” on page 526.

Each COPY statement must be preceded by a space and ended with a separator period.

A COPY statement can appear in the source text anywhere a character string or a separator can appear.

COPY statements can be nested, and any COPY statement in a chain of nested COPY statements can have the REPLACING phrase, provided there is only one such COPY statement in the chain. When the REPLACING phrase is specified for a COPY statement that appears in a chain of nested COPY statements, the REPLACING phrase applies to all library text that is included by COPY statements nested under the COPY statement that has the REPLACING phrase.
A nested COPY statement cannot cause recursion. That is, a COPY member can be named only once in a set of nested COPY statements until the end-of-file for that COPY member is reached. For example, assume that the source text contains the statement: COPY X. and library text X contains the statement: COPY Y.

In this case, library text contained in Y must not have a COPY X or a COPY Y statement.

For details, see “Comparison and replacement rules” on page 527.

Library text copied from the library is placed into the same area of the resultant program as it is in the library. Library text must conform to the rules for the 85 COBOL Standard format. Library text can consist of or include any words, identifiers, or literals that can be written in the source text. This includes DBCS user-defined words, DBCS literals, and national literals.

**Note:** Characters outside those defined for COBOL words and separators must not appear in library text or pseudo-text except in comment lines, inline comments, comment-entries, alphanumeric literals, DBCS literals, or national literals.

### SUPPRESS phrase

The SUPPRESS phrase specifies that the library text is not to be printed on the source listing.

### REPLACING phrase

When the REPLACING phrase is specified, the library text is copied, and each properly matched occurrence of operand-1 or partial-word-1 within the library text is replaced by the associated operand-2 or partial-word-2.

In the discussion that follows, when the LEADING or TRAILING keyword of the REPLACING phrase is specified, each operand of the REPLACING phrase must be a partial-word. Otherwise, each operand can consist of one of the following items:

- Pseudo-text
- An identifier
- A literal
- A COBOL word (except the word COPY)
- A function-identifier

### pseudo-text

A sequence of text words that are bounded by, but not including, pseudo-text delimiters (==). Both characters of each pseudo-text delimiter must appear on one line.

Individual text words within pseudo-text can be up to 322 characters long. They can be continued subject to the normal continuation rules for source code format.

A text word must be delimited by separators. For more information, see Chapter 1, “Characters,” on page 3.

*pseudo-text-1* refers to pseudo-text when used for operand-1, and *pseudo-text-2* refers to pseudo-text when used for operand-2.

*pseudo-text-1* can be one or more text words. It can consist solely of the separator comma or separator semicolon. *pseudo-text-2* can be zero or more text words. It can consist solely of space characters, comment lines, or inline comments.

Each text word in *pseudo-text-2* that is to be copied into the program is placed in the same area of the resultant program as the area in which it appears in *pseudo-text-2*.

Pseudo-text can consist of or include any words (except COPY), identifiers, or literals that can be written in the source text. This includes DBCS user-defined words, DBCS literals, and national literals.
DBCS user-defined words must be wholly formed; that is, there is no partial-word replacement for DBCS words.

Words or literals containing DBCS characters cannot be continued across lines.

Use pseudo-text when you replace a PICTURE character-string. To avoid ambiguities, the entire PICTURE clause, including the keyword PICTURE or PIC, should be specified in pseudo-text-1.

**identifier**
Can be defined in any section of the DATA DIVISION.

**literal**
Can be numeric, alphanumeric, DBCS, or national.

**word**
Can be any single COBOL word (except COPY), including DBCS user-defined words. DBCS user-defined words must be wholly formed. You cannot replace part of a DBCS word.

You can include the nonseparator COBOL characters (for example, + * / $ < > =) as part of a COBOL word when used as REPLACING operands. In addition, a hyphen or underscore can be at the beginning of the word or a hyphen can be at the end of the word.

**function-identifier**
A sequence of character strings and separators that uniquely references the data item that results from the evaluation of a function. For more information, see “Function-identifier” on page 73.

**partial-word**
A single text word that is bounded by, but not including, pseudo-text delimiters (==). Both characters of each pseudo-text delimiter must appear on one line. However, the text word within a partial-word can be continued.

The following rules apply to partial-word-1 and partial-word-2:

- partial-word-1 consists of one text word.
- partial-word-2 consists of zero or one text word.
- partial-word-1 and partial-word-2 cannot be an alphanumeric literal, national literal, DBCS literal, or DBCS word.

For purposes of matching, each identifier, literal, word, or function-identifier is treated as pseudo-text that contains only that identifier, literal, word, or function-identifier, respectively.

**Comparison and replacement rules**
This topic introduces detailed rules for comparison and replacement.

- Arithmetic and logical operators are considered text words and can be replaced only through a pseudo-text operand.
- Beginning and ending blanks are not included in the text comparison process. Embedded blanks are used in the text comparison process to separate multiple text words.
- When operand-1 is a figurative constant, operand-1 matches only the same exact figurative constant. For example, if ALL "AB" is specified in operand-1, "ABAB" in the library text is not considered a match; only ALL "AB" is considered a match.
- Any separator comma, semicolon, or space that precedes the leftmost word in the library text is copied into the source text. Beginning with the leftmost library text word and the first operand-1 or partial-word-1 specified in the REPLACING phrase, the entire REPLACING operand that precedes the keyword BY is compared to an equivalent number of contiguous library text words.
- operand-1 matches the library text only if the ordered sequence of text words that forms operand-1 is equal, character for character, to the ordered sequence of library words. For national characters, the sequence of national characters must be equal, national character for national character, to the ordered sequence of library words.
• When the LEADING phrase is specified, partial-word-1 matches the library text only if the contiguous sequence of characters that forms partial-word-1 is equal, character for character, to an equal number of contiguous characters that start with the leftmost character position of a library text word.

When the TRAILING phrase is specified, partial-word-1 matches the library text only if the contiguous sequence of characters that forms partial-word-1 is equal, character for character, to an equal number of contiguous characters that end with the rightmost character position of a library text word.

• For matching purposes, each occurrence of a separator comma, a separator semicolon, or a sequence of one or more separator spaces is considered to be a single space.

However, when operand-1 or partial-word-1 consists solely of a separator comma or separator semicolon, the operand-1 or partial-word-1 participates in the match as a text word. In this case, the space that follows the comma or semicolon separator can be omitted.

When the library text contains a closing quotation mark that is not immediately followed by a separator space, a separator comma, a separator semicolon, or a separator period, the closing quotation mark is considered a separator quotation mark.

• If no match occurs, the comparison is repeated with each successive operand-1 or partial-word-1 if specified, until either a match is found or no further REPLACING operands exist.

• Whenever a match occurs between operand-1 and the library text, the corresponding operand-2 is copied into the source text. Whenever a match occurs between partial-word-1 and the library text word, the matched characters of that library text word are either replaced by partial-word-2 or deleted when partial-word-2 consists of zero text words.

• When all operands are compared and no match occurs, the leftmost library text word is copied into the source text.

• Once a library text word is finished being processed and is either copied into the source text unchanged or replaced because of a match, the next successive uncopied library text word is then considered to be the leftmost text word, and the comparison cycle starts again, beginning with the first occurrence of operand-1 or partial-word-1. The process continues until the rightmost library text word is compared.

• The sequence of text words in the library text, pseudo-text-1, and partial-word-1 is determined by the rules for reference format. For more information, see Chapter 6, “Reference format,” on page 51.

• When text words are placed in the source text, additional spaces are introduced only between text words where there already exists a space, including the assumed space between source lines.

The following rules apply to comment lines, inline comments, and blank lines:

– Comment lines, inline comments, or blank lines in the library text, pseudo-text-1, or partial-word-1 are ignored for purposes of matching.

– Comment lines, inline comments, or blank lines in the library text are copied into the resultant source text unchanged with the following exception: a comment line, an inline comment, or a blank line in the library text is not copied if that comment line, inline comment, or blank line appears in the sequence of text words that match pseudo-text-1 or partial-word-1.

– Comment lines, inline comments, or blank lines in pseudo-text-2 or partial-word-2 are copied into the resultant program unchanged whenever pseudo-text-2 or partial-word-2 is placed into the source text as a result of text replacement.

– If the word COPY appears in a comment-entry, or in the place where a comment-entry can appear, it is considered part of the comment-entry.

• COPY REPLACING does not affect the EJECT, SKIP1, SKIP2, SKIP3, or TITLE compiler-directing statements.

• Lines that contain *CONTROL (*CBL), EJECT, SKIP1, SKIP2, SKIP3, or TITLE statements can appear in the library text. Such lines are copied into the resultant source text unchanged.

• The following rules apply to debugging lines:

– Debugging lines are permitted in library text and pseudo-text. Text words within a debugging line participate in the matching rules as if the letter "D" did not appear in the indicator area. A debugging
line is specified within pseudo-text if the debugging line begins in the source text after the opening pseudo-text delimiter but before the matching closing pseudo-text delimiter.

- If more lines are introduced into the source text as a result of a COPY statement, each text word that is introduced appears on a debugging line if the COPY statement begins on a debugging line or if the text word that is introduced appears on a debugging line in library text. When a text word specified in the BY phrase is introduced, it appears on a debugging line if the first library text word that is being replaced is specified on a debugging line.

- When a COPY statement is specified on a debugging line, the copied text is treated as though it appeared on a debugging line, except that comment lines in the text appear as comment lines in the resulting source text.

- After all COPY and REPLACE statements are processed, a debugging line will be considered to have all the characteristics of a comment line, if the WITH DEBUGGING MODE clause is not specified in the SOURCE-COMPUTER paragraph.

- The syntactic correctness of the entire COBOL source text cannot be determined until all COPY and REPLACE statements have been completely processed, because the syntactic correctness of the library text cannot be independently determined.

- (This rule applies to pseudo-text only.) If the source text has occurrences of a dummy operand :TAG: that is delimited by colons in the program text, the compiler replaces the dummy operand with the required text. Example 3 shows how it is used with the dummy operand :TAG:. The colons serve as separators and make TAG a stand-alone operand.

- The COPY statement with REPLACING phrase can be used to replace parts of words, either by using the LEADING|TRAILING partial-word-1 BY partial-word-2 phrase, or by using the pseudo-text :TAG: method.

- After replacement, text words are placed in the source text according to the 85 COBOL Standard format rules.

**Comparison and replacement examples**

This topic shows examples of comparison and replacement.

Sequences of code (such as file and data descriptions, error, and exception routines) that are common to a number of programs can be saved in a library, and then used with the COPY statement. If naming conventions are established for such common code, the REPLACING phrase need not be specified. If the names change from one program to another, the REPLACING phrase can be used to supply meaningful names for this program.

**Example 1**

In this example, the library text PAYLIB consists of the following DATA DIVISION entries:

```plaintext
01 A.
  02 B    PIC S99.
  02 C    PIC S9(5)V99.
  02 D    PIC S9999 OCCURS 1 TO 52 TIMES DEPENDING ON B OF A.
```

You can use the COPY statement in the DATA DIVISION of a program as follows:

```plaintext
COPY PAYLIB.
```

In this program, the library text is copied. The resulting text is treated as if it were written as follows:

```plaintext
01 A.
  02 B    PIC S99.
  02 C    PIC S9(5)V99.
  02 D    PIC S9999 OCCURS 1 TO 52 TIMES DEPENDING ON B OF A.
```
**Example 2**

To change some or all of the names within the library text, you can use the REPLACING phrase:

```
COPY PAYLIB REPLACING A BY PAYROLL
   B BY PAY-CODE
   C BY GROSS-PAY
   D BY HOURS.
```

In this program, the library text is copied. The resulting text is treated as if it were written as follows:

```
01  PAYROLL.
02  PAY-CODE    PIC S99.
02  GROSS-PAY   PIC S9(5)V99.
02  HOURS       PIC S9999 OCCURS 1 TO 52 TIMES
               DEPENDING ON PAY-CODE OF PAYROLL.
```

The changes that are shown are made only for this program. The text remains unchanged as it appears in the library.

**Example 3**

If the following conventions are followed in the library text, parts of names (for example, the prefix portion of data names) can be changed with the REPLACING phrase.

In this example, the library text `PAYLIB` consists of the following DATA DIVISION entries:

```
01  :TAG:.
02  :TAG:-WEEK          PIC S99.
02  :TAG:-GROSS-PAY     PIC S9(5)V99.
02  :TAG:-HOURS         PIC S999  OCCURS 1 TO 52 TIMES
                        DEPENDING ON :TAG:-WEEK OF :TAG:.
```

You can use the COPY statement in the DATA DIVISION of a program as follows:

```
COPY PAYLIB REPLACING ==:TAG:== BY ==Payroll==.
```

**Usage Note:** In this example, the required use of colons or parentheses as delimiters in the library text. Colons are recommended for clarity because parentheses can be used for a subscript, for instance in referencing a table element.

In this program, the library text is copied. The resulting text is treated as if it were written as follows:

```
01  PAYROLL.
02  PAYROLL-WEEK        PIC S99.
02  PAYROLL-GROSS-PAY   PIC S9(5)V99.
02  PAYROLL-HOURS       PIC S999 OCCURS 1 TO 52 TIMES
                        DEPENDING ON PAYROLL-WEEK OF PAYROLL.
```

The changes that are shown are made only for this program. The text remains unchanged as it appears in the library.

**Example 4**

This example shows how to selectively replace level numbers without replacing the numbers in the PICTURE clause:

```
COPY xxx REPLACING ==(01)== BY ==(01)==
   ==(01)== BY ==(05)==.
```

**Example 5**
This example demonstrates use of the LEADING keyword of the REPLACING phrase in the COPY statement. The library text PAYLIB consists of the following DATA DIVISION entries:

```
01  DEPT.
  02  DEPT-WEEK       PIC S99.
  02  DEPT-GROSS-PAY  PIC S9(5)V99.
  02  DEPT-HOURS      PIC S999 OCCURS 1 TO 52 TIMES
                     DEPENDING ON DEPT-WEEK OF DEPT.
```

You can use the COPY statement in the DATA DIVISION of a program as follows:

```
COPY PAYLIB REPLACING LEADING == DEPT == BY == PAYROLL ==.
```

In this program, the library text is copied. The resulting text is treated as if it were written as follows:

```
01  PAYROLL.
  02  PAYROLL-WEEK       PIC S99.
  02  PAYROLL-GROSS-PAY  PIC S9(5)V99.
  02  PAYROLL-HOURS      PIC S999 OCCURS 1 TO 52 TIMES
                     DEPENDING ON PAYROLL-WEEK OF PAYROLL.
```

The changes that are shown are made only for this program. The text remains unchanged as it appears in the library.

**Example 6**

This example demonstrates use of the TRAILING keyword of the REPLACING phrase in the COPY statement. The library text PAYLIB consists of the following DATA DIVISION entries:

```
01  PAYROLL.
  02  PAYROLL-WEEK       PIC S99.
  02  PAYROLL-GROSS-PAY  PIC S9(5)V99.
  02  PAYROLL-HOURS      PIC S999 OCCURS 1 TO 52 TIMES
                     DEPENDING ON PAYROLL-WEEK OF PAYROLL.
```

You can use the COPY statement in the DATA DIVISION of a program as follows:

```
COPY PAYLIB REPLACING TRAILING == GROSS-PAY == BY == NET-PAY ==.
```

In this program, the library text is copied. The resulting text is treated as if it were written as follows:

```
01  PAYROLL.
  02  PAYROLL-WEEK       PIC S99.
  02  PAYROLL-NET-PAY    PIC S9(5)V99.
  02  PAYROLL-HOURS      PIC S999 OCCURS 1 TO 52 TIMES
                     DEPENDING ON PAYROLL-WEEK OF PAYROLL.
```

The changes that are shown are made only for this program. The text remains unchanged as it appears in the library.

**Example 7**

This example demonstrates a scenario where two types of partial-word replacement are specified in a single REPLACING phrase. The library text PAYLIB consists of the following DATA DIVISION entries:

```
01  PAYROLL.
  02  PAYROLL-WEEK       PIC S99.
  02  :TAG:-GROSS-PAY    PIC S9(5)V99.
  02  PAYROLL-HOURS      PIC S999 OCCURS 1 TO 52 TIMES
                     DEPENDING ON PAYROLL-WEEK OF PAYROLL.
```
You can use the COPY statement in the DATA DIVISION of a program as follows:

COPY PAYLIB REPLACING == :TAG: == BY == PAYROLL ==
TRAILING == GROSS-PAY == BY == NET-PAY ==.

In this program, the library text is copied. The resulting text is treated as if it were written as follows:

01 PAYROLL.
  02 PAYROLL-WEEK       PIC S99.
  02 PAYROLL-GROSS-PAY  PIC S9(5)V99.
  02 PAYROLL-HOURS      PIC S999 OCCURS 1 TO 52 TIMES
                       DEPENDING ON PAYROLL-WEEK OF PAYROLL.

Two types of partial-word replacement are specified in the same REPLACING operation, but as usual, one replacement is done on a single library text word. Therefore, even though :TAG:-GROSS-PAY is considered a match with the first operand of both replacement operations, after the first match and replacement is performed, no more replacement is performed on that word.

The changes that are shown are made only for this program. The text remains unchanged as it appears in the library.

Example 8

This example demonstrates support for the REPLACING phrase in a chain of nested COPY statements. In this example, it is the outermost COPY statement that has the REPLACING phrase, but note that the REPLACING phrase can be specified on any of the nested COPY statements in the chain, provided it is specified on only one of them. The library text PAYLIB consists of the following DATA DIVISION entries:

01 PAYROLL.
  02 PAYROLL-WEEK     PIC S99.
  02 :TAG:-GROSS-PAY  PIC S9(5)V99.
  02 PAYROLL-HOURS    PIC S999 OCCURS 1 TO 52 TIMES
                       DEPENDING ON PAYROLL-WEEK OF PAYROLL.
COPY PAYLIB2

The library text PAYLIB2 consists of the following DATA DIVISION entries:

01 PAYROLL2.
  02 PAYROLL2-WEEK      PIC S99.
  02 PAYROLL2-HOURS     PIC S999 OCCURS 1 TO 52 TIMES
                       DEPENDING ON PAYROLL2-WEEK OF PAYROLL2.

You can use the COPY statement in the DATA DIVISION of a program as follows:

COPY PAYLIB REPLACING == :TAG: == BY == PAYROLL ==
TRAILING == GROSS-PAY == BY == NET-PAY ==.

In this program, the library text is copied. The resulting text is treated as if it were written as follows:

01 PAYROLL.
  02 PAYROLL-WEEK       PIC S99.
  02 PAYROLL-NET-PAY    PIC S9(5)V99.
  02 PAYROLL-HOURS      PIC S999 OCCURS 1 TO 52 TIMES
                       DEPENDING ON PAYROLL-WEEK OF PAYROLL.

01 PAYROLL2.
  02 PAYROLL2-WEEK      PIC S99.
  02 PAYROLL2-NET-PAY   PIC S9(5)V99.
  02 PAYROLL2-HOURS     PIC S999 OCCURS 1 TO 52 TIMES
                       DEPENDING ON PAYROLL2-WEEK OF PAYROLL2.

The REPLACING phrase in the outermost COPY statement applies not only to the library text in PAYLIB but also to the text in PAYLIB2.
The changes that are shown are made only for this program. The text remains unchanged as it appears in the library.

**Copy member search order**

This topic discusses the search order of copy members when a COPY statement is processed.

When the compiler is invoked from JCL or any method other than from z/OS UNIX, the search for a copy member is performed according to the following rules:

- If an explicit library name is specified in the COPY statement, the name is assumed to correspond to a ddname that has been allocated to a concatenation of datasets where the compiler will search for the copy member. If no library name is specified in the COPY statement, the concatenation of datasets allocated to the SYSLIB ddname is searched.

- If the copy member was not found during the above search, then all locations specified via the COPYLOC option will be searched in the order that they were specified. These locations can include a mix of z/OS UNIX directories and PDS or PDSE datasets.

When the compiler is invoked in z/OS UNIX using the cob2 command, the search for a copy member is performed according to the following rules:

- If an explicit library name is specified in the COPY statement and the name is a literal, the name will be interpreted as a z/OS UNIX directory and that directory will be searched.

- If an explicit library name is specified in the COPY statement and it is not a literal, the name is interpreted to be an environment variable that specifies a colon-separated list of z/OS UNIX directories. If the environment variable exists, those z/OS UNIX directories will be searched in the order they appear in the list. If the environment variable does not exist, the current directory will be searched.

- If an explicit library name was not specified in the COPY statement, the compiler searches for copy members in this order:
  - The current z/OS UNIX directory is searched for the copy member.
  - Any z/OS UNIX directories specified via the -I option of cob2 are searched.
  - Any z/OS UNIX directories specified via the SYSLIB environment variable are searched.

- If the copy member was not found during the above search; then all locations specified via the COPYLOC option are searched in the order that they were specified. These locations can include a mix of z/OS UNIX directories and PDS or PDSE datasets.

**Note:** When the compiler searches for a copy member in a z/OS UNIX directory and the copy member name in the corresponding COPY statement is not a literal and is not found to be referring to an environment variable, then the compiler searches for multiple different versions of the provided name with the following extensions: .cpy, .CPY, .cbl, .CBL, .cob, and .COB.

**DELETE statement**

The DELETE statement is an extended source library statement. It removes COBOL statements from a source program that was included by a BASIS statement.

**Format**

```
  sequence-number  DELETE  sequence-number-field
```

**sequence-number**

Can optionally appear in columns 1 through 6, followed by a space. The content of this field is ignored.

**DELETE**

Can appear anywhere within columns 1 through 72. The keyword DELETE must be followed by a space and the `sequence-number-field`. There must be no other text in the statement.
sequence-number-field

Each number must be equal to a sequence-number in the BASIS source program. This sequence-number is the six-digit number the programmer assigns in columns 1 through 6 of the COBOL coding form. The numbers referenced in the sequence-number-field of INSERT or DELETE statements must always be specified in ascending numeric order.

The sequence-number-field must be one of the following options:

- A single number
- A series of single numbers
- A range of numbers (indicated by separating the two bounding numbers of the range by a hyphen)
- A series of ranges of numbers
- Any combination of one or more single numbers and one or more ranges of numbers

Each entry in the sequence-number-field must be separated from the preceding entry by a comma followed by a space. For example:

000250 DELETE 000010-000050, 000400, 000450

Source program statements can follow a DELETE statement. These source program statements are then inserted into the BASIS source program before the statement following the last statement deleted (that is, in the example above, before the next statement following deleted statement 000450).

If a DELETE statement begins in column 12 or higher and a valid sequence-number-field does not follow the keyword DELETE, the compiler assumes that this DELETE statement is a COBOL DELETE statement.

Usage note: If INSERT or DELETE statements are used to modify the COBOL source program provided by a BASIS statement, the sequence field of the COBOL source program must contain numeric sequence numbers in ascending order. The source file remains unchanged. Any INSERT or DELETE statements referring to these sequence numbers must occur in ascending order.

EJECT statement

The EJECT statement specifies that the next source statement is to be printed at the top of the next page.

Format

EJECT

The EJECT statement must be the only statement on the line. It can be written in either Area A or Area B, and can be terminated with a separator period.

The EJECT statement must be embedded in a program source. For example, in the case of batch applications, the EJECT statement must be placed between the CBL (PROCESS) statement and the end of the program (or the END PROGRAM marker, if specified).

The EJECT statement has no effect on the compilation of the source unit itself.

ENTER statement

The ENTER statement is designed to facilitate the use of more than one source language in the same source program. However, only COBOL is allowed in the source program.

The ENTER statement is syntax checked but has no effect on the execution of the program.
**Format**

```
language-name-1
    routine-name-1
```

**language-name-1**
A system name that has no defined meaning. It must be either a correctly formed user-defined word or the word “COBOL.” At least one character must be alphabetic.

**routine-name-1**
Must follow the rules for formation of a user-defined word. At least one character must be alphabetic.

---

**INSERT statement**

The INSERT statement is a library statement that adds COBOL statements to a source program that was included by a BASIS statement.

```
sequence-number
    INSERT
        sequence-number-field
```

**sequence-number**
Can optionally appear in columns 1 through 6, followed by a space. The content of this field is ignored.

**INSERT**
Can appear anywhere within columns 1 through 72, followed by a space and the **sequence-number-field**. There must be no other text in the statement.

**sequence-number-field**
A number that must be equal to a **sequence-number** in the BASIS source program. This **sequence-number** is a six-digit number that the programmer assigns in columns 1 through 6 of the COBOL source line.

The numbers referenced in the **sequence-number-field** of INSERT or DELETE statements must always be specified in ascending numeric order.

The **sequence-number-field** must be a single number (for example, 000130). At least one new source program statement must follow the INSERT statement for insertion after the statement number specified by the **sequence-number-field**.

New source program statements following the INSERT statement can include any COBOL syntax.

**Usage note:** If INSERT or DELETE statements are used to modify the COBOL source program provided by a BASIS statement, the sequence field of the COBOL source program must contain numeric sequence numbers in ascending order. The source file remains unchanged. Any INSERT or DELETE statements referring to these sequence numbers must occur in ascending order.
**READY or RESET TRACE statement**

The READY or RESET TRACE statement was designed to trace the execution of procedures. The READY or RESET TRACE statement can appear only in the PROCEDURE DIVISION, but has no effect on your program.

```
Format
READY TRACE .
RESET TRACE .
```

You can trace the execution of procedures by using the USE FOR DEBUGGING declarative as described in Example: USE FOR DEBUGGING in the Enterprise COBOL Programming Guide.

**REPLACE statement**

The REPLACE statement is used to replace source text.

A REPLACE statement can occur anywhere in the source text that a character-string can occur. It must be preceded by a separator period except when it is the first statement in a separately compiled program. It must ends with a separator period.

The REPLACE statement provides a means of applying a change to an entire COBOL compilation group, or part of a compilation group, without manually having to find and modify all places that need to be changed. It is an easy method of doing simple string substitutions. It is similar in action to the REPLACING phrase of the COPY statement, except that it acts on the entire source text, not just on the text in COPY libraries.

If the word REPLACE appears in a comment-entry or in the place where a comment-entry can appear, it is considered part of the comment-entry.

```
Format 1
REPLACE
  == pseudo-text-1 == BY == pseudo-text-2 ==
  LEADING == partial-word-1 == BY == partial-word-2 ==
  TRAILING ==
.
```

Each matched occurrence of pseudo-text-1 in the source text is replaced by the corresponding pseudo-text-2.

```
Format 2
REPLACE OFF.
```

Any text replacement currently in effect is discontinued with the format-2 form of REPLACE. If format 2 is not specified, a specific occurrence of the REPLACE statement is in effect from the point at which it is specified until the next occurrence of a REPLACE statement or the end of the separately compiled program.
**pseudo-text-1, pseudo-text-2**

A sequence of text words that are bounded by, but not including, pseudo-text delimiters (==). Both characters of each pseudo-text delimiter must appear on one line.

Individual text words within pseudo-text can be up to 322 characters long. They can be continued subject to the normal continuation rules for source code format.

A text word must be delimited by separators. For more information, see Chapter 1, “Characters,” on page 3.

**pseudo-text-1** can be one or more text words. It can consist solely of the separator comma or separator semicolon. **pseudo-text-2** can be zero or more text words. It can consist solely of space characters, comment lines, or inline comments.

Each text word in **pseudo-text-2** that is to be copied into the program is placed in the same area of the resultant program as the area in which it appears in **pseudo-text-2**.

Pseudo-text can consist of or include any words (except COPY), identifiers, or literals that can be written in the source text. This includes DBCS user-defined words, DBCS literals, and national literals.

DBCS user-defined words must be wholly formed; that is, there is no partial-word replacement for DBCS words.

Words or literals containing DBCS characters cannot be continued across lines.

**partial-word-1, partial-word-2**

A single text word that is bounded by, but not including, pseudo-text delimiters (==). Both characters of each pseudo-text delimiter must appear on one line. However, the text word within a partial-word can be continued.

The following rules apply to **partial-word-1** and **partial-word-2**:

- **partial-word-1** consists of one text word.
- **partial-word-2** consists of zero or one text word.
- **partial-word-1** and **partial-word-2** cannot be an alphanumeric literal, national literal, DBCS literal, or DBCS word.

The compiler processes REPLACE statements in source text after the processing of any COPY statements. COPY must be processed first to assemble complete source text. Then, REPLACE can be used to modify that source text, performing simple string substitution. REPLACE statements cannot themselves contain COPY statements.

The text that is produced as a result of the processing of a REPLACE statement must not contain a REPLACE statement.

**Continuation rules for pseudo-text and partial-word**

The character-strings and separators that comprise pseudo-text and partial-words can start in either Area A or Area B. However, if a hyphen is in the indicator area of a line, and that hyphen follows the opening pseudo-text or partial-word delimiter, Area A of the line must be blank, and the normal rules for continuation of lines apply to the formation of text words. See “Continuation lines” on page 54.

**Comparison rules**

The comparison operation that determines text replacement starts with the leftmost source text word that follows the REPLACE statement, and with the first word of **pseudo-text-1** or **partial-word-1**.

- **pseudo-text-1** is compared to an equivalent number of contiguous source text words. **pseudo-text-1** matches the source text only if the ordered sequence of text words that forms **pseudo-text-1** is equal, character for character, to the ordered sequence of source text words. For national characters, the sequence of national characters must be equal, national character for national character, to the ordered sequence of library words.
• When the LEADING phrase is specified, partial-word-1 matches the source text only if the contiguous sequence of characters that forms partial-word-1 is equal, character for character, to an equal number of contiguous characters that start with the leftmost character position of a source text word.

When the TRAILING phrase is specified, partial-word-1 matches the source text only if the contiguous sequence of characters that forms partial-word-1 is equal, character for character, to an equal number of contiguous characters that end with the rightmost character position of a source text word.

• For matching purposes, each occurrence of a separator comma, a separator semicolon, or a sequence of one or more separator spaces is considered to be a single space.

However, when pseudo-text-1 or partial-word-1 consists solely of a separator comma or separator semicolon, the comma or semicolon participates in the match as a text word. In this case, the space that follows the comma or semicolon separator can be omitted.

When the source text contains a closing quotation mark that is not immediately followed by a separator space, a separator comma, a separator semicolon, or a separator period, the closing quotation mark is considered a separator quotation mark.

• If no match occurs, the comparison is repeated with each successive occurrence of pseudo-text-1 or partial-word-1 if specified, until either a match is found or no further REPLACING operands exist.

• When all occurrences of pseudo-text-1 or partial-word-1 are compared and no match occurs, the next successive source text word is then considered to be the leftmost source text word, and the comparison cycle starts again, beginning with the first occurrence of pseudo-text-1 or partial-word-1.

• Whenever a match occurs between pseudo-text-1 and the source text, the corresponding pseudo-text-2 replaces the matched text in the source text. Whenever a match occurs between partial-word-1 and the source text word, the matched characters of that source text word are either replaced by partial-word-2 or deleted if partial-word-2 consists of zero text words. The source text word that immediately follows the rightmost text word that participated in the match is then considered as the leftmost source text word. The comparison cycle starts again, beginning with the first occurrence of pseudo-text-1 or partial-word-1.

• The comparison operation continues until the rightmost text word in the source text that is within the scope of the REPLACE statement has either participated in a match, or been considered as a leftmost source text word and participated in a complete comparison cycle.

Replacement rules

This topic introduces detailed rules for replacement.

• The sequence of text words in the source text, pseudo-text-1, and partial-word-1 is determined by the rules for reference format. For more information, see Chapter 6, “Reference format,” on page 51.

• Text words that are inserted into the source text as a result of processing a REPLACE statement are placed in the source text according to the rules for reference format. When inserting text words of pseudo-text-2 or partial-word-2 into the source text, additional spaces are introduced only between text words where there already exists a space, including the assumed space between source lines.

• If more lines are introduced into the source text as a result of the processing of REPLACE statements, the indicator area of the introduced lines contains the same character as the line on which the text being replaced begins, unless that line contains a hyphen, in which case the introduced line contains a space.

• If any literal within pseudo-text-2 or partial-word-2 is of a length too great to be accommodated on a single line without continuation to another line in the resultant program, and the literal is not being placed on a debugging line, more continuation lines are introduced that contain the remainder of the literal. If replacement requires the continued literal to be continued on a debugging line, the program is in error.

• Each word in pseudo-text-2 or partial-word-2 that is to be placed into the resultant program begins in the same area of the resultant program as it appears in pseudo-text-2 or partial-word-2.

• The following rules apply to comment lines, inline comments, and blank lines:
– Comment lines, inline comments, or blank lines in the source text, *pseudo-text-1*, or *partial-word-1* are ignored for purposes of matching.
– Comment lines, inline comments, or blank lines in the source text are copied into the resultant source text unchanged with the following exception: a comment line, an inline comment, or a blank line in the source text is not copied if that comment line, inline comment, or blank line appears in the sequence of text words that match *pseudo-text-1* or *partial-word-1*.
– Comment lines, inline comments, or blank lines in *pseudo-text-2* or *partial-word-2* are copied into the resultant program unchanged whenever *pseudo-text-2* or *partial-word-2* is placed into the source text as a result of text replacement.
• Lines that contain *CONTROL* (*CBL), EJECT, SKIP1, SKIP2, SKIP3, or TITLE statements can appear in the source text. Such lines are copied into the resultant source text unchanged.
• The following rules apply to debugging lines:
  – Debugging lines are permitted in pseudo-text or partial-words. Text words within a debugging line participate in the matching rules as if the letter "D" did not appear in the indicator area.
  – When a REPLACE statement is specified on a debugging line, the statement is treated as if the letter "D" did not appear in the indicator area.
  – After all COPY and REPLACE statements are processed, a debugging line is considered to have all the characteristics of a comment line if the WITH DEBUGGING MODE clause is not specified in the SOURCE-COMPUTER paragraph.
• Except for COPY and REPLACE statements, the syntactic correctness of the source text cannot be determined until all COPY and REPLACE statements are completely processed.
• (This rule only applies to pseudo-text.) If the source text has occurrences of a dummy operand :TAG: that is delimited by colons in the program text, the compiler replaces the dummy operand with the required text. The colons serve as separators and make TAG a stand-alone operand.

For example, you can use the REPLACE statement in the DATA DIVISION of a program as follows:

```
REPLACE ==:TAG:== BY ==Payroll==.
01  :TAG:.
   02  :TAG:-WEEK          PIC S99.
   02  :TAG:-GROSS-PAY     PIC S9(5)V99.
   02  :TAG:-HOURS         PIC S999  OCCURS 1 TO 52 TIMES
                             DEPENDING ON :TAG:-WEEK OF :TAG:.
```

• The REPLACE statement can be used to replace parts of words, either by using the LEADING/TRAILING *partial-word-1* BY *partial-word-2* phrase, or by using the pseudo-text :TAG: method.

**SERVICE LABEL statement**

This statement is generated by the CICS integrated language translator (and the separate CICS translator) to indicate control flow. It is also used after calls to CEE3SRP when using Language Environment condition handling. For more information about CEE3SRP, see the Language Environment Programming Guide.

**Format**

```
 SERVICE LABEL
```

The SERVICE LABEL statement can appear only in the PROCEDURE DIVISION, but not in the declaratives section.
SERVICE RELOAD statement

The SERVICE RELOAD statement is syntax checked, but has no effect on the execution of the program.

**Format**

```
SERVICE RELOAD   — identifier-1
```

SKIP statements

The SKIP1, SKIP2, and SKIP3 statements specify blank lines that the compiler should add when printing the source listing. SKIP statements have no effect on the compilation of the source text itself.

**Format**

```
SKIP1

SKIP2

SKIP3
```

**SKIP1**

Specifies a single blank line to be inserted in the source listing.

**SKIP2**

Specifies two blank lines to be inserted in the source listing.

**SKIP3**

Specifies three blank lines to be inserted in the source listing.

SKIP1, SKIP2, or SKIP3 can be written anywhere in either Area A or Area B, and can be terminated with a separator period. It must be the only statement on the line.

The SKIP statements must be embedded in a program source. For example, in the case of batch applications, a SKIP1, SKIP2, or SKIP3 statement must be placed between the CBL (PROCESS) statement and the end of the program or class (or the END CLASS marker or END PROGRAM marker, if specified).

TITLE statement

The TITLE statement specifies a title to be printed at the top of each page of the source listing produced during compilation.

If no TITLE statement is found, a title containing the identification of the compiler and the current release level is generated. The title is left-justified on the title line.

**Format**

```
TITLE   — literal
```

**literal**

Must be an alphanumeric literal, DBCS literal, or national literal and can be followed by a separator period.

Must not be a figurative constant.

In addition to the default or chosen title, the right side of the title line contains the following items:
• For programs, the name of the program from the PROGRAM-ID paragraph for the outermost program. (This space is blank on pages preceding the PROGRAM-ID paragraph for the outermost program.)
• For classes, the name of the class from the CLASS-ID paragraph.
• Current page number.
• Date and time of compilation.

The TITLE statement:
• Forces a new page immediately, if the SOURCE compiler option is in effect
• Is not itself printed on the source listing
• Has no other effect on compilation
• Has no effect on program execution
• Cannot be continued on another line
• Can appear anywhere in any of the divisions

A title line is produced for each page in the listing produced by the LIST option. This title line uses the last TITLE statement found in the source statements or the default.

The word TITLE can begin in either Area A or Area B.

The TITLE statement must be embedded in a class or program source. For example, in the case of batch applications, the TITLE statement must be placed between the CBL (PROCESS) statement and the end of the class or program (or the END CLASS marker or END PROGRAM marker, if specified).

No other statement can appear on the same line as the TITLE statement.

**USE statement**

The USE statement defines the conditions under which the procedures that follow the statement will be executed.

The formats for the USE statement are:
• EXCEPTION/ERROR declarative
• DEBUGGING declarative

For general information about declaratives, see “Declaratives” on page 241.

**EXCEPTION/ERROR declarative**

The EXCEPTION/ERROR declarative specifies procedures for input/output exception or error handling that are to be executed in addition to the standard system procedures.

The words EXCEPTION and ERROR are synonymous and can be used interchangeably.
file-name-1
Valid for all files. When this option is specified, the procedure is executed only for the files named. No file-name can refer to a sort or merge file. For any given file, only one EXCEPTION/ERROR procedure can be specified; thus, file-name specification must not cause simultaneous requests for execution of more than one EXCEPTION/ERROR procedure.

A USE AFTER EXCEPTION/ERROR declarative statement specifying the name of a file takes precedence over a declarative statement specifying the open mode of the file.

INPUT
Valid for all files. When this option is specified, the procedure is executed for all files opened in INPUT mode or in the process of being opened in INPUT mode that get an error.

OUTPUT
Valid for all files. When this option is specified, the procedure is executed for all files opened in OUTPUT mode or in the process of being opened in OUTPUT mode that get an error.

I-O
Valid for all direct-access files. When this option is specified, the procedure is executed for all files opened in I-O mode or in the process of being opened in I-O mode that get an error.

EXTEND
Valid for all files. When this option is specified, the procedure is executed for all files opened in EXTEND mode or in the process of being opened in EXTEND mode that get an error.

The EXCEPTION/ERROR procedure is executed:
• Either after completing the system-defined input/output error routine, or
• Upon recognition of an INVALID KEY or AT END condition when an INVALID KEY or AT END phrase has not been specified in the input/output statement, or
• Upon recognition of an IBM-defined condition that causes file status key 1 to be set to 9. (See “File status key” on page 276.)

After execution of the EXCEPTION/ERROR procedure, control is returned to the invoking routine in the input/output control system. If the input/output status value does not indicate a critical input/output error, the input/output control system returns control to the next executable statement following the input/output statement whose execution caused the exception.

An applicable EXCEPTION/ERROR procedure is activated when an input/output error occurs during execution of a READ, WRITE, REWRITE, START, OPEN, CLOSE, or DELETE statement. To determine what conditions are errors, see “Common processing facilities” on page 276.

The following rules apply to declarative procedures:
• A declarative procedure can be performed from a nondeclarative procedure.
• A nondeclarative procedure can be performed from a declarative procedure.
• A declarative procedure can be referenced in a GO TO statement in a declarative procedure.
• A nondeclarative procedure can be referenced in a GO TO statement in a declarative procedure.

You can include a statement that executes a previously called USE procedure that is still in control. However, to avoid an infinite loop, you must be sure that there is an eventual exit at the bottom.

You cannot use a GOBACK statement or a STOP RUN statement when an EXCEPTION/ERROR declarative is active due to a QSAM abend for a READ, WRITE, or REWRITE statement. You cannot use an EXIT PROGRAM statement in a non-nested subprogram when an EXCEPTION/ERROR declarative is active due to a QSAM abend for a READ, WRITE, or REWRITE statement. When a QSAM abend occurs during a READ, WRITE, or REWRITE statement, the file status code can be "34" or "90".

You cannot use a GOBACK statement or an EXIT PROGRAM statement while a declarative is active in a nested program. You cannot use a GOBACK statement or an EXIT METHOD statement while a declarative is active in a method.

EXCEPTION/ERROR procedures can be used to check the file status key values whenever an input/output error occurs.

Precedence rules for nested programs
Special precedence rules are followed when programs are contained within other programs.

In applying these rules, only the first qualifying declarative is selected for execution. The order of precedence for selecting a declarative is:

1. A file-specific declarative (that is, a declarative of the form USE AFTER ERROR ON file-name-1) within the program that contains the statement that caused the qualifying condition.
2. A mode-specific declarative (that is, a declarative of the form USE AFTER ERROR ON INPUT) within the program that contains the statement that caused the qualifying condition.
3. A file-specific declarative that specifies the GLOBAL phrase and is within the program directly containing the program that was last examined for a qualifying declarative.
4. A mode-specific declarative that specifies the GLOBAL phrase and is within the program directly containing the program that was last examined for a qualifying condition.

Steps 3 and 4 are repeated until the last examined program is the outermost program, or until a qualifying declarative has been found.

DEBUGGING declarative
Debugging sections are permitted only in the outermost program; they are not valid in nested programs. Debugging sections are never triggered by procedures contained in nested programs.

Debugging sections are not permitted in:
• A method
• A program defined with the recursive attribute
• A program compiled with the THREAD compiler option

The WITH DEBUGGING MODE clause of the SOURCE-COMPUTER paragraph activates all debugging sections and lines that have been compiled into the object code. See Appendix D, “Source language debugging,” on page 587 for additional details.

When the debugging mode is suppressed by not specifying the WITH DEBUGGING MODE clause, all USE FOR DEBUGGING declarative procedures and all debugging lines are inhibited.

Automatic execution of a debugging section is not caused by a statement that appears in a debugging section.
USE FOR DEBUGGING

All debugging statements must be written together in a section immediately after the DECLARATIVES header.

Except for the USE FOR DEBUGGING sentence itself, within the debugging procedure there must be no reference to any nondeclarative procedures.

procedure-name-1

Must not be defined in a debugging session.

Table 64 on page 544 shows, for each valid option, the points during execution when the USE FOR DEBUGGING procedures are executed.

Any given procedure-name can appear in only one USE FOR DEBUGGING sentence, and only once in that sentence. All procedures must appear in the outermost program.

ALL PROCEDURES

procedure-name-1 must not be specified in any USE FOR DEBUGGING sentences. The ALL PROCEDURES phrase can be specified only once in a program. Only the procedures contained in the outermost program will trigger execution of the debugging section.

<table>
<thead>
<tr>
<th>Table 64. Execution of debugging declaratives</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>USE FOR DEBUGGING operand</strong></td>
</tr>
</tbody>
</table>
| procedure-name-1 | Before each execution of the named procedure  
After the execution of an ALTER statement referring to the named procedure |
| ALL PROCEDURES | Before each execution of each nondebugging procedure in the outermost program  
After the execution of each ALTER statement in the outermost program (except ALTER statements in declarative procedures) |
Chapter 23. Compiler directives

A compiler directive is a statement that causes the compiler to take a specific action during compilation.

CALLINTERFACE

The CALLINTERFACE directive specifies the interface convention for CALL and SET statements. The convention specified stays in effect until another CALLINTERFACE directive is encountered in the source.

Format

```
>>>CALLINTERFACE
>>>CALLINT
```

| DLL | Specifies that the interface convention for subsequent CALL statements is a call to a DLL, and that subsequent SET function-pointer and procedure-pointer statements are treated as if the DLL compiler option was in effect. |
| DYNAMIC | Specifies that the interface convention for subsequent CALL literal statements and subsequent SET function-pointer and procedure-pointer statements is dynamic (as if the DYNAM compiler option was in effect). |
| STATIC | Specifies that the interface convention for subsequent CALL statements and subsequent SET function pointer and procedure pointer statements is static (as if the NODLL and NODYNAM compiler options were in effect). |

If the `>>>CALLINTERFACE` directive has no suboptions, the interface convention for subsequent CALL and SET statements is determined by the setting of the DLL and DYNAM compiler options.

The `>>>CALLINTERFACE` directive has no effect on the CANCEL statement.

The `>>>CALLINTERFACE` directive can only appear in the procedure division.

The positions of CALL statements relative to the CALLINTERFACE directive are determined following any processing of COPY and REPLACE statements. For example, CALL statements and CALLINTERFACE directives in COPY text are processed by the rules specified for the CALLINTERFACE directive.

Syntax and general rules

You must specify `>>>CALLINTERFACE` on a line by itself, in Area B.

You cannot specify `>>>CALLINTERFACE` in the following cases:

- Within a COPY or REPLACE statement
- Between the lines of a continued character string
- In the middle of a COBOL statement

The `>>>CALLINTERFACE` specification is limited to the current compilation unit.

The REPLACE statement and REPLACING phrase of the COPY statement do not affect the CALLINTERFACE directive.
Precedence of CALLINTERFACE directive versus DLL and DYNAM compiler options

If you specify both the CALLINTERFACE directive (with suboptions) and the DLL or the DYNAM compiler option, the directive overrides the compiler option in effect and determines the interface to be used for subsequent CALL and SET statements.

If you specify the CALLINTERFACE directive without any suboptions, the DLL or the DYNAM compiler option will determine the interface to use for subsequent CALL and SET statements.

INLINE

The INLINE directive lets you selectively prevent the compiler from considering procedures eligible for inlining. Specifying >>INLINE OFF prevents the compiler from inlining procedures referenced by PERFORM statements. If the NOINLINE compiler option is in effect, all INLINE directives are ignored. The relative location of an INLINE directive to the definition of a procedure name is important for determining the inlining behavior for that procedure.

Format

```
    >>INLINE ON
    >>INLINE OFF
```

ON
- Specifies that the compiler determines if the procedures within the scope of the directive are inlined in a specific PERFORM statement when OPTIMIZE(1) or OPTIMIZE(2) is in effect.

OFF
- Specifies that procedures within the scope of the directive will not be inlined when referenced by PERFORM statements, no matter which optimization level setting is in effect.

Note: The word *inlining* here implies that the compiler might choose to replace the PERFORM of a procedure (paragraph or section) that is performed more than once with a copy of that procedure’s code. By inserting the procedure code at the location of the PERFORM, the compiler saves the overhead of branching logic to and from the procedure. Note that if a procedure is only performed once, the compiler might move the code for that procedure to the PERFORM site, even with >>INLINE OFF.

Syntax and general rules

You must specify >>INLINE ON or >>INLINE OFF on a line by itself, in either Area A or Area B.

You cannot specify >>INLINE ON or >>INLINE OFF in the following cases:
- Within a COPY or REPLACE statement
- Between the lines of a continued character string
- In the middle of a COBOL statement

The >>INLINE ON or >>INLINE OFF specification is limited to the current compilation unit.

Note: The INLINE directive can appear multiple times in the program.

Inlining eligibility of procedures for PERFORM statements

A procedure is “in the scope of” a particular INLINE directive when:
- This INLINE directive appears before the definition of the procedure name
  and
- There are no other intervening INLINE directives between the definition of the procedure name and this particular INLINE directive
A procedure is eligible for inlining for statements of the form "PERFORM procedure-name-1 [THROUGH procedure-name-2]", if and only if:

- The procedure is in the scope of an INLINE ON directive
  
  or

- The INLINE compiler option is in effect and the procedure is not in the scope of an INLINE OFF directive

For a section that contains paragraphs, it is possible that this section is under the scope of one INLINE directive while some of its paragraphs are under the scope of another INLINE directive.

**Example**

Following is an example of the effect of the inlining directive on a section that is comprised of one or more paragraphs:

```cobol
>>INLINE ON
  MY-SUBROUTINE SECTION.
  MY-PARAGRAPH-ONE.
  .
>>INLINE OFF
  MY-PARAGRAPH-TWO.
  .
  MY-PARAGRAPH-THREE.
  .
EXIT.
```

**Notes:**

1. Procedure MY-SUBROUTINE will be eligible for inlining in any statements that PERFORM it.
2. Procedure MY-PARAGRAPH-ONE will be eligible for inlining in any statements that PERFORM it.
3. Procedure MY-PARAGRAPH-TWO will not be eligible for inlining in any statements that PERFORM it.
4. Procedure MY-PARAGRAPH-THREE will not be eligible for inlining in any statements that PERFORM it.
5. Procedures MY-PARAGRAPH-ONE through MY-PARAGRAPH-THREE will be eligible for inlining in any statements that PERFORM "MY-PARAGRAPH-ONE THRU MY-PARAGRAPH-THREE", because MY-PARAGRAPH-ONE is eligible for inlining in a PERFORM statement.

**Related references**

INLINE compiler option *(Enterprise COBOL Programming Guide)*

---

**Conditional compilation**

Conditional compilation provides a way of including or omitting selected lines of source code depending on the values of literals specified by the DEFINE directive. In this way, you can create multiple variants of the same program without the need to maintain separate source streams.

The compiler directives that are used for conditional compilation are the DEFINE directive, the EVALUATE directive, and the IF directive. The DEFINE directive is used to define compilation variables that are referenced in the EVALUATE and IF directives to select lines of source code that are to be included or omitted in a compilation group.

Conditional compilation directives are processed according to the following rules:

- Within a source file, if a conditional compilation directive appears before a COPY or REPLACE statement, it is processed before the COPY or REPLACE statement is processed. This means that conditional compilation directives may be used to exclude COPY and REPLACE statements from a program.
• Conditional compilation directives are not affected by substitutions made as the result of REPLACE statements or the REPLACING phrase of COPY statements.
• Conditional compilation directives may appear in copybooks.

**Note:** Conditional compilation directives can be included in a file that contains the BASIS statement, but in that file, conditional compilation directives do not control the inclusion or exclusion of source from that file. Instead, the conditional compilation directives will be processed like any other source lines in the BASIS file that are not INSERT or DELETE statements, and those directives will be passed through to the source that is being assembled to be processed later during the library phase.

**Related references**
“DEFINE” on page 548
“EVALUATE” on page 549
“IF” on page 551

Example: conditional compilation output (Enterprise COBOL Programming Guide)

**DEFINE**
The DEFINE directive defines or undefines a compilation variable. The compilation variables can be used within any of the conditional compilation directives (DEFINE, EVALUATE, and IF). The compilation variable is treated as a symbolic reference to the literal value it currently represents.

**Format**
```
>>DEFINE  compilation-variable-name-1
          AS
          arith-expr-1
          literal-1
          PARAMETER
          OVERRIDE
          OFF
```

**>>DEFINE**
Must begin on a new line in area A or B and must be specified entirely on that line.

**compilation-variable-name-1**
Must not be the same as a conditional compiler directive keyword and must not be one of the predefined compilation variable names.

If a DEFINE directive does not specify the OFF or the OVERRIDE phrase, then one of the following conditions must be true:
• **compilation-variable-name-1** was not declared previously within the same compilation group.
• The previous DEFINE directive referring to **compilation-variable-name-1** must have been specified with the OFF phrase.
• The previous DEFINE directive referring to **compilation-variable-name-1** must have specified the same value.

**literal-1**
Must be one of the following items:
• An alphanumeric literal, which can be specified as a regular alphanumeric literal (‘abcd’) or as a hex literal (‘xF1F2F3’). National literals, DBCS literals, and null-terminated alphanumeric literals (Z literals) are not supported.
• An integer literal.
• A boolean literal (only B‘0’ and B‘1’ are supported).
**arith-expr-1**

Must be formed in accordance with the arithmetic expression rules as described in “Compile-time arithmetic expressions” on page 555.

**General rules**

- DEFINE directives that appear in code that is omitted as the result of other conditional compilation directives are not processed.
- In the text that follows a DEFINE directive that defines `compilation-variable-name-1` and does not include the OFF phrase, `compilation-variable-name-1` can be used in a conditional compilation directive wherever a literal of the category associated with the name is permitted, including in a defined condition and a boolean condition.
- In the text that follows a DEFINE directive specifying `compilation-variable-name-1` with the OFF phrase, `compilation-variable-name-1` can be used only in a defined condition, unless `compilation-variable-name-1` is redefined in a subsequent DEFINE directive without the OFF phrase.
- If the OVERRIDE phrase is specified, `compilation-variable-name-1` is unconditionally set to reference the value of the specified operand.
- If the AS PARAMETER phrase is specified, the value referenced by `compilation-variable-name-1` is obtained from a DEFINE option for `compilation-variable-name-1`, if such an option exists. If there is no DEFINE option for `compilation-variable-name-1`, `compilation-variable-name-1` is not defined.
- If `arith-expr-1` is specified, `arith-expr-1` is evaluated according to “Compile-time arithmetic expressions” on page 555, and `compilation-variable-name-1` references the resultant value.
- If literal-1 is specified, `compilation-variable-name-1` references `literal-1`.

**Related references**

“Defined conditions” on page 554
“Predefined compilation variables” on page 555
DEFINE (Enterprise COBOL Programming Guide)

**EVALUATE**

The EVALUATE directive provides a multi-branch method of choosing the source lines to include in a compilation group.
For descriptive purposes, in this topic:

- **operand-1** refers to **literal-1** or **arith-expr-1** in format 1, and to the TRUE keyword in format 2.
- **operand-2** refers to **literal-2** or **arith-expr-2** in format 1, and to **constant-conditional-expression-1** in format 2.
- **operand-3** refers to **literal-3** or **arith-expr-3** in format 1.

All formats:

**>>EVALUATE, >>WHEN, >>WHEN OTHER, >>END-EVALUATE**

- Must begin on a new line in area A or B and must be specified entirely on that line.
- **text-1**, **text-2**
  - Must begin on a new line and may consist of multiple lines.
  - May be any kind of source lines, including compiler directives. **text-1** or **text-2** may also include COPY statements.
  - The phrases of a given EVALUATE directive must be specified all in the same library text or all in source text. For purposes of this rule, **text-1** and **text-2** are not considered phrases of the EVALUATE directive. A nested EVALUATE directive specified in **text-1** or **text-2** is considered a new EVALUATE directive.

Format 1:

**>>EVALUATE**

- All operands of one EVALUATE directive must be of the same category. For this rule, an arithmetic expression is of category numeric.

**literal-1, arith-expr-1**

- Selection subjects.

**literal-2, literal-3, arith-expr-2, arith-expr-3**

- Selection objects.

**THROUGH, THRU**

- The words THROUGH and THRU are equivalent. If the THROUGH phrase is specified, all selection subjects and selection objects must be of category numeric.

**arith-expr-1, arith-expr-2, arith-expr-3**

- Must be formed in accordance with the arithmetic expression rules as described in “Compile-time arithmetic expressions” on page 555.

Format 2:

**constant-conditional-expression-1**

- Must be formed in accordance with the constant conditional expression rules as described in “Constant conditional expressions” on page 553.
General rules

All Formats:

• `text-1` and `text-2` are not part of the EVALUATE compiler directive line. `text-1` and `text-2` that are in the first true branch of the EVALUATE statement are subject to the matching and replacing rules of the COPY statement and REPLACE statement.

• If the END-EVALUATE phrase is reached without any WHEN phrase evaluating to TRUE, or without encountering a WHEN OTHER phrase, all lines of `text-1` associated with all WHEN phrases are omitted from the resultant text.

Format 1:

• The selection subject is compared against the values specified in each WHEN phrase in turn as follows:
  – If the THROUGH phrase is not specified, a TRUE result is returned if the selection subject is equal to `operand-2`.
  – If the THROUGH phrase is specified, a TRUE result is returned if the selection subject is greater than or equal to `operand-2` and less than or equal to `operand-3`.

• If a WHEN phrase evaluates to TRUE, all the lines of `text-1` associated with that WHEN phrase are included in the resultant text. All lines of `text-1` associated with other WHEN phrases in that EVALUATE directive and all lines of `text-2` associated with a WHEN OTHER phrase are omitted from the resultant text.

• If no WHEN phrase evaluates to TRUE, all lines of `text-2` associated with the WHEN OTHER phrase, if specified, are included in the resultant text. All lines of `text-1` associated with other WHEN phrases are omitted from the resultant text.

• If `literal-1` is an alphanumeric literal, a character-by-character comparison for equality based on the binary value of each character’s encoding is used. If the literals are of unequal length, they are not equal.

Format 2:

• For each WHEN phrase in turn, the `constant-conditional-expression-1` is evaluated in accordance with the rules in “Constant conditional expressions” on page 553.

• If a WHEN phrase evaluates to TRUE, all lines of `text-1` associated with that WHEN phrase are included in the resultant text. All lines of `text-1` associated with other WHEN phrases of that EVALUATE directive and all lines of `text-2` associated with a WHEN OTHER phrase are omitted from the resultant text.

• If no WHEN phrase evaluates to TRUE, all lines of `text-2` associated with the WHEN OTHER phrase, if specified, are included in the resultant text. All lines of `text-1` associated with other WHEN phrases are omitted from the resultant text.

Related references

“COPY statement” on page 524
“REPLACE statement” on page 536

IF

The IF directive provides for a one-way or two-way conditional compilation.
**IF, ELSE, END-IF**

Must begin on a new line in area A or B and must be specified entirely on that line.

**constant-conditional-expression-1**

Must be formed in accordance with the constant conditional expression rules as described in “Constant conditional expressions” on page 553.

**text-1, text-2**

Must begin on a new line and may consist of multiple lines.

May be any kind of source lines, including compiler directives. *text-1* or *text-2* may also include COPY statements.

The phrases of a given IF directive must be specified all in the same library text or all in source text. For purposes of this rule, *text-1* and *text-2* are not considered phrases of the IF directive. A nested IF directive specified in *text-1* or in *text-2* is considered a new IF directive.

**General Rules**

- *text-1* and *text-2* are not part of the IF compiler directive line. The text in the IF directive (either *text-1* or *text-2*) is subject to the matching and replacing rules of the COPY and REPLACE statements.
- If *constant-conditional-expression-1* evaluates to TRUE, all lines of *text-1* are included in the resultant text and all lines of *text-2* are omitted from the resultant text.
- If *constant-conditional-expression-1* evaluates to FALSE, all lines of *text-2* are included in the resultant text and all lines of *text-1* are omitted from the resultant text.

**Related references**

“COPY statement” on page 524
“REPLACE statement” on page 536

**Examples of conditional compilation**

**Example 1: use predefined compilation variables to enable a program to be used in both CICS and BATCH:**

Suppose that the CICS compiler option is in effect and the compiler works with the integrated CICS translator:

```plaintext
>>IF IGY-CICS
   EXEC CICS READ FILE-1... *> Read a record in CICS
>>ELSE
   READ FILE-2              *> Read a record in BATCH
>>END-IF
```

**Example 2: import numeric variable value from outside the source and test it**

Suppose that DEFINE(VAR1=10) is in effect:

```plaintext
>>DEFINE VAR1 AS PARAMETER
   . . .
>>DEFINE VAR2 AS VAR1 + 2
   . . .
>>IF VAR2 < 12
   compute x = x + 1 *> this code should NOT be included
>>ELSE
   compute x = x - 1 *> this code should be included
>>END-IF
```

**Example 3: use the format 1 EVALUATE directive with numeric compilation variables**

```plaintext
>>DEFINE VAR1 AS 6
>>DEFINE VAR2 AS 1
   . . .
>>EVALUATE VAR1
   . . .
```
Example 4: use the format 2 EVALUATE directive with alphanumeric compilation variables

```plaintext
>>DEFINE VAR1 AS 'MOO'
>>EVALUATE TRUE
>>WHEN VAR2 IS DEFINED
  compute x = x + 1 *> this code should NOT be included
>>WHEN VAR1 IS EQUAL TO 'GOO' OR VAR1 IS EQUAL TO 'MOO'
  compute x = x - 1 *> this code should be included
>>END-EVALUATE
```

Example 5: use OVERRIDE and OFF in the DEFINE directive

```plaintext
>>DEFINE VAR AS 12
>>DEFINE VAR OFF
>>IF VAR IS DEFINED
  compute x = x + 1 *> this code should NOT be included
>>ELSE
  compute x = x - 1 *> this code should be included
>>END-IF
>>DEFINE VAR AS 16
>>DEFINE VAR AS VAR - 2 OVERRIDE
>>IF VAR IS EQUAL TO 16
  compute x = x + 1 *> this code should NOT be included
>>ELSE
  compute x = x - 1 *> this code should be included
>>END-IF
```

Example 6: general use of boolean literals and compilation variables

```plaintext
>>DEFINE B1 B'1' *> B1 is category boolean
>>DEFINE B2 B'0' *> B2 is category boolean
>>IF B1 AND B2
  display "Both B1 and B2 are true" *> not included
>>ELSE
  display "Only B1 is true" *> included
  >>IF B1
    display "Only B1 is true" *> included
  >>ELSE
    display "Only B2 is true" *> not included
    >>IF B2
      display "Only B2 is true" *> not included
    >>ELSE
      display "Neither B1 nor B2 is true" *> not included
  >>END-IF
>>END-IF
>>END-IF
```

Constant conditional expressions

A constant conditional expression is an expression that is specified in conditional compilation directives and evaluated during the processing of those directives to determine the text that is included in the resultant program.

**Note:** In this topic, "literals" also include compilation variables, which means that you can use compilation variables in constant conditional expressions.

A constant conditional expression shall be one of the following items:
• A relation condition in which both operands are literals or arithmetic expressions that contain only literal terms. The condition shall follow the rules for relation conditions, with the following additions:
  – The operands shall be of the same category. An arithmetic expression is of the category numeric.
  – If literals are specified and they are not numeric literals, the relational operator shall be “IS EQUAL TO”, “IS NOT EQUAL TO”, “IS =”, “IS NOT =”, or “IS <>”.

• A defined condition.
• A boolean condition.
• A complex condition formed by combining the above forms of simple conditions into complex conditions by using AND, OR, and NOT. Abbreviated combined relation conditions shall not be specified.

Related references
“Relation conditions” on page 249
“Defined conditions” on page 554
“Abbreviated combined relation conditions” on page 264

Defined conditions
A defined condition expression tests whether a compilation variable is defined.

Format

```
compilation-variable-name-1
```

account-name-1 IS NOT DEFINED

Must not be the same as a conditional compiler directive keyword.

IS DEFINED
A defined condition that uses the IS DEFINED syntax evaluates to TRUE if the compilation-variable-name-1 is defined.

If a defined condition references a compilation variable that was defined via a DEFINE compiler option, but preceding the defined condition in the program there is neither a corresponding DEFINE directive with the AS PARAMETER phrase nor a DEFINE directive without the OFF phrase for the compilation variable, then the defined condition for the compilation variable evaluates to FALSE.

IS NOT DEFINED
A defined condition that uses the IS NOT DEFINED syntax evaluates to TRUE if the compilation-variable-name-1 is not defined.

A compilation variable whose most recent definition is via a DEFINE directive with the OFF phrase is considered to be not defined.

Related references
“Predefined compilation variables” on page 555
DEFINE (Enterprise COBOL Programming Guide)

Boolean conditions
A boolean condition determines whether a boolean literal is true or false.

Format

```
boolean-literal-1
```

account-name-1

Evaluates to true if it is B'1', and evaluates to false if it is B'0'.

The condition NOT boolean-literal-1 evaluates to the reverse truth-value of boolean-literal-1.
 Compile-time arithmetic expressions
You can specify a compile-time arithmetic expression in the DEFINE and EVALUATE directives and as part of a constant conditional expression, such as those found in IF directives or WHEN phrases of the EVALUATE directives.

Note: In this topic, "literals" also include compilation variables, which means that you can use compilation variables in compile-time arithmetic expressions.

A compile-time arithmetic expression follows the usual arithmetic expression rules, with the following exceptions:

• The exponentiation operator shall not be specified.
• All operands shall be integer numeric literals or arithmetic expressions in which all operands are integer numeric literals.
• The expression shall be specified in such a way that a division by zero does not occur.
• Intermediate results are computed according to the rules described in Fixed-point data and intermediate results in the Enterprise COBOL Programming Guide. For that purpose, the integer operands of compile-time arithmetic expressions can be considered fixed-point numbers with 0 decimal digits. The ARITH(COMPAT|EXTEND) option setting is taken into account when deciding how many digits of precision to maintain for intermediate results.

 Related references
“Arithmetic expressions” on page 244
ARITH (Enterprise COBOL Programming Guide)

Predefined compilation variables
There are compilation variables that are defined automatically by the compiler. These compilation variables listed in this topic can be referenced in conditional compilation directives wherever a compilation variable is allowed.

<table>
<thead>
<tr>
<th>Predefined compilation variable name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IGY-ARCH</td>
<td>Indicates the target architecture for which the source code is being compiled.</td>
<td>The value of the ARCH option that was used to compile the program: 8, 9, 10, 11, 12 or 13.</td>
</tr>
<tr>
<td>IGY-CICS</td>
<td>Indicates whether embeddedCICS statements are accepted.</td>
<td>B'1' if the CICS compiler option is in effect; B'0' otherwise.</td>
</tr>
</tbody>
</table>
| IGY-COMPILER-VRM                     | Indicates the version of the compiler. | An integer in the formatVVRRMM, where:
• VV represents the version number.
• RR represents the release number.
• MM represents the modification number. For example, compiler version 6.3.0 has a COMPILER-VRM value of 060300. |
<table>
<thead>
<tr>
<th>Predefined compilation variable name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IGY-DLL</td>
<td>Indicates whether the program is compiled as DLL code.</td>
<td>B'1' if the DLL compiler option is in effect; B'0' otherwise.</td>
</tr>
<tr>
<td>IGY-DYNAM</td>
<td>Indicates whether programs invoked through the CALL literal statement will be loaded or deleted dynamically at run time.</td>
<td>B'1' if the DYNAM compiler option is in effect; B'0' otherwise.</td>
</tr>
<tr>
<td>IGY-LP</td>
<td>Indicates whether a 31-bit or 64-bit program should be generated with the related language features enabled.</td>
<td>The value of the LP option that is used to compile the program: 32 or 64.</td>
</tr>
<tr>
<td>IGY-OPTIMIZE</td>
<td>Indicates the optimization level.</td>
<td>The optimization level that was used to compile the program: 0, 1 or 2.</td>
</tr>
<tr>
<td>IGY-SQL</td>
<td>Indicates whether processing of embedded SQL statements is enabled.</td>
<td>B'1' if the SQL compiler option is in effect; B'0' otherwise.</td>
</tr>
<tr>
<td>IGY-SQLIMS</td>
<td>Indicates whether processing of embedded SQLIMS statements is enabled.</td>
<td>B'1' if the SQLIMS compiler option is in effect; B'0' otherwise.</td>
</tr>
<tr>
<td>IGY-THREAD</td>
<td>Indicates whether the program is compiled with multithread support enabled.</td>
<td>B'1' if the THREAD compiler option is in effect; B'0' otherwise.</td>
</tr>
</tbody>
</table>

1. Old variables without the IGY- prefix are tolerated. It is suggested that the IGY- prefix be used when you define the compilation variables.

Related references
“DEFINE” on page 548
DEFINE (Enterprise COBOL Programming Guide)
Appendix A. IBM extensions

IBM extensions are features, syntax rules, or behaviors defined by IBM rather than by the COBOL standards.

The COBOL standards are listed in Appendix H, “Industry specifications,” on page 613.

Table 66 on page 557 lists IBM extensions with a brief description. Standard behavior is shown in brackets, [], when the standard behavior is not obvious. Extensions are described in more detail throughout this document, but they are not further identified as extensions.

Many IBM extensions are distinguished from standard language by their syntax. For others, you use compiler options to choose between standard and extension behavior. Generally, the related compiler options are noted in the detailed rules. For information about compiler options, see Option settings for 85 COBOL Standard conformance in the Enterprise COBOL Programming Guide.

If an item is listed as an extension, all related rules are also extensions. For example, USAGE DISPLAY-1 for DBCS characters is listed as an extension; its many uses in statements and clauses are also extensions, but are not listed separately.

<table>
<thead>
<tr>
<th>Language area</th>
<th>Extension elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>COBOL words</td>
<td>User-defined words written in DBCS characters</td>
</tr>
<tr>
<td></td>
<td>Computer-name written in DBCS characters</td>
</tr>
<tr>
<td></td>
<td>Class-names (for object orientation)</td>
</tr>
<tr>
<td></td>
<td>Method-names</td>
</tr>
<tr>
<td></td>
<td>User-defined words can include an underscore, but not as the first character.</td>
</tr>
<tr>
<td>National character support (Unicode support)</td>
<td>Support for UTF-16 with USAGE NATIONAL</td>
</tr>
<tr>
<td></td>
<td>Allowance of UTF-8 with USAGE DISPLAY</td>
</tr>
<tr>
<td></td>
<td>Usage NATIONAL for data categories national, national-edited, numeric, numeric-edited, external decimal, and external floating-point</td>
</tr>
<tr>
<td></td>
<td>GROUP-USAGE clause with the NATIONAL phrase</td>
</tr>
<tr>
<td></td>
<td>National literals (basic and hexadecimal)</td>
</tr>
<tr>
<td></td>
<td>National character value for figurative constants SPACE, ZERO, QUOTE, HIGH-VALUES, LOW-VALUES, ALL literal</td>
</tr>
<tr>
<td></td>
<td>Intrinsic functions for data conversion:</td>
</tr>
<tr>
<td></td>
<td>• DISPLAY-OF</td>
</tr>
<tr>
<td></td>
<td>• NATIONAL-OF</td>
</tr>
<tr>
<td></td>
<td>Extended case mapping with UPPER-CASE and LOWER-CASE functions</td>
</tr>
</tbody>
</table>
### Table 66. IBM extension language elements (continued)

<table>
<thead>
<tr>
<th>Language area</th>
<th>Extension elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implicit items</td>
<td>Special object references:</td>
</tr>
<tr>
<td></td>
<td>• SELF</td>
</tr>
<tr>
<td></td>
<td>• SUPER</td>
</tr>
<tr>
<td></td>
<td>Special registers:</td>
</tr>
<tr>
<td></td>
<td>• ADDRESS OF</td>
</tr>
<tr>
<td></td>
<td>• JNIENVPTR</td>
</tr>
<tr>
<td></td>
<td>• JSON-CODE</td>
</tr>
<tr>
<td></td>
<td>• JSON-STATUS</td>
</tr>
<tr>
<td></td>
<td>• LENGTH OF</td>
</tr>
<tr>
<td></td>
<td>• RETURN-CODE</td>
</tr>
<tr>
<td></td>
<td>• SHIFT-IN</td>
</tr>
<tr>
<td></td>
<td>• SHIFT-OUT</td>
</tr>
<tr>
<td></td>
<td>• SORT-CONTROL</td>
</tr>
<tr>
<td></td>
<td>• SORT-CORE-SIZE</td>
</tr>
<tr>
<td></td>
<td>• SORT-FILE-SIZE</td>
</tr>
<tr>
<td></td>
<td>• SORT-MESSAGE</td>
</tr>
<tr>
<td></td>
<td>• SORT-MODE-SIZE</td>
</tr>
<tr>
<td></td>
<td>• SORT-RETURN</td>
</tr>
<tr>
<td></td>
<td>• TALLY</td>
</tr>
<tr>
<td></td>
<td>• WHEN-compiled</td>
</tr>
<tr>
<td></td>
<td>• XML-CODE</td>
</tr>
<tr>
<td></td>
<td>• XML-EVENT</td>
</tr>
<tr>
<td></td>
<td>• XML-INFORMATION</td>
</tr>
<tr>
<td></td>
<td>• XML-NAMESPACE</td>
</tr>
<tr>
<td></td>
<td>• XML-NAMESPACE-PREFIX</td>
</tr>
<tr>
<td></td>
<td>• XML-NNAMESPACE</td>
</tr>
<tr>
<td></td>
<td>• XML-NNAMESPACE-PREFIX</td>
</tr>
<tr>
<td></td>
<td>• XML-NTEXT</td>
</tr>
<tr>
<td></td>
<td>• XML-TEXT</td>
</tr>
<tr>
<td>Figurative constants</td>
<td>Selection of apostrophe (‘’) as the value of the figurative constant QUOTE</td>
</tr>
<tr>
<td></td>
<td>NULL and NULLS for pointers and object references</td>
</tr>
<tr>
<td>Language area</td>
<td>Extension elements</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| Literals               | Use of apostrophe (‘) as an alternative to the quotation mark (" ) in opening and closing delimiters  
Mixed single-byte and double-byte characters in alphanumeric literals (mixed literals)  
Hexadecimal notation for alphanumeric literals, defined by opening delimiters X" and X’  
Null-terminated alphanumeric literals, defined by opening delimiters Z" and Z’  
DBCS literals, defined by opening delimiters N", N’, G", and G’. N” and N’ are defined as DBCS when the NSYMBOL(DBCS) compiler option is in effect.  
Consecutive alphanumeric literals (coding two consecutive alphanumeric literals by ending the first literal in column 72 of a continued line and starting the next literal with a quotation mark in the continuation line)  
National literals N", N’, NX", NX’ for storing literal content as national characters. N” and N’ are defined as national when the NSYMBOL(NATIONAL) compiler option is in effect.  
19- to 31-digit fixed-point numeric literals. [The 85 COBOL Standard specifies a maximum of 18 digits.]  
Floating-point numeric literals |
| Comments               | Comment lines before the IDENTIFICATION DIVISION header  
Comment lines and comment entries containing multibyte characters  
Inline comments                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| End markers            | The following end markers:  
• END CLASS  
• END FACTORY  
• END METHOD  
• END OBJECT                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| Indexing and subscripting | Referencing a table with an index-name defined for a different table  
Specifying a positive signed integer literal following the operator + or - in relative subscripting                                                                                                                                                                                                                                                                                                                                                     |
Table 66. **IBM extension language elements** (continued)

<table>
<thead>
<tr>
<th>Language area</th>
<th>Extension elements</th>
</tr>
</thead>
</table>
| IDENTIFICATION DIVISION for programs | Abbreviation ID for IDENTIFICATION RECURSIVE clause  
An optional separator period following PROGRAM-ID, AUTHOR, INSTALLATION, DATE-WRITTEN, and SECURITY paragraph headers. [The 85 COBOL Standard requires a period following each of these paragraph headers.]  
An optional separator period following program-name in the PROGRAM-ID paragraph. [The 85 COBOL Standard requires a period following program-name.]  
An alphanumeric literal for program-name in the PROGRAM-ID paragraph; characters $, #, and @ in the name of the outermost program, and the underscore can be the first character; program-name up to 160 characters in length. [The 85 COBOL Standard requires that program-name be specified as a user-defined word.] |
| End markers                          | Program-name in a literal. [The 85 COBOL Standard requires that program-name be specified as a user-defined word.]                                                                                                      |
| Object-oriented structure            | In a class definition:  
• CLASS-ID paragraph  
• INHERITS clause  
• END CLASS marker  
In a method definition:  
• METHOD-ID paragraph  
• EXIT METHOD statement  
• END METHOD marker                                                                                                                                             |
<p>| Configuration section                | Repository paragraph                                                                                                                                                                                             |</p>
<table>
<thead>
<tr>
<th>Language area</th>
<th>Extension elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL-NAMES paragraph</td>
<td>The optional order of clauses. [The 85 COBOL Standard requires that the clauses be coded in the order presented in the syntax diagram.]</td>
</tr>
<tr>
<td></td>
<td>Optionality of a period after the last clause when no clauses are coded. [The 85 COBOL Standard requires a period, even when no clauses are coded.]</td>
</tr>
<tr>
<td></td>
<td>Multiple CURRENCY SIGN clauses. [The 85 COBOL Standard allows a single CURRENCY SIGN clause.]</td>
</tr>
<tr>
<td></td>
<td>WITH PICTURE SYMBOL phrase in the CURRENCY SIGN clause</td>
</tr>
<tr>
<td></td>
<td>Multiple-character and mixed-case currency signs in the CURRENCY SIGN clause (when the WITH PICTURE SYMBOL phrase is specified). [The 85 COBOL Standard allows only one character, and it is both the currency sign and the currency picture symbol. The standard currency sign must not be: ]</td>
</tr>
<tr>
<td></td>
<td>• The same character as any standard picture symbol</td>
</tr>
<tr>
<td></td>
<td>• A digit 0-9</td>
</tr>
<tr>
<td></td>
<td>• One of the special characters * + - , ; ( ) &quot; = /</td>
</tr>
<tr>
<td></td>
<td>• A space</td>
</tr>
<tr>
<td></td>
<td>Use of lower-case alphabetic characters as a currency sign. [The 85 COBOL Standard allows only uppercase characters.]</td>
</tr>
<tr>
<td>Language area</td>
<td>Extension elements</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------</td>
</tr>
</tbody>
</table>
| INPUT-OUTPUT SECTION, FILE-CONTROL paragraph | Optionality of "FILE-CONTROL." when the INPUT-OUTPUT SECTION is specified, no file-control-paragraph is specified, and there are no files defined in the compilation unit. [The 85 COBOL Standard requires that "FILE-CONTROL." be coded if "INPUT-OUTPUT SECTION." is coded.]
| | Optionality of the file-control-paragraph when the "FILE CONTROL." syntax is specified and there are no files defined in the compilation unit. [The 85 COBOL Standard requires that a file-control-paragraph be coded if "INPUT-OUTPUT SECTION." is coded.]
| | PASSWORD clause
| | The second data-name in the FILE STATUS clause
| | Optionality of RECORD in the ALTERNATE RECORD KEY clause. [The 85 COBOL Standard requires the word RECORD.]
| | A numeric, numeric-edited, alphanumeric-edited, alphabetic, internal floating-point, external floating-point, national, national-edited, or DBCS primary or alternate record key data item. [The 85 COBOL Standard requires that the key be alphanumeric.]
| | A primary or alternate record key defined outside the minimum record size for indexed files containing variable-length records. [The 85 COBOL Standard requires that the primary and alternate record keys be within the minimum record size.]
| | A numeric data item of usage DISPLAY or NATIONAL in the FILE STATUS clause. [The 85 COBOL Standard requires an alphanumeric file status data item.]
| | The ORGANIZATION IS LINE SEQUENTIAL clause and line-sequential file control format
| | National literal in the PADDING CHARACTER clause |

| INPUT-OUTPUT SECTION, I-O-CONTROL paragraph | APPLY WRITE-ONLY clause
| | Specifying only one file-name in the SAME clause in the sequential, indexed, and sort-merge formats of the I-O-control entry. [The 85 COBOL Standard requires at least two file-names.]
| | Optionality of the keyword ON in the RERUN clause. [The 85 COBOL Standard requires that ON be coded.]
| | The line-sequential format I-O-control entry
<p>| | The RERUN clause in the sort-merge I-O-control entry |</p>
<table>
<thead>
<tr>
<th>Language area</th>
<th>Extension elements</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DATA DIVISION</strong></td>
<td><strong>LOCAL-STOREAGE SECTION</strong></td>
</tr>
<tr>
<td></td>
<td>The <code>GLOBAL</code> clause in the <code>LINKAGE SECTION</code></td>
</tr>
<tr>
<td></td>
<td>Specifying level numbers that are lower than other level numbers at the same hierarchical level in a data description entry. [The 85 COBOL Standard requires that all elementary or group items at the same level in the hierarchy be assigned identical level numbers.]</td>
</tr>
<tr>
<td></td>
<td>Data categories internal floating-point, external floating-point, DBCS, national, and national-edited.</td>
</tr>
<tr>
<td></td>
<td>Data category numeric with usage NATIONAL.</td>
</tr>
<tr>
<td></td>
<td>Data category numeric-edited with usage NATIONAL.</td>
</tr>
<tr>
<td><strong>FILE SECTION</strong></td>
<td><code>data-name</code> in the <code>LABEL RECORDS</code> clause, for specifying user labels</td>
</tr>
<tr>
<td></td>
<td><code>RECORDING MODE</code> clause</td>
</tr>
<tr>
<td></td>
<td>Line-sequential format file description entry</td>
</tr>
<tr>
<td><strong>Sort/merge file description entry</strong></td>
<td>The following clauses:</td>
</tr>
<tr>
<td></td>
<td>• <code>BLOCK CONTAINS</code></td>
</tr>
<tr>
<td></td>
<td>• <code>LABEL RECORDS</code></td>
</tr>
<tr>
<td></td>
<td>• <code>VALUE OF</code></td>
</tr>
<tr>
<td></td>
<td>• <code>LINAGE</code></td>
</tr>
<tr>
<td></td>
<td>• <code>CODE-SET</code></td>
</tr>
<tr>
<td></td>
<td>• <code>WITH FOOTING</code></td>
</tr>
<tr>
<td></td>
<td>• <code>LINES AT</code></td>
</tr>
<tr>
<td><strong>BLOCK CONTAINS clause</strong></td>
<td><code>BLOCK CONTAINS 0</code> for QSAM files. [The 85 COBOL Standard requires that at least 1 CHARACTER or RECORD be specified in the <code>BLOCK CONTAINS</code> clause.]</td>
</tr>
<tr>
<td><strong>VALUE OF clause</strong></td>
<td>The lack of <code>VALUE</code> clause effect on execution when specified under an SD</td>
</tr>
<tr>
<td><strong>DATA RECORDS clause</strong></td>
<td>Optionality of an 01 record description entry for a specified <code>data-name</code>. [The 85 COBOL Standard requires that an 01 record with the same <code>data-name</code> be specified.]</td>
</tr>
<tr>
<td><strong>LINAGE clause</strong></td>
<td>Specifying LINAGE for files opened in EXTEND mode</td>
</tr>
<tr>
<td><strong>BLANK WHEN ZERO clause</strong></td>
<td>Alternative spellings ZEROS and ZEROES for ZERO</td>
</tr>
<tr>
<td><strong>GLOBAL clause</strong></td>
<td>Specifying GLOBAL in the <code>LINKAGE SECTION</code></td>
</tr>
<tr>
<td><strong>INDEXED BY phrase</strong></td>
<td>Nonunique unreferenced index names</td>
</tr>
<tr>
<td>Language area</td>
<td>Extension elements</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>OCCURS clause</td>
<td>Omission of &quot;integer-1 TO&quot; for variable-length tables</td>
</tr>
<tr>
<td></td>
<td>Complex OCCURS DEPENDING ON. [The 85 COBOL Standard requires that an entry containing OCCURS DEPENDING ON be followed only by subordinate entries, and that no entry containing OCCURS DEPENDING ON be subordinate to an entry containing OCCURS DEPENDING ON.]</td>
</tr>
<tr>
<td></td>
<td>Implicit qualification of a key specified without qualifiers when the key name is not unique</td>
</tr>
<tr>
<td></td>
<td>Reference to a table through indexing when no INDEXED BY phrase is specified</td>
</tr>
<tr>
<td></td>
<td>Keys of usages COMPUTATIONAL-1, COMPUTATIONAL-2, COMPUTATIONAL-3, COMPUTATIONAL-4, and COMPUTATIONAL-5 in the ASCENDING/DESCENDING KEY phrase</td>
</tr>
<tr>
<td></td>
<td>Acceptance of nonunique index-names that are not referenced</td>
</tr>
<tr>
<td></td>
<td>Unbounded tables and groups</td>
</tr>
<tr>
<td>PICTURE clause</td>
<td>A picture character-string containing 31 to 50 characters. [The 85 COBOL Standard allows a maximum of 30 characters.]</td>
</tr>
<tr>
<td></td>
<td>Picture symbols G and N</td>
</tr>
<tr>
<td></td>
<td>Picture symbol E and the external floating-point picture format</td>
</tr>
<tr>
<td></td>
<td>Coding a trailing comma insertion character or trailing period insertion character immediately followed by a separator comma or separator semicolon in a PICTURE clause that is not the last clause of a data description entry. [The 85 COBOL Standard requires that a PICTURE clause containing a picture ending with a comma or period be the last clause in the entry and that it be followed immediately by a separator period.]</td>
</tr>
<tr>
<td></td>
<td>Selecting a currency sign and currency symbol with the CURRENCY compiler option</td>
</tr>
<tr>
<td></td>
<td>Case-sensitive currency symbols</td>
</tr>
<tr>
<td></td>
<td>The maximum of 31 digits for numeric items of usages DISPLAY and PACKED-DECIMAL and for numeric-edited items of USAGE DISPLAY</td>
</tr>
<tr>
<td></td>
<td>The effect of the TRUNC compiler option on the value of data items described with a usage of BINARY, COMPUTATIONAL, or COMPUTATIONAL-4</td>
</tr>
<tr>
<td>REDEFINES clause</td>
<td>Specifying REDEFINES of a redefined data item</td>
</tr>
<tr>
<td></td>
<td>At a subordinate level, specifying a redefining data item that has a size greater than the size of the redefined data item</td>
</tr>
<tr>
<td>SYNCHRONIZED clause</td>
<td>Specifying SYNCHRONIZED for a level 01 entry</td>
</tr>
<tr>
<td>Language area</td>
<td>Extension elements</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>USAGE clause</td>
<td>The following phrases:&lt;br&gt;  • NATIVE  &lt;br&gt;  • COMP-1 and COMPUTATIONAL-1  &lt;br&gt;  • COMP-2 and COMPUTATIONAL-2  &lt;br&gt;  • COMP-3 and COMPUTATIONAL-3  &lt;br&gt;  • COMP-4 and COMPUTATIONAL-4  &lt;br&gt;  • COMP-5 and COMPUTATIONAL-5  &lt;br&gt;  • DISPLAY-1  &lt;br&gt;  • OBJECT REFERENCE  &lt;br&gt;  • NATIONAL  &lt;br&gt;  • POINTER  &lt;br&gt;  • PROCEDURE-POINTER  &lt;br&gt;  • FUNCTION-POINTER  &lt;br&gt; Use of the SYNCHRONIZED clause for items of usage INDEX</td>
</tr>
<tr>
<td>VALUE clause for condition-name entries</td>
<td>A VALUE clause in file and LINKAGE SECTION other than in condition-name entries  &lt;br&gt; A VALUE clause for a condition-name entry on a group that has usages other than DISPLAY  &lt;br&gt; VALUE IS NULL and VALUE IS NULLS</td>
</tr>
<tr>
<td>VOLATILE clause</td>
<td>A VOLATILE clause in a format 1 data description entry</td>
</tr>
<tr>
<td>PROCEDURE DIVISION</td>
<td>Omission of a section-name  &lt;br&gt; Omission of a paragraph-name when a section-name is omitted  &lt;br&gt; A method, factory, or object procedure division  &lt;br&gt; Referencing data items in the LINKAGE SECTION without a USING phrase in the PROCEDURE DIVISION header (when those data-names are the operand of an ADDRESS OF phrase or ADDRESS OF special register)  &lt;br&gt; The following statements:&lt;br&gt;  • ENTRY  &lt;br&gt;  • EXIT METHOD  &lt;br&gt;  • GOBACK  &lt;br&gt;  • INVOKE  &lt;br&gt;  • JSON PARSE  &lt;br&gt;  • JSON GENERATE  &lt;br&gt;  • XML PARSE  &lt;br&gt;  • XML GENERATE</td>
</tr>
<tr>
<td>Language area</td>
<td>Extension elements</td>
</tr>
<tr>
<td>----------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PROCEDURE DIVISION header</td>
<td>The BY VALUE phrase</td>
</tr>
<tr>
<td></td>
<td>The RETURNING phrase</td>
</tr>
<tr>
<td></td>
<td>Specifying a data item in the USING phrase when the data item has a REDEFINES clause in its description</td>
</tr>
<tr>
<td></td>
<td>Specifying multiple instances of a given data item in the USING phrase</td>
</tr>
<tr>
<td></td>
<td>The formats for method, factory, and object definitions</td>
</tr>
<tr>
<td>Declarative Procedures</td>
<td>Performing a nondeclarative procedure from a declarative procedure</td>
</tr>
<tr>
<td></td>
<td>Referencing a declarative procedure or nondeclarative procedure in a GO TO statement from a declarative procedure. [The 85 COBOL Standard specifies that a declarative procedure must not reference a nondeclarative procedure. A reference to a declarative procedure from either another declarative procedure or a nondeclarative procedure is allowed only with a PERFORM statement.]</td>
</tr>
<tr>
<td></td>
<td>Executing an active declarative</td>
</tr>
<tr>
<td>Procedures</td>
<td>Specifying priority-number as a positive signed numeric literal. [The 85 COBOL Standard requires an unsigned integer.]</td>
</tr>
<tr>
<td></td>
<td>Omitting the section-header after the declaratives or when there are no declaratives. [The 85 COBOL Standard requires a section-header following the &quot;DECLARATIVES.&quot; syntax and following the &quot;END DECLARATIVES.&quot; syntax.]</td>
</tr>
<tr>
<td></td>
<td>Omitting an initial paragraph-name if there are no declaratives. [The 85 COBOL Standard requires a paragraph-name in the following circumstances:</td>
</tr>
<tr>
<td></td>
<td>• After the USE statement if there are statements in the declarative procedure</td>
</tr>
<tr>
<td></td>
<td>• Following a section header outside declarative procedures</td>
</tr>
<tr>
<td></td>
<td>• Before any procedural statement if there are no declaratives and the 85 COBOL Standard requires that procedural statements be within a paragraph.]</td>
</tr>
<tr>
<td></td>
<td>Specifying paragraphs that are not contained within a section, even if some paragraphs are so contained. [The 85 COBOL Standard requires that paragraphs be within a section except when there are no declaratives. The 85 COBOL Standard requires that either all paragraphs be in sections or that none be.]</td>
</tr>
<tr>
<td>Conditional expressions</td>
<td>DBCS and KANJI class conditions</td>
</tr>
<tr>
<td></td>
<td>Specifying data items of usage COMPUTATIONAL-3 or usage PACKED-DECIMAL in a NUMERIC class test</td>
</tr>
</tbody>
</table>
### Table 66. *IBM extension language elements* (continued)

<table>
<thead>
<tr>
<th>Language area</th>
<th>Extension elements</th>
</tr>
</thead>
</table>
| Relation condition      | Enclosing an alphanumeric, DBCS, or national literal in parentheses  
                          | The data-pointer format, the procedure-pointer and function-pointer format, and the object-reference format  
                          | Comparison of an index-name with an arithmetic expression  
                          | Use of parentheses within abbreviated combined relation conditions  
                          | **Note:** Enterprise COBOL supports most parenthesis usage as IBM extensions with the following exceptions:  
                          | • Within the scope of an abbreviated combined relation condition, Enterprise COBOL does not support relational operators inside parentheses. For example:  
                          | ![Example](image-url)  
                          | • Incorrect usages of parentheses in relation conditions are not accepted by Enterprise COBOL. For example:  
                          | ![Example](image-url)  
                          |                                                                                                           |
| CORRESPONDING phrase    | Specifying an identifier that is subordinate to a filler item                                                                                                                                                    |
| INVALID KEY phrase      | Omission of both the INVALID KEY phrase and an applicable EXCEPTION/ERROR procedure. [The 85 COBOL Standard requires at least one of them.]                                                                 |
| ACCEPT statement        | The environment-name operand of the FROM phrase  
                          | The DATE YYYYMMDD phrase  
                          | The DAY YYYYDDD phrase                                                                                                                                           |
| ADD statement           | A composite of operands greater than 18 digits                                                                                                                                                                   |
| ALLOCATE statement      | The LOC phrase                                                                                                                                                                                                    |
| CALL statement          | The procedure-pointer and function-pointer operands for identifying the program to be called  
                          | The following phrases and parameters:  
                          | • ADDRESS OF  
                          | • LENGTH OF  
                          | • OMITTED  
                          | • BY VALUE  
                          | • RETURNING  
                          | Specifying a *file-name* as an argument  
                          | Specifying the called program-name in an alphabetic or zoned-decimal data item  
<pre><code>                      | Specifying an argument defined as a subordinate group item. [The 85 COBOL Standard requires that arguments be an elementary data item or a group item defined with level 01.] |
</code></pre>
<table>
<thead>
<tr>
<th>Language area</th>
<th>Extension elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANCEL statement</td>
<td>Specifying the name of the program to be canceled in an alphabetic or zoned-decimal data item</td>
</tr>
<tr>
<td></td>
<td>The effect of the PGMNAME compiler option on the name of the program to be canceled</td>
</tr>
<tr>
<td>CLOSE statement</td>
<td>WITH NO REWIND phrase</td>
</tr>
<tr>
<td></td>
<td>The line-sequential format</td>
</tr>
<tr>
<td>COMPUTE statement</td>
<td>The use of the word EQUAL in place of the equal sign (=)</td>
</tr>
<tr>
<td>DISPLAY statement</td>
<td>The environment-name operand of the UPON phrase</td>
</tr>
<tr>
<td></td>
<td>Displaying signed numeric literals and noninteger numeric literals</td>
</tr>
<tr>
<td>DIVIDE statement</td>
<td>A composite of operands greater than 18 digits</td>
</tr>
<tr>
<td>EXIT statement</td>
<td>Specifying the EXIT statement in a sentence that has statements before or after the EXIT statement or in a paragraph that has other sentences. [The 85 COBOL Standard requires that the EXIT statement be specified in a sentence by itself and that the sentence be the only sentence in the paragraph.]</td>
</tr>
<tr>
<td>EXIT PROGRAM statement</td>
<td>Specifying EXIT PROGRAM before the last statement in a sequence of imperative statements. [The 85 COBOL Standard requires that the EXIT PROGRAM statement be specified as the last statement in a sequence of imperative statements.]</td>
</tr>
<tr>
<td>GO TO statement</td>
<td>Coding the unconditional format before the last statement in a sequence of imperative statements. [The 85 COBOL Standard requires that an unconditional GO TO be coded:</td>
</tr>
<tr>
<td></td>
<td>• Only in a single-statement paragraph if no procedure-name is specified</td>
</tr>
<tr>
<td></td>
<td>• Otherwise, as the last statement of a sentence.</td>
</tr>
<tr>
<td>IF statement</td>
<td>The use of END-IF with the NEXT SENTENCE phrase. [The 85 COBOL Standard disallows use of END-IF with NEXT SENTENCE.]</td>
</tr>
<tr>
<td>INITIALIZE statement</td>
<td>DBCS, EGCS, NATIONAL, and NATIONAL-EDITED in the REPLACING phrase</td>
</tr>
<tr>
<td></td>
<td>Initializing a data item that contains the DEPENDING phrase of the OCCURS clause</td>
</tr>
<tr>
<td>MERGE statement</td>
<td>Specifying file-names in a SAME clause</td>
</tr>
<tr>
<td>MULTIPLY statement</td>
<td>A composite of operands greater than 18 digits</td>
</tr>
<tr>
<td>OPEN statement</td>
<td>The line-sequential format</td>
</tr>
<tr>
<td></td>
<td>Specifying the EXTEND phrase for files that have a LINAGE clause</td>
</tr>
<tr>
<td>PERFORM statement</td>
<td>An empty in-line PERFORM statement</td>
</tr>
<tr>
<td></td>
<td>A common exit for two or more active PERFORMS</td>
</tr>
<tr>
<td>Language area</td>
<td>Extension elements</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| READ statement  | Omission of both the AT END phrase and an applicable declarative procedure  
|                 | Omission of both the INVALID KEY phrase and an applicable declarative procedure  
|                 | Read into an item that is neither an alphanumeric group item nor an elementary alphanumeric item                                                    |
| RETURN statement| Return into an item that is neither an alphanumeric group item nor an elementary alphanumeric item                                               |
| REWRITE statement| Omission of both the INVALID KEY phrase and an applicable declarative procedure  
|                 | Rewriting a record with a different number of character positions than the number of character positions in the record being rewritten          |
| SEARCH statement | Specifying END SEARCH with NEXT SENTENCE  
|                 | Omission of both the NEXT SENTENCE phrase and imperative statements in the binary search format                                                  |
| SET statement    | The data-pointer format  
|                 | The procedure-pointer and function-pointer format  
|                 | The object reference format                                                                                                                     |
| SORT statement   | Specifying GIVING file-names in the SAME clause                                                                                                 |
| START statement  | Omission of both the INVALID KEY phrase and an applicable exception procedure  
|                 | Use of a key of a category other than alphanumeric                                                                                               |
| STOP statement   | Specifying a noninteger fixed-point literal or a signed numeric integer or noninteger fixed-point literal  
|                 | Coding STOP as other than the last statement in a sentence                                                                                     |
| STRING statement | Reference modification of the data item specified in the INTO phrase                                                                             |
| SUBTRACT statement | A composite of operands greater than 18 digits                                                                                                    |
| UNSTRING statement| Reference modification of the sending field                                                                                                       |
| WRITE statement  | INVALID KEY and NOT ON INVALID KEY phrases  
|                 | The line-sequential format  
|                 | For a relative file, writing a different number of character positions than the number of character positions in the record being replaced  
|                 | Specifying both the ADVANCING PAGE and END-OF-PAGE phrases in a single WRITE statement  
|                 | The effect of the ADV compiler option on the length of the record written to a file  
|                 | Using WRITE ADVANCING with stacker selection for a card punch file  
<p>|                 | For a relative or indexed file, omission of both the INVALID KEY phrase and an applicable exception procedure                                  |</p>
<table>
<thead>
<tr>
<th>Language area</th>
<th>Extension elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intrinsic functions</td>
<td>The effect of the INTDATE compiler options on the INTEGER-OF-DAY and INTEGER-OF-DAY functions.</td>
</tr>
<tr>
<td></td>
<td>The following functions:</td>
</tr>
<tr>
<td></td>
<td>• “BIT-OF” on page 473</td>
</tr>
<tr>
<td></td>
<td>• “BIT-TO-CHAR” on page 474</td>
</tr>
<tr>
<td></td>
<td>• “DATE-TO-YYYYMMDD” on page 477</td>
</tr>
<tr>
<td></td>
<td>• “DAY-TO-YYYYDDD” on page 478</td>
</tr>
<tr>
<td></td>
<td>• “DISPLAY-OF” on page 479</td>
</tr>
<tr>
<td></td>
<td>• “HEX-OF” on page 481</td>
</tr>
<tr>
<td></td>
<td>• “HEX-TO-CHAR” on page 482</td>
</tr>
<tr>
<td></td>
<td>• “NATIONAL-OF” on page 490</td>
</tr>
<tr>
<td></td>
<td>• ULENGTH</td>
</tr>
<tr>
<td></td>
<td>• UPOS</td>
</tr>
<tr>
<td></td>
<td>• USUBSTR</td>
</tr>
<tr>
<td></td>
<td>• USUPPLEMENTARY</td>
</tr>
<tr>
<td></td>
<td>• UVALID</td>
</tr>
<tr>
<td></td>
<td>• UWIDTH</td>
</tr>
<tr>
<td></td>
<td>• “YEAR-TO-YYYY” on page 517</td>
</tr>
<tr>
<td>FACTORIAL function</td>
<td>The effect of the ARITH(EXTEND) compiler option on the range of values permitted in the argument.</td>
</tr>
<tr>
<td>LENGTH function</td>
<td>Specifying a pointer, the ADDRESS OF special register, or the LENGTH OF special register as an argument to the function.</td>
</tr>
<tr>
<td>NUMVAL function</td>
<td>The effect of the ARITH(EXTEND) compiler option on the maximum number of digits allowed in the argument.</td>
</tr>
<tr>
<td>NUMVAL-C function</td>
<td>The effect of the ARITH(EXTEND) compiler option on the maximum number of digits allowed in the argument.</td>
</tr>
<tr>
<td>NUMVAL-F function</td>
<td>The effect of the ARITH(EXTEND) compiler option on the maximum number of digits allowed in the argument.</td>
</tr>
<tr>
<td>TEST-NUMVAL function</td>
<td>The effect of the ARITH(EXTEND) compiler option on the maximum number of digits allowed in the argument.</td>
</tr>
<tr>
<td>TEST-NUMVAL-C function</td>
<td>The effect of the ARITH(EXTEND) compiler option on the maximum number of digits allowed in the argument.</td>
</tr>
<tr>
<td>TEST-NUMVAL-F function</td>
<td>The effect of the ARITH(EXTEND) compiler option on the maximum number of digits allowed in the argument.</td>
</tr>
<tr>
<td>Language area</td>
<td>Extension elements</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Compiler-directing statements</td>
<td>The following statements:</td>
</tr>
<tr>
<td></td>
<td>• BASIS</td>
</tr>
<tr>
<td></td>
<td>• CBL(PROCESS)</td>
</tr>
<tr>
<td></td>
<td>• *CONTROL and *CBL</td>
</tr>
<tr>
<td></td>
<td>• DELETE</td>
</tr>
<tr>
<td></td>
<td>• EJECT</td>
</tr>
<tr>
<td></td>
<td>• INSERT</td>
</tr>
<tr>
<td></td>
<td>• READY or RESET TRACE</td>
</tr>
<tr>
<td></td>
<td>• SERVICE LABEL</td>
</tr>
<tr>
<td></td>
<td>• SERVICE RELOAD</td>
</tr>
<tr>
<td></td>
<td>• SKIP1, SKIP2, and SKIP3</td>
</tr>
<tr>
<td></td>
<td>• TITLE</td>
</tr>
<tr>
<td>Compiler directives</td>
<td>CALLINTERFACE directive</td>
</tr>
<tr>
<td>COPY statement</td>
<td>The optionality of the syntax &quot;OF library-name&quot; for specifying a text-name qualifier</td>
</tr>
<tr>
<td></td>
<td>Literals for specifying text-name and library-name</td>
</tr>
<tr>
<td></td>
<td>SUPPRESS phrase</td>
</tr>
<tr>
<td></td>
<td>Nested COPY statements</td>
</tr>
<tr>
<td></td>
<td>Hyphen as the first or last character in the word form of REPLACING operands</td>
</tr>
<tr>
<td></td>
<td>The use of any character (other than a COBOL separator) in the word form of REPLACING operands. [The 85 COBOL Standard accepts only the characters used in formation of user-defined words.]</td>
</tr>
</tbody>
</table>
Appendix B. Compiler limits

Enterprise COBOL has the following limits for programs and class definitions.

Although the COBOL compiler supports addressing various memory areas in a compile unit up to the limits described in this appendix, a complete application, typically consisting of multiple compile units, is still restricted by the amount of private storage available in the address space in which it runs. In other words, an application might run out of storage before reaching the described limits.

You can sum the length fields in the MAP (also requires either OFFSET or LIST option output's PPA4 section for the static memory footprint of a compile unit. For automatic (local-storage) requirements, see Example: DSA memory map (stack storage map) in Enterprise COBOL Programming Guide

In general, the maximum size of tables and of elementary alphanumeric data items in LP(64) is 2,147,483,646 bytes. See the table below for more details.

The LOCAL-STORAGE SECTION is allocated on the Language Environment stack. Its total size is limited by the settings in the Language Environment as well as the requirements of internal variables used by the compiler. The actual limit available for COBOL programs is less than 2,147,483,646 bytes.

The Language Environment STACK64 runtime option controls the allocation of stack storage for AMODE 64 applications. The default value is STACK64(1M,1M,128M). Use this runtime option to specify the maximum size of the stack required by your application.

The WORKING-STORAGE SECTION is allocated on the Language Environment heap. Its total size is limited by the 64-bit storage capacity of the machine.

Table 67. Compiler limits

<table>
<thead>
<tr>
<th>Language element</th>
<th>Compiler limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum length of user-defined words (for example, data-name, file-name, class-name)</td>
<td>30 bytes</td>
</tr>
<tr>
<td>Size of program</td>
<td>999,999 lines</td>
</tr>
<tr>
<td>Size of the following data types:</td>
<td></td>
</tr>
<tr>
<td>USAGE IS INDEX</td>
<td></td>
</tr>
<tr>
<td>USAGE IS POINTER</td>
<td></td>
</tr>
<tr>
<td>USAGE IS FUNCTION-POINTER</td>
<td></td>
</tr>
<tr>
<td>USAGE IS OBJECT-REFERENCE</td>
<td></td>
</tr>
<tr>
<td>With LP(32): 4 bytes</td>
<td>With LP(64): 8 bytes</td>
</tr>
<tr>
<td>Size of the following data types:</td>
<td></td>
</tr>
<tr>
<td>USAGE IS PROCEDURE-POINTER</td>
<td></td>
</tr>
<tr>
<td>With LP(32): 8 bytes</td>
<td>With LP(64): 8 bytes</td>
</tr>
<tr>
<td>LENGTH OF special register</td>
<td></td>
</tr>
<tr>
<td>With LP(32): USAGE IS BINARY PICTURE 9(9)</td>
<td>With LP(64): USAGE IS BINARY PICTURE 9(18)</td>
</tr>
<tr>
<td>ADDRESS OF special register</td>
<td></td>
</tr>
<tr>
<td>With LP(32): 4 bytes</td>
<td>With LP(64): 8 bytes</td>
</tr>
<tr>
<td>Number of literals</td>
<td>4,194,303(Note 1)</td>
</tr>
<tr>
<td>Total length of literals</td>
<td>4,194,303 bytes(Note 1)</td>
</tr>
<tr>
<td>Reserved word table entries</td>
<td>1536</td>
</tr>
<tr>
<td>COPY REPLACING . . . BY . . . (items per COPY statement)</td>
<td>No limit</td>
</tr>
<tr>
<td>Number of COPY libraries</td>
<td>No limit</td>
</tr>
<tr>
<td>Block size of COPY library</td>
<td>32,760 bytes</td>
</tr>
<tr>
<td>Language element</td>
<td>Compiler limit</td>
</tr>
<tr>
<td>--------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td><strong>IDENTIFICATION DIVISION</strong></td>
<td></td>
</tr>
<tr>
<td><strong>ENVIRONMENT DIVISION</strong></td>
<td></td>
</tr>
<tr>
<td>Configuration section</td>
<td></td>
</tr>
<tr>
<td>Special-names paragraph</td>
<td></td>
</tr>
<tr>
<td><code>mnemonic-name</code> IS</td>
<td>18</td>
</tr>
<tr>
<td>UPSI-<code>n</code> ... (switches)</td>
<td>0-7</td>
</tr>
<tr>
<td><code>alphabet-name</code> IS</td>
<td>No limit</td>
</tr>
<tr>
<td>Literal THRU ... or ALSO ...</td>
<td>256</td>
</tr>
<tr>
<td><strong>Input-Output section</strong></td>
<td></td>
</tr>
<tr>
<td>File-control paragraph</td>
<td></td>
</tr>
<tr>
<td><code>SELECT file-name</code> ...</td>
<td>A maximum of 65,535 file names can be assigned external names</td>
</tr>
<tr>
<td><code>ASSIGN system-name</code> ...</td>
<td>No limit</td>
</tr>
<tr>
<td><code>ALTERNATE RECORD KEY data-name</code> ...</td>
<td>253</td>
</tr>
<tr>
<td><code>RECORD KEY length</code></td>
<td>No limit(Notes 3)</td>
</tr>
<tr>
<td><code>RESERVE integer</code> (buffers)</td>
<td>255(Notes 4)</td>
</tr>
<tr>
<td><strong>I-O-control paragraph</strong></td>
<td></td>
</tr>
<tr>
<td><code>RERUN ON system-name</code> ...</td>
<td>32,767</td>
</tr>
<tr>
<td><code>RERUN integer RECORDS</code></td>
<td>16,777,215</td>
</tr>
<tr>
<td><code>SAME RECORD AREA</code></td>
<td>255</td>
</tr>
<tr>
<td><code>SAME RECORD AREA FOR file-name</code> ...</td>
<td>255</td>
</tr>
<tr>
<td><code>SAME SORT/MERGE AREA</code></td>
<td>No limit(Notes 2)</td>
</tr>
<tr>
<td><code>MULTIPLE FILE file-name</code> ...</td>
<td>No limit(Notes 2)</td>
</tr>
<tr>
<td><strong>DATA DIVISION</strong></td>
<td></td>
</tr>
<tr>
<td><code>77 data item size</code></td>
<td>With LP(32): 999,999,999 bytes With LP(64): 2,147,483,646 bytes</td>
</tr>
<tr>
<td><code>01-49 data item size</code></td>
<td>With LP(32): 999,999,999 bytes With LP(64): 2,147,483,646 bytes</td>
</tr>
<tr>
<td>Total 01 + 77 (data items)</td>
<td>No limit</td>
</tr>
<tr>
<td><code>88 condition-names</code> ...</td>
<td>No limit</td>
</tr>
<tr>
<td><code>88 level VALUE clause</code></td>
<td>No limit</td>
</tr>
<tr>
<td><code>66 RENAMES</code> ...</td>
<td>No limit</td>
</tr>
<tr>
<td><code>PICTURE clause, number of characters in character-string</code></td>
<td>50</td>
</tr>
<tr>
<td><code>PICTURE clause, numeric item digit positions</code></td>
<td>With ARITH(COMPAT): 18 With ARITH(EXTEND): 31</td>
</tr>
<tr>
<td>Language element</td>
<td>Compiler limit</td>
</tr>
<tr>
<td>------------------------------------------------------</td>
<td>-----------------------------------------------------</td>
</tr>
<tr>
<td>PICTURE clause, numeric-edited character positions</td>
<td>249</td>
</tr>
<tr>
<td>Picture symbol replication ( )</td>
<td>With LP(32): 999,999,999 bytes</td>
</tr>
<tr>
<td></td>
<td>With LP(64): 2,147,483,646 bytes</td>
</tr>
<tr>
<td>Picture symbol replication (editing)</td>
<td>32,767</td>
</tr>
<tr>
<td>Picture symbol replication ( ), class DBCS items</td>
<td>With LP(32): 499,999,999 bytes</td>
</tr>
<tr>
<td></td>
<td>With LP(64): 1,073,741,823 bytes</td>
</tr>
<tr>
<td>Picture symbol replication ( ), class national items</td>
<td>With LP(32): 499,999,999 bytes</td>
</tr>
<tr>
<td></td>
<td>With LP(64): 1,073,741,823 bytes</td>
</tr>
<tr>
<td>Elementary item size</td>
<td>With LP(32): 999,999,999 bytes</td>
</tr>
<tr>
<td></td>
<td>With LP(64): 2,147,483,646 bytes</td>
</tr>
<tr>
<td>OCCURS integer</td>
<td>With LP(32): 999,999,999 bytes</td>
</tr>
<tr>
<td></td>
<td>With LP(64): 2,147,483,646 bytes</td>
</tr>
<tr>
<td>Total number of ODOs</td>
<td>4,194,303(^{\text{Note 1}})</td>
</tr>
<tr>
<td>Table size</td>
<td>With LP(32): 999,999,999 bytes</td>
</tr>
<tr>
<td></td>
<td>With LP(64): 2,147,483,646 bytes</td>
</tr>
<tr>
<td>Table element size</td>
<td>With LP(32): 999,999,999 bytes</td>
</tr>
<tr>
<td></td>
<td>With LP(64): 2,147,483,646 bytes</td>
</tr>
<tr>
<td>ASCENDING or DESCENDING KEY . . . (per OCCURS clause)</td>
<td>12 KEYS</td>
</tr>
<tr>
<td>Total length of keys (per OCCURS clause)</td>
<td>256 bytes</td>
</tr>
<tr>
<td>INDEXED BY . . . (index names per OCCURS clause)</td>
<td>12</td>
</tr>
<tr>
<td>Total number of indexes (index names) per class or</td>
<td>65,535</td>
</tr>
<tr>
<td>program</td>
<td></td>
</tr>
<tr>
<td>Size of relative index</td>
<td>32,765</td>
</tr>
</tbody>
</table>

**FILE SECTION**

| FD record description entry                          | 1,048,575 bytes                                     |
| FD file-name . . .                                     | 65,535                                              |
| LABEL data-name . . . (if no optional clauses)        | 255                                                 |
| Label record length                                  | 80 bytes                                            |
| BLOCK CONTAINS integer                                | 2,147,483,647\(^{\text{Note 8}}\)                  |
| RECORD CONTAINS integer                               | 1,048,575\(^{\text{Note 5}}\)                      |
| LINAGE clause values                                  | 99,999,999                                          |
| SD file-name . . .                                     | 65,535                                              |
| DATA RECORD data-name . .                             | No limit\(^{\text{Note 2}}\)                       |

**LINKAGE SECTION**

<p>| Total size                                           | With LP(32): 2,147,483,646 bytes                    |
|                                                     | With LP(64): Unlimited, up to the available 64-bit  |
|                                                     | addressing capacity of the machine.                 |</p>
<table>
<thead>
<tr>
<th>Language element</th>
<th>Compiler limit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LOCAL-STORAGE SECTION</strong></td>
<td></td>
</tr>
<tr>
<td>Total size</td>
<td>With LP(32): 2,147,483,646 bytes</td>
</tr>
<tr>
<td></td>
<td>With LP(64): 2,147,483,646 bytes</td>
</tr>
<tr>
<td><strong>WORKING-STORAGE SECTION</strong></td>
<td></td>
</tr>
<tr>
<td>Total size of items without the external attribute</td>
<td>With LP(32): 2,147,483,646 bytes</td>
</tr>
<tr>
<td></td>
<td>With LP(64): Unlimited, up to the available 64-bit addressing capacity of the machine.</td>
</tr>
<tr>
<td>Total size of items with the external attribute</td>
<td>With LP(32): 2,147,483,646 bytes</td>
</tr>
<tr>
<td></td>
<td>With LP(64): Unlimited, up to the available 64-bit addressing capacity of the machine.</td>
</tr>
<tr>
<td><strong>PROCEDURE DIVISION</strong></td>
<td></td>
</tr>
<tr>
<td>Procedure and constant area</td>
<td>4,194,303 bytes</td>
</tr>
<tr>
<td></td>
<td><strong>(Note 1)</strong></td>
</tr>
<tr>
<td>PROCEDURE DIVISION USING <em>identifier</em> . . .</td>
<td>32,767</td>
</tr>
<tr>
<td>Procedure-names</td>
<td>1,048,575</td>
</tr>
<tr>
<td></td>
<td><strong>(Note 1)</strong></td>
</tr>
<tr>
<td>Subscripted data-names per statement</td>
<td>32,767</td>
</tr>
<tr>
<td>Statements per line (TEST)</td>
<td>7</td>
</tr>
<tr>
<td>ACCEPT statement, record length on input device</td>
<td>32,760</td>
</tr>
<tr>
<td>ADD <em>identifier</em> . . .</td>
<td>No limit</td>
</tr>
<tr>
<td>ALTER <em>procedure-name-1</em> TO <em>procedure-name-2</em> . . .</td>
<td>4,194,303</td>
</tr>
<tr>
<td></td>
<td><strong>(Note 1)</strong></td>
</tr>
<tr>
<td>CALL . . . BY CONTENT <em>identifier</em></td>
<td>2,147,483,647 bytes</td>
</tr>
<tr>
<td>CALL <em>identifier</em> or literal* USING <em>identifier</em> or literal . . .</td>
<td>16,380</td>
</tr>
<tr>
<td>CALL literal . . .</td>
<td>4,194,303</td>
</tr>
<tr>
<td></td>
<td><strong>(Note 1)</strong></td>
</tr>
<tr>
<td>Active programs in a run unit</td>
<td>32,767</td>
</tr>
<tr>
<td>Number of names called (DYN option)</td>
<td>No limit</td>
</tr>
<tr>
<td>CANCEL <em>identifier</em> or literal . .</td>
<td>No limit</td>
</tr>
<tr>
<td>CLOSE <em>file-name</em> . .</td>
<td>No limit</td>
</tr>
<tr>
<td>COMPUTE <em>identifier</em> . .</td>
<td>No limit</td>
</tr>
<tr>
<td>DISPLAY <em>identifier</em> or literal . .</td>
<td>No limit</td>
</tr>
<tr>
<td>DIVIDE <em>identifier</em> . .</td>
<td>No limit</td>
</tr>
<tr>
<td>ENTRY USING <em>identifier</em> or literal . .</td>
<td>No limit</td>
</tr>
<tr>
<td>EVALUATE . . . . subjects</td>
<td>64</td>
</tr>
<tr>
<td>EVALUATE . . . . WHEN clauses</td>
<td>256</td>
</tr>
<tr>
<td>GO <em>procedure-name</em> . . DEPENDING</td>
<td>255</td>
</tr>
<tr>
<td>INSPECT TALLYING and REPLACING clauses</td>
<td>No limit</td>
</tr>
<tr>
<td>MERGE <em>file-name</em> ASC or DES KEY . .</td>
<td>No limit</td>
</tr>
<tr>
<td>Total merge key length</td>
<td>4,092 bytes</td>
</tr>
<tr>
<td></td>
<td><strong>(Note 6)</strong></td>
</tr>
<tr>
<td>Language element</td>
<td>Compiler limit</td>
</tr>
<tr>
<td>------------------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>MERGE USING file-name . . .</td>
<td>16 (Note 7)</td>
</tr>
<tr>
<td>MOVE identifier or literal TO identifier . .</td>
<td>No limit</td>
</tr>
<tr>
<td>MULTIPLY identifier . .</td>
<td>No limit</td>
</tr>
<tr>
<td>OPEN file-name . .</td>
<td>No limit</td>
</tr>
<tr>
<td>PERFORM</td>
<td>4,194,303</td>
</tr>
<tr>
<td>PERFORM . . TIMES identifier or literal</td>
<td>999,999,999</td>
</tr>
<tr>
<td>SEARCH ALL . . maximum key length</td>
<td>No limit</td>
</tr>
<tr>
<td>SEARCH ALL . . total length of keys</td>
<td>No limit</td>
</tr>
<tr>
<td>SEARCH . . WHEN . .</td>
<td>No limit</td>
</tr>
<tr>
<td>SET index or identifier . . TO</td>
<td>No limit</td>
</tr>
<tr>
<td>SET index . . UP/DOWN</td>
<td>No limit</td>
</tr>
<tr>
<td>SORT file-name ASC or DES KEY</td>
<td>No limit</td>
</tr>
<tr>
<td>Total sort key length</td>
<td>4,092 bytes (Note 6)</td>
</tr>
<tr>
<td>SORT USING file-name . .</td>
<td>16 (Note 7)</td>
</tr>
<tr>
<td>STRING identifier . .</td>
<td>No limit</td>
</tr>
<tr>
<td>STRING DELIMITED identifier or literal . .</td>
<td>No limit</td>
</tr>
<tr>
<td>UNSTRING DELIMITED identifier or literal . .</td>
<td>No limit</td>
</tr>
<tr>
<td>UNSTRING INTO identifier or literal . .</td>
<td>No limit</td>
</tr>
<tr>
<td>USE . . ON file-name . .</td>
<td>No limit</td>
</tr>
<tr>
<td>XML PARSE statement, maximum size of identifier</td>
<td>999,999,999 bytes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>With LP(32): Return maximum 9-digit integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>LENGTH</td>
<td>With LP(64): Return maximum 18-digit integer</td>
</tr>
<tr>
<td>UPOS</td>
<td></td>
</tr>
<tr>
<td>UVALID</td>
<td></td>
</tr>
<tr>
<td>ULENGTH</td>
<td></td>
</tr>
<tr>
<td>USUPPLEMENTARY</td>
<td></td>
</tr>
</tbody>
</table>

Notes:
1. Items included in 4,194,303 byte limit for procedure plus constant area.
2. Syntax checked, but has no effect on the execution of the program; there is no limit.
3. No compiler limit, but VSAM limits it to 255 bytes.
4. QSAM.
5. Compiler limit shown, but QSAM limits it to 32,760 bytes.
6. For QSAM and VSAM, the limit is 4088 bytes if EQUALS is coded on the OPTION control statement.
7. SORT limit for QSAM and VSAM.
8. Requires large block interface (LBI) support provided by OS/390 DFSMS Version 2 Release 10.0 or later. On OS/390 systems with earlier releases of DFSMS, the limit is 32,760 bytes. For more information about using large block sizes, see Setting block sizes in the Enterprise COBOL Programming Guide.
Appendix C. EBCDIC and ASCII collating sequences

A collating sequence defines the order of characters within a coded character set or COBOL alphabet for purposes of sorting, merging, and comparing data and for processing files with indexed organization.

The ascending collating sequences for both the single-byte EBCDIC (Extended Binary Coded Decimal Interchange Code) and single-byte ASCII (American National Standard Code for Information Interchange) character sets are shown in this appendix. The collating sequence is defined by the ordinal number of characters in the character set, relative to 1.

The symbols and associated meanings shown for the EBCDIC collating sequence are those defined in the EBCDIC code page defined with CCSID 1140. Symbols and meanings can vary for other EBCDIC code pages, but the collating sequence is unchanged.

**EBCDIC collating sequence**

The EBCDIC collating sequence is the order in which characters are defined in EBCDIC.

The following table presents the collating sequence for single-byte EBCDIC code page 1140. Ellipsis (...) indicates omission of a range of ordinal numbers between predecessor and successor ordinal numbers.

<table>
<thead>
<tr>
<th>Ordinal number</th>
<th>Symbol</th>
<th>Meaning</th>
<th>Decimal representation</th>
<th>Hex representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>Space</td>
<td></td>
<td>64</td>
<td>40</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>¢</td>
<td>Cent sign</td>
<td>74</td>
<td>4A</td>
</tr>
<tr>
<td>76</td>
<td>.</td>
<td>Period, decimal point</td>
<td>75</td>
<td>4B</td>
</tr>
<tr>
<td>77</td>
<td>&lt;</td>
<td>Less than sign</td>
<td>76</td>
<td>4C</td>
</tr>
<tr>
<td>78</td>
<td>(</td>
<td>Left parenthesis</td>
<td>77</td>
<td>4D</td>
</tr>
<tr>
<td>79</td>
<td>+</td>
<td>Plus sign</td>
<td>78</td>
<td>4E</td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
<td>Vertical bar, logical OR</td>
<td>79</td>
</tr>
<tr>
<td>81</td>
<td>&amp;</td>
<td>Ampersand</td>
<td>80</td>
<td>50</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>91</td>
<td>!</td>
<td>Exclamation point</td>
<td>90</td>
<td>5A</td>
</tr>
<tr>
<td>92</td>
<td>$</td>
<td>Dollar sign</td>
<td>91</td>
<td>5B</td>
</tr>
<tr>
<td>93</td>
<td>*</td>
<td>Asterisk</td>
<td>92</td>
<td>5C</td>
</tr>
<tr>
<td>94</td>
<td>)</td>
<td>Right parenthesis</td>
<td>93</td>
<td>5D</td>
</tr>
<tr>
<td>95</td>
<td>;</td>
<td>Semicolon</td>
<td>94</td>
<td>5E</td>
</tr>
<tr>
<td>96</td>
<td>¬</td>
<td>Logical NOT</td>
<td>95</td>
<td>5F</td>
</tr>
<tr>
<td>97</td>
<td>-</td>
<td>Minus, hyphen</td>
<td>96</td>
<td>60</td>
</tr>
</tbody>
</table>
Table 68. **EBCDIC collating sequence** (continued)

<table>
<thead>
<tr>
<th>Ordinal number</th>
<th>Symbol</th>
<th>Meaning</th>
<th>Decimal representation</th>
<th>Hex representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>98</td>
<td>/</td>
<td>Slash</td>
<td>97</td>
<td>61</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>108</td>
<td>,</td>
<td>Comma</td>
<td>107</td>
<td>6B</td>
</tr>
<tr>
<td>109</td>
<td>%</td>
<td>Percent sign</td>
<td>108</td>
<td>6C</td>
</tr>
<tr>
<td>110</td>
<td>_</td>
<td>Underscore</td>
<td>109</td>
<td>6D</td>
</tr>
<tr>
<td>111</td>
<td>&gt;</td>
<td>Greater than sign</td>
<td>110</td>
<td>6E</td>
</tr>
<tr>
<td>112</td>
<td>?</td>
<td>Question mark</td>
<td>111</td>
<td>6F</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>122</td>
<td>\</td>
<td>Grave accent</td>
<td>121</td>
<td>79</td>
</tr>
<tr>
<td>123</td>
<td>:</td>
<td>Colon</td>
<td>122</td>
<td>7A</td>
</tr>
<tr>
<td>124</td>
<td>#</td>
<td>Number sign, pound sign</td>
<td>123</td>
<td>7B</td>
</tr>
<tr>
<td>125</td>
<td>@</td>
<td>At sign</td>
<td>124</td>
<td>7C</td>
</tr>
<tr>
<td>126</td>
<td>'</td>
<td>Apostrophe, prime sign</td>
<td>125</td>
<td>7D</td>
</tr>
<tr>
<td>127</td>
<td>=</td>
<td>Equal sign</td>
<td>126</td>
<td>7E</td>
</tr>
<tr>
<td>128</td>
<td>&quot;</td>
<td>Quotation marks</td>
<td>127</td>
<td>7F</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>130</td>
<td>a</td>
<td></td>
<td>129</td>
<td>81</td>
</tr>
<tr>
<td>131</td>
<td>b</td>
<td></td>
<td>130</td>
<td>82</td>
</tr>
<tr>
<td>132</td>
<td>c</td>
<td></td>
<td>131</td>
<td>83</td>
</tr>
<tr>
<td>133</td>
<td>d</td>
<td></td>
<td>132</td>
<td>84</td>
</tr>
<tr>
<td>134</td>
<td>e</td>
<td></td>
<td>133</td>
<td>85</td>
</tr>
<tr>
<td>135</td>
<td>f</td>
<td></td>
<td>134</td>
<td>86</td>
</tr>
<tr>
<td>136</td>
<td>g</td>
<td></td>
<td>135</td>
<td>87</td>
</tr>
<tr>
<td>137</td>
<td>h</td>
<td></td>
<td>136</td>
<td>88</td>
</tr>
<tr>
<td>138</td>
<td>i</td>
<td></td>
<td>137</td>
<td>89</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>146</td>
<td>j</td>
<td></td>
<td>145</td>
<td>91</td>
</tr>
<tr>
<td>147</td>
<td>k</td>
<td></td>
<td>146</td>
<td>92</td>
</tr>
<tr>
<td>148</td>
<td>l</td>
<td></td>
<td>147</td>
<td>93</td>
</tr>
<tr>
<td>149</td>
<td>m</td>
<td></td>
<td>148</td>
<td>94</td>
</tr>
<tr>
<td>150</td>
<td>n</td>
<td></td>
<td>149</td>
<td>95</td>
</tr>
<tr>
<td>151</td>
<td>o</td>
<td></td>
<td>150</td>
<td>96</td>
</tr>
<tr>
<td>152</td>
<td>p</td>
<td></td>
<td>151</td>
<td>97</td>
</tr>
<tr>
<td>153</td>
<td>q</td>
<td></td>
<td>152</td>
<td>98</td>
</tr>
<tr>
<td>Ordinal number</td>
<td>Symbol</td>
<td>Meaning</td>
<td>Decimal representation</td>
<td>Hex representation</td>
</tr>
<tr>
<td>----------------</td>
<td>--------</td>
<td>----------------------</td>
<td>------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>154</td>
<td>r</td>
<td></td>
<td>153</td>
<td>99</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>€</td>
<td>Euro currency sign</td>
<td>159</td>
<td>9F</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>162</td>
<td>~</td>
<td>Tilde</td>
<td>161</td>
<td>A1</td>
</tr>
<tr>
<td>163</td>
<td>s</td>
<td></td>
<td>162</td>
<td>A2</td>
</tr>
<tr>
<td>164</td>
<td>t</td>
<td></td>
<td>163</td>
<td>A3</td>
</tr>
<tr>
<td>165</td>
<td>u</td>
<td></td>
<td>164</td>
<td>A4</td>
</tr>
<tr>
<td>166</td>
<td>v</td>
<td></td>
<td>165</td>
<td>A5</td>
</tr>
<tr>
<td>167</td>
<td>w</td>
<td></td>
<td>166</td>
<td>A6</td>
</tr>
<tr>
<td>168</td>
<td>x</td>
<td></td>
<td>167</td>
<td>A7</td>
</tr>
<tr>
<td>169</td>
<td>y</td>
<td></td>
<td>168</td>
<td>A8</td>
</tr>
<tr>
<td>170</td>
<td>z</td>
<td></td>
<td>169</td>
<td>A9</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>177</td>
<td>^</td>
<td>Caret</td>
<td>176</td>
<td>B0</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>188</td>
<td>[</td>
<td>Opening square bracket</td>
<td>187</td>
<td>BA</td>
</tr>
<tr>
<td>189</td>
<td>]</td>
<td>Closing square bracket</td>
<td>188</td>
<td>BB</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>193</td>
<td>{</td>
<td>Opening brace</td>
<td>192</td>
<td>C0</td>
</tr>
<tr>
<td>194</td>
<td>A</td>
<td></td>
<td>193</td>
<td>C1</td>
</tr>
<tr>
<td>195</td>
<td>B</td>
<td></td>
<td>194</td>
<td>C2</td>
</tr>
<tr>
<td>196</td>
<td>C</td>
<td></td>
<td>195</td>
<td>C3</td>
</tr>
<tr>
<td>197</td>
<td>D</td>
<td></td>
<td>196</td>
<td>C4</td>
</tr>
<tr>
<td>198</td>
<td>E</td>
<td></td>
<td>197</td>
<td>C5</td>
</tr>
<tr>
<td>199</td>
<td>F</td>
<td></td>
<td>198</td>
<td>C6</td>
</tr>
<tr>
<td>200</td>
<td>G</td>
<td></td>
<td>199</td>
<td>C7</td>
</tr>
<tr>
<td>201</td>
<td>H</td>
<td></td>
<td>200</td>
<td>C8</td>
</tr>
<tr>
<td>202</td>
<td>I</td>
<td></td>
<td>201</td>
<td>C9</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>209</td>
<td>}</td>
<td>Closing brace</td>
<td>208</td>
<td>D0</td>
</tr>
<tr>
<td>210</td>
<td>J</td>
<td></td>
<td>209</td>
<td>D1</td>
</tr>
<tr>
<td>211</td>
<td>K</td>
<td></td>
<td>210</td>
<td>D2</td>
</tr>
<tr>
<td>212</td>
<td>L</td>
<td></td>
<td>211</td>
<td>D3</td>
</tr>
</tbody>
</table>
The following table presents the collating sequence for the US English ASCII code page. The collating sequence is the order in which characters are defined in ANSI INCITS 4, the 7-Bit American National
Standard Code for Information Interchange (7-Bit ASCII), and in the International Reference Version of ISO/IEC 646, 7-Bit Coded Character Set for Information Interchange.

Ellipsis ( . . . ) indicates omission of a range of ordinal numbers between predecessor and successor ordinal numbers.

Table 69. ASCII collating sequence

<table>
<thead>
<tr>
<th>Ordinal number</th>
<th>Symbol</th>
<th>Meaning</th>
<th>Decimal representation</th>
<th>Hex representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Null</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>. . .</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>Space</td>
<td></td>
<td>32</td>
<td>20</td>
</tr>
<tr>
<td>34</td>
<td>!</td>
<td>Exclamation point</td>
<td>33</td>
<td>21</td>
</tr>
<tr>
<td>35</td>
<td>&quot;</td>
<td>Quotation mark</td>
<td>34</td>
<td>22</td>
</tr>
<tr>
<td>36</td>
<td>#</td>
<td>Number sign</td>
<td>35</td>
<td>23</td>
</tr>
<tr>
<td>37</td>
<td>$</td>
<td>Dollar sign</td>
<td>36</td>
<td>24</td>
</tr>
<tr>
<td>38</td>
<td>%</td>
<td>Percent sign</td>
<td>37</td>
<td>25</td>
</tr>
<tr>
<td>39</td>
<td>&amp;</td>
<td>Ampersand</td>
<td>38</td>
<td>26</td>
</tr>
<tr>
<td>40</td>
<td>'</td>
<td>Apostrophe, prime sign</td>
<td>39</td>
<td>27</td>
</tr>
<tr>
<td>41</td>
<td>(</td>
<td>Opening parenthesis</td>
<td>40</td>
<td>28</td>
</tr>
<tr>
<td>42</td>
<td>)</td>
<td>Closing parenthesis</td>
<td>41</td>
<td>29</td>
</tr>
<tr>
<td>43</td>
<td>*</td>
<td>Asterisk</td>
<td>42</td>
<td>2A</td>
</tr>
<tr>
<td>44</td>
<td>+</td>
<td>Plus sign</td>
<td>43</td>
<td>2B</td>
</tr>
<tr>
<td>45</td>
<td>,</td>
<td>Comma</td>
<td>44</td>
<td>2C</td>
</tr>
<tr>
<td>46</td>
<td>-</td>
<td>Hyphen, minus</td>
<td>45</td>
<td>2D</td>
</tr>
<tr>
<td>47</td>
<td>.</td>
<td>Period, decimal point</td>
<td>46</td>
<td>2E</td>
</tr>
<tr>
<td>48</td>
<td>/</td>
<td>Slash, solidus</td>
<td>47</td>
<td>2F</td>
</tr>
<tr>
<td>49</td>
<td>0</td>
<td></td>
<td>48</td>
<td>30</td>
</tr>
<tr>
<td>50</td>
<td>1</td>
<td></td>
<td>49</td>
<td>31</td>
</tr>
<tr>
<td>51</td>
<td>2</td>
<td></td>
<td>50</td>
<td>32</td>
</tr>
<tr>
<td>52</td>
<td>3</td>
<td></td>
<td>51</td>
<td>33</td>
</tr>
<tr>
<td>53</td>
<td>4</td>
<td></td>
<td>52</td>
<td>34</td>
</tr>
<tr>
<td>54</td>
<td>5</td>
<td></td>
<td>53</td>
<td>35</td>
</tr>
<tr>
<td>55</td>
<td>6</td>
<td></td>
<td>54</td>
<td>36</td>
</tr>
<tr>
<td>56</td>
<td>7</td>
<td></td>
<td>55</td>
<td>37</td>
</tr>
<tr>
<td>57</td>
<td>8</td>
<td></td>
<td>56</td>
<td>38</td>
</tr>
<tr>
<td>58</td>
<td>9</td>
<td></td>
<td>57</td>
<td>39</td>
</tr>
<tr>
<td>59</td>
<td>:</td>
<td>Colon</td>
<td>58</td>
<td>3A</td>
</tr>
<tr>
<td>60</td>
<td>;</td>
<td>Semicolon</td>
<td>59</td>
<td>3B</td>
</tr>
<tr>
<td>Ordinal number</td>
<td>Symbol</td>
<td>Meaning</td>
<td>Decimal representation</td>
<td>Hex representation</td>
</tr>
<tr>
<td>----------------</td>
<td>--------</td>
<td>-----------------------</td>
<td>------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>61</td>
<td>&lt;</td>
<td>Less than sign</td>
<td>60</td>
<td>3C</td>
</tr>
<tr>
<td>62</td>
<td>=</td>
<td>Equal sign</td>
<td>61</td>
<td>3D</td>
</tr>
<tr>
<td>63</td>
<td>&gt;</td>
<td>Greater than sign</td>
<td>62</td>
<td>3E</td>
</tr>
<tr>
<td>64</td>
<td>?</td>
<td>Question mark</td>
<td>63</td>
<td>3F</td>
</tr>
<tr>
<td>65</td>
<td>@</td>
<td>Commercial At sign</td>
<td>64</td>
<td>40</td>
</tr>
<tr>
<td>66</td>
<td>A</td>
<td></td>
<td>65</td>
<td>41</td>
</tr>
<tr>
<td>67</td>
<td>B</td>
<td></td>
<td>66</td>
<td>42</td>
</tr>
<tr>
<td>68</td>
<td>C</td>
<td></td>
<td>67</td>
<td>43</td>
</tr>
<tr>
<td>69</td>
<td>D</td>
<td></td>
<td>68</td>
<td>44</td>
</tr>
<tr>
<td>70</td>
<td>E</td>
<td></td>
<td>69</td>
<td>45</td>
</tr>
<tr>
<td>71</td>
<td>F</td>
<td></td>
<td>70</td>
<td>46</td>
</tr>
<tr>
<td>72</td>
<td>G</td>
<td></td>
<td>71</td>
<td>47</td>
</tr>
<tr>
<td>73</td>
<td>H</td>
<td></td>
<td>72</td>
<td>48</td>
</tr>
<tr>
<td>74</td>
<td>I</td>
<td></td>
<td>73</td>
<td>49</td>
</tr>
<tr>
<td>75</td>
<td>J</td>
<td></td>
<td>74</td>
<td>4A</td>
</tr>
<tr>
<td>76</td>
<td>K</td>
<td></td>
<td>75</td>
<td>4B</td>
</tr>
<tr>
<td>77</td>
<td>L</td>
<td></td>
<td>76</td>
<td>4C</td>
</tr>
<tr>
<td>78</td>
<td>M</td>
<td></td>
<td>77</td>
<td>4D</td>
</tr>
<tr>
<td>79</td>
<td>N</td>
<td></td>
<td>78</td>
<td>4E</td>
</tr>
<tr>
<td>80</td>
<td>O</td>
<td></td>
<td>79</td>
<td>4F</td>
</tr>
<tr>
<td>81</td>
<td>P</td>
<td></td>
<td>80</td>
<td>50</td>
</tr>
<tr>
<td>82</td>
<td>Q</td>
<td></td>
<td>81</td>
<td>51</td>
</tr>
<tr>
<td>83</td>
<td>R</td>
<td></td>
<td>82</td>
<td>52</td>
</tr>
<tr>
<td>84</td>
<td>S</td>
<td></td>
<td>83</td>
<td>53</td>
</tr>
<tr>
<td>85</td>
<td>T</td>
<td></td>
<td>84</td>
<td>54</td>
</tr>
<tr>
<td>86</td>
<td>U</td>
<td></td>
<td>85</td>
<td>55</td>
</tr>
<tr>
<td>87</td>
<td>V</td>
<td></td>
<td>86</td>
<td>56</td>
</tr>
<tr>
<td>88</td>
<td>W</td>
<td></td>
<td>87</td>
<td>57</td>
</tr>
<tr>
<td>89</td>
<td>X</td>
<td></td>
<td>88</td>
<td>58</td>
</tr>
<tr>
<td>90</td>
<td>Y</td>
<td></td>
<td>89</td>
<td>59</td>
</tr>
<tr>
<td>91</td>
<td>Z</td>
<td></td>
<td>90</td>
<td>5A</td>
</tr>
<tr>
<td>92</td>
<td>[</td>
<td>Opening bracket</td>
<td>91</td>
<td>5B</td>
</tr>
<tr>
<td>93</td>
<td>\</td>
<td>Backslash, reverse solidus</td>
<td>92</td>
<td>5C</td>
</tr>
<tr>
<td>94</td>
<td>]</td>
<td>Closing bracket</td>
<td>93</td>
<td>5D</td>
</tr>
<tr>
<td>Ordinal number</td>
<td>Symbol</td>
<td>Meaning</td>
<td>Decimal representation</td>
<td>Hex representation</td>
</tr>
<tr>
<td>----------------</td>
<td>--------</td>
<td>------------------</td>
<td>------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>95</td>
<td>^</td>
<td>Caret</td>
<td>94</td>
<td>5E</td>
</tr>
<tr>
<td>96</td>
<td>_</td>
<td>Underscore</td>
<td>95</td>
<td>5F</td>
</tr>
<tr>
<td>97</td>
<td>`</td>
<td>Grave accent</td>
<td>96</td>
<td>60</td>
</tr>
<tr>
<td>98</td>
<td>a</td>
<td></td>
<td>97</td>
<td>61</td>
</tr>
<tr>
<td>99</td>
<td>b</td>
<td></td>
<td>98</td>
<td>62</td>
</tr>
<tr>
<td>100</td>
<td>c</td>
<td></td>
<td>99</td>
<td>63</td>
</tr>
<tr>
<td>101</td>
<td>d</td>
<td></td>
<td>100</td>
<td>64</td>
</tr>
<tr>
<td>102</td>
<td>e</td>
<td></td>
<td>101</td>
<td>65</td>
</tr>
<tr>
<td>103</td>
<td>f</td>
<td></td>
<td>102</td>
<td>66</td>
</tr>
<tr>
<td>104</td>
<td>g</td>
<td></td>
<td>103</td>
<td>67</td>
</tr>
<tr>
<td>105</td>
<td>h</td>
<td></td>
<td>104</td>
<td>68</td>
</tr>
<tr>
<td>106</td>
<td>i</td>
<td></td>
<td>105</td>
<td>69</td>
</tr>
<tr>
<td>107</td>
<td>j</td>
<td></td>
<td>106</td>
<td>6A</td>
</tr>
<tr>
<td>108</td>
<td>k</td>
<td></td>
<td>107</td>
<td>6B</td>
</tr>
<tr>
<td>109</td>
<td>l</td>
<td></td>
<td>108</td>
<td>6C</td>
</tr>
<tr>
<td>110</td>
<td>m</td>
<td></td>
<td>109</td>
<td>6D</td>
</tr>
<tr>
<td>111</td>
<td>n</td>
<td></td>
<td>110</td>
<td>6E</td>
</tr>
<tr>
<td>112</td>
<td>o</td>
<td></td>
<td>111</td>
<td>6F</td>
</tr>
<tr>
<td>113</td>
<td>p</td>
<td></td>
<td>112</td>
<td>70</td>
</tr>
<tr>
<td>114</td>
<td>q</td>
<td></td>
<td>113</td>
<td>71</td>
</tr>
<tr>
<td>115</td>
<td>r</td>
<td></td>
<td>114</td>
<td>72</td>
</tr>
<tr>
<td>116</td>
<td>s</td>
<td></td>
<td>115</td>
<td>73</td>
</tr>
<tr>
<td>117</td>
<td>t</td>
<td></td>
<td>116</td>
<td>74</td>
</tr>
<tr>
<td>118</td>
<td>u</td>
<td></td>
<td>117</td>
<td>75</td>
</tr>
<tr>
<td>119</td>
<td>v</td>
<td></td>
<td>118</td>
<td>76</td>
</tr>
<tr>
<td>120</td>
<td>w</td>
<td></td>
<td>119</td>
<td>77</td>
</tr>
<tr>
<td>121</td>
<td>x</td>
<td></td>
<td>120</td>
<td>78</td>
</tr>
<tr>
<td>122</td>
<td>y</td>
<td></td>
<td>121</td>
<td>79</td>
</tr>
<tr>
<td>123</td>
<td>z</td>
<td></td>
<td>122</td>
<td>7A</td>
</tr>
<tr>
<td>124</td>
<td>{</td>
<td>Opening brace</td>
<td>123</td>
<td>7B</td>
</tr>
<tr>
<td>125</td>
<td></td>
<td></td>
<td>124</td>
<td>7C</td>
</tr>
<tr>
<td>126</td>
<td>}</td>
<td>Closing brace</td>
<td>125</td>
<td>7D</td>
</tr>
<tr>
<td>127</td>
<td>~</td>
<td>Tilde</td>
<td>126</td>
<td>7E</td>
</tr>
</tbody>
</table>
Appendix D. Source language debugging

Several COBOL language elements implement the debugging feature. COBOL language elements are:

- Debugging lines
- Debugging sections
- DEBUG-ITEM special register
- Compile-time switch (WITH DEBUGGING MODE clause)
- Object-time switch

### Debugging lines

A *debugging line* is a statement that is compiled only when the compile-time switch is activated. Debugging lines allow you, for example, to check the value of a data item at certain points in a procedure.

To specify a debugging line in your program, code a D in column 7 (the indicator area). You can include successive debugging lines, but each must have a D in column 7. You cannot break character-strings across two lines.

All your debugging lines must be written so that the program is syntactically correct, whether the debugging lines are compiled or treated as comments.

You can code debugging lines anywhere in your program after the OBJECT-COMPUTER paragraph. A debugging line that contains only spaces in Area A and in Area B is treated as a blank line.

### Debugging sections

Debugging sections are permitted only in the outermost program; they are not valid in nested programs. Debugging sections are never triggered by procedures contained in nested programs.

Debugging sections are declarative procedures. Declarative procedures are described under “USE statement” on page 541. A debugging section can be called, for example, by a PERFORM statement that causes repeated execution of a procedure. Any associated *procedure-name* debugging declarative section is executed once for each repetition.

A debugging section executes *only* if both the compile-time switch and the object-time switch are activated.

The debug feature recognizes each separate occurrence of an imperative statement *within* an imperative statement as the beginning of a separate statement.

You cannot refer to a procedure defined within a debugging section from a statement outside of the debugging section.

References to the DEBUG-ITEM special register can be made only from within a debugging declarative procedure.

### DEBUG-ITEM special register

The DEBUG-ITEM special register provides information for a debugging declarative procedure about the conditions that cause debugging section execution.

For details of the DEBUG-ITEM special register, see “DEBUG-ITEM” on page 17.
Activate compile-time switch

The compile-time switch activates the debugging lines and sections. To place the compile-time switch in effect, specify WITH DEBUGGING MODE in the SOURCE COMPUTER paragraph of the configuration section.

**Format**

```
SOURCE-COMPUTER.   
computer-name WITH DEBUGGING MODE.
```

**WITH DEBUGGING MODE**

When WITH DEBUGGING MODE is specified, all debugging sections and debugging lines are compiled.

When WITH DEBUGGING MODE is omitted, all debugging sections and debugging lines are treated as comments.

**Usage note:** If you include a COPY statement as a debugging line, the letter "D" must appear on the first line of the COPY statement. The compiler treats the copied text as the debugging line or lines. The COPY statement is executed, regardless of whether WITH DEBUGGING MODE is specified or not.

Activate object-time switch

The object-time switch is set when the runtime option DEBUG or NODEBUG is specified. (NODEBUG is the default supplied by IBM.)

For details on the format, see the *Language Environment Programming Guide*.

The USE FOR DEBUGGING declarative procedures are activated when DEBUG is in effect and inhibited when NODEBUG is in effect.

The debugging lines (lines with "D" or "d" in column 7) are not affected by the DEBUG or NODEBUG option; they are always active if they have been compiled.

When WITH DEBUGGING MODE is *not* specified in the SOURCE-COMPUTER paragraph, the object-time switch has no effect on execution of the object program.

You do not have to recompile the source unit to activate or deactivate the object-time switch.
Appendix E. Reserved words

A *reserved word* is a character-string with a predefined meaning in a COBOL source unit.

The following table identifies words that are reserved in Enterprise COBOL and words that you should avoid because they might be reserved in a future release of Enterprise COBOL.

- **Words marked X under *Reserved* are reserved for function implemented in Enterprise COBOL. If used as user-defined names, these words are flagged with an S-level message.**
- **Words marked X under *Standard only* are 85 COBOL Standard reserved words for function not implemented in Enterprise COBOL. (Some of the function is implemented in the Report Writer Precompiler.) Use of these words as user-defined names is flagged with an S-level message.**
- **Words marked X under *Potential reserved words* are words that might be reserved in a future release of Enterprise COBOL. IBM recommends that you not use these words as user-defined names. Use of these words as user-defined names is flagged with an I-level message.**

This column includes words reserved in the 2002 COBOL Standard.

The default reserved word table is shown below. You can select a different reserved word table by using the WORD compiler option. For details, see *WORD* in the *Enterprise COBOL Programming Guide*.

<table>
<thead>
<tr>
<th>Word</th>
<th>Reserved</th>
<th>Standard only</th>
<th>Potential reserved words</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>**</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*&gt;</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACCESS</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACTIVE-CLASS</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 1991, 2020
<table>
<thead>
<tr>
<th>Word</th>
<th>Reserved</th>
<th>Standard only</th>
<th>Potential reserved words</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDRESS</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADVANCING</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AFTER</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALIGNED</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>ALL</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALLOCATE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALPHABET</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALPHABETIC</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALPHABETIC-LOWER</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALPHABETIC-UPPER</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALPHANUMERIC</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALPHANUMERIC-EDITED</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALSO</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALTER</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALTERNATE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANY</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANYCASE</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>APPLY</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AREA</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AREAS</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASCENDING</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASSIGN</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AUTHOR</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B-AND</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>B-NOT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B-OR</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B-XOR</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BASED</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BASIS</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEFORE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Table 70. Reserved words (continued)

<table>
<thead>
<tr>
<th>Word</th>
<th>Reserved</th>
<th>Standard only</th>
<th>Potential reserved words</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGINNING</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BINARY</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BINARY-CHAR</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BINARY-DOUBLE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BINARY-LONG</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BINARY-SHORT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIT</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>BLANK</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLOCK</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOOLEAN</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>BOTTOM</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BY</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BYTE-LENGTH</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CANCEL</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CBL</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CD</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CF</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CH</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHARACTERS</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLASS</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLASS-ID</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLOCK-UNITS</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLOSE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COBOL</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CODE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CODE-SET</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COL</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>COLLATING</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COLS</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>COLUMN</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>COLUMNS</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>COM-REG</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Word</td>
<td>Reserved</td>
<td>Standard only</td>
<td>Potential reserved words</td>
</tr>
<tr>
<td>----------------</td>
<td>----------</td>
<td>---------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>COMMA</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMON</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMUNICATION</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>COMP</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMP-1</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMP-2</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMP-3</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMP-4</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMP-5</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPUTATIONAL</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Computational-1</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Computational-2</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Computational-3</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Computational-4</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Computational-5</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPUTE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONDITION</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONFIGURATION</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONSTANT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONTAINS</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONTENT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONTINUE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONTROL</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONTROLS</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONVERTING</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COPY</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CORR</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CORRESPONDING</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COUNT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRT</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CURRENCY</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CURSOR</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>DATA</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATA-PROCESSOR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 70. **Reserved words** (continued)
<table>
<thead>
<tr>
<th>Word</th>
<th>Reserved</th>
<th>Standard only</th>
<th>Potential reserved words</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATE-COMPILED</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATE-Written</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DAY</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DAY-OF-WEEK</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBCS</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>DEBUG-CONTENTS</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEBUG-ITEM</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEBUG-LINE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEBUG-NAME</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEBUG-SUB-1</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEBUG-SUB-2</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEBUG-SUB-3</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEBUGGING</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DECIMAL-POINT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DECLARATIVES</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEFAULT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELETE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELIMITED</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELIMITER</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEPENDING</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DESCENDING</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DESTINATION</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DETAIL</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DISABLE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DISPLAY</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DISPLAY-1</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVIDE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVISION</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOWN</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DUPLICATES</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DYNAMIC</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EC</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Word</td>
<td>Reserved</td>
<td>Standard only</td>
<td>Potential reserved words</td>
</tr>
<tr>
<td>--------------</td>
<td>----------</td>
<td>---------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>EGCS</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EGI</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EJECT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ELSE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EMI</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENABLE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END-ACCEPT</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>END-ADD</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END-CALL</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END-COMPUTE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END-DELETE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END-DISPLAY</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>END-DIVIDE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END-EVALUATE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END-EXEC</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END-IF</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END-INVOKE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END-JSON</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END-MULTIPLY</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END-OF-PAGE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END-PERFORM</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END-READ</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END-RECEIVE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>END-RETURN</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END-REWRITE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END-SEARCH</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END-START</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END-STRING</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END-SUBTRACT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END-UNSTRING</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END-WRITE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END-XML</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENDING</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Word</td>
<td>Reserved</td>
<td>Standard only</td>
<td>Potential reserved words</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------</td>
<td>---------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>ENTER</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENTRY</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENVIRONMENT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EO</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>EOP</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EQUAL</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ERROR</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESI</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>EVALUATE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVERY</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXCEPTION</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXCEPTION-OBJECT</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>EXEC</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXECUTE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXIT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXTEND</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FACTORY</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FALSE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FD</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FILE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FILE-CONTROL</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FILLER</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FINAL</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>FIRST</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOAT-EXTENDED</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>FLOAT-LONG</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>FLOAT-SHORT</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>FOOTING</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FOR</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FORMAT</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>FREE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FROM</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FUNCTION</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Word</td>
<td>Reserved</td>
<td>Standard only</td>
<td>Potential reserved words</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------</td>
<td>---------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>FUNCTION-ID</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>FUNCTION-POINTER</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GENERATE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GET</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>GIVING</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GLOBAL</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GO</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GOBACK</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GREATER</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GROUP</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>GROUP-USAGE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HEADING</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>HIGH-VALUE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HIGH-VALUES</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I-O</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I-O-CONTROL</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDENTIFICATION</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IF</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INDEX</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INDEXED</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INDICATE</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>INHERITS</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INITIAL</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INITIALIZE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INITIATE</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>INPUT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INPUT-OUTPUT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSPECT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSTALLATION</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTERFACE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>INTERFACE-ID</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Word</td>
<td>Reserved</td>
<td>Standard only</td>
<td>Potential reserved words</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------</td>
<td>---------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>INTO</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INVALID</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INVOKE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IS</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JAVA</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNIENVPTR</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JSON</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JSON-CODE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JSON-STATUS</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JUST</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JUSTIFIED</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KANJI</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KEY</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LABEL</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAST</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>LEADING</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEFT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LENGTH</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LESS</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LIMIT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LIMITS</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>LINAGE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINAGE-COUNTER</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINE-COUNTER</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINES</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINKAGE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOCAL-_STORAGE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOCALE</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>LOCK</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOW-VALUE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOW-VALUES</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEMORY</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MERGE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Word</td>
<td>Reserved</td>
<td>Standard only</td>
<td>Potential reserved words</td>
</tr>
<tr>
<td>--------------------------</td>
<td>----------</td>
<td>---------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>MESSAGE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>METHOD</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>METHOD-ID</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>MINUS</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>MODE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>MODULES</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>MORE-LABELS</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>MOVE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>MULTIPLE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>MULTIPLY</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>NATIONAL</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>NATIONAL-EDITED</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>NATIVE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>NEGATIVE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>NESTED</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>NEXT</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>NO</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>NOT</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>NULLS</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>NUMBER</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>NUMERIC</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>NUMERIC-EDITED</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>OBJECT</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>OBJECT-COMPUTER</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>OBJECT-REFERENCE</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>OCCURS</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>OF</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>OFF</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>OMITTED</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>ON</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>OPEN</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>OPTIONAL</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>OPTIONS</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Word</td>
<td>Reserved</td>
<td>Standard only</td>
<td>Potential reserved words</td>
</tr>
<tr>
<td>--------------</td>
<td>----------</td>
<td>---------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>OR</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORDER</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORGANIZATION</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OTHER</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OVERFLOW</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OVERRIDE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PACKED-DECIMAL</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PADDING</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PAGE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PAGE-COUNTER</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>PASSWORD</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PERFORM</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PF</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>PH</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>PIC</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PICTURE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PLUS</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>POINTER</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POINTER-24</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>POINTER-31</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>POINTER-32</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POINTER-64</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>POSITION</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POSITIVE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRESENT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRINTING</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>PROCEDURE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROCEDURE-POINTER</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROCEDURES</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROCEED</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROCESSING</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROGRAM</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROGRAM-ID</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Word</td>
<td>Reserved</td>
<td>Standard only</td>
<td>Potential reserved words</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------</td>
<td>---------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>PROGRAM-POINTER</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>PROPERTY</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>PROTOTYPE</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>PURGE</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>QUEUE</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>QUOTE</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>QUOTES</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>RAISE</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>RAISING</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>RANDOM</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>RD</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>READ</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>READY</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>RECEIVE</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>RECORD</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>RECORDING</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>RECORDS</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>RECURSIVE</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>REDEFINES</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>REEL</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>REFERENCE</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>REFERENCES</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>RELATIVE</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>RELEASE</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>RELOAD</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>REMAINDER</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>REMOVAL</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>RENAMES</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>REPLACE</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>REPLACING</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>REPORT</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>REPORTING</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>REPORTS</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>REPOSITORY</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Word</td>
<td>Reserved</td>
<td>Standard only</td>
<td>Potential reserved words</td>
</tr>
<tr>
<td>--------------</td>
<td>----------</td>
<td>---------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>RERUN</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESERVE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESET</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESUME</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>RETRY</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>RETURN</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>RETURN-CODE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RETURNING</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>REVERSED</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REWRITE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>RF</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>RH</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>RIGHT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROUNDED</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RUN</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAME</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCREEN</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SD</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEARCH</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SECTION</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SECURITY</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEGMENT</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SEGMENT-LIMIT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SELECT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SELF</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEND</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SENTENCE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEPARATE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEQUENCE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEQUENTIAL</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SERVICE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SET</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHARING</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Word</td>
<td>Reserved</td>
<td>Standard only</td>
<td>Potential reserved words</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------</td>
<td>---------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>SHIFT-IN</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SHIFT-OUT</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SIGN</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SIZE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SKIP1</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SKIP2</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SKIP3</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SORT</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SORT-CONTROL</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SORT-CORE-SIZE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SORT-FILE-SIZE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SORT-MERGE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SORT-MESSAGE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SORT-MODE-SIZE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SORT-RETURN</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SOURCE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SOURCE-COMPUTER</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SOURCES</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SPACE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SPACES</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SPECIAL-NAMES</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SQL</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SQLIMS</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>STANDARD</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>STANDARD-1</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>STANDARD-2</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>START</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>STATUS</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>STOP</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>STRING</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SUB-QUEUE-1</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SUB-QUEUE-2</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SUB-QUEUE-3</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SUBTRACT</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Word</td>
<td>Reserved</td>
<td>Standard only</td>
<td>Potential reserved words</td>
</tr>
<tr>
<td>------------------</td>
<td>----------</td>
<td>---------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>SUM</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SUPER</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SUPPRESS</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SYMBOLIC</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SYNC</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SYNCHRONIZED</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SYSTEM-DEFAULT</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>TABLE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>TALLY</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>TALLYING</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>TAPE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>TERMINAL</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>TERMINATE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>TEST</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>TEXT</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>THAN</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>THEN</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>THROUGH</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>THRU</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>TIME</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>TIMES</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>TITLE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>TO</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>TOP</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>TRACE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>TRAILING</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>TRUE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>TYPE</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>TYPEDEF</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>UNIT</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>UNIVERSAL</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>UNLOCK</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>UNSTRING</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>UNTIL</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Word</td>
<td>Reserved</td>
<td>Standard only</td>
<td>Potential reserved words</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------</td>
<td>---------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>UP</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UPON</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USAGE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USER-DEFAULT</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>USING</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UTF-8</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VAL-STATUS</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>VALID</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VALIDATE</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>VALIDATE-STATUS</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>VALUE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VALUES</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VARYING</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VOLATILE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WHEN</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WHEN-COMPILED</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WITH</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WORDS</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WORKING-STORAGE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE-ONLY</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XML</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XML-CODE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XML-EVENT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XML-INFORMATION</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XML-NAMESPACE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XML-NAMESPACE-PREFIX</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XML-NAMESPACE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XML-NAMESPACE-PREFIX</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XML-NTEXT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XML-SCHEMA</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XML-TEXT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZERO</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 70. *Reserved words* (continued)

<table>
<thead>
<tr>
<th>Word</th>
<th>Reserved</th>
<th>Standard only</th>
<th>Potential reserved words</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZEROES</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>ZEROS</td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

**Related references**
Appendix F, “Context-sensitive words,” on page 607
Appendix F. Context-sensitive words

A context-sensitive word is a COBOL word that is reserved only in the general formats in which it is specified. If a context-sensitive word is used where the context-sensitive word is permitted in the general format, the word is treated as a keyword; otherwise it is treated as a user-defined word.

Table 71. Context-sensitive words

<table>
<thead>
<tr>
<th>Context-sensitive word</th>
<th>Language construct or context</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE-LENGTH</td>
<td>PICTURE clause</td>
</tr>
<tr>
<td>CYCLE</td>
<td>EXIT statement</td>
</tr>
<tr>
<td>FIXED</td>
<td>CALL ... USING parameters</td>
</tr>
<tr>
<td>INITIALIZED</td>
<td>ALLOCATE statement</td>
</tr>
<tr>
<td>INTRINSIC</td>
<td>REPOSITORY paragraph</td>
</tr>
<tr>
<td>LOC</td>
<td>ALLOCATE statement</td>
</tr>
<tr>
<td>NAME</td>
<td>JSON GENERATE statement</td>
</tr>
<tr>
<td></td>
<td>JSON PARSE statement</td>
</tr>
<tr>
<td></td>
<td>XML GENERATE statement</td>
</tr>
<tr>
<td>PARAGRAPH</td>
<td>EXIT statement</td>
</tr>
<tr>
<td>RECURSIVE</td>
<td>PROGRAM-ID paragraph</td>
</tr>
<tr>
<td>YYYYDDD</td>
<td>ACCEPT statement</td>
</tr>
<tr>
<td>YYYYYMDDD</td>
<td>ACCEPT statement</td>
</tr>
</tbody>
</table>

Related references
“User-defined words” on page 10

Appendix E, “Reserved words,” on page 589
Appendix G. ASCII considerations

The compiler supports the American National Standard Code for Information Interchange (ASCII) for magnetic tape files. Thus, the programmer can create and process tape files recorded in accordance with several standards.

The standards are:

- American National Standard Code for Information Interchange, X3.4-1977
- American National Standard Magnetic Tape Labels for Information Interchange, X3.27-1978
- American National Standard Recorded Magnetic Tape for Information Interchange (800 CPI, NRZI), X3.22-1967

Single-byte ASCII-encoded tape files, when read into the system, are automatically translated in the buffers into single-byte EBCDIC. Internal manipulation of data is performed exactly as if the ASCII files were single-byte EBCDIC-encoded files. For an output file, the system translates the EBCDIC characters into single-byte ASCII in the buffers before writing the file on tape. Therefore, there are special considerations concerning ASCII-encoded files when they are processed in COBOL.

This appendix also applies (with appropriate modifications) to the International Reference Version of the ISO 7-bit code defined in International Standard 646, 7-Bit Coded Character Set for Information Processing Interchange (ISCII). The ISCII code set differs from ASCII only in the graphic representation of two code points:

- Ordinal number 37, which is a dollar sign in ASCII, but a lozenge in ISCII
- Ordinal number 127, which is a tilde (~) in ASCII, but an overline (or optionally a tilde) in ISCII.

The following paragraphs discuss the special considerations concerning ASCII-encoded (or ISCII-encoded) files. The information given for STANDARD-1 also applies to STANDARD-2 except where otherwise specified.

ENVIRONMENT DIVISION

In the ENVIRONMENT DIVISION, the OBJECT-COMPUTER, SPECIAL-NAMES, and FILE-CONTROL paragraphs are affected by the use of ASCII-encoded files.

OBJECT-COMPUTER and SPECIAL-NAMES paragraphs

When at least one file in the program is an ASCII-encoded file, the alphabet-name clause of the SPECIAL-NAMES paragraph must be specified; the alphabet-name must be associated with STANDARD-1 or STANDARD-2 (for ASCII or ISCII collating sequence or CODE SET, respectively).

When alphanumeric comparisons within the object program are to use the ASCII collating sequence, the PROGRAM COLLATING SEQUENCE clause of the OBJECT-COMPUTER paragraph must be specified; the alphabet-name used must also be specified as an alphabet-name in the SPECIAL-NAMES paragraph, and associated with STANDARD-1. For example:

```
Object-computer.  IBM-system
Program collating sequence is ASCII-sequence.
Special-names.  Alphabet ASCII-sequence is standard-1.
```

When both clauses are specified, the ASCII collating sequence is used in this program to determine the truth value of the following alphanumeric comparisons:

- Those explicitly specified in relation conditions
- Those explicitly specified in condition-name conditions
• Any alphanumeric sort or merge keys (unless the COLLATING SEQUENCE phrase is specified in the
MERGE or SORT statement).

When the PROGRAM COLLATING SEQUENCE clause is omitted, the EBCDIC collating sequence is used for
such comparisons.

The PROGRAM COLLATING SEQUENCE clause, in conjunction with the alphabet-name clause, can be
used to specify EBCDIC alphanumeric comparisons for an ASCII-encoded tape file or ASCII alphanumeric
comparisons for an EBCDIC-encoded tape file.

The literal option of the alphabet-name clause can be used to process internal data in a collating
sequence other than NATIVE or STANDARD-1.

FILE-CONTROL paragraph

For ASCII files, the ASSIGN clause assignment-name has the following format:

<table>
<thead>
<tr>
<th>Format: assignment-name for QSAM files</th>
</tr>
</thead>
<tbody>
<tr>
<td>label-</td>
</tr>
</tbody>
</table>

The file must be a QSAM file assigned to a magnetic tape device.

**label-**

Documents the device and device class to which a file is assigned. If specified, it must end with a
hyphen.

**s-**

The organization field. Optional for QSAM files, which always have sequential organization.

**name**

A required one-character to eight-character field that specifies the external name for this file.

I-O-CONTROL paragraph

The assignment-name in a RERUN clause must not specify an ASCII-encoded file.

ASCII-encoded files that contain checkpoint records cannot be processed.

DATA DIVISION

In the DATA DIVISION, there are special considerations for the FD entry and for data description entries.

For each logical file defined in the ENVIRONMENT DIVISION, there must be a corresponding FD entry and
level-01 record description entry in the FILE SECTION of the DATA DIVISION.

FD Entry: CODE-SET clause

The FD Entry for an ASCII-encoded file must contain a CODE-SET clause; the alphabet-name must be
associated with STANDARD-1 (for the ASCII code set) in the SPECIAL-NAMES paragraph.

For example:

```
Special-names. Alphabet ASCII-sequence is standard-1.
FD  ASCII-file label records standard
    Recording mode is f
    Code-set is ASCII-sequence.
```
**Data description entries**

For ASCII files, the data description considerations listed in the topic apply.

- PICTURE clause specifications are valid for the following categories of data:
  - Alphabetic
  - Alphanumeric
  - Alphanumeric-edited
  - Numeric
  - Numeric-edited
- For signed numeric items, the SIGN clause with the SEPARATE CHARACTER phrase must be specified.
- For the USAGE clause, only the DISPLAY phrase is valid.

**PROCEDURE DIVISION**

An ASCII-collated sort or merge operation can be specified in two ways as described in the topic.

- Through the PROGRAM COLLATING SEQUENCE clause in the OBJECT-COMPUTER paragraph. In this case, the ASCII collating sequence is used for alphanumeric comparisons explicitly specified in relation conditions and condition-name conditions.
- Through the COLLATING SEQUENCE phrase of the SORT or MERGE statement. In this case, only this sort or merge operation uses the ASCII collating sequence.

In either case, alphabet-name must be associated with STANDARD-1 (for ASCII collating sequence) in the SPECIAL-NAMES paragraph.

For this sort or merge operation, the COLLATING SEQUENCE phrase of the SORT or MERGE statement takes precedence over the PROGRAM COLLATING SEQUENCE clause in the OBJECT-COMPUTER paragraph.

If both the PROGRAM COLLATING SEQUENCE clause and the COLLATING SEQUENCE phrase are omitted (or if the one in effect specifies an EBCDIC collating sequence), the sort or merge is performed using the EBCDIC collating sequence.
Appendix H. Industry specifications

Enterprise COBOL supports various industry standards.

- ISO COBOL standards
  - ISO 1989:1985, Programming languages - COBOL
    ISO 1989:1985 is identical to ANSI INCITS 23-1985 (R2001), Programming Languages - COBOL
    ISO/IEC 1989/AMD1:1992 is identical to ANSI INCITS 23a-1989 (R2001), Programming Languages - Intrinsic Function Module for COBOL
  - ISO/IEC 1989/AMD2:1994, Programming languages - Correction and clarification amendment for COBOL
    ISO/IEC 1989/AMD2:1994 is identical to ANSI INCITS 23b-1993 (R2001), Programming Language - Correction Amendment for COBOL

  All required modules are supported at the highest level defined by the standard.

  The following optional modules of the standard are supported:
  - Intrinsic Functions (1 ITR 0,1)
  - Debug (1 DEB 0,2)
  - Segmentation (2 SEG 0,2)

  The Report Writer optional module of the standard is supported with the optional IBM COBOL Report Writer Precompiler and Libraries (5798-DYR).

  The following optional modules of the standard are not supported:
  - Communications
  - Debug (2 DEB 0,2)

    For a list of 2002 COBOL Standard features that are implemented since Enterprise COBOL Version 3, see Appendix I, “2002/2014 COBOL Standard features implemented in Enterprise COBOL Version 3 or later versions,” on page 615.

- ANSI COBOL standards
  - ANSI INCITS 23-1985 (R2001), Programming Languages - COBOL
  - ANSI INCITS 23a-1989 (R2001), Programming Languages - Intrinsic Function Module for COBOL
  - ANSI INCITS 23b-1993 (R2001), Programming Language - Correction Amendment for COBOL

  All required modules are supported at the highest level defined by the standard.

  The following optional modules of the standard are supported:
  - Intrinsic Functions (1 ITR 0,1)
  - Debug (1 DEB 0,2)
- Segmentation (2 SEG 0,2)
The following optional modules of the standard are not supported:
- Communications
- Debug (2 DEB 0,2)
  - ANSI INCITS 1989-2014, *Information technology - Programming languages, their environments and system software interfaces - Programming language COBOL* (partial support)

- International Reference Version of *ISO/IEC 646, 7-Bit Coded Character Set for Information Interchange*
- The 7-bit coded character set defined in *American National Standard X3.4-1977, Code for Information Interchange*

Enterprise COBOL has the following restriction related to COBOL standards:

- OPEN EXTEND is not supported for ASCII-encoded tapes (CODE-SET STANDARD-1 or STANDARD-2).

See *Option settings for 85 COBOL Standard conformance* in the *Enterprise COBOL Programming Guide* for specification of the compiler options and Language Environment runtime options that are required to support the above standards.
Appendix I. 2002/2014 COBOL Standard features implemented in Enterprise COBOL Version 3 or later versions

Beginning with Enterprise COBOL Version 3, substantive changes are implemented according to the 2002 COBOL Standard and 2014 COBOL Standard. This topic lists those changes that will potentially affect existing COBOL programs and those changes that will not affect existing COBOL programs.

<table>
<thead>
<tr>
<th>Features</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL-NAMES paragraph, CURRENCY SIGN clause</td>
<td>The letters 'N', 'n', 'E' and 'e' are now used as picture symbols. The letter 'N' or 'E' can no longer be specified as a currency sign in the CURRENCY SIGN clause.</td>
</tr>
<tr>
<td>SAME clause</td>
<td>File-names referenced in a SAME clause shall be described in the FILE-CONTROL paragraph of the source element that contains the SAME clause.</td>
</tr>
<tr>
<td>Executable code production</td>
<td>The implementor is not required to produce an executable object program if a fatal exception condition for which checking is not enabled is detected by the compiler.</td>
</tr>
<tr>
<td>Exponentiation</td>
<td>If the value of an expression to be raised to a power is less than zero, the following condition shall be true for the exponent: (FUNCTION FRACTION-PART exponent) = 0. Otherwise, the EC-SIZE-EXPONENTIATION exception exists and the size error condition is raised.</td>
</tr>
<tr>
<td>CORRESPONDING order</td>
<td>The order of execution of the implied statements created for corresponding operands for ADD, MOVE, and SUBTRACT with the CORRESPONDING phrase is defined to be the order of the specification of the operands in the group following the word CORRESPONDING. The previous COBOL standard did not specify an order. In addition, the evaluation of subscripts for the implied statements is done only once, at the start of the execution of the actual ADD, MOVE, or SUBTRACT statement. The previous COBOL standard implied this, but was not specific.</td>
</tr>
<tr>
<td>Sending and receiving operands</td>
<td>The terms sending operand and receiving operand have been defined.</td>
</tr>
<tr>
<td>Incompatible data clarification</td>
<td>The conditions that cause the incompatible data condition are specified explicitly. They are limited to boolean, numeric, and numeric-edited sending operands.</td>
</tr>
</tbody>
</table>
Table 72. 2002/2014 COBOL Standard features implemented in V3 or later versions that will potentially affect existing programs (continued)

<table>
<thead>
<tr>
<th>Features</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVALUATE statement, sequence of execution</td>
<td>The sequence of evaluation of selection subjects and objects is now defined to be from left to right and selection objects are evaluated as each WHEN phrase is processed. When a WHEN phrase is selected, no more selection objects are evaluated.</td>
</tr>
<tr>
<td>SIZE ERROR phrase for intermediate results</td>
<td>If the SIZE ERROR phrase is specified for a statement containing an arithmetic expression and a size error is detected during evaluation of an intermediate result of that arithmetic expression, the size error condition is set to exist.</td>
</tr>
</tbody>
</table>

Table 73. 2002/2014 COBOL Standard features implemented in V3 or later versions that will not affect existing programs

<table>
<thead>
<tr>
<th>Features</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCEPT statement four-digit year</td>
<td>The capability to access the four-digit year of the Gregorian calendar is added to the ACCEPT statement.</td>
</tr>
<tr>
<td>Apostrophe as quotation mark</td>
<td>The apostrophe character and the quotation mark can be used in the opening and closing delimiters of alphanumeric, boolean, and national literals. A given literal can use either the apostrophe or the quotation mark, but both the starting and ending characters are required to be the same. Whichever character is used, it is necessary to double that character to represent one occurrence of the character within the literal. Both formats can be used in a single source element.</td>
</tr>
<tr>
<td>Arithmetic operators</td>
<td>No space is required between a left parenthesis and unary operator or between a unary operator and a left parenthesis.</td>
</tr>
<tr>
<td>AT END phrase</td>
<td>The AT END phrase of the READ statement does not have to be specified if there is no applicable USE statement.</td>
</tr>
<tr>
<td>BINARY and floating point data</td>
<td>Two new representations of numeric data type are introduced, a binary representation that holds data in a machine-specific way and is not restricted to decimal ranges of values, and a floating-point representation. The floating-point type exists both in a numeric form, with a machine-specific representation, and in a numeric-edited form.</td>
</tr>
<tr>
<td>CALL argument level numbers (any)</td>
<td>CALL arguments can be elementary or groups with any level number. Formerly, they had to be elementary or have a level number of 1 or 77.</td>
</tr>
<tr>
<td>CALL BY CONTENT parameter difference</td>
<td>A parameter passed by content does not have to have the same description as the matching parameter in the called program.</td>
</tr>
<tr>
<td>Features</td>
<td>Notes</td>
</tr>
<tr>
<td>----------</td>
<td>-------</td>
</tr>
<tr>
<td>CALL parameter length difference</td>
<td>The size of an argument in the USING phrase of the CALL statement can be greater than the size of the matching formal parameter if either the argument or the formal parameter is a group item. Formerly, the sizes were required to be the same.</td>
</tr>
<tr>
<td>CALL recursively</td>
<td>The capability of calling an active COBOL program has been added to COBOL.</td>
</tr>
<tr>
<td>COBOL words reserved in context</td>
<td>Certain words added to the COBOL standard are reserved only in the contexts in which they are specified and were not added to the reserved word list. See Appendix F, “Context-sensitive words,” on page 607 for details.</td>
</tr>
<tr>
<td>CODE clause (Report Writer)</td>
<td>The identifier phrase is provided in the CODE clause in the report description entry.</td>
</tr>
<tr>
<td>COLUMN clause</td>
<td>A relative form is provided using PLUS integer, by analogy with LINE; COLUMN RIGHT and COLUMN CENTER are provided, allowing alignment of a printable item at the right or center; and COL, COLS, and COLUMNS are allowed as synonyms for COLUMN.</td>
</tr>
<tr>
<td>COLUMN, LINE, SOURCE, and VALUE clauses</td>
<td>These clauses can have more than one operand in a report group description entry.</td>
</tr>
<tr>
<td>Comment lines anywhere in a compilation group</td>
<td>Comment lines can be written as any line in a compilation group, including before the identification division header.</td>
</tr>
<tr>
<td>Compiler directives</td>
<td>• The CALLINTERFACE compiler directive specifies the interface convention for CALL and SET statements. • The DEFINE directive defines or undefines a compilation variable. • The EVALUATE directive provides a multi-branch method of choosing the source lines to include in a compilation group. • The IF directive provides for a one-way or two-way conditional compilation.</td>
</tr>
<tr>
<td>Control data name</td>
<td>This is allowed to be omitted on TYPE CH/CF if only one control is defined.</td>
</tr>
<tr>
<td>Conversion from two-digit year to four-digit year</td>
<td>There are three functions for converting from a two-digit year to a four-digit year. DATE-TO-YYYYMMDD, DAY-TO-YYYYDDD, and YEAR-TO-YYYY convert from YYnnnn to YYYYnnnn, YYnnn to YYYYnnn, and YY to YYYY, respectively.</td>
</tr>
<tr>
<td>Features</td>
<td>Notes</td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>COPY statement</td>
<td>An alphanumeric literal can be specified in place of text-name-1 or library-name-1.</td>
</tr>
<tr>
<td></td>
<td>When more than one COBOL library is referenced, text-name need not be qualified when the library text resides in the default library.</td>
</tr>
<tr>
<td></td>
<td>The ability to nest COPY statements is provided.</td>
</tr>
<tr>
<td></td>
<td>Library text incorporated as a result of processing a COPY statement without a REPLACING phrase can contain a COPY statement without a REPLACING phrase.</td>
</tr>
<tr>
<td></td>
<td>A SUPPRESS PRINTING phrase is added to the COPY statement to suppress listing of library text incorporated as a result of COPY statement processing.</td>
</tr>
<tr>
<td>COPY and REPLACE statement partial word replacement</td>
<td>LEADING and TRAILING phrases of the REPLACE statement and the REPLACING phrase of the COPY statement allow replacement of partial words in source text and library text. This is useful for prefixing and postfixing names.</td>
</tr>
<tr>
<td>Currency sign extensions</td>
<td>The CURRENCY SIGN clause has been extended to allow for national characters and for multiple distinct currency signs, which can have any length.</td>
</tr>
<tr>
<td>DISPLAY statement</td>
<td>If the literal in a DISPLAY statement is numeric, it can be signed.</td>
</tr>
<tr>
<td>DIVIDE BY zero</td>
<td>DIVIDE BY zero conforms to the 2002 COBOL Standard. Nothing has changed in IBM COBOL, but the Standard is changed so that now Enterprise COBOL is conforming. (Enterprise COBOL was not conforming to the 85 COBOL Standard for DIVIDE BY zero.)</td>
</tr>
<tr>
<td>Dynamic-length elementary items</td>
<td>The addition of the DYNAMIC LENGTH clause provides the ability to describe a data item of varying size.</td>
</tr>
<tr>
<td>Dynamic storage allocation</td>
<td>ALLOCATE and FREE statements are provided for obtaining storage dynamically. This storage is addressed by pointer data items.</td>
</tr>
<tr>
<td>EXIT statement</td>
<td>The ability to immediately exit an inline PERFORM statement, a paragraph, or a section has been added.</td>
</tr>
<tr>
<td>EXIT PROGRAM allowed as other than last statement</td>
<td>EXIT PROGRAM is allowed to appear as other than the last statement in a consecutive sequence of imperative statements.</td>
</tr>
<tr>
<td>FILLER</td>
<td>FILLER is allowed in the report section.</td>
</tr>
<tr>
<td>Features</td>
<td>Notes</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| Floating comment delimiters          | • The floating comment indicator (*>) indicates a comment line if it is the first character string in the program-text area (Area A plus Area B), or indicates an inline comment if it is after one or more character strings in the program-text area.  
• The compiler directive indicator (>>) indicates a compiler directive line when followed by a compiler directive word - with or without an intervening space. |
| FUNCTION All INTRINSIC               | The REPOSITORY paragraph supports a function specifier, and for some intrinsic functions, the keyword FUNCTION is optional when a function is included in the REPOSITORY paragraph. |
| GOBACK statement                     | A GOBACK statement has been added that always returns control, either to the operating system or to the calling runtime element.                                                                         |
| Hexadecimal literals                 | The ability was added to specify alphanumeric, boolean, and national literals using hexadecimal (radix 16) notation.                                                                                   |
| Inline comment                       | A comment can be written on a line following any character-strings of the source text or library text that are written on that line. An inline comment is introduced by the two contiguous characters ‘*>’.                                |
| Index data item                      | The definition of an index data item can include the SYNCHRONIZED clause.                                                                                                                             |
| INITIALIZE statement, FILLER phrase | FILLER data items can be initialized with the INITIALIZE statement.                                                                                                                                    |
| INITIALIZE statement, VALUE phrase  | A VALUE phrase can be specified in the INITIALIZE statement to cause initialization of elementary data items to the literal specified in the VALUE clause of the associated data description entry.                        |
| Intrinsic function facility          | Previously, the intrinsic function facility was a separate module and its implementation was optional. The intrinsic function facility is integrated into the specification and it shall be implemented by a conforming implementation |
### Table 73. 2002/2014 COBOL Standard features implemented in V3 or later versions that will not affect existing programs (continued)

<table>
<thead>
<tr>
<th>Features</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>New intrinsic functions</td>
<td>New intrinsic functions are added:</td>
</tr>
<tr>
<td></td>
<td>• ABS</td>
</tr>
<tr>
<td></td>
<td>• BYTE-LENGTH</td>
</tr>
<tr>
<td></td>
<td>• E</td>
</tr>
<tr>
<td></td>
<td>• EXP</td>
</tr>
<tr>
<td></td>
<td>• EXP10</td>
</tr>
<tr>
<td></td>
<td>• DATE-TO-YYYYMMDD</td>
</tr>
<tr>
<td></td>
<td>• DAY-TO-YYYYDDDD</td>
</tr>
<tr>
<td></td>
<td>• DISPLAY-OF</td>
</tr>
<tr>
<td></td>
<td>• NATIONAL-OF</td>
</tr>
<tr>
<td></td>
<td>• NUMVAL-F</td>
</tr>
<tr>
<td></td>
<td>• PI</td>
</tr>
<tr>
<td></td>
<td>• SIGN</td>
</tr>
<tr>
<td></td>
<td>• TEST-NUMVAL</td>
</tr>
<tr>
<td></td>
<td>• TEST-NUMVAL-C</td>
</tr>
<tr>
<td></td>
<td>• TEST-NUMVAL-F</td>
</tr>
<tr>
<td></td>
<td>• TRIM</td>
</tr>
<tr>
<td></td>
<td>• YEAR-TO-YYYY</td>
</tr>
<tr>
<td>INVALID KEY phrase</td>
<td>The INVALID KEY phrase does not have to be specified if there is no applicable USE statement.</td>
</tr>
<tr>
<td>LOCAL-STORAGE SECTION</td>
<td>A facility was added to define data that is set to its initial values each time a function, method, or program is activated. Each instance of this source element has its own copy of this data.</td>
</tr>
<tr>
<td>National character handling</td>
<td>The capability is added for using large character sets, such as ISO/IEC 10646-1, in source text and library text and in data at execution time. Class national and categories national and national-edited are specified by picture character-strings containing the symbol 'N'; national literals are identified by a separator N&quot;, N', NX&quot;, or NX'. Usage national specifies representation of data in a national character set. User-defined words can contain extended letters. Processing of data of class national is comparable to processing data of class alphanumeric, though there are some minor differences. Conversions between data of classes alphanumeric and national are provided by intrinsic functions.</td>
</tr>
<tr>
<td>Object orientation</td>
<td>Support for object-oriented programming has been added.</td>
</tr>
<tr>
<td>OCCURS clause</td>
<td>Repetition vertically or horizontally and a STEP phrase are added for Report Writer.</td>
</tr>
<tr>
<td>Optional word OF</td>
<td>Allowed after SUM.</td>
</tr>
<tr>
<td>Features</td>
<td>Notes</td>
</tr>
<tr>
<td>----------</td>
<td>-------</td>
</tr>
<tr>
<td>Optional word FOR and ON</td>
<td>Allowed after TYPE CH or CF.</td>
</tr>
<tr>
<td>OR PAGE phrase of the CONTROL HEADING phrase</td>
<td>This enables the control heading group to be printed at the top of each page and after a control break.</td>
</tr>
<tr>
<td>PAGE FOOTING report group</td>
<td>Such a group is allowed to have all relative LINE clauses.</td>
</tr>
<tr>
<td>PAGE LIMIT clause</td>
<td>New COLUMNS phrase is provided to define maximum number of horizontal print positions in each report line and a LAST CONTROL HEADING phrase was added.</td>
</tr>
<tr>
<td>Paragraph-name</td>
<td>A paragraph-name is not required at the beginning of the procedure division or a section.</td>
</tr>
<tr>
<td>PERFORM statement</td>
<td>A common exit for multiple active PERFORM statements is allowed.</td>
</tr>
<tr>
<td>PICTURE clause</td>
<td>The maximum number of characters that can be specified in a picture character-string is increased from 30 to 50. The symbol 'E' can be used in a PICTURE character-string to specify a floating-point numeric-edited data item. The symbol 'N' can be used in a PICTURE character-string to specify a national or a national-edited data item. When the last symbol of a PICTURE character-string is a period or a comma, one or more spaces can precede the following separator period. It was unclear in the previous standard whether a space could precede the separator period in this context.</td>
</tr>
<tr>
<td>PLUS and MINUS</td>
<td>The symbol + or - is synonymous with PLUS or MINUS, respectively, in the COLUMN and LINE clauses.</td>
</tr>
<tr>
<td>Pointer data</td>
<td>A new class of data is introduced, a pointer type that holds data and program addresses in a machine-specific or system-specific way.</td>
</tr>
<tr>
<td>PRESENT WHEN clause</td>
<td>The PRESENT WHEN clauses allows lines and printable items, or groups of them, to be printed or not, depending on the truth value of a condition.</td>
</tr>
<tr>
<td>Program-names as literals</td>
<td>The option to specify a literal as the program-name to be externalized was added for names that are not valid COBOL words or need to be case-sensitive.</td>
</tr>
<tr>
<td>RECORD KEY and ALTERNATE RECORD KEY</td>
<td>Keys for indexed files can be made up from more than one component.</td>
</tr>
<tr>
<td>Report Writer</td>
<td>Previously, the Report Writer was a separate module and its implementation was optional. The Report Writer facility is integrated into the specification and it shall be implemented by a conforming implementation.</td>
</tr>
</tbody>
</table>
Table 73. 2002/2014 COBOL Standard features implemented in V3 or later versions that will not affect existing programs (continued)

<table>
<thead>
<tr>
<th>Features</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report Writer national character support</td>
<td>The capability of printing national characters and alphanumeric characters in a report is provided.</td>
</tr>
<tr>
<td>Function specifier</td>
<td>The REPOSITORY paragraph supports function specifier, and for some intrinsic functions, the keyword FUNCTION is optional when a function is included in the REPOSITORY paragraph.</td>
</tr>
<tr>
<td>SIGN clause in a report description entry</td>
<td>The SEPARATE phrase is no longer required in a report description entry and the SIGN clause can be specified at the group level.</td>
</tr>
<tr>
<td>SORT statement</td>
<td>A SORT statement can be used to sort a table. This sort can be done using the fields specified in the KEY phrase defining the table, by using the entire table element as the key, or by using specified key data items.</td>
</tr>
<tr>
<td></td>
<td>GIVING files in a SORT statement can now be specified in the same SAME RECORD AREA clause.</td>
</tr>
<tr>
<td>SOURCE clause in a report description entry</td>
<td>The sending operand can be a function-identifier.</td>
</tr>
<tr>
<td></td>
<td>An arithmetic-expression is allowed as an operand and a ROUNDED phrase was added.</td>
</tr>
<tr>
<td>SUM clause in a report description entry</td>
<td>The SUM clause was extended in the following ways:</td>
</tr>
<tr>
<td></td>
<td>• Extension to total a repeating entry.</td>
</tr>
<tr>
<td></td>
<td>• Now allowed in any TYPE of report group, not only control footing.</td>
</tr>
<tr>
<td></td>
<td>• SUM of arithmetic-expression format.</td>
</tr>
<tr>
<td></td>
<td>• Checks for overflow of a sum counter during totalling.</td>
</tr>
<tr>
<td></td>
<td>• Any numeric report section item can be totalled, not just another sum counter.</td>
</tr>
<tr>
<td></td>
<td>• ROUNDED phrase.</td>
</tr>
<tr>
<td>Underscore (_) character</td>
<td>The basic special characters of the COBOL character repertoire have been expanded to include the underscore (_) character, which can be used in the formation of COBOL words.</td>
</tr>
<tr>
<td>UNSTRING statement</td>
<td>The sending operand can be reference modified.</td>
</tr>
<tr>
<td>USE BEFORE REPORTING</td>
<td>The effect of GLOBAL in a report description and a USE declarative is further elucidated.</td>
</tr>
<tr>
<td>VALUE clause, WHEN SET TO FALSE phrase in data division</td>
<td>The WHEN SET TO FALSE phrase allows specification of a FALSE condition value. This value is moved to the associated conditional variable when the SET TO FALSE statement is executed for the associated condition-name.</td>
</tr>
<tr>
<td>Features</td>
<td>Notes</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>VARYING clause</td>
<td>A VARYING clause is provided in the validate and Report Writer facilities to be used with an OCCURS clause.</td>
</tr>
</tbody>
</table>
| WITH RESET phrase | This was added to the NEXT PAGE phrase of the NEXT GROUP clause to reset PAGE-COUNTER back to 1.  
**Note:** REPORT WRITER features such as CODE clause, COLUMN clause, control data name, CONTROL HEADING, PAGE FOOTING, PAGE LIMIT, SUM clause, and report description entries are supported via the optional Report Writer Precompiler. |
Accessibility features assist users who have a disability, such as restricted mobility or limited vision, to use information technology content successfully. The accessibility features in z/OS provide accessibility for Enterprise COBOL for z/OS.

Accessibility features

z/OS includes the following major accessibility features:

- Interfaces that are commonly used by screen readers and screen-magnifier software
- Keyboard-only navigation
- Ability to customize display attributes such as color, contrast, and font size

z/OS uses the latest W3C Standard, WAI-ARIA 1.0 (http://www.w3.org/TR/wai-aria/), to ensure compliance to US Section 508 (http://www.access-board.gov/guidelines-and-standards/communications-and-it/about-the-section-508-standards/section-508-standards) and Web Content Accessibility Guidelines (WCAG) 2.0 (http://www.w3.org/TR/WCAG20/). To take advantage of accessibility features, use the latest release of your screen reader in combination with the latest web browser that is supported by this product.

The Enterprise COBOL for z/OS online product documentation in IBM Knowledge Center is enabled for accessibility. The accessibility features of IBM Knowledge Center are described at http://www.ibm.com/support/knowledgecenter/en/about/releasenotes.html.

Keyboard navigation

Users can access z/OS user interfaces by using TSO/E or ISPF.

Users can also access z/OS services by using IBM Developer for z/OS.

For information about accessing these interfaces, see the following publications:

- z/OS TSO/E Primer (http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/ikj4p120)
- IBM Developer for z/OS Knowledge Center (http://www.ibm.com/support/knowledgecenter/SSQ2R2/rdz_welcome.html?lang=en)

These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

Interface information

The Enterprise COBOL for z/OS online product documentation is available in IBM Knowledge Center, which is viewable from a standard web browser.

PDF files have limited accessibility support. With PDF documentation, you can use optional font enlargement, high-contrast display settings, and can navigate by keyboard alone.

To enable your screen reader to accurately read syntax diagrams, source code examples, and text that contains period or comma PICTURE symbols, you must set the screen reader to speak all punctuation.
Assistive technology products work with the user interfaces that are found in z/OS. For specific guidance information, see the documentation for the assistive technology product that you use to access z/OS interfaces.

**Related accessibility information**

In addition to standard IBM help desk and support websites, IBM has established a TTY telephone service for use by deaf or hard of hearing customers to access sales and support services:

TTY service
800-IBM-3383 (800-426-3383)
(within North America)

**IBM and accessibility**

For more information about the commitment that IBM has to accessibility, see [IBM Accessibility](http://www.ibm.com/able).
This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who want to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Intellectual Property Dept. for Rational Software
IBM Corporation
5 Technology Park Drive
Westford, MA 01886
U.S.A.
Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM’s future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1991, 2020.

PRIVACY POLICY CONSIDERATIONS:

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, or to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at http://www.ibm.com/privacy and IBM's Online Privacy Statement at http://www.ibm.com/privacy/details in the section entitled "Cookies, Web Beacons and Other Technologies,"

Programming interface information

This Language Reference documents intended Programming Interfaces that allow the customer to write programs to obtain the services of Enterprise COBOL.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.html.

Intel is a registered trademark of Intel Corporation in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other product and service names might be trademarks of IBM or other companies.
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:


American National Standard definitions are preceded by an asterisk (*).

A

* abbreviated combined relation condition
The combined condition that results from the explicit omission of a common subject or a common subject and common relational operator in a consecutive sequence of relation conditions.

abend
Abnormal termination of a program.

above the 16 MB line
Storage above the so-called 16 MB line (or boundary) but below the 2 GB bar. This storage is addressable only in 31-bit mode. Before IBM introduced the MVS/XA architecture in the 1980s, the virtual storage for a program was limited to 16 MB. Programs that have been compiled with a 24-bit mode can address only 16 MB of space, as though they were kept under an imaginary storage line. Since VS COBOL II, a program that has been compiled with a 31-bit mode can be above the 16 MB line.

* access mode
The manner in which records are to be operated upon within a file.

* actual decimal point
The physical representation, using the decimal point characters period (.) or comma (,), of the decimal point position in a data item.

actual document encoding
For an XML document, one of the following encoding categories that the XML parser determines by examining the first few bytes of the document:

- ASCII
- EBCDIC
- UTF-8
- UTF-16, either big-endian or little-endian
- Other unsupported encoding
- No recognizable encoding

* alphabet-name
A user-defined word, in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION, that assigns a name to a specific character set or collating sequence or both.
**alphabetic character**
A letter or a space character.

**alphanumeric character position**
See *character position*.

**alphabetic data item**
A data item that is described with a PICTURE character string that contains only the symbol A. An alphabetic data item has USAGE DISPLAY.

**alphanumeric character**
Any character in the single-byte character set of the computer.

**alphanumeric data item**
A general reference to a data item that is described implicitly or explicitly as USAGE DISPLAY, and that has category alphanumeric, alphanumeric-edited, or numeric-edited.

**alphanumeric-edited data item**
A data item that is described by a PICTURE character string that contains at least one instance of the symbol A or X and at least one of the simple insertion symbols B, 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**
A numeric literal, an identifier representing a numeric elementary item, such identifiers and literals separated by arithmetic operators, two arithmetic expressions separated by an arithmetic operator, or an arithmetic expression enclosed in parentheses.

**arithmetic operation**
The process caused by the execution of an arithmetic statement, or the evaluation of an arithmetic expression, that results in a mathematically correct solution to the arguments presented.

**arithmetic operator**
A single character, or a fixed two-character combination that belongs to the following set:

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
</tbody>
</table>
### Character Meaning

* Multiplication
\(/\) Division
** Exponentiation

### * arithmetic statement
A statement that causes an arithmetic operation to be executed. The arithmetic statements are ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT.

### array
An aggregate that consists of data objects, each of which can be uniquely referenced by subscripting. An array is roughly analogous to a COBOL table.

### * ascending key
A key upon the values of which data is ordered, starting with the lowest value of the key up to the highest value of the key, in accordance with the rules for comparing data items.

### ASCII
American National Standard Code for Information Interchange. The standard code uses a coded character set that is based on 7-bit coded characters (8 bits including parity check). The standard is used for information interchange between data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters.

IBM has defined an extension to ASCII (characters 128-255).

### ASCII DBCS
See double-byte ASCII.

### assignment-name
A name that identifies the organization of a COBOL file and the name by which it is known to the system.

### * assumed decimal point
A decimal point position that does not involve the existence of an actual character in a data item. The assumed decimal point has logical meaning but no physical representation.

### AT END condition
A condition that is caused during the execution of a READ, RETURN, or SEARCH statement under certain conditions:

- A READ statement runs on a sequentially accessed file when no next logical record exists in the file, or when the number of significant digits in the relative record number is larger than the size of the relative key data item, or when an optional input file is not available.
- A RETURN statement runs when no next logical record exists for the associated sort or merge file.
- A SEARCH statement runs when the search operation terminates without satisfying the condition specified in any of the associated WHEN phrases.

### B

### basic character set
The basic set of characters used in writing words, character-strings, and separators of the language. The basic character set is implemented in single-byte EBCDIC. The extended character set includes DBCS characters, which can be used in comments, literals, and user-defined words.

Synonymous with COBOL character set in the 85 COBOL Standard.

### big-endian
The default format that the mainframe and the AIX® workstation use to store binary data and UTF-16 characters. In this format, the least significant byte of a binary data item is at the highest address and the least significant byte of a UTF-16 character is at the highest address. Compare with little-endian.
binary item
A numeric data item that is represented in binary notation (on the base 2 numbering system). The decimal equivalent consists of the decimal digits 0 through 9, plus an operational sign. The leftmost bit of the item is the operational sign.

binary search
A dichotomizing search in which, at each step of the search, the set of data elements is divided by two; some appropriate action is taken in the case of an odd number.

* block
A physical unit of data that is normally composed of one or more logical records. For mass storage files, a block can contain a portion of a logical record. The size of a block has no direct relationship to the size of the file within which the block is contained or to the size of the logical records that are either contained within the block or that overlap the block. Synonymous with physical record.

boolean condition
A boolean condition determines whether a boolean literal is true or false. A boolean condition can only be used in a constant conditional expression.

boolean literal
Can be either B'1', indicating a true value, or B'0', indicating a false value. Boolean literals can only be used in constant conditional expressions.

breakpoint
A place in a computer program, usually specified by an instruction, where external intervention or a monitor program can interrupt the program as it runs.

buffer
A portion of storage that is used to hold input or output data temporarily.

built-in function
See intrinsic function.

business method
A method of an enterprise bean that implements the business logic or rules of an application. (Oracle)

byte
A string that consists of a certain number of bits, usually eight, treated as a unit, and representing a character or a control function.

byte order mark (BOM)
A Unicode character that can be used at the start of UTF-16 or UTF-32 text to indicate the byte order of subsequent text; the byte order can be either big-endian or little-endian.

bytecode
Machine-independent code that is generated by the Java compiler and executed by the Java interpreter. (Oracle)

C

callable services
In Language Environment, a set of services that a COBOL program can invoke by using the conventional Language Environment-defined call interface. All programs that share the Language Environment conventions can use these services.

called program
A program that is the object of a CALL statement. At run time the called program and calling program are combined to produce a run unit.

* calling program
A program that executes a CALL to another program.

canonical decomposition
A way to represent a single precomposed Unicode character using two or more Unicode characters. A canonical decomposition is typically used to separate latin letters with a diacritical mark so that the latin letter and the diacritical mark are represented individually. See precomposed character for an example showing a precomposed Unicode character and its canonical decomposition.
**case structure**
A program-processing logic in which a series of conditions is tested in order to choose between a number of resulting actions.

**cataloged procedure**
A set of job control statements that are placed in a partitioned data set called the procedure library (SYS1.PROCLIB). You can use cataloged procedures to save time and reduce errors in coding JCL.

**CCSID**
See *coded character set identifier*.

**century window**
A 100-year interval within which any two-digit year is unique. Several types of century window are available to COBOL programmers:
- For the windowing intrinsic functions `DATE-T0-YYYYMMDD`, `DAY-T0-YYYYDDD`, and `YEAR-T0-YYYY`, you specify the century window with `argument-2`.
- For Language Environment callable services, you specify the century window in CEESCEN.

**character**
The basic indivisible unit of the language.

**character encoding unit**
A unit of data that corresponds to one code point in a coded character set. One or more character encoding units are used to represent a character in a coded character set. Also known as *encoding unit*.

For USAGE NATIONAL, a character encoding unit corresponds to one 2-byte code point of UTF-16.

For USAGE DISPLAY, a character encoding unit corresponds to a byte.

For USAGE DISPLAY-1, a character encoding unit corresponds to a 2-byte code point in the DBCS character set.

**character position**
The amount of physical storage or presentation space required to hold or present one character. The term applies to any class of character. For specific classes of characters, the following terms apply:
- *Alphanumeric character position*, for characters represented in USAGE DISPLAY
- *DBCS character position*, for DBCS characters represented in USAGE DISPLAY-1
- *National character position*, for characters represented in USAGE NATIONAL; synonymous with character encoding unit for UTF-16

**character set**
A collection of elements that are used to represent textual information, but for which no coded representation is assumed. See also *coded character set*.

**character string**
A sequence of contiguous characters that form a COBOL word, a literal, a PICTURE character string, or a comment-entry. A character string must be delimited by separators.

**checkpoint**
A point at which information about the status of a job and the system can be recorded so that the job step can be restarted later.

**class**
The entity that defines common behavior and implementation for zero, one, or more objects. The objects that share the same implementation are considered to be objects of the same class. Classes can be defined hierarchically, allowing one class to inherit from another.

**class (object-oriented)**
The entity that defines common behavior and implementation for zero, one, or more objects. The objects that share the same implementation are considered to be objects of the same class.
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.

The COBOL source unit that defines a class.

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.

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.

The name of an object-oriented COBOL class definition.

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.

The runtime object that represents a class.

An ordered set of consecutive COBOL character strings whose purpose is to specify an attribute of an entry.

In object-oriented programming, a program or method that requests services from one or more methods in a class.

The set of characters used in writing COBOL syntax. The complete COBOL character set consists of these characters:

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1,...,9</td>
<td>Digit</td>
</tr>
<tr>
<td>A,B,...,Z</td>
<td>Uppercase letter</td>
</tr>
<tr>
<td>a,b,...,z</td>
<td>Lowercase letter</td>
</tr>
<tr>
<td></td>
<td>Space</td>
</tr>
<tr>
<td></td>
<td>Plus sign</td>
</tr>
<tr>
<td></td>
<td>Minus sign (hyphen)</td>
</tr>
<tr>
<td></td>
<td>Asterisk</td>
</tr>
<tr>
<td></td>
<td>Slant (forward slash)</td>
</tr>
<tr>
<td></td>
<td>Equal sign</td>
</tr>
<tr>
<td></td>
<td>Currency sign</td>
</tr>
<tr>
<td></td>
<td>Comma</td>
</tr>
<tr>
<td></td>
<td>Semicolon</td>
</tr>
<tr>
<td></td>
<td>Period (decimal point, full stop)</td>
</tr>
<tr>
<td></td>
<td>Quotation mark</td>
</tr>
<tr>
<td></td>
<td>Apostrophe</td>
</tr>
<tr>
<td>Character</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>(</td>
<td>Left parenthesis</td>
</tr>
<tr>
<td>)</td>
<td>Right parenthesis</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>:</td>
<td>Colon</td>
</tr>
<tr>
<td>_</td>
<td>Underscore</td>
</tr>
</tbody>
</table>

* **COBOL word**
  See word.

**code page**
An assignment of graphic characters and control function meanings to all code points. For example, one code page could assign characters and meanings to 256 code points for 8-bit code, and another code page could assign characters and meanings to 128 code points for 7-bit code. For example, one of the IBM code pages for English on the workstation is IBM-1252 and on the host is IBM-1047. A **coded character set**.

**code point**
A unique bit pattern that is defined in a coded character set (code page). Graphic symbols and control characters are assigned to code points.

**coded character set**
A set of unambiguous rules that establish a character set and the relationship between the characters of the set and their coded representation. Examples of coded character sets are the character sets as represented by ASCII or EBCDIC code pages or by the UTF-16 encoding scheme for Unicode.

**coded character set identifier (CCSID)**
An IBM-defined number in the range 1 to 65,535 that identifies a specific code page.

* **collating sequence**
The sequence in which the characters that are acceptable to a computer are ordered for purposes of sorting, merging, comparing, and for processing indexed files sequentially.

* **column**
A byte position within a print line or within a reference format line. The columns are numbered from 1, by 1, starting at the leftmost position of the line and extending to the rightmost position of the line. A column holds one single-byte character.

* **combined condition**
A condition that is the result of connecting two or more conditions with the AND or the OR logical operator. See also condition and negated combined condition.

**combining characters**
A Unicode character used to modify other succeeding or preceding Unicode characters. Combining characters are typically Unicode diacritical mark used to modify latin letters. See precomposed character for an example of combining character U+0308 (¨) used with latin letter U+0061 (a).

* **comment-entry**
An entry in the IDENTIFICATION DIVISION that is used for documentation and has no effect on execution.

**comment line**
A source program line represented by an asterisk (*) in the indicator area of the line or by an asterisk followed by greater-than sign (>) as the first character string in the program text area (Area A plus Area B), and any characters from the character set of the computer that follow in Area A and Area B of that line. A comment line serves only for documentation. A special form of comment line represented by a slant (/) in the indicator area of the line and any characters from the character set of the computer in Area A and Area B of that line causes page ejection before the comment is printed.
*common program*
A program that, despite being directly contained within another program, can be called from any program directly or indirectly contained in that other program.

*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 arithmetic expression*
A subset of arithmetic expressions that are specified in the DEFINE and EVALUATE directives or in a constant conditional expression. The difference between compile-time arithmetic expressions and regular arithmetic expressions is that in a compile-time arithmetic expression:

• The exponentiation operator shall not be specified.
• All operands shall be integer numeric literals or arithmetic expressions in which all operands are integer numeric literals.
• The expression shall be specified in such a way that a division by zero does not occur.

*compile time*
The time at which COBOL source code is translated, by a COBOL compiler, to a COBOL object program.

*composition variable*
A symbolic name for a particular literal value or the value of a compile-time arithmetic expression as specified by the DEFINE directive or by the DEFINE compiler option.

*complex condition*
A condition in which one or more logical operators act upon one or more conditions. See also condition, negated simple condition, and negated combined condition.

*complex ODO*
Certain forms of the OCCURS DEPENDING ON clause:

• Variably located item or group: A data item described by an OCCURS clause with the DEPENDING ON option is followed by a nonsubordinate data item or group. The group can be an alphanumeric group or a national group.
• Variably located table: A data item described by an OCCURS clause with the DEPENDING ON option is followed by a nonsubordinate data item described by an OCCURS clause.
• Table with variable-length elements: A data item described by an OCCURS clause contains a subordinate data item described by an OCCURS clause with the DEPENDING ON option.
• Index name for a table with variable-length elements.
• Element of a table with variable-length elements.

*component*
(1) A functional grouping of related files. (2) In object-oriented programming, a reusable object or program that performs a specific function and is designed to work with other components and applications. JavaBeans is Oracle's architecture for creating components.

*composed form*
Representation of a precomposed Unicode character through a canonical decomposition. See precomposed character for details.
computer-name
A system-name that identifies the computer where the program is to be compiled or run.

c conditional (exception)
An exception that has been enabled, or recognized, by Language Environment and thus is eligible to activate user and language condition handlers. Any alteration to the normal programmed flow of an application. Conditions can be detected by the hardware or the operating system and result in an interrupt. They can also be detected by language-specific generated code or language library code.

c conditional (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.

c 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.

c 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.

c 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.

c conditional variable
A data item one or more values of which has a condition-name assigned to it.

c 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.

c condition-name condition
The proposition (for which a truth value can be determined) that the value of a conditional variable is a member of the set of values attributed to a condition-name associated with the conditional variable.

CONFIGURATION SECTION
A section of the ENVIRONMENT DIVISION that describes overall specifications of source and object programs and class definitions.

CONSOLE
A COBOL environment-name associated with the operator console.

constant conditional expression
A subset of conditional expressions that may be used in IF directives or WHEN phrases of the EVALUATE directives.

A constant conditional expression shall be one of the following items:

• A relation condition in which both operands are literals or arithmetic expressions that contain only literal terms. The condition shall follow the rules for relation conditions, with the following additions:
  – The operands shall be of the same category. An arithmetic expression is of the category numeric.
  – If literals are specified and they are not numeric literals, the relational operator shall be “IS EQUAL TO”, “IS NOT EQUAL TO”, “IS =”, “IS NOT =”, or “IS <>”.

  See also relation condition.

• A defined condition. See also defined condition.

• A boolean condition. See also boolean condition.
A complex condition formed by combining the above forms of simple conditions into complex conditions by using AND, OR, and NOT. Abbreviated combined relation conditions shall not be specified. See also complex condition.

contained program
A COBOL program that is nested within another COBOL program.

* contiguous items
Items that are described by consecutive entries in the DATA DIVISION, and that bear a definite hierarchic relationship to each other.

copybook
A file or library member that contains a sequence of code that is included in the source program at compile time using the COPY statement. The file can be created by the user, supplied by COBOL, or supplied by another product. Synonymous with copy file.

* counter
A data item used for storing numbers or number representations in a manner that permits these numbers to be increased or decreased by the value of another number, or to be changed or reset to zero or to an arbitrary positive or negative value.

cross-reference listing
The portion of the compiler listing that contains information on where files, fields, and indicators are defined, referenced, and modified in a program.

currency sign value
A character string that identifies the monetary units stored in a numeric-edited item. Typical examples are $, USD, and EUR. A currency sign value can be defined by either the CURRENCY compiler option or the CURRENCY SIGN clause in the SPECIAL- NAMES paragraph of the ENVIRONMENT DIVISION. If the CURRENCY SIGN clause is not specified and the NOCURRENCY compiler option is in effect, the dollar sign ($) is used as the default currency sign value. See also currency symbol.

currency symbol
A character used in a PICTURE clause to indicate the position of a currency sign value in a numeric-edited item. A currency symbol can be defined by either the CURRENCY compiler option or the CURRENCY SIGN clause in the SPECIAL- NAMES paragraph of the ENVIRONMENT DIVISION. If the CURRENCY SIGN clause is not specified and the NOCURRENCY compiler option is in effect, the dollar sign ($) is used as the default currency sign value and currency symbol. Multiple currency symbols and currency sign values can be defined. See also currency sign value.

* current record
In file processing, the record that is available in the record area associated with a file.

* current volume pointer
A conceptual entity that points to the current volume of a sequential file.

D

** data clause
A clause, appearing in a data description entry in the DATA DIVISION of a COBOL program, that provides information describing a particular attribute of a data item.

* data description entry
An entry in the DATA DIVISION of a COBOL program that is composed of a level-number followed by a data-name, if required, and then followed by a set of data clauses, as required.

DATA DIVISION
The division of a COBOL program or method that describes the data to be processed by the program or method: the files to be used and the records contained within them; internal WORKING-STORAGE records that will be needed; data to be made available in more than one program in the COBOL run unit.

* data item
A unit of data (excluding literals) defined by a COBOL program or by the rules for function evaluation.
**data set**

Synonym for file.

* **data-name**

A user-defined word that names a data item described in a data description entry. When used in the general formats, data-name represents a word that must not be reference-modified, subscripted, or qualified unless specifically permitted by the rules for the format.

**DBCS**

See double-byte character set (DBCS).

**DBCS character**

Any character defined in IBM's double-byte character set.

**DBCS character position**

See character position.

**DBCS data item**

A data item that is described by a PICTURE character string that contains at least one symbol G, or, when the NSYMBOL(DBCS) compiler option is in effect, at least one symbol N. A DBCS data item has USAGE DISPLAY-1.

* **debugging line**

Any line with a D in the indicator area of the line.

* **debugging section**

A section that contains a USE FOR DEBUGGING statement.

* **declarative sentence**

A compiler-directing sentence that consists of a single USE statement terminated by the separator period.

* **declaratives**

A set of one or more special-purpose sections, written at the beginning of the PROCEDURE DIVISION, the first of which is preceded by the key word DECLARATIVE and the last of which is followed by the key words END DECLARATIVES. A declarative is composed of a section header, followed by a USE compiler-directing sentence, followed by a set of zero, one, or more associated paragraphs.

* **de-edit**

The logical removal of all editing characters from a numeric-edited data item in order to determine the unedited numeric value of the item.

**defined condition**

A compile-time condition that tests whether a compilation variable is defined. Defined conditions are specified in IF directives or WHEN phrases of the EVALUATE directives.

* **delimited scope statement**

Any statement that includes its explicit scope terminator.

* **delimiter**

A character or a sequence of contiguous characters that identify the end of a string of characters and separate that string of characters from the following string of characters. A delimiter is not part of the string of characters that it delimits.

**dependent region**

In IMS, the MVS virtual storage region that contains message-driven programs, batch programs, or online utilities.

* **descending key**

A key upon the values of which data is ordered starting with the highest value of key down to the lowest value of key, in accordance with the rules for comparing data items.

**digit**

Any of the numerals from 0 through 9. In COBOL, the term is not used to refer to any other symbol.

* **digit position**

The amount of physical storage required to store a single digit. This amount can vary depending on the usage specified in the data description entry that defines the data item.
**direct access**
The facility to obtain data from storage devices or to enter data into a storage device in such a way that the process depends only on the location of that data and not on a reference to data previously accessed.

**display floating-point data item**
A data item that is described implicitly or explicitly as USAGE DISPLAY and that has a PICTURE character string that describes an external floating-point data item.

**division**
A collection of zero, one, or more sections or paragraphs, called the division body, that are formed and combined in accordance with a specific set of rules. Each division consists of the division header and the related division body. There are four divisions in a COBOL program: Identification, Environment, Data, and Procedure.

**division header**
A combination of words followed by a separator period that indicates the beginning of a division. The division headers are:

```plaintext
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
```

**DLL**
See *dynamic link library (DLL)*.

**DLL application**
An application that references imported programs, functions, or variables.

**DLL linkage**
A CALL in a program that has been compiled with the DLL and NODYNAM options; the CALL resolves to an exported name in a separate module, or to an INVOKE of a method that is defined in a separate module.

**do construct**
In structured programming, a DO statement is used to group a number of statements in a procedure. In COBOL, an inline PERFORM statement functions in the same way.

**do-until**
In structured programming, a do-until loop will be executed at least once, and until a given condition is true. In COBOL, a TEST AFTER phrase used with the PERFORM statement functions in the same way.

**do-while**
In structured programming, a do-while loop will be executed if, and while, a given condition is true. In COBOL, a TEST BEFORE phrase used with the PERFORM statement functions in the same way.

**document type declaration**
An XML element that contains or points to markup declarations that provide a grammar for a class of documents. This grammar is known as a document type definition, or DTD.

**document type definition (DTD)**
The grammar for a class of XML documents. See *document type declaration*.

**double-byte ASCII**
An IBM character set that includes DBCS and single-byte ASCII characters. (Also known as ASCII DBCS.)

**double-byte EBCDIC**
An IBM character set that includes DBCS and single-byte EBCDIC characters. (Also known as EBCDIC DBCS.)

**double-byte character set (DBCS)**
A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points,
require double-byte character sets. Because each character requires 2 bytes, entering, displaying, and printing DBCS characters requires hardware and supporting software that are DBCS-capable.

**DWARF**

DWARF was developed by the UNIX International Programming Languages Special Interest Group (SIG). It is designed to meet the symbolic, source-level debugging needs of different languages in a unified fashion by supplying language-independent debugging information. A DWARF file contains debugging data organized into different elements. For more information, see *DWARF program information* in the *DWARF/ELF Extensions Library Reference*.

* dynamic access
  An access mode in which specific logical records can be obtained from or placed into a mass storage file in a nonsequential manner and obtained from a file in a sequential manner during the scope of the same OPEN statement.

**dynamic CALL**

A CALL literal statement in a program that has been compiled with the DYNAM option and the NODLL option, or a CALL identifier statement in a program that has been compiled with the NODLL option.

**dynamic-length**

An adjective describing an item whose logical length might change at runtime.

**dynamic-length elementary item**

An elementary data item whose data declaration entry contains the DYNAMIC LENGTH clause.

**dynamic-length group**

A group item that contains a subordinate dynamic-length elementary item.

**dynamic link library (DLL)**

A file that contains executable code and data that are bound to a program at load time or run time, rather than during linking. Several applications can share the code and data in a DLL simultaneously. Although a DLL is not part of the executable file for a program, it can be required for an executable file to run properly.

**dynamic storage area (DSA)**

Dynamically acquired storage composed of a register save area and an area available for dynamic storage allocation (such as program variables). A DSA is allocated upon invocation of a program or function and persists for the duration of the invocation instance. DSAs are generally allocated within stack segments managed by Language Environment.

**E**

* EBCDIC (Extended Binary-Coded Decimal Interchange Code)*

A coded character set based on 8-bit coded characters.

**EBCDIC character**

Any one of the symbols included in the EBCDIC (Extended Binary-Coded-Decimal Interchange Code) set.

**EBCDIC DBCS**

See double-byte EBCDIC.

**edited data item**

A data item that has been modified by suppressing zeros or inserting editing characters or both.

* editing character

A single character or a fixed two-character combination belonging to the following set:

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Space</td>
</tr>
<tr>
<td>+</td>
<td>Zero</td>
</tr>
<tr>
<td>-</td>
<td>Plus</td>
</tr>
<tr>
<td>CR</td>
<td>Minus</td>
</tr>
<tr>
<td>Credit</td>
<td></td>
</tr>
<tr>
<td>Character</td>
<td>Meaning</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
</tr>
<tr>
<td>DB</td>
<td>Debit</td>
</tr>
<tr>
<td>Z</td>
<td>Zero suppress</td>
</tr>
<tr>
<td>*</td>
<td>Check protect</td>
</tr>
<tr>
<td>$</td>
<td>Currency sign</td>
</tr>
<tr>
<td>,</td>
<td>Comma (decimal point)</td>
</tr>
<tr>
<td>.</td>
<td>Period (decimal point)</td>
</tr>
<tr>
<td>/</td>
<td>Slant (forward slash)</td>
</tr>
</tbody>
</table>

**EGCS**
See extended graphic character set (EGCS).

**EJB**
See Enterprise JavaBeans.

**EJB container**
A container that implements the EJB component contract of the J2EE architecture. This contract specifies a runtime environment for enterprise beans that includes security, concurrency, life cycle management, transaction, deployment, and other services. An EJB container is provided by an EJB or J2EE server. (Oracle)

**EJB server**
Software that provides services to an EJB container. An EJB server can host one or more EJB containers. (Oracle)

**element (text element)**
One logical unit of a string of text, such as the description of a single data item or verb, preceded by a unique code identifying the element type.

**elementary item**
A data item that is described as not being further logically subdivided.

**encapsulation**
In object-oriented programming, the technique that is used to hide the inherent details of an object. The object provides an interface that queries and manipulates the data without exposing its underlying structure. Synonymous with information hiding.

**enclave**
When running under Language Environment, an enclave is analogous to a run unit. An enclave can create other enclaves by using LINK and by using the system() function in C.

**encoding unit**
See character encoding unit.

**end class marker**
A combination of words, followed by a separator period, that indicates the end of a COBOL class definition. The end class marker is:

```
END CLASS class-name.
```

**end method marker**
A combination of words, followed by a separator period, that indicates the end of a COBOL method definition. The end method marker is:

```
END METHOD method-name.
```

**end of PROCEDURE DIVISION**
The physical position of a COBOL source program after which no further procedures appear.
**end program marker**
A combination of words, followed by a separator period, that indicates the end of a COBOL source program. The end program marker is:

```
END PROGRAM program-name.
```

**enterprise bean**
A component that implements a business task and resides in an EJB container. (Oracle)

**Enterprise JavaBeans**
A component architecture defined by Oracle for the development and deployment of object-oriented, distributed, enterprise-level applications.

**entry**
Any descriptive set of consecutive clauses terminated by a separator period and written in the IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, or DATA DIVISION of a COBOL program.

**environment clause**
A clause that appears as part of an ENVIRONMENT DIVISION entry.

**ENVIRONMENT DIVISION**
One of the four main component parts of a COBOL program, class definition, or method definition. The ENVIRONMENT DIVISION describes the computers where the source program is compiled and those where the object program is run. It provides a linkage between the logical concept of files and their records, and the physical aspects of the devices on which files are stored.

**environment-name**
A name, specified by IBM, that identifies system logical units, printer and card punch control characters, report codes, program switches or all of these. When an environment-name is associated with a mnemonic-name in the ENVIRONMENT DIVISION, the mnemonic-name can be substituted in any format in which such substitution is valid.

**environment variable**
Any of a number of variables that define some aspect of the computing environment, and are accessible to programs that operate in that environment. Environment variables can affect the behavior of programs that are sensitive to the environment in which they operate.

**execution time**
See run time.

**execution-time environment**
See runtime environment.

**explicit scope terminator**
A reserved word that terminates the scope of a particular PROCEDURE DIVISION statement.

**exponent**
A number that indicates the power to which another number (the base) is to be raised. Positive exponents denote multiplication; negative exponents denote division; and fractional exponents denote a root of a quantity. In COBOL, an exponential expression is indicated with the symbol ** followed by the exponent.

**expression**
An arithmetic or conditional expression.

**extend mode**
The state of a file after execution of an OPEN statement, with the EXTEND phrase specified for that file, and before the execution of a CLOSE statement, without the REEL or UNIT phrase for that file.

**extended graphic character set (EGCS)**
A graphic character set, such as a kanji character set, that requires two bytes to identify each graphic character. It is refined and replaced by double-byte character set (DBCS).

**Extensible Markup Language**
See XML.
COBOL syntax and semantics supported by IBM compilers in addition to those described in the 85 COBOL Standard.

For XML documents, the value specified by the CODEPAGE compiler option.

The data that is described in a program as external data items and external file connectors.

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.

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.

See zoned decimal data item and national decimal data item.

A file connector that is accessible to one or more object programs in the run unit.

See display floating-point data item and national floating-point data item.

The outermost program. A program that is not nested.

A hardware or software device, defined and named by the implementor, which is used to indicate that one of two alternate states exists.

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.

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.

A compiler-generated value referenced through the use of certain reserved words.

A collection of logical records.

An unsuccessful conflict condition 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.

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).

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.

The name of an ENVIRONMENT DIVISION paragraph in which the data files for a given source program are declared.
**file control block**
Block containing the addresses of I/O routines, information about how they were opened and closed, and a pointer to the file information block.

**file control entry**
A SELECT clause and all its subordinate clauses that declare the relevant physical attributes of a file.

**FILE-CONTROL paragraph**
A paragraph in the ENVIRONMENT DIVISION in which the data files for a given source unit are declared.

**file description entry**
An entry in the FILE SECTION of the DATA DIVISION that is composed of the level indicator FD, followed by a file-name, and then followed by a set of file clauses as required.

**file-name**
A user-defined word that names a file connector described in a file description entry or a sort-merge file description entry within the FILE SECTION of the DATA DIVISION.

**file organization**
The permanent logical file structure established at the time that a file is created.

**file position indicator**
A conceptual entity that contains the value of the current key within the key of reference for an indexed file, or the record number of the current record for a sequential file, or the relative record number of the current record for a relative file, or indicates that no next logical record exists, or that an optional input file is not available, or that the AT END condition already exists, or that no valid next record has been established.

**FILE SECTION**
The section of the DATA DIVISION that contains file description entries and sort-merge file description entries together with their associated record descriptions.

**file system**
The collection of files that conform to a specific set of data-record and file-description protocols, and a set of programs that manage these files.

**fixed file attributes**
Information about a file that is established when a file is created and that cannot subsequently be changed during the existence of the file. These attributes include the organization of the file (sequential, relative, or indexed), the prime record key, the alternate record keys, the code set, the minimum and maximum record size, the record type (fixed or variable), the collating sequence of the keys for indexed files, the blocking factor, the padding character, and the record delimiter.

**fixed-length record**
A record associated with a file whose file description or sort-merge description entry requires that all records contain the same number of bytes.

**fixed-point item**
A numeric data item defined with a PICTURE clause that specifies the location of an optional sign, the number of digits it contains, and the location of an optional decimal point. The format can be either binary, packed decimal, or external decimal.

**floating comment indicators (>*>)**
A floating comment indicator indicates a comment line if it is the first character string in the program-text area (Area A plus Area B), or indicates an inline comment if it is after one or more character strings in the program-text area.

**floating point**
A format for representing numbers in which a real number is represented by a pair of distinct numerals. In a floating-point representation, the real number is the product of the fixed-point part (the first numeral) and a value obtained by raising the implicit floating-point base to a power denoted by the exponent (the second numeral). For example, a floating-point representation of the number 0.0001234 is 0.1234 -3, where 0.1234 is the mantissa and -3 is the exponent.
**floating-point data item**
A numeric data item that contains a fraction and an exponent. Its value is obtained by multiplying the fraction by the base of the numeric data item raised to the power that the exponent specifies.

**format**
A specific arrangement of a set of data.

**function**
A temporary data item whose value is determined at the time the function is referenced during the execution of a statement.

**function-identifier**
A syntactically correct combination of character strings and separators that references a function. The data item represented by a function is uniquely identified by a function-name with its arguments, if any. A function-identifier can include a reference-modifier. A function-identifier that references an alphanumeric function can be specified anywhere in the general formats that an identifier can be specified, subject to certain restrictions. A function-identifier that references an integer or numeric function can be referenced anywhere in the general formats that an arithmetic expression can be specified.

**function-name**
A word that names the mechanism whose invocation, along with required arguments, determines the value of a function.

**function-pointer data item**
A data item in which a pointer to an entry point can be stored. A data item defined with the USAGE IS FUNCTION-POINTER clause contains the address of a function entry point. Typically used to communicate with C and Java programs.

**garbage collection**
The automatic freeing by the Java runtime system of the memory for objects that are no longer referenced.

**global name**
A name that is declared in only one program but that can be referenced from the program and from any program contained within the program. Condition-names, data-names, file-names, record-names, report-names, and some special registers can be global names.

**global reference**
A reference to an object that is outside the scope of a method.

**group item**
(1) A data item that is composed of subordinate data items. See alphanumeric group item and national group item. (2) When not qualified explicitly or by context as a national group or an alphanumeric group, the term refers to groups in general.

**grouping separator**
A character used to separate units of digits in numbers for ease of reading. The default is the character comma.

**header label**
(1) A data-set label that precedes the data records in a unit of recording media. (2) Synonym for beginning-of-file label.

**hide (a method)**
To redefine (in a subclass) a factory or static method defined with the same method-name in a parent class. Thus, the method in the subclass hides the method in the parent class.

**high-order end**
The leftmost character of a string of characters.

**hiperspace**
In a z/OS environment, a range of up to 2 GB of contiguous virtual storage addresses that a program can use as a buffer.
IBM COBOL extension

COBOL syntax and semantics supported by IBM compilers in addition to those described in the 85 COBOL Standard.

IDENTIFICATION DIVISION

One of the four main component parts of a COBOL program, class definition, or method definition. The IDENTIFICATION DIVISION identifies the program, class, or method. The IDENTIFICATION DIVISION can include the following documentation: author name, installation, or date.

* identifier

A syntactically correct combination of character strings and separators that names a data item. When referencing a data item that is not a function, an identifier consists of a data-name, together with its qualifiers, subscripts, and reference-modifier, as required for uniqueness of reference. When referencing a data item that is a function, a function-identifier is used.

IGZCBSN

The bootstrap routine for COBOL/370 Release 1. It must be link-edited with any module that contains a COBOL/370 Release 1 program.

IGZCBSO

The bootstrap routine for COBOL for MVS & VM Release 2, COBOL for OS/390 & VM and Enterprise COBOL. It must be link-edited with any module that contains a COBOL for MVS & VM Release 2, COBOL for OS/390 & VM or Enterprise COBOL program.

IGZEBST

The bootstrap routine for VS COBOL II. It must be link-edited with any module that contains a VS COBOL II program.

ILC

InterLanguage Communication. Interlanguage communication is defined as programs that call or are called by other high-level languages. Assembler is not considered a high-level language; thus, calls to and from assembler programs are not considered ILC.

* imperative statement

A statement that either begins with an imperative verb and specifies an unconditional action to be taken or is a conditional statement that is delimited by its explicit scope terminator (delimited scope statement). An imperative statement can consist of a sequence of imperative statements.

* implicit scope terminator

A separator period that terminates the scope of any preceding unterminated statement, or a phrase of a statement that by its occurrence indicates the end of the scope of any statement contained within the preceding phrase.

IMS

Information Management System, IBM licensed product. IMS supports hierarchical databases, data communication, translation processing, and database backout and recovery.

* index

A computer storage area or register, the content of which represents the identification of a particular element in a table.

* index data item

A data item in which the values associated with an index-name can be stored in a form specified by the implementor.

indexed data-name

An identifier that is composed of a data-name, followed by one or more index-names enclosed in parentheses.

* indexed file

A file with indexed organization.

* indexed organization

The permanent logical file structure in which each record is identified by the value of one or more keys within that record.
indexing
Synonymous with subscripting using index-names.

* index-name
A user-defined word that names an index associated with a specific table.

inheritance
A mechanism for using the implementation of a class as the basis for another class. By definition, the inheriting class conforms to the inherited classes. Enterprise COBOL does not support multiple inheritance; a subclass has exactly one immediate superclass.

inheritance hierarchy
See class hierarchy.

* initial program
A program that is placed into an initial state every time the program is called in a run unit.

* initial state
The state of a program when it is first called in a run unit.

inline
In a program, instructions that are executed sequentially, without branching to routines, subroutines, or other programs.

inline comments
An inline comment is identified by a floating comment indicator (*>) preceded by one or more character-strings in the program-text area, and can be written on any line of a compilation group. All characters that follow the floating comment indicator up to the end of area B are comment text.

* input file
A file that is opened in the input mode.

* input mode
The state of a file after execution of an OPEN statement, with the INPUT phrase specified, for that file and before the execution of a CLOSE statement, without the REEL or UNIT phrase for that file.

* input-output file
A file that is opened in the I-O mode.

**INPUT-OUTPUT SECTION**
The section of the ENVIRONMENT DIVISION that names the files and the external media required by an object program or method and that provides information required for transmission and handling of data at run time.

* input-output statement
A statement that causes files to be processed by performing operations on individual records or on the file as a unit. The input-output statements are ACCEPT (with the identifier phrase), CLOSE, DELETE, DISPLAY, OPEN, READ, REWRITE, SET (with the TO ON or TO OFF phrase), START, and WRITE.

* input procedure
A set of statements, to which control is given during the execution of a format 1 SORT statement, for the purpose of controlling the release of specified records to be sorted.

instance data
Data that defines the state of an object. The instance data introduced by a class is defined in the WORKING-STORAGE SECTION of the DATA DIVISION in the OBJECT paragraph of the class definition. The state of an object also includes the state of the instance variables introduced by classes that are inherited by the current class. A separate copy of the instance data is created for each object instance.

* integer
(1) A numeric literal that does not include any digit positions to the right of the decimal point. (2) A numeric data item defined in the DATA DIVISION that does not include any digit positions to the right of the decimal point. (3) A numeric function whose definition provides that all digits to the right of the decimal point are zero in the returned value for any possible evaluation of the function.
**integer function**
A function whose category is numeric and whose definition does not include any digit positions to the right of the decimal point.

**Interactive System Productivity Facility (ISPF)**
An IBM software product that provides a menu-driven interface for the TSO or VM user. ISPF includes library utilities, a powerful editor, and dialog management.

**interlanguage communication (ILC)**
The ability of routines written in different programming languages to communicate. ILC support lets you readily build applications from component routines written in a variety of languages.

**intermediate result**
An intermediate field that contains the results of a succession of arithmetic operations.

**internal data**
The data that is described in a program and excludes all external data items and external file connectors. Items described in the LINKAGE SECTION of a program are treated as internal data.

**internal data item**
A data item that is described in one program in a run unit. An internal data item can have a global name.

**internal decimal data item**
A data item that is described as USAGE PACKED-DECIMAL or USAGE COMP-3, and that has a PICTURE character string that defines the item as numeric (a valid combination of symbols 9, S, P, or V). Synonymous with packed-decimal data item.

**internal file connector**
A file connector that is accessible to only one object program in the run unit.

**internal floating-point data item**
A data item that is described as USAGE COMP-1 or USAGE COMP-2. COMP-1 defines a single-precision floating-point data item. COMP-2 defines a double-precision floating-point data item. There is no PICTURE clause associated with an internal floating-point data item.

**intrapage data structure**
The entire collection of groups and elementary data items from a logical record that a contiguous subset of the data description entries defines. These data description entries include all entries whose level-number is greater than the level-number of the first data description entry describing the intra-record data structure.

**intrinsic function**
A predefined function, such as a commonly used arithmetic function, called by a built-in function reference.

**invalid key condition**
A condition, at run time, caused when a specific value of the key associated with an indexed or relative file is determined to be not valid.

**I-O-CONTROL**
The name of an ENVIRONMENT DIVISION paragraph in which object program requirements for rerun points, sharing of same areas by several data files, and multiple file storage on a single input-output device are specified.

**I-O-CONTROL entry**
An entry in the I-O-CONTROL paragraph of the ENVIRONMENT DIVISION; this entry contains clauses that provide information required for the transmission and handling of data on named files during the execution of a program.

**I-O mode**
The state of a file after execution of an OPEN statement, with the I-O phrase specified, for that file and before the execution of a CLOSE statement without the REEL or UNIT phase for that file.

**I-O status**
A conceptual entity that contains the two-character value indicating the resulting status of an input-output operation. This value is made available to the program through the use of the FILE STATUS clause in the file control entry for the file.
is-a
A relationship that characterizes classes and subclasses in an inheritance hierarchy. Subclasses that have an is-a relationship to a class inherit from that class.

ISPF
See Interactive System Productivity Facility (ISPF).

iteration structure
A program processing logic in which a series of statements is repeated while a condition is true or until a condition is true.

J

J2EE
See Java 2 Platform, Enterprise Edition (J2EE).

Java 2 Platform, Enterprise Edition (J2EE)
An environment for developing and deploying enterprise applications, defined by Oracle. The J2EE platform consists of a set of services, application programming interfaces (APIs), and protocols that provide the functionality for developing multitiered, Web-based applications. (Oracle)

Java Batch Launcher and Toolkit for z/OS (JZOS)
A set of tools that helps you develop z/OS Java applications that run in a traditional batch environment, and that access z/OS system services.

Java batch-processing program (JBP)
An IMS batch-processing program that has access to online databases and output message queues. JBPs run online, but like programs in a batch environment, they are started with JCL or in a TSO session.

Java batch-processing region
An IMS dependent region in which only Java batch-processing programs are scheduled.

Java Database Connectivity (JDBC)
A specification from Oracle that defines an API that enables Java programs to access databases.

Java message-processing program (JMP)
A Java application program that is driven by transactions and has access to online IMS databases and message queues.

Java message-processing region
An IMS dependent region in which only Java message-processing programs are scheduled.

Java Native Interface (JNI)
A programming interface that lets Java code that runs inside a Java virtual machine (JVM) interoperate with applications and libraries written in other programming languages.

Java virtual machine (JVM)
A software implementation of a central processing unit that runs compiled Java programs.

JavaBeans
A portable, platform-independent, reusable component model. (Oracle)

JBP
See Java batch-processing program (JBP).

JDBC
See Java Database Connectivity (JDBC).

JMP
See Java message-processing program (JMP).

job control language (JCL)
A control language used to identify a job to an operating system and to describe the job’s requirements.

JSON
JSON (JavaScript Object Notation) is a lightweight data-interchange format.

JVM
See Java virtual machine (JVM).
**JZOS**

See *Java Batch Launcher and Toolkit for z/OS.*

**K**

A data item that identifies the location of a record, or a set of data items that serve to identify the ordering of data.

**key of reference**

The key, either prime or alternate, currently being used to access records within an indexed file.

**keyword**

A context-sensitive word or a reserved word whose presence is required when the format in which the word appears is used in a source unit.

**kilobyte (KB)**

One kilobyte equals 1024 bytes.

**Language Environment**

Short form of z/OS Language Environment. A set of architectural constructs and interfaces that provides a common runtime environment and runtime services for C, C++, COBOL, FORTRAN and PL/I applications. It is required for programs compiled by Language Environment-conforming compilers and for Java applications.

**Language Environment-conforming**

A characteristic of compiler products (such as Enterprise COBOL, COBOL for OS/390 & VM, COBOL for MVS & VM, C/C++ for MVS & VM, PL/I for MVS & VM) that produce object code conforming to the Language Environment conventions.

**last-used state**

A state that a program is in if its internal values remain the same as when the program was exited (the values are not reset to their initial values).

**letter**

A character belonging to one of the following two sets:

2. Lowercase letters: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z

**level indicator**

Two alphabetic characters that identify a specific type of file or a position in a hierarchy. The level indicators in the DATA DIVISION are: CD, FD, and SD.

**level-number**

A user-defined word (expressed as a two-digit number) that indicates the hierarchical position of a data item or the special properties of a data description entry. Level-numbers in the range from 1 through 49 indicate the position of a data item in the hierarchical structure of a logical record. Level-numbers in the range 1 through 9 can be written either as a single digit or as a zero followed by a significant digit. Level-numbers 66, 77, and 88 identify special properties of a data description entry.

**library-name**

A user-defined word that names a COBOL library that the compiler is to use for compiling a given source program.

**library text**

A sequence of text words, comment lines, inline comments, the separator space, or the separator pseudo-text delimiter in a COBOL library.
Lilian date
The number of days since the beginning of the Gregorian calendar. Day one is Friday, October 15, 1582. The Lilian date format is named in honor of Luigi Lilio, the creator of the Gregorian calendar.

* lineage-counter
A special register whose value points to the current position within the page body.

link
(1) The combination of the link connection (the transmission medium) and two link stations, one at each end of the link connection. A link can be shared among multiple links in a multipoint or token-ring configuration. (2) To interconnect items of data or portions of one or more computer programs; for example, linking object programs by a linkage-editor to produce an executable file.

LINKAGE SECTION
The section in the DATA DIVISION of the called program or invoked method that describes data items available from the calling program or invoking method. Both the calling program or invoking method and the called program or invoked method can refer to these data items.

linker
A term that refers to either the z/OS binder (linkage-editor).

literal
A character string whose value is specified either by the ordered set of characters comprising the string or by the use of a figurative constant.

little-endian
The default format that Intel processors use to store binary data and UTF-16 characters. In this format, the most significant byte of a binary data item is at the highest address and the most significant byte of a UTF-16 character is at the highest address. Compare with big-endian.

local reference
A reference to an object that is within the scope of your method.

locale
A set of attributes for a program execution environment that indicates culturally sensitive considerations, such as character code page, collating sequence, date and time format, monetary value representation, numeric value representation, or language.

* LOCAL-STORAGE SECTION
The section of the DATA DIVISION that defines storage that is allocated and freed on a per-invocation basis, depending on the value assigned in the VALUE clauses.

* logical operator
One of the reserved words AND, OR, or NOT. In the formation of a condition, either AND, or OR, or both can be used as logical connectives. NOT can be used for logical negation.

* logical record
The most inclusive data item. The level-number for a record is 01. A record can be either an elementary item or a group of items. Synonymous with record.

* low-order end
The rightmost character of a string of characters.

M

main program
In a hierarchy of programs and subroutines, the first program that receives control when the programs are run within a process.

makefile
A text file that contains a list of the files for your application. The make utility uses this file to update the target files with the latest changes.

* mass storage
A storage medium in which data can be organized and maintained in both a sequential manner and a nonsequential manner.

* mass storage device
A device that has a large storage capacity, such as a magnetic disk.
* **mass storage file**
  A collection of records that is stored in a mass storage medium.

* **megabyte (MB)**
  One megabyte equals 1,048,576 bytes.

* **merge file**
  A collection of records to be merged by a MERGE statement. The merge file is created and can be used only by the merge function.

**message-processing program (MPP)**
An IMS application program that is driven by transactions and has access to online IMS databases and message queues.

**message queue**
The data set on which messages are queued before being processed by an application program or sent to a terminal.

**method**
Procedural code that defines an operation supported by an object and that is executed by an INVOKE statement on that object.

* **method definition**
The COBOL source code that defines a method.

* **method identification entry**
An entry in the METHOD-ID paragraph of the IDENTIFICATION DIVISION; this entry contains a clause that specifies the method-name.

**method invocation**
A communication from one object to another that requests the receiving object to execute a method.

**method-name**
The name of an object-oriented operation. When used to invoke the method, the name can be an alphanumeric or national literal or a category alphanumeric or category national data item. When used in the METHOD-ID paragraph to define the method, the name must be an alphanumeric or national literal.

**method hiding**
See hide.

**method overloading**
See overload.

**method overriding**
See override.

* **mnemonic-name**
  A user-defined word that is associated in the ENVIRONMENT DIVISION with a specified implementor-name.

**module definition file**
A file that describes the code segments within a program object.

**MPP**
See message-processing program (MPP).

**multitasking**
A mode of operation that provides for the concurrent, or interleaved, execution of two or more tasks.

**multithreading**
Concurrent operation of more than one path of execution within a computer. Synonymous with multiprocessing.

**N**

**name**
A word (composed of not more than 30 characters) that defines a COBOL operand.

**namespace**
See XML namespace.
**national character**
(1) A UTF-16 character in a USAGE NATIONAL data item or national literal. (2) Any character represented in UTF-16.

**national character data**
A general reference to data represented in UTF-16.

**national character position**
See character position.

**national data**
See national character data.

**national data item**
A data item of category national, national-edited, or numeric-edited of USAGE NATIONAL.

**national decimal data item**
An external decimal data item that is described implicitly or explicitly as USAGE NATIONAL and that contains a valid combination of PICTURE symbols 9, S, P, and V.

**national-edited data item**
A data item that is described by a PICTURE character string that contains at least one instance of the symbol N and at least one of the simple insertion symbols B, 0, or /. A national-edited data item has USAGE NATIONAL.

**national floating-point data item**
An external floating-point data item that is described implicitly or explicitly as USAGE NATIONAL and that has a PICTURE character string that describes a floating-point data item.

**national group item**
A group item that is explicitly or implicitly described with a GROUP-USAGE NATIONAL clause. A national group item is processed as though it were defined as an elementary data item of category national for operations such as INSPECT, STRING, and UNSTRING. This processing ensures correct padding and truncation of national characters, as contrasted with defining USAGE NATIONAL data items within an alphanumeric group item. For operations that require processing of the elementary items within a group, such as MOVE CORRESPONDING, ADD CORRESPONDING, and INITIALIZE, a national group is processed using group semantics.

**native character set**
The implementor-defined character set associated with the computer specified in the OBJECT-COMPUTER paragraph.

**native collating sequence**
The implementor-defined collating sequence associated with the computer specified in the OBJECT-COMPUTER paragraph.

**native method**
A Java method with an implementation that is written in another programming language, such as COBOL.

**negated combined condition**
The NOT logical operator immediately followed by a parenthesized combined condition. See also condition and combined condition.

**negated simple condition**
The NOT logical operator immediately followed by a simple condition. See also condition and simple condition.

**nested program**
A program that is directly contained within another program.

**next executable sentence**
The next sentence to which control will be transferred after execution of the current statement is complete.

**next executable statement**
The next statement to which control will be transferred after execution of the current statement is complete.
The record that logically follows the current record of a file.

Elementary data items in the WORKING-STORAGE SECTION and LINKAGE SECTION that bear no hierarchic relationship to other data items.

Elementary data items in the WORKING-STORAGE and LINKAGE SECTIONs that bear no hierarchic relationship to other data items.

A data item whose description permits its content to be composed of any combination of characters taken from the computer's character set. Certain categories of nonnumeric items may be formed from more restricted character sets.

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.

A character that belongs to the following set of digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

(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.

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.

A function whose class and category are numeric but that for some possible evaluation does not satisfy the requirements of integer functions.

A data item whose description restricts its content to a value represented by characters chosen from the digits from '0' through '9'; if signed, the item may also contain a '+' or '-' or other representation of an operational sign.

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.

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.

Output from a compiler or assembler that is itself executable machine code or is suitable for processing to produce executable machine code.

The name of an ENVIRONMENT DIVISION paragraph in which the computer environment, where the object program is run, is described.

An entry in the OBJECT-COMPUTER paragraph of the ENVIRONMENT DIVISION; this entry contains clauses that describe the computer environment in which the object program is to be executed.
object deck
A portion of an object program suitable as input to a linkage-editor. Synonymous with object module and text deck.

object instance
A single object, of possibly many, instantiated from the specifications in the object paragraph of a COBOL class definition. An object instance has a copy of all the data described in its class definition and all inherited data. The methods associated with an object instance includes the methods defined in its class definition and all inherited methods.

An object instance can be an instance of a Java class.

object module
Synonym for object deck or text deck.

* object of entry
A set of operands and reserved words, within a DATA DIVISION entry of a COBOL program, that immediately follows the subject of the entry.

object-oriented programming
A programming approach based on the concepts of encapsulation and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates on the data objects that comprise the problem and how they are manipulated, not on how something is accomplished.

object program
A set or group of executable machine-language instructions and other material designed to interact with data to provide problem solutions. In this context, an object program is generally the machine language result of the operation of a COBOL compiler on a source program or class definition. Where there is no danger of ambiguity, the word program can be used in place of object program.

object reference
A value that identifies an instance of a class. If the class is not specified, the object reference is universal and can apply to instances of any class.

* object time
The time at which an object program is executed. Synonymous with run time.

* obsolete element
A COBOL language element in the 85 COBOL Standard that was deleted from the 2002 COBOL Standard.

ODO object
In the example below, X is the object of the OCCURS DEPENDING ON clause (ODO object).

```
WORKING-STORAGE SECTION.
01 TABLE-1.
   05 X PIC S9.
   05 Y OCCURS 3 TIMES DEPENDING ON X PIC X.
```

The value of the ODO object determines how many of the ODO subject appear in the table.

ODO subject
In the example above, Y is the subject of the OCCURS DEPENDING ON clause (ODO subject). The number of Y ODO subjects that appear in the table depends on the value of X.

* open mode
The state of a file after execution of an OPEN statement for that file and before the execution of a CLOSE statement without the REEL or UNIT phrase for that file. The particular open mode is specified in the OPEN statement as either INPUT, OUTPUT, I-O, or EXTEND.

* operand
(1) The general definition of operand is "the component that is operated upon." (2) For the purposes of this document, any lowercase word (or words) that appears in a statement or entry format can be considered to be an operand and, as such, is an implied reference to the data indicated by the operand.
operation
A service that can be requested of an object.

* operational sign
An algebraic sign that is associated with a numeric data item or a numeric literal, to indicate whether its value is positive or negative.

optional file
A file that is declared as being not necessarily available each time the object program is run.

* optional word
A reserved word that is included in a specific format only to improve the readability of the language. Its presence is optional to the user when the format in which the word appears is used in a source unit.

* output file
A file that is opened in either output mode or extend mode.

* output mode
The state of a file after execution of an OPEN statement, with the OUTPUT or EXTEND phrase specified, for that file and before the execution of a CLOSE statement without the REEL or UNIT phrase for that file.

* output procedure
A set of statements to which control is given during execution of a format 1 SORT statement after the sort function is completed, or during execution of a MERGE statement after the merge function reaches a point at which it can select the next record in merged order when requested.

overflow condition
A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage.

overload
To define a method with the same name as another method that is available in the same class, but with a different signature. See also signature.

override
To redefine an instance method (inherited from a parent class) in a subclass.

P

package
A group of related Java classes, which can be imported individually or as a whole.

packed-decimal data item
See internal decimal data item.

padding character
An alphanumeric or national character that is used to fill the unused character positions in a physical record.

page
A vertical division of output data that represents a physical separation of the data. The separation is based on internal logical requirements or external characteristics of the output medium or both.

* page body
That part of the logical page in which lines can be written or spaced or both.

* paragraph
In the PROCEDURE DIVISION, a paragraph-name followed by a separator period and by zero, one, or more sentences. In the IDENTIFICATION DIVISION and ENVIRONMENT DIVISION, a paragraph header followed by zero, one, or more entries.
**paragraph header**
A reserved word, followed by the separator period, that indicates the beginning of a paragraph in the IDENTIFICATION DIVISION and ENVIRONMENT DIVISION. The permissible paragraph headers in the IDENTIFICATION DIVISION are:

- PROGRAM-ID. (Program IDENTIFICATION DIVISION)
- CLASS-ID. (Class IDENTIFICATION DIVISION)
- METHOD-ID. (Method IDENTIFICATION DIVISION)
- AUTHOR.
- INSTALLATION.
- DATE-WRITTEN.
- DATE-COMPILED.
- SECURITY.

The permissible paragraph headers in the ENVIRONMENT DIVISION are:

- SOURCE-COMPUTER.
- OBJECT-COMPUTER.
- SPECIAL-NAMES.
- REPOSITORY. (Program or Class CONFIGURATION SECTION)
- FILE-CONTROL.
- I-O-CONTROL.

**paragraph-name**
A user-defined word that identifies and begins a paragraph in the PROCEDURE DIVISION.

**parameter**
(1) Data passed between a calling program and a called program. (2) A data element in the USING phrase of a method invocation. Arguments provide additional information that the invoked method can use to perform the requested operation.

**Persistent Reusable JVM**
A JVM that can be serially reused for transaction processing by resetting the JVM between transactions. The reset phase restores the JVM to a known initialization state.

**phrase**
An ordered set of one or more consecutive COBOL character strings that form a portion of a COBOL procedural statement or of a COBOL clause.

**physical record**
See block.

**pointer data item**
A data item in which address values can be stored. Data items are explicitly defined as pointers with the USAGE IS POINTER clause. ADDRESS OF special registers are implicitly defined as pointer data items. Pointer data items can be compared for equality or moved to other pointer data items.

**port**
(1) To modify a computer program to enable it to run on a different platform. (2) In the Internet suite of protocols, a specific logical connector between the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP) and a higher-level protocol or application. A port is identified by a port number.

**portability**
The ability to transfer an application program from one application platform to another with relatively few changes to the source program.

**precomposed character**
A single Unicode character that can be represented using two or more Unicode characters through a canonical decomposition. A precomposed character does not have the same physical representation as its composed character form. For example, Unicode character U+00E4 (ä) is a precomposed character that can be represented as a combination of Unicode characters U+0061 + U+0308 (ä) - latin small letter a + combining diaeresis. A precomposed character is typically used to represent a latin letter with a diacritical mark or some other combining character.
preinitialization
The initialization of the COBOL runtime environment in preparation for multiple calls from programs, especially non-COBOL programs. The environment is not terminated until an explicit termination.

* prime record key
A key whose contents uniquely identify a record within an indexed file.

* priority-number
A user-defined word that classifies sections in the PROCEDURE DIVISION for purposes of segmentation. Segment numbers can contain only the characters 0 through 9. A segment number can be expressed as either one or two digits.

private
As applied to factory data or instance data, accessible only by methods of the class that defines the data.

* procedure
A paragraph or group of logically successive paragraphs, or a section or group of logically successive sections, within the PROCEDURE DIVISION.

* procedure branching statement
A statement that causes the explicit transfer of control to a statement other than the next executable statement in the sequence in which the statements are written in the source code. The procedure branching statements are: ALTER, CALL, EXIT, EXIT PROGRAM, GO TO, MERGE (with the OUTPUT PROCEDURE phrase), PERFORM and SORT (with the INPUT PROCEDURE or OUTPUT PROCEDURE phrase), XML PARSE.

PROCEDURE DIVISION
The COBOL division that contains instructions for solving a problem.

procedure integration
One of the functions of the COBOL optimizer is to simplify calls to performed procedures or contained programs.

PERFORM procedure integration is the process whereby a PERFORM statement is replaced by its performed procedures. Contained program procedure integration is the process where a call to a contained program is replaced by the program code.

* procedure-name
A user-defined word that is used to name a paragraph or section in the PROCEDURE DIVISION. It consists of a paragraph-name (which can be qualified) or a section-name.

procedure pointer
A data item in which a pointer to an entry point can be stored. A data item defined with the USAGE IS PROCEDURE-POINTER clause contains the address of a procedure entry point.

procedure-pointer data item
A data item in which a pointer to an entry point can be stored. A data item defined with the USAGE IS PROCEDURE-POINTER clause contains the address of a procedure entry point. Typically used to communicate with COBOL and Language Environment programs.

process
The course of events that occurs during the execution of all or part of a program. Multiple processes can run concurrently, and programs that run within a process can share resources.

program
(1) A sequence of instructions suitable for processing by a computer. Processing may include the use of a compiler to prepare the program for execution, as well as a runtime environment to execute it. (2) A logical assembly of one or more interrelated modules. Multiple copies of the same program can be run in different processes.

program-name
In the IDENTIFICATION DIVISION and the end program marker, a user-defined word or an alphanumeric literal that identifies a COBOL source program.
* program identification entry
  In the PROGRAM-ID paragraph of the IDENTIFICATION DIVISION, an entry that contains clauses that specify the program-name and assign selected program attributes to the program.

program-name
  In the IDENTIFICATION DIVISION and the end program marker, a user-defined word or alphanumeric literal that identifies a COBOL source program.

project
  The complete set of data and actions that are required to build a target, such as a dynamic link library (DLL) or other executable (EXE).

* pseudo-text
  A sequence of text words, comment lines, inline comments, or the separator space in a source program or COBOL library bounded by, but not including, pseudo-text delimiters.

* pseudo-text delimiter
  Two contiguous equal sign characters (==) used to delimit pseudo-text.

* punctuation character
  A character that belongs to the following set:

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>,</td>
<td>Comma</td>
</tr>
<tr>
<td>;</td>
<td>Semicolon</td>
</tr>
<tr>
<td>:</td>
<td>Colon</td>
</tr>
<tr>
<td>.</td>
<td>Period (full stop)</td>
</tr>
<tr>
<td>&quot;</td>
<td>Quotation mark</td>
</tr>
<tr>
<td>(</td>
<td>Left parenthesis</td>
</tr>
<tr>
<td>)</td>
<td>Right parenthesis</td>
</tr>
<tr>
<td>=</td>
<td>Space</td>
</tr>
<tr>
<td></td>
<td>Equal sign</td>
</tr>
</tbody>
</table>

Q

QSAM (Queued Sequential Access Method)
  An extended version of the basic sequential access method (BSAM). When this method is used, a queue is formed of input data blocks that are awaiting processing or of output data blocks that have been processed and are awaiting transfer to auxiliary storage or to an output device.

* qualified data-name
  An identifier that is composed of a data-name followed by one or more sets of either of the connectives OF and IN followed by a data-name qualifier.

* qualifier
  (1) A data-name or a name associated with a level indicator that is used in a reference either together with another data-name (which is the name of an item that is subordinate to the qualifier) or together with a condition-name. (2) A section-name that is used in a reference together with a paragraph-name specified in that section. (3) A library-name that is used in a reference together with a text-name associated with that library.

R

* random access
  An access mode in which the program-specified value of a key data item identifies the logical record that is obtained from, deleted from, or placed into a relative or indexed file.

* record
  See logical record.
**record area**
A storage area allocated for the purpose of processing the record described in a record description entry in the FILE SECTION of the DATA DIVISION. In the FILE SECTION, the current number of character positions in the record area is determined by the explicit or implicit RECORD clause.

**record description**
See record description entry.

**record description entry**
The total set of data description entries associated with a particular record. Synonymous with record description.

**recording mode**
The format of the logical records in a file. Recording mode can be F (fixed-length), V (variable-length), S (spanned), or U (undefined).

**record key**
A key whose contents identify a record within an indexed file.

**record-name**
A user-defined word that names a record described in a record description entry in the DATA DIVISION of a COBOL program.

**record number**
The ordinal number of a record in the file whose organization is sequential.

**recording mode**
The format of the logical records in a file. Recording mode can be F (fixed length), V (variable length), S (spanned), or U (undefined).

**recursion**
A program calling itself or being directly or indirectly called by one of its called programs.

**recursively capable**
A program is recursively capable (can be called recursively) if the RECURSIVE attribute is on the PROGRAM-ID statement.

**reel**
A discrete portion of a storage medium, the dimensions of which are determined by each implementor that contains part of a file, all of a file, or any number of files. Synonymous with unit and volume.

**reentrant**
The attribute of a program or routine that lets more than one user share a single copy of a program object.

**reference format**
A format that provides a standard method for describing COBOL source programs.

**reference modification**
A method of defining a new category alphanumeric, category DBCS, or category national data item by specifying the leftmost character and length relative to the leftmost character position of a USAGE DISPLAY, DISPLAY-1, or NATIONAL data item.

**reference-modifier**
A syntactically correct combination of character strings and separators that defines a unique data item. It includes a delimiting left parenthesis separator, the leftmost character position, a colon separator, optionally a length, and a delimiting right parenthesis separator.

**relation**
See relational operator or relation condition.

**relation character**
A character that belongs to the following set:

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
</tbody>
</table>

Glossary 663
* 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:

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS GREATER THAN</td>
<td>Greater than</td>
</tr>
<tr>
<td>IS &gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>IS NOT GREATER THAN</td>
<td>Not greater than</td>
</tr>
<tr>
<td>IS NOT &gt;</td>
<td>Not greater than</td>
</tr>
<tr>
<td>IS LESS THAN</td>
<td>Less than</td>
</tr>
<tr>
<td>IS &lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>IS NOT LESS THAN</td>
<td>Not less than</td>
</tr>
<tr>
<td>IS NOT &lt;</td>
<td>Not less than</td>
</tr>
<tr>
<td>IS EQUAL TO</td>
<td>Equal to</td>
</tr>
<tr>
<td>IS =</td>
<td>Equal to</td>
</tr>
<tr>
<td>IS NOT EQUAL TO</td>
<td>Not equal to</td>
</tr>
<tr>
<td>IS NOT =</td>
<td>Not equal to</td>
</tr>
<tr>
<td>IS GREATER THAN OR EQUAL TO</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>IS &gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>IS LESS THAN OR EQUAL TO</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>IS &lt;=</td>
<td>Less than or equal to</td>
</tr>
</tbody>
</table>

* relative file
  A file with relative organization.

* relative key
  A key whose contents identify a logical record in a relative file.

* relative organization
  The permanent logical file structure in which each record is uniquely identified by an integer value greater than zero, which specifies the logical ordinal position of the record in the file.

* relative record number
  The ordinal number of a record in a file whose organization is relative. This number is treated as a numeric literal that is an integer.

* reserved word
  A COBOL word that is specified in the list of words that can be used in a COBOL source program, but that must not appear in the program as a user-defined word or system-name.
* resource
A facility or service, controlled by the operating system, that an executing program can use.

* resultant identifier
A user-defined data item that is to contain the result of an arithmetic operation.

reusable environment
A reusable environment is created when you establish an assembler program as the main program by using either the old COBOL interfaces for preinitialization (RTEREUS runtime option), or the Language Environment interface, CEEPIPI.

routine
A set of statements in a COBOL program that causes the computer to perform an operation or series of related operations. In Language Environment, refers to either a procedure, function, or subroutine.

* routine-name
A user-defined word that identifies a procedure written in a language other than COBOL.

* run time
The time at which an object program is executed. Synonymous with object time.

runtime environment
The environment in which a COBOL program executes.

* run unit
A stand-alone object program, or several object programs, that interact by means of COBOL CALL or INVOKE statements and function at run time as an entity.

S

SBCS
See single-byte character set (SBCS).

scope terminator
A COBOL reserved word that marks the end of certain PROCEDURE DIVISION statements. It can be either explicit (END-ADD, for example) or implicit (separator period).

* section
A set of zero, one, or more paragraphs or entities, called a section body, the first of which is preceded by a section header. Each section consists of the section header and the related section body.

* section header
A combination of words followed by a separator period that indicates the beginning of a section in any of these divisions: ENVIRONMENT, DATA, or PROCEDURE. In the ENVIRONMENT DIVISION and DATA DIVISION, a section header is composed of reserved words followed by a separator period. The permissible section headers in the ENVIRONMENT DIVISION are:

```
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
```

The permissible section headers in the DATA DIVISION are:

```
FILE SECTION.
WORKING-STORAGE SECTION.
LOCAL-STORAGE SECTION.
LINKAGE SECTION.
```

In the PROCEDURE DIVISION, a section header is composed of a section-name, followed by the reserved word SECTION, followed by a separator period.

* section-name
A user-defined word that names a section in the PROCEDURE DIVISION.

segmentation
A feature of Enterprise COBOL that is based on the 85 COBOL Standard segmentation module. The segmentation feature uses priority-numbers in section headers to assign sections to fixed segments.
or independent segments. Segment classification affects whether procedures contained in a segment receive control in initial state or last-used state.

**selection structure**
A program processing logic in which one or another series of statements is executed, depending on whether a condition is true or false.

* **sentence**
A sequence of one or more statements, the last of which is terminated by a separator period.

* **separately compiled program**
A program that, together with its contained programs, is compiled separately from all other programs.

* **separator**
A character or two or more contiguous characters used to delimit character strings.

* **separator comma**
A comma (,) followed by a space used to delimit character strings.

* **separator period**
A period (.) followed by a space used to delimit character strings.

* **separator semicolon**
A semicolon (;) followed by a space used to delimit character strings.

**sequence structure**
A program processing logic in which a series of statements is executed in sequential order.

* **sequential access**
An access mode in which logical records are obtained from or placed into a file in a consecutive predecessor-to-successor logical record sequence determined by the order of records in the file.

* **sequential file**
A file with sequential organization.

* **sequential organization**
The permanent logical file structure in which a record is identified by a predecessor-successor relationship established when the record is placed into the file.

**serial search**
A search in which the members of a set are consecutively examined, beginning with the first member and ending with the last.

**session bean**
In EJB, an enterprise bean that is created by a client and that usually exists only for the duration of a single client/server session. (Oracle)

* **77-level-description-entry**
A data description entry that describes a noncontiguous data item that has level-number 77.

* **sign condition**
The proposition (for which a truth value can be determined) that the algebraic value of a data item or an arithmetic expression is either less than, greater than, or equal to zero.

**signature**
(1) The name of an operation and its parameters. (2) The name of a method and the number and types of its formal parameters.

* **simple condition**
Any single condition chosen from this set:
  * Relation condition
  * Class condition
  * Condition-name condition
  * Switch-status condition
  * Sign condition
See also condition and negated simple condition.
**single-byte character set (SBCS)**
A set of characters in which each character is represented by a single byte. See also ASCII and EBCDIC (Extended Binary-Coded Decimal Interchange Code).

**slack bytes (within records)**
Bytes inserted by the compiler between data items to ensure correct alignment of some elementary data items. Slack bytes contain no meaningful data. The SYNCHRONIZED clause instructs the compiler to insert slack bytes when they are needed for proper alignment.

**slack bytes (between records)**
Bytes inserted by the programmer between blocked logical records of a file, to ensure correct alignment of some elementary data items. In some cases, slack bytes between records improve performance for records processed in a buffer.

**sort file**
A collection of records to be sorted by a format 1 SORT statement. The sort file is created and can be used by the sort function only.

**sort-merge file description entry**
An entry in the FILE SECTION of the DATA DIVISION that is composed of the level indicator SD, followed by a file-name, and then followed by a set of file clauses as required.

**SOURCE-COMPUTER**
The name of an ENVIRONMENT DIVISION paragraph in which the computer environment, where the source program is compiled, is described.

**source computer entry**
An entry in the SOURCE-COMPUTER paragraph of the ENVIRONMENT DIVISION; this entry contains clauses that describe the computer environment in which the source program is to be compiled.

**source item**
An identifier designated by a SOURCE clause that provides the value of a printable item.

**source program**
Although a source program can be represented by other forms and symbols, in this document the term always refers to a syntactically correct set of COBOL statements. A COBOL source program commences with the IDENTIFICATION DIVISION or a COPY statement and terminates with the end program marker, if specified, or with the absence of additional source program lines.

**source unit**
A unit of COBOL source code that can be separately compiled: a program or a class definition. Also known as a compilation unit.

**special character**
A character that belongs to the following set:

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Plus sign</td>
</tr>
<tr>
<td>-</td>
<td>Minus sign (hyphen)</td>
</tr>
<tr>
<td>*</td>
<td>Asterisk</td>
</tr>
<tr>
<td>/</td>
<td>Slant (forward slash)</td>
</tr>
<tr>
<td>=</td>
<td>Equal sign</td>
</tr>
<tr>
<td>$</td>
<td>Currency sign</td>
</tr>
<tr>
<td>.</td>
<td>Comma</td>
</tr>
<tr>
<td>;</td>
<td>Semicolon</td>
</tr>
<tr>
<td>.</td>
<td>Period (decimal point, full stop)</td>
</tr>
<tr>
<td>&quot;</td>
<td>Quotation mark</td>
</tr>
<tr>
<td>'</td>
<td>Apostrophe</td>
</tr>
</tbody>
</table>
SPECIAL-NAMES
The name of an ENVIRONMENT DIVISION paragraph in which environment-names are related to user-specified mnemonic-names.

* special names entry
An entry in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION; this entry provides means for specifying the currency sign; choosing the decimal point; specifying symbolic characters; relating implementor-names to user-specified mnemonic-names; relating alphabet-names to character sets or collating sequences; and relating class-names to sets of characters.

* special registers
Certain compiler-generated storage areas whose primary use is to store information produced in conjunction with the use of a specific COBOL feature.

* standard data format
The concept used in describing the characteristics of data in a COBOL DATA DIVISION under which the characteristics or properties of the data are expressed in a form oriented to the appearance of the data on a printed page of infinite length and breadth, rather than a form oriented to the manner in which the data is stored internally in the computer, or on a particular external medium.

* statement
A syntactically valid combination of words, literals, and separators, beginning with a verb, written in a COBOL source program.

structured programming
A technique for organizing and coding a computer program in which the program comprises a hierarchy of segments, each segment having a single entry point and a single exit point. Control is passed downward through the structure without unconditional branches to higher levels of the hierarchy.

* subclass
A class that inherits from another class. When two classes in an inheritance relationship are considered together, the subclass is the inheritor or inheriting class; the superclass is the inheritee or inherited class.

* subject of entry
An operand or reserved word that appears immediately following the level indicator or the level-number in a DATA DIVISION entry.

* subprogram
See called program.

* subscript
An occurrence number that is represented by either an integer, a data-name optionally followed by an integer with the operator + or -, or an index-name optionally followed by an integer with the operator + or -, that identifies a particular element in a table. A subscript can be the word ALL when the subscripted identifier is used as a function argument for a function allowing a variable number of arguments.

* subscripted data-name
An identifier that is composed of a data-name followed by one or more subscripts enclosed in parentheses.
substitution character
A character that is used in a conversion from a source code page to a target code page to represent a character that is not defined in the target code page.

* superclass
A class that is inherited by another class. See also subclass.

surrogate pair
In the UTF-16 format of Unicode, a pair of encoding units that together represents a single Unicode graphic character. The first unit of the pair is called a high surrogate and the second a low surrogate. The code value of a high surrogate is in the range X'D800' through X'DBFF'. The code value of a low surrogate is in the range X'DC00' through X'DFFF'. Surrogate pairs provide for more characters than the 65,536 characters that fit in the Unicode 16-bit coded character set.

switch-status condition
The proposition (for which a truth value can be determined) that an UPSI switch, capable of being set to an on or off status, has been set to a specific status.

* symbolic-character
A user-defined word that specifies a user-defined figurative constant.

syntax

* system-name
A COBOL word that is used to communicate with the operating environment.

T

* table
A set of logically consecutive items of data that are defined in the DATA DIVISION by means of the OCCURS clause.

* table element
A data item that belongs to the set of repeated items comprising a table.

text deck
Synonym for object deck or object module.

* text-name
A user-defined word that identifies library text.

* text word
A character or a sequence of contiguous characters between margin A and margin R in a COBOL library, source program, or pseudo-text that is any of the following characters:

- A separator, except for space; a pseudo-text delimiter; and the opening and closing delimiters for alphanumeric literals. The right parenthesis and left parenthesis characters, regardless of context within the library, source program, or pseudo-text, are always considered text words.
- A literal including, in the case of alphanumeric literals, the opening quotation mark and the closing quotation mark that bound the literal.
- Any other sequence of contiguous COBOL characters except comment lines and the word COPY bounded by separators that are neither a separator nor a literal.

thread
A stream of computer instructions (initiated by an application within a process) that is in control of a process.

token
In the COBOL editor, a unit of meaning in a program. A token can contain data, a language keyword, an identifier, or other part of the language syntax.
**top-down design**
The design of a computer program using a hierarchic structure in which related functions are performed at each level of the structure.

**top-down development**
See structured programming.

**trailer-label**
(1) A data-set label that follows the data records on a unit of recording medium. (2) Synonym for end-of-file label.

**troubleshoot**
To detect, locate, and eliminate problems in using computer software.

**truth value**
The representation of the result of the evaluation of a condition in terms of one of two values: true or false.

**typed object reference**
A data-name that can refer only to an object of a specified class or any of its subclasses.

**U**

**unary operator**
A plus (+) or a minus (-) sign that precedes a variable or a left parenthesis in an arithmetic expression and that has the effect of multiplying the expression by +1 or -1, respectively.

**unbounded table**
A table with OCCURS int-1 to UNBOUNDED instead of specifying int-2 as the upper bound.

**Unicode**
A universal character encoding standard that supports the interchange, processing, and display of text that is written in any of the languages of the modern world. There are multiple encoding schemes to represent Unicode, including UTF-8, UTF-16, and UTF-32. Enterprise COBOL supports Unicode using UTF-16 in big-endian format as the representation for the national data type.

**Uniform Resource Identifier (URI)**
A sequence of characters that uniquely names a resource; in Enterprise COBOL, the identifier of a namespace. URI syntax is defined by the document Uniform Resource Identifier (URI): Generic Syntax.

**unit**
A module of direct access, the dimensions of which are determined by IBM.

**universal object reference**
A data-name that can refer to an object of any class.

**unrestricted storage**
Storage below the 2 GB bar. It can be above or below the 16 MB line. If it is above the 16 MB line, it is addressable only in 31-bit mode.

**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.
* variable
A data item whose value can be changed by execution of the object program. A variable used in an arithmetic expression must be a numeric elementary item.

variable-length item
A group item that contains a table described with the DEPENDING phrase of the OCCURS clause.

* variable-length record
A record associated with a file whose file description or sort-merge description entry permits records to contain a varying number of character positions.

* variable-occurrence data item
A variable-occurrence data item is a table element that is repeated a variable number of times. Such an item must contain an OCCURS DEPENDING ON clause in its data description entry or be subordinate to such an item.

* variably located group
A group item following, and not subordinate to, a variable-length table in the same record. The group item can be an alphanumeric group or a national group.

* variably located item
A data item following, and not subordinate to, a variable-length table in the same record.

* verb
A word that expresses an action to be taken by a COBOL compiler or object program.

volume
A module of external storage. For tape devices it is a reel; for direct-access devices it is a unit.

volume switch procedures
System-specific procedures that are executed automatically when the end of a unit or reel has been reached before end-of-file has been reached.

VSAM file system
A file system that supports COBOL sequential, relative, and indexed organizations.

W

web service
A modular application that performs specific tasks and is accessible through open protocols like HTTP and SOAP.

white space
Characters that introduce space into a document. They are:

- Space
- Horizontal tabulation
- Carriage return
- Line feed
- Next line
as named in the Unicode Standard.

* word
A character string of not more than 30 characters that forms a user-defined word, a system-name, a reserved word, or a function-name.

* WORKING-STORAGE SECTION
The section of the DATA DIVISION that describes WORKING-STORAGE data items, composed either of noncontiguous items or WORKING-STORAGE records or of both.

workstation
A generic term for computers, including personal computers, 3270 terminals, intelligent workstations, and UNIX terminals. Often a workstation is connected to a mainframe or to a network.
**wrapper**

An object that provides an interface between object-oriented code and procedure-oriented code. Using wrappers lets programs be reused and accessed by other systems.

**X**

The symbol in a PICTURE clause that can hold any character in the character set of the computer.

**XML**

Extensible Markup Language. A standard metalanguage for defining markup languages that was derived from and is a subset of SGML. XML omits the more complex and less-used parts of SGML and makes it much easier to write applications to handle document types, author and manage structured information, and transmit and share structured information across diverse computing systems. The use of XML does not require the robust applications and processing that is necessary for SGML. XML is developed under the auspices of the World Wide Web Consortium (W3C).

**XML data**

Data that is organized into a hierarchical structure with XML elements. The data definitions are defined in XML element type declarations.

**XML declaration**

XML text that specifies characteristics of the XML document such as the version of XML being used and the encoding of the document.

**XML document**

A data object that is well formed as defined by the W3C XML specification.

**XML namespace**

A mechanism, defined by the W3C XML Namespace specifications, that limits the scope of a collection of element names and attribute names. A uniquely chosen XML namespace ensures the unique identity of an element name or attribute name across multiple XML documents or multiple contexts within an XML document.

**XML schema**

A mechanism, defined by the W3C, for describing and constraining the structure and content of XML documents. An XML schema, which is itself expressed in XML, effectively defines a class of XML documents of a given type, for example, purchase orders.

**Z**

**z/OS UNIX file system**

A collection of files and directories that are organized in a hierarchical structure and can be accessed by using z/OS UNIX.

**zoned decimal data item**

An external decimal data item that is described implicitly or explicitly as USAGE DISPLAY and that contains a valid combination of PICTURE symbols 9, S, P, and V. The content of a zoned decimal data item is represented in characters 0 through 9, optionally with a sign. If the PICTURE string specifies a sign and the SIGN IS SEPARATE clause is specified, the sign is represented as characters + or -. If SIGN IS SEPARATE is not specified, the sign is one hexadecimal digit that overlays the first 4 bits of the sign position (leading or trailing).

#

**85 COBOL Standard**

The COBOL language defined by the following standards:

2002 COBOL Standard
The COBOL language defined by the following standard:


2014 COBOL Standard
The COBOL language defined by the following standard:

List of resources

Enterprise COBOL for z/OS

COBOL for z/OS publications
You can find the following publications in the Enterprise COBOL for z/OS library:

- *Customization Guide*, SC27-8712-02
- *Language Reference*, SC27-8713-02
- *Programming Guide*, SC27-8714-02
- *Migration Guide*, GC27-8715-02
- *Performance Tuning Guide*, SC27-9202-01
- *Messages and Codes*, SC27-4648-01
- *Program Directory*, GI13-4526-02
- *Licensed Program Specifications*, GI13-4532-02

Softcopy publications

- z/OS Software Products Collection
- z/OS and Software Products DVD Collection

Support

If you have a problem using Enterprise COBOL for z/OS, see the following site that provides up-to-date support information: [https://www.ibm.com/support/home/product/B984385H82239E03/Enterprise_COBOL_for_z/OS](https://www.ibm.com/support/home/product/B984385H82239E03/Enterprise_COBOL_for_z/OS).

Related publications

z/OS library publications
You can find the following publications in the z/OS library.

Run-Time Library Extensions
- *Common Debug Architecture Library Reference*
- *Common Debug Architecture User’s Guide*
- *DWARF/ELF Extensions Library Reference*

z/Architecture®
- *Principles of Operation*

z/OS DFSMS
- *Access Method Services for Catalogs*
- *Checkpoint/Restart*
- *Macro Instructions for Data Sets*
You can find the following publications in the CICS library:

- Developing CICS Applications
You can find the following publications in the Db2 library:

- Application Programming and SQL Guide
- Command Reference
- SQL Reference

IBM Debug for z/OS (formerly IBM Debug for z Systems and IBM Debug Tool for z/OS)

You can find information about IBM Debug for z/OS in the IBM Debug for z/OS library.

IBM Debug for z/OS supersedes IBM Debug for z Systems® and IBM Debug Tool for z/OS. Not all references to IBM Debug for z Systems and IBM Debug Tool for z/OS have been changed in the COBOL documentation library. It is recommended that you upgrade your debugger to the latest level in order to have the full range of debugging features available. In some cases, you must upgrade your debugger to a certain version depending on what level of Enterprise COBOL you are using to create the COBOL application:

- IBM Debug Tool V13.1 supports Enterprise COBOL V5.1 and earlier versions
- IBM Debug for z Systems V14.0 supports Enterprise COBOL V6.1 and earlier versions
- IBM Debug for z Systems V14.1 supports Enterprise COBOL V6.2 and earlier versions
- IBM Debug for z/OS V14.2 supports Enterprise COBOL V6.3 and earlier versions

To find out which IBM debug product best suits your needs, see https://www.ibm.com/support/knowledgecenter/SSQ2R2_14.2.0/com.ibm.debug.cg.doc/common/dcompo.html?sc=SSQ2R2_latest.

IBM Developer for z/OS (formerly IBM Developer for z Systems)

You can find information about IBM Developer for z Systems in the IBM Developer for z/OS library.

Note: IBM Developer for z/OS supersedes IBM Developer for z Systems and Rational® Developer for z Systems.

You can find the following publications by searching their publication numbers in the IBM Publications Center.

IMS
- Application Programming API Reference, SC18-9699
- Application Programming Guide, SC18-9698

WebSphere® Application Server for z/OS
- Applications, SA22-7959
Softcopy publications for z/OS
The following collection kit contains z/OS and related product publications:
• z/OS CD Collection Kit, SK3T-4269

Java
• IBM SDK for Java - Tools Documentation, publib.boulder.ibm.com/infocenter/javasdk/tools/index.jsp
• Java 2 on z/OS, www.ibm.com/servers/eserver/zseries/software/java/
• The Java EE 5 Tutorial, download.oracle.com/javaee/5/tutorial/doc/
• The Java Native Interface, download.oracle.com/javase/1.5.0/docs/guide/jni/
• JDK 5.0 Documentation, download.oracle.com/javase/1.5.0/docs/

JSON
• JavaScript Object Notation (JSON), www.json.org

Unicode and character representation
• Unicode, www.unicode.org/
• Character Data Representation Architecture Reference and Registry, SC09-2190

XML
• Extensible Markup Language (XML), www.w3.org/XML/
• Namespaces in XML 1.0, www.w3.org/TR/xml-names/
• Namespaces in XML 1.1, www.w3.org/TR/xml-names11/
• XML specification, www.w3.org/TR/xml/
Index

Special Characters
- (minus)
  insertion character 203, 204
  SIGN clause 212
  symbol in PICTURE clause 193
, (comma)
  insertion character 202
  symbol in PICTURE clause 191, 193
: (colon)
  description 46
  required use of 530
/ or *> (comment line 56
(period) symbol in PICTURE clause 191
* symbol in PICTURE clause 191
*> (floating comment indicator) 56
*CBL (*CONTROL) statement 522
*CONTROL (*CBL) statement 522
( slash
  insertion character 202
  symbol in PICTURE clause 193
+ (plus)
  insertion character 203–205
  SIGN clause 212
  symbol in PICTURE clause 193
< (less than) 250
<= (less than or equal to) 250
= (equal) 250
> (greater than) 250
>= (greater than or equal to) 250
>> (compiler directive indicator)
  CALLINTERFACE directive 545
  compiler directive 545
  DEFINE directive 548
  EVALUATE directive 549
  IF directive 551
  INLINE directive 546
$ (default currency symbol)
  in PICTURE clause 193
  insertion character 203, 204
  symbol in PICTURE clause 192

Numerics
0
  insertion character 202
  symbol in PICTURE clause 193
0 symbol in PICTURE clause 191
2002 COBOL Standard features 615
2014 COBOL Standard features 615
64-bit
  considerations xxiv
66, RENAMES data description entry 210
66, renames level-number 152
77, elementary item level-number 152
88, condition-name data description entry 176
88, conditional variable level number 152
9 symbol in PICTURE clause 191
9, symbol in PICTURE clause 193

A
A symbol in PICTURE clause 190
abbreviated combined relation condition
  examples 266
  using parentheses in 264
ABS function 471
ACCEPT statement
  description and format 283
  FROM phrase 283
  mnemonic-name in 283
  overlapping operands, unpredictable results 275
  system information transfer 284
access mode
  description 132
  dynamic
    DELETE statement 306
    description 133
    READ statement 390
  DYNAMIC 133
  random
    DELETE statement 306
    description 133
    READ statement 389
  RANDOM 133
  sequential
    DELETE statement 306
    description 133
    READ statement 388
SEQUENTIAL 132
ACCESS MODE clause 132
accessibility
  keyboard navigation 625
  of Enterprise COBOL for z/OS 625
  of this information 625
  using z/OS 625
accessibility features for this product 625
ACOS function 471
ADD statement
  common phrases 271
  CORRESPONDING phrase 288
  description and format 287
  END-ADD phrase 288
  GIVING phrase 287
  NOT ON SIZE ERROR phrase 288
  ON SIZE ERROR phrase 288
  ROUNDED phrase 288
ADDRESS OF special register 17
ADV compiler option 435
advanced function printing 438
ADVANCING phrase 434
AFTER phrase
  INSPECT statement 333
  PERFORM statement 379
AFTER phrase (continued)
with REPLACING 330
with TALLYING 329
WRITE statement 435
alignment rules 158
ALL literal
  figurative constant 14
  STOP statement 418
  STRING statement 420
ALL phrase
  INSPECT statement 329, 330
  SEARCH statement 399
  UNSTRING statement 427
ALL subscripting 68, 464
ALLOCATE statement
description and format 289
UNBOUNDED tables 291
ALPHABET clause 111
alphabet-name
description 111
  MERGE statement 360
  PROGRAM COLLATING SEQUENCE clause 107
  SORT statement 412
alphabet category 155
alphabet character in ACCEPT 283
ALPHABETIC class test 247
alphabetic function arguments 462
alphabetic items
  alignment rules 158
  elementary move rules 365
  how to define 194
  PICTURE clause 194
ALPHABETIC-LOWER class test 247
ALPHABETIC-UPPER class test 247
alphanumeric category 155
alphanumeric comparisons 253
alphanumeric function arguments 462
alphanumeric functions 460, 461
alphanumeric group items 152
alphanumeric items
  alignment rules 158
  elementary move rules 365
  how to define 194
  PICTURE clause 194
alphanumeric literals
  in hexadecimal notation 36
  with DBCS characters 35
alphanumeric operands, comparing 253
alphanumeric-edited category 156
alphanumeric-edited items
  alignment rules 158
  elementary move rules 365
  how to define 195
  PICTURE clause 195
ALSO phrase
  ALPHABET clause 111
  EVALUATE statement 314
ALTER statement
description and format 292
  GO TO statement and 321
  segmentation considerations 293
  altered GO TO statement 321
alternate key data item 135
ALTERNATE RECORD KEY clause
  ALTERNATE RECORD KEY clause (continued)
  DUPLICATES phrase 136
  AND logical operator 261
  ANNUITY function 472
  ANSI COBOL standards 613
  ANSI X3.22 609
  ANSI X3.27 609
  ANSI X3.4 609
  APPLY WRITE-ONLY clause 142
  Area A (cols. 8-11) 52
  Area B (cols. 12-72) 53
  arguments 462
  arithmetic expression
    COMPUTE statement 304
    description 244
    EVALUATE statement 315
    relation condition 250
  arithmetic operators
    description 244
    permissible symbol pairs 245
  arithmetic statements
    ADD 287
    common phrases 271
    COMPUTE 304
    DIVIDE 309
    list of 274
    multiple results 275
    MULTIPLY 369
    operands 274
    programming notes 275
    SUBTRACT 423
ASCENDING KEY phrase
collating sequence 184
description 359
  MERGE statement 359
  OCCURS clause 183
  SORT statement 409, 410
ASCII
collating sequence 582
  specifying in SPECIAL-NAMES paragraph 111
ASCII considerations
  ASSIGN clause 610
  CODE-SET clause 610
  data description entries 611
  DATA DIVISION 610
  ENVIRONMENT DIVISION 609
  I-O-CONTROL paragraph 610
  OBJECT-COMPUTER paragraph 609
  PROCEDURE DIVISION 611
  PROGRAM COLLATING SEQUENCE clause 609
  SPECIAL-NAMES paragraph 609
ASCII standard 613
ASIN function 472
ASSIGN clause
  ASCII considerations 610
  description 126
  format 122
  SELECT clause and 126
assigning index values 401
assignment-name
  ASSIGN clause 126
  environment variable 126
  RECORD DELIMITER clause 132
  RERUN clause 139
assistive technologies 626
asterisk (*)
  comment line 56
  insertion character 205
AT END phrase
  READ statement 387
  RETURN statement 393
  SEARCH statement 396
  SEARCH statement (binary search) 399
  SEARCH statement (serial search) 397
AT END-OF-PAGE phrases 436
at-end condition
  READ statement 389
  RETURN statement 393
ATAN function 473
ATTRIBUTE-CHARACTER XML event 25
ATTRIBUTE-CHARACTERS XML event 25
ATTRIBUTE-NAME XML event 25
ATTRIBUTE-NATIONAL-CHARACTER XML event 25
ATTRIBUTES phrase 444
AUTHOR paragraph
  description 100
  format 93
B
  insertion character 202
    symbol in PICTURE clause 189
  basic character set 3
  basic PERFORM statement
    format and description 376
  Basic UTF-8 literals 39
  BASIS statement 521
  basis-name 60
  batch compile 81
BEFORE phrase
  INSPECT statement 333
  PERFORM statement 379
  with REPLACING 330
  with TALLYING 329
  WRITE statement 435
Bibliography 675
  big-endian 5
  binary arithmetic operators 244
  binary data item, DISPLAY statement 307
  BINARY phrase in USAGE clause 219
  binary search 399
  BIT-OF function 473
  BIT-TO-CHAR function 474
  blank lines 57
BLANK WHEN ZERO clause
  description and format 177
  INDEX phrase in USAGE clause 222
BLOCK CONTAINS clause
  description 167
  format 161
  boolean conditions
    description 554
  branching
    GO TO statement 320
    out-of-line PERFORM statement 377
BY CONTENT phrase
  CALL statement 296
  CALL statement 295
  BY VALUE phrase
    CALL statement 297
  INVOKE statement 336
  BYTE-LENGTH function 474
C
call convention 545, 546
CALL statement
  CANCEL statement and 300
    description and format 293
  LINKAGE SECTION 241
  ON OVERFLOW phrase 293
  PROCEDURE DIVISION header 238, 241
  program termination 293
  subprogram linkage 293
  transfer of control 75
  USING phrase 241
called and calling programs, description 293
CALLINTERFACE directive 545
CANCEL statement 300
carriage control character 435
category
  of group items 152
  relationship to classes of data 153
  relationship to usages of data 153
category descriptions 155
category of data
  alphabetic 155, 194
  alphanumeric 155, 194
  alphanumeric-edited 156, 195
  DBCS 156, 195
  external floating-point 156
  internal floating-point 156
  national 156, 197
  national-edited 156, 197
  numeric 157, 198
  numeric-edited 157, 199
  UTF-8 157, 200
category of functions 154
category of literals 155
CBL (PROCESS) statement 522
CBLQDA runtime option 374
CCSID 5
CHAR function 475
character code set, specifying 111
character encoding unit 5
character sets 5
character-strings
  COBOL words 9
  representation in PICTURE clause 193
  size determination 158
CHARACTERS BY phrase 330
CHARACTERS phrase
  BLOCK CONTAINS clause 167
  INSPECT statement 329
  MEMORY SIZE clause 107
  USAGE clause and 167
characters, valid in COBOL program 3
checkpoint processing, RERUN clause 139
CICS
  restrictions

Index 681
CICS (continued)
restrictions (continued)
parsing with validation using FILE 452
class (object-oriented) 85
class (of data)
of data items 153
of figurative constants 154
of functions 154
of group items 152
of literals 154
CLASS clause 113
class condition 246
class definition
class procedure division 237
CLASS-ID paragraph 98
description 85
effect of SELF and SUPER 335
factory procedure division 237
IDENTIFICATION DIVISION 94
object procedure division 237
requirements for indexed tables 184
class identification division 93
class IDENTIFICATION DIVISION 98
class procedure division 237
CLASS-ID paragraph 98
class-name 11, 12, 60
class-name class test 247
class-name, OO 60
clauses
definition 50
syntactical hierarchy 49
CLOSE statement
format and description 301
COBOL
class definition 85
language structure 3
method definition 89
program structure 79
reference format 51
COBOL classes 85
COBOL objects 85
COBOL standards 613
COBOL words
with DBCS characters 9
with single-byte characters 9
code page names 5
code pages 5
CODE-SET clause
ALPHABET clause and 111
ASCII considerations 610
description 174
format 161
NATIVE phrase and 174
CODEPAGE compiler option 5
collating sequence
ASCENDING/DESCENDING KEY phrase and 184
ASCII 582
EBCDIC 579
specified in OBJECT-COMPUTER paragraph 107
specified in SPECIAL-NAMES paragraph 111
COLLATING SEQUENCE phrase
ALPHABET clause 111
MERGE statement 360
COLLATING SEQUENCE phrase (continued)
SORT statement 412
colon character
description 46
required use of 530
column 7
indicator area 54
specifying comments 56
combined condition
description 262
evaluation rules 263
logical operators and evaluation results 263
order of evaluation 264
permissible element sequences 262
comma ()
DECIMAL-POINT IS COMMA clause 115
insertion character 202
comment lines
description 56
in IDENTIFICATION DIVISION 100
in library text 528
in source text 538
COMMENT XML event 25
comments
sending xxiv
COMMON clause 97
common processing facilities 276
COMP-1 through COMP-5 data items 220
comparison tables 251
comparison types 251
comparisons
alphanumeric operands 253
cycle, INSPECT statement 333
DBCS operands 254
Dynamic-length elementary item operands 255
function pointer operands 259
group operands 257
in EVALUATE statement 316
index data items 257
index-names 257
national operands 255
numeric operands 257
object reference operands 260
procedure pointer operands 259
rules for COPY statement 527
UTF-8 operands 256
compile-time arithmetic expressions
description 555
compiler directive 545
compiler limits 573
compiler options
ADV 435
CODEPAGE 5
controlling listing output 522
NUMPROC 260
PGMNAME 300
specifying 522
THREAD 184
TRUNC 159
compiler-directing statements
*CBL (*CONTROL) 522
*CBL (*CONTROL 522
BASIS 521
CBL (PROCESS) 522
compiler-directing statements (continued)
COPY 524
DELETE 533
EJECT 534
ENTER 534
INSERT 535
PROCESS (CBL) 522
READY TRACE 536
REPLACE 536
RESET TRACE 536
SERVICE LABEL 539
SERVICE RELOAD 540
SKIP 1 540
SKIP 2 540
SKIP 3 540
TITLE 540
USE 541
complex conditions
abbreviated combined relation 264
combined condition 262
description 261
negated simple 262
complex OCCURS DEPENDING ON (CODO) 188
composite of operands 274
COMPUTATIONAL data items 219
COMPUTATIONAL phrases in USAGE clause 220
COMPUTE statement
common phrases 273
description and format 304
computer-name 12, 106, 107
condition
abbreviated combined relation 264
class 246
combined 262
complex 261
description
condition-name 248
EVALUATE statement 315
IF statement 322
negated simple 262
PERFORM UNTIL statement 379
relation 250
SEARCH statement (binary search) 399
SEARCH statement (serial search) 398
sign 260
simple 246
switch-status 261
condition-name
and conditional variable 176
description and format 248
rules for values 230
SEARCH statement 400
SET statement 404
SPECIAL-NAMES paragraph 111
switch status condition 111
conditional compilation
description 547
directives
DEFINE directive 548
EVALUATE directive 549
IF directive 551
examples 552
predefined compilation variables 555
conditional expressions
boolean conditions 554
conditional expressions (continued)
compile-time arithmetic expressions 555
constant conditional expressions 553
DBCS operands 254
defined condition expressions 554
description 246
Dynamic-length elementary item operands 255
index-names and index data items 257
order of evaluation of operands 263
parentheses in abbreviated combined relation conditions 264
UTF-8 operands 256
conditional statements
description 269
GO TO statement 321
IF statement 322
list of 269
PERFORM statement 379
conditional variable 176
configuration section
description (programs, classes, methods) 105
REPOSITORY paragraph 116
SOURCE-COMPUTER paragraph 106
SPECIAL-NAMES paragraph 108
conformance rules
SET...USAGE OBJECT REFERENCE 407
current condition expressions
description 553
contained programs 79
CONTENT-CHARACTER XML event 25
CONTENT-CHARACTERS XML event 25
CONTENT-NATIONAL-CHARACTER XML event 25
collection-sensitive word 607
continuation
area 51
lines 54, 56
CONTINUE statement 305
CONTROL statement (*CONTROL) 522
conversion of data, DISPLAY statement 307
CONVERTING phrase
JSON GENERATE statement 345
JSON PARSE statement 353
COPY libraries 64
COPY statement
collection rules 527
description and format 524
description
replacement rules 527
REPLACING phrase 526
searching order 533
SUPPRESS option 526
CORRESPONDING (CORR) phrase
ADD statement 288
description 288
MOVE statement 363
SUBTRACT statement 424
with ON SIZE ERROR phrase 274
COS function 475
COUNT IN phrase
UNSTRING statement 428
XML GENERATE statement 443
COUNT phrase
JSON GENERATE statement 344
CR (credit)
data
  alignment 158
  categories 153, 194
  classes 153
  hierarchies used in qualification 150
  organization 130
  signed 159
  truncation of 159, 180
data category
  alphabetic 194
  alphanumeric 194
  alphanumeric-edited 195
  DBCS 195
  national 197
  national-edited 197
  numeric 198
  numeric-edited 199
  UTF-8 200
data category descriptions 155
data conversion, DISPLAY statement 307
data description entries
  ASCII considerations 611
data description entry
  BLANK WHEN ZERO clause 177
data-name 176
  DYNAMIC LENGTH clause 178
  FILLER phrase 177
  GLOBAL clause 179
  indentation and 152
  JUSTIFIED clause 180
  level-66 format (previously defined items) 176
  level-88 format (condition-names) 176
  level-number description 176
  OCCURS clause 182
  OCCURS DEPENDING ON (ODO) clause
    format 185
  PICTURE clause 188
  REDEFINES clause 206
  RENAMES clause 210
  SIGN clause 211
  SYNCHRONIZED clause 212
  USAGE clause 218
  USAGE IS NATIONAL clause and 180
  VALUE clause 226
  VOLATILE clause 232
  data division
  (continued)
Index 685

data-name (continued)
definition 59
data-pointer
  USAGE clause 223, 224
DATE 285
DATE YYYYMMDD 286
DATE-COMPILED paragraph
description 100
  format 93
DATE-OF-INTEGER function 477
DATE-TO-YYYYMMDD function 477
DATE-WRITTEN paragraph
description 100
  format 93
DAY 286
DAY YYYYDDD 286
DAY-OF-INTEGER function 478
DAY-OF-WEEK 286
DAY-TO-YYYYDDD function 478
DB (debit)
symbol in PICTURE clause 191
DBCS (Double-Byte Character Set)
elementary move rules 365
  using in comments 100
DBCS category 156
DBCS character set 3
DBCS characters
  in COBOL words 10
  in literals 36
DBCS class condition 247
DBCS comparisons 254
DBCS function arguments 462
DBCS items
  alignment rules 158
  how to define 195
  in ACCEPT 283
  PICTURE clause 195
DBCS literals
  in ACCEPT 283
DBCS notation xxii
de-editing 366
DEBUG-CONTENTS 17
DEBUG-ITEM special register 17, 587
DEBUG-LINE 17
DEBUG-NAME 17
debugging 587
DEBUGGING declarative 541, 543
debugging lines 57, 106, 587
debugging mode
  compile-time switch 588
  object-time switch 588
DEBUGGING MODE clause 106, 543, 587, 588
debugging sections 587
decimal point (.) 273
DECIMAL-POINT IS COMMA clause
description 115
  NUMVAL function 491
  NUMVAL-C function 492
declarative procedures
description and format 241
  PERFORM statement 376
  USE statement 241
declaratives (continued)
DEBUGGING 543
EXCEPTION/ERROR 541
  precedence rules for nested programs 543
DECLARATIVES key word
  begin in Area A 53
  description 241
  declaratives section 241
DEFINE directive 548
defined condition expressions
description 554
DELETE statement
description and format 533
  dynamic access 306
  format and description 305
  INVALID KEY phrase 306
  random access 306
  sequential access 306
DELIMITED BY phrase
  STRING 419
UNSTRING statement 427
delimited scope statement 270
delimiter
  INSPECT statement 331
  UNSTRING statement 427
DELEMITER IN phrase, UNSTRING statement 428
DEPENDING phrase
  GO TO statement 321
  OCCURS clause 185
derived class 85
DESCENDING KEY phrase
collating sequence 184
description 359
  MERGE statement 359
  SORT statement 409, 410
disability 625
display floating-point 196
DISPLAY phrase in USÄGE clause 221
DISPLAY statement
description and format 307
DISPLAY-OF function 479
DIVIDE statement
description and format 309
  COMMON phrases 273
  REMAINDER phrase 312
division header
  format, ENVIRONMENT DIVISION 105
  format, IDENTIFICATION DIVISION 93
  format, PROCEDURE DIVISION 238
  specification of 52
DO-UNTIL structure, PERFORM statement 379
DO-WHILE structure, PERFORM statement 379
DOCUMENT-TYPE-DECLARATION XML event 25
Double-Byte Character Set (DBCS)
  PICTURE clause and 195
  using in comments 100
DOWN BY phrase, SET statement 403
DUPLICATES phrase
  SORT statement 412
dynamic access mode
data organization and 134
DELETE statement 306
description 133
  READ statement 390
DYNAMIC LENGTH clause
  description 178
  format 178
Dynamic-length elementary item comparisons 255
dynamic-length items
  dynamic-length elementary items 159
dynamic-length group items 159

E

E function 480
E symbol in PICTURE clause 190
EBCDIC
  code page 1140
  CODE-SET clause and 174
  collating sequence 579
  specifying in SPECIAL-NAMES paragraph 111
editing
  fixed insertion 203
  floating insertion 204
  replacement 205
  signs 159
  simple insertion 202
  special insertion 202
  suppression 205
editing sign control symbol 191
editing signs 159
EGCS 324, 645
eject page 56
EJECT statement 534
elementary items
  alignment rules 158
  basic subdivisions of a record 150
  MOVE statement 364
  size determination in program 158
  size determination in storage 158
elementary move rules 364
ELSE NEXT SENTENCE phrase 322
ENCODING phrase
  XML GENERATE statement 443
ENCODING phrase, in XML PARSE 452
encoding units 5
ENCODING-DECLARATION XML event 25
der class marker 53
END DECLARATIVES key word 241
end markers 53
development marker 53
END PROGRAM 81
dev program marker 53
END-ADD phrase 288
END-CALL phrase 299
END-IF phrase 322
END-INVOKE phrase 339
END-JSOM phrase
  JSON GENERATE statement 346
  JSON PARSE statement 354
END-OF-CDATA-SECTION XML event 25
END-OF-DIGEST XML event 25
END-OF-ELEMENT XML event 25
end-of-file processing 301
END-OF-INPUT XML event 25
END-OF-PAGE phrases 436
END-PERFORM phrase 376
END-SUBTRACT phrase 425
END-WRITE phrase 437
END-XML phrase
  XML GENERATE statement 447
  XML PARSE statement 454
entries
  definition 50
  syntactical hierarchy 49
ENTRY statement
  description and format 312
  subprogram linkage 312
environment division
  configuration section
    ALPHABET clause 111
    CURRENCY SIGN clause 113
    OBJECT-COMPUTER paragraph 107
    REPOSITORY paragraph 116
    SPECIAL-NAMES paragraph 113
    SYMBOLIC CHARACTERS clause 115
    XML-SCHEMA clause 115
    REPOSITORY paragraph 116
ENVIRONMENT DIVISION
  ASCII considerations 609
  configuration section
    SOURCE-COMPUTER paragraph 106
    SPECIAL-NAMES paragraph 108
  input-output section
    FILE-CONTROL paragraph 122
environment variable
  assignment-name 126
  DSN option 116, 126
  for a line sequential file 126
  for a QSAM file 126
  for a VSM file 126
  for an XML schema file 116
  PATH option 116, 126
  XML schema file 116
environment-name
  SPECIAL-NAMES paragraph 110, 111
EOP phrases 436
equal sign (=) 250
EQUAL TO relational operator 250
EUC 5
Euro currency sign
  specifying in CURRENCY SIGN clause 113
EVALUATE directive 549
EVALUATE statement
  comparing operands 316
  determining truth value 315
  format and description 313
evaluation rules
  combined conditions 263
  EVALUATE statement 316
  nested IF statement 323
EXCEPTION XML event 25
EXCEPTION/ERROR declarative
  CLOSE statement 302
  DELETE statement 306
description and format 541
execution flow
  ALTER statement 292
  basic PERFORM statement 376
  PERFORM statement 376
EXIT METHOD statement
  format and description 317
fixed-length
  records 167
floating comment indicator (*>)
  comment lines 56
  description 56
  inline comment 56
floating comment indicators 13
floating comment indicators (*>) 647
floating insertion editing 204
floating-point
  DISPLAY statement 307
floating-point literals 41
FOOTING phrase of LINAGE clause 171
FOR REMOVAL phrase 301, 302
format notation, rules for xix
FREE statement
  description and format 319
  UNBOUNDED tables 291
FROM phrase
  ACCEPT statement 283
  REWRITE statement 393
  SUBTRACT statement 423
    with identifier 281
  WRITE statement 434
function arguments 462
function definitions 465
function pointer
  in SET statement 401
function pointer data items
  relation condition 259
function type 460
function-identifier 73
function-names 12
function-pointer data items
  in SET statement 401
FUNCTION-POINTER phrase in USAGE clause 221
functions
  arguments 462
  categories 154
  class and category of 154
  classes 154
  description 459
  rules for usage 461
  types of functions 460

G
G symbol in PICTURE clause 190
garbage collection 85
GIVING phrase
  ADD statement 287
  arithmetic 273
  DIVIDE statement 312
  MERGE statement 361
  MULTIPLY statement 370
  SORT statement 414
  SUBTRACT statement 424
GLOBAL clause
  with data item 179
    with file name 167
Glossary 631
GO TO statement
  altered 321
    conditional 321
GO TO statement (continued)
  format and description 320
SEARCH statement 396, 400
  unconditional 320
GO TO, DEPENDING ON phrase 292
GOBACK statement 319
graphic character 5
GREATER THAN OR EQUAL TO symbol (>=) 250
GREATER THAN symbol (>) 250
group comparisons 257
group items
  alphanumeric 152
  class and category of 152
  description 150
  MOVE statement 368
  national 153, 180
    usage of 152
    utf8 153
group move rules 368
GROUP-USAGE clause
  description 180
  format 180
GROUP-USAGE NATIONAL clause 180
groups
  categories 154
  classes 154

H
halting execution 418
HEX-OF function 481
HEX-TO-CHAR function 482
hexadecimal notation
  for alphanumeric literals 36
  for national literals 43
Hexadecimal notation 40
Hexadecimal notation for UTF-8 literals 40
hiding 100
hierarchy of data 150
HIGH-VALUE figurative constant 13, 111
HIGH-VALUES figurative constant 13, 111
hyphen (-), in indicator area 54

I
I-O-CONTROL paragraph
  APPLY WRITE-ONLY clause 142
  ASCII considerations 610
  checkpoint processing in 139
    description 121, 138
  MULTIPLE FILE TAPE clause 142
  order of entries 138
  RERUN clause 139
  SAME AREA clause 140
  SAME RECORD AREA clause 141
  SAME SORT AREA clause 141
  SAME SORT-MERGE AREA clause 142
IBM extensions xxi, 557
  identification division
    FACTORY paragraph 99
    METHOD-ID paragraph 99
    OBJECT paragraph 99
IDENTIFICATION DIVISION
IDENTIFICATION DIVISION (continued)
  CLASS-ID paragraph 98
    format 93
    format (program, class, method) 93
    optional paragraphs 100
  PROGRAM-ID paragraph 96
identifier 244
identifiers 65, 243
  IF directive 551
  IF statement 322
imperative statement 267
implementor-name 12
  implicit
    redefinition of storage area 166, 208
    scope terminators 271
implicit attributes, of data 74
indentation 54, 152
  independent segments 243
index
  data item 257, 363
  relative indexing 70
  SET statement 70
index data item 68
INDEX phrase in USAGE clause 222
index-name
  assigning values 401
  comparisons 257
  OCCURS clause 185
  PERFORM statement 384
  SET statement 401, 402
INDEXED BY phrase 184
indexed files
  CLOSE statement 302
  DELETE statement 306
  FILE-CONTROL paragraph format 122
  I-O-CONTROL paragraph format 138
  organization 131
  permissible statements for 374
  READ statement 389
  REWRITE statement 395
  START statement 417
indexed organization
  description 131
  FILE-CONTROL paragraph format 122
  I-O-CONTROL paragraph format 138
indexing
  description 68
  MOVE statement evaluation 364
  OCCURS clause 68, 182
  relative 70
  SET statement and 70
indicator area 51
industry specifications 613
inheritance 85, 99
INHERITS clause 98
INITIAL clause 97
initial state of program 97
INITIALIZE statement
  format and description 324
  overlapping operands, unpredictable results 275
inline comments 44, 650
INLINE directive 546
INPUT phrase
  OPEN statement 372
  INTO phrase
    DIVIDE statement 309
    READ statement 385
    RETURN statement 392
INPUT phrase (continued)
  USE statement 541
  USE PROCEDURE phrase
    RELEASE statement 391
    SORT statement 413
Input-Output section
  description 121
  file control paragraph 121
  FILE-CONTROL keyword 121
  FILE-CONTROL paragraph 122
  format 121
  I-O-CONTROL paragraph 138
input-output statements
  ACCEPT 283
  CLOSE 301
  common processing facilities 276
  DELETE 305
  DISPLAY 307
  EXCEPTION/ERROR procedures 542
  general description 276
  OPEN 371
  READ 385
  REWRITE 393
  START 416
  WRITE 432
INSERT statement 535
insertion editing
  fixed (numeric-edited items) 203
  floating (numeric-edited items) 204
  simple 202
  special (numeric-edited items) 202
INSPECT statement
  AFTER phrase 331
  BEFORE phrase 331
  comparison cycle 333
  CONVERTING phrase 331
  overlapping operands, unpredictable results 275
  REPLACING phrase 329
INSTALLATION paragraph
  description 100
  format 93
  instance data 85, 89, 150
  instance definition
    format and description 87
  instance method 85, 89
  instance variable 85
  integer arguments 462
  INTEGER function 483
  integer function arguments 462
  integer functions 460, 461
  INTEGER-OF-DAY function 483
  INTEGER-OF-DAY function 483
  INTEGER-PART function 484
  internal floating-point
    DISPLAY statement 307
    size of items 159
  internal floating-point category 156
  internal floating-point items
    alignment rules 158
    how to define 194
INTO phrase
  DIVIDE statement 309
  READ statement 385
  RETURN statement 392
 INTO phrase (continued)
  STRING statement 419
  UNSTRING statement 428

with identifier 281

intrinsic functions
  ABS 471
  ACOS 471
  alphanumeric functions 460
  ANNUITY 472
  ASIN 472
  ATAN 473
  BIT-OF 473
  BIT-TO-CHAR 474
  BYTE-LENGTH 474
  categories 154
  CHAR 475
  classes 154
  COS 475
  CURRENT-DATE 476
  DATE-OF-INTEGER 477
  DATE-TO-YYYYMMDD 477
  DAY-OF-INTEGER 478
  DAY-TO-YYYYDDD 478
  DISPLAY-OF 479
  E 480
  EXP 480
  EXP10 481
  FACTORIAL 481
  floating-point literals 463
  HEX-OF 481
  HEX-TO-CHAR 482
  INTEGER 483
  integer functions 460
  INTEGER-OF-DATE 483
  INTEGER-OF-DAY 483
  INTEGER-PART 484
  LENGTH 484
  LOG 486
  LOG10 486
  LOWER-CASE 486
  MAX 487
  MEAN 488
  MEDIAN 488
  MIDRANGE 488
  MIN 489
  MOD 490
  national functions 460
  NATIONAL-OF 490
  numeric functions 460
  NUMVAL 491
  NUMVAL-C 492
  NUMVAL-F 494
  ORD 495
  ORD-MAX 495
  ORD-MIN 496
  PI 496
  PRESENT-VALUE 496
  RANDOM 497
  RANGE 497
  REM 498
  REVERSE 498
  SIGN 499
  SIN 500
  SQRT 500

intrinsic functions (continued)
  STANDARD-DEVIAITON 500
  SUM 501
  summary of 466
  TAN 501
  TEST-NUMVAL 502
  TEST-NUMVAL-C 502
  TEST-NUMVAL-F 504
  TRIM 505
  ULENGTH 506
  under LP(64) xxiv
  UPOS 507
  UPPER-CASE 508
  USUBSTR 509
  USUPPLEMENTARY 510
  UUID4 511
  UVALID 512
  UWIDTH 514
  VARIANCE 515
  WHEN-COMPILED 516
  YEAR-TO-YYYY 517

invalid key condition 280

INVALID KEY phrase
  DELETE statement 306
  READ statement 387
  REWRITE statement 394
  START statement 417
  WRITE statement 436

INVOKE statement
  BY VALUE phrase 336
  format and description 335
  LENGTH OF special register 337
  NEW phrase 336
  NOT ON EXCEPTION phrase 339
  ON EXCEPTION phrase 338
  RETURNING phrase 337
  SELF special object identifier 336
  SUPER special object identifier 336
  USING phrase 336

ISCII considerations 609

ISCII standard 613

ISO 646 609

ISO COBOL standards 613

J

Java
  class-name 118
  package 118

Java classes 85

Java interoperability
  data types 337, 339, 341
  literal types 337

Java interoperability 85

Java Native Interface (JNI) 18, 85

Java objects 85

Java String data 85

java.lang.Object 85

JNI environment pointer 85

JNIENVPTR special register 18, 85

JSON GENERATE statement
  CONVERTING phrase 345
  COUNT phrase 344
  description 342
local-storage
  requirement for indexed items 184
LOCAL-STORAGE
defining with RECURSIVE clause 97
LOG function 486
LOG10 function 486
logical operator
  complex condition 261
  in evaluation of combined conditions 263
  list of 261
logical record
  definition 149
  file data 149
  program data 150
  record description entry and 149
RECORDS phrase 168
LOW-VALUE figurative constant 14, 111
LOW-VALUES figurative constant 14, 111
LOWER-CASE function 486
lowercase letters
  in PICTURE clause 189
LP(64)
  intrinsic functions xxiv
  POINTER xxiv
  POINTER-32 xxiv
  SYNCHRONIZED clause xxiv
M
MAX function 487
maximum index value 70
MEAN function 488
MEDIAN function 488
MEMORY SIZE clause 107
MERGE statement
  ASCENDING/DESCENDING KEY phrase 359
  COLLATING SEQUENCE phrase 360
  format and description 358
  GIVING phrase 361
  OUTPUT PROCEDURE phrase 362
  segmentation considerations 362
  USING phrase 361
method data 150
method data division
  format 145
method definition
  effect of SELF and SUPER 335
  format and description 89
  IDENTIFICATION DIVISION 95
  method procedure division 237
  METHOD-ID paragraph 99
method FILE SECTION 147
method hiding 100
method identification division 93, 99
method LOCAL-STORAGE 148, 149
method overloading 99
method overriding 100
method procedure division 237, 238
method procedure division header 239
method WORKING-STORAGE 147
METHOD-ID paragraph 99
method-name 60
methods
  available to subclasses 99
methods (continued)
  exiting 317
  invoking 335
  method definition
    inheritance rules 99
    recursively reentering 97
  reusing 98
MIDRANGE function 488
MIN function 489
minus sign (-)
  COBOL character 3
  fixed insertion symbol 203
  floating insertion symbol 204, 205
  SIGN clause 212
mnemonic-name
  ACCEPT statement 283
  DISPLAY statement 307
  SET statement 403
  SPECIAL-NAMES paragraph 111
  WRITE statement 435
MOD function 490
MOVE statement
  CORRESPONDING phrase 363
  elementary moves 364
  format and description 363
  group moves 368
  record area 368
MULTIPLE FILE TAPE clause 142
multiple record processing, READ statement 387
multiple results, arithmetic statements 275
MULTIPLY statement
  common phrases 273
  format and description 369
N
N symbol in PICTURE clause 190
NAME phrase
  JSON GENERATE statement 344
  JSON PARSE statement 352
  XML GENERATE statement 445
NAMESPACE phrase 444
NAMESPACE-DECLARATION XML event 25
NAMESPACE-PREFIX phrase 444
national category 156
national comparisons 255
national data items
  elementary move rules 366
  in a class condition 246
  in ACCEPT 283
  in UNSTRING statement 426
  SEARCH statement 399
national floating-point 196
national function arguments 462
national functions 460, 461
national groups
  CORRESPONDING phrase 181
  description 181
  INITIALIZE statement 181
  qualification of data-names 181
  RENAMES clause 181
  where processed as group 181
  XML GENERATE statement 181
national items
<table>
<thead>
<tr>
<th>NULL</th>
<th>figurative constant 15</th>
</tr>
</thead>
<tbody>
<tr>
<td>null block branch, CONTINUE statement 305</td>
<td></td>
</tr>
<tr>
<td>null-terminated alphanumeric literals 37</td>
<td></td>
</tr>
<tr>
<td>NULL/NULLS</td>
<td></td>
</tr>
<tr>
<td>data pointer 258, 404</td>
<td></td>
</tr>
<tr>
<td>figurative constant 232</td>
<td></td>
</tr>
<tr>
<td>function-pointer 259, 406</td>
<td></td>
</tr>
<tr>
<td>object reference 260, 407</td>
<td></td>
</tr>
<tr>
<td>procedure-pointer 259, 406</td>
<td></td>
</tr>
<tr>
<td>NULLS</td>
<td></td>
</tr>
<tr>
<td>figurative constant 15</td>
<td></td>
</tr>
<tr>
<td>numeric arguments 462, 463</td>
<td></td>
</tr>
<tr>
<td>numeric category 157</td>
<td></td>
</tr>
<tr>
<td>NUMERIC class test 247</td>
<td></td>
</tr>
<tr>
<td>numeric comparisons 257</td>
<td></td>
</tr>
<tr>
<td>numeric function arguments 463</td>
<td></td>
</tr>
<tr>
<td>numeric functions 460, 461</td>
<td></td>
</tr>
<tr>
<td>numeric items</td>
<td></td>
</tr>
<tr>
<td>alignment rules 158</td>
<td></td>
</tr>
<tr>
<td>how to define 199</td>
<td></td>
</tr>
<tr>
<td>millennium dates 199</td>
<td></td>
</tr>
<tr>
<td>PICTURE clause 198</td>
<td></td>
</tr>
<tr>
<td>numeric literals 41</td>
<td></td>
</tr>
<tr>
<td>numeric-edited category 157</td>
<td></td>
</tr>
<tr>
<td>numeric-edited item</td>
<td></td>
</tr>
<tr>
<td>editing signs 159</td>
<td></td>
</tr>
<tr>
<td>elementary move rules 366</td>
<td></td>
</tr>
<tr>
<td>numeric-edited items</td>
<td></td>
</tr>
<tr>
<td>alignment rules 158</td>
<td></td>
</tr>
<tr>
<td>how to define 199</td>
<td></td>
</tr>
<tr>
<td>PICTURE clause 198</td>
<td></td>
</tr>
<tr>
<td>NUMVAL function 491</td>
<td></td>
</tr>
<tr>
<td>NUMVAL-C function 492</td>
<td></td>
</tr>
<tr>
<td>NUMVAL-F function 494</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>object data division</th>
</tr>
</thead>
<tbody>
<tr>
<td>format 146</td>
<td></td>
</tr>
<tr>
<td>object definition</td>
<td></td>
</tr>
<tr>
<td>OBJECT paragraph 99</td>
<td></td>
</tr>
<tr>
<td>object identification division 93, 99</td>
<td></td>
</tr>
<tr>
<td>object instance data 150</td>
<td></td>
</tr>
<tr>
<td>OBJECT paragraph 99</td>
<td></td>
</tr>
<tr>
<td>object procedure division 237</td>
<td></td>
</tr>
<tr>
<td>object program 79</td>
<td></td>
</tr>
<tr>
<td>object reference</td>
<td></td>
</tr>
<tr>
<td>in SET statement 401</td>
<td></td>
</tr>
<tr>
<td>OBJECT REFERENCE phrase 222</td>
<td></td>
</tr>
<tr>
<td>object WORKING-STORAGE 147</td>
<td></td>
</tr>
<tr>
<td>OBJECT-COMPUTER paragraph</td>
<td></td>
</tr>
<tr>
<td>ASCII considerations 609</td>
<td></td>
</tr>
<tr>
<td>object-oriented class-name 60</td>
<td></td>
</tr>
<tr>
<td>object-oriented COBOL</td>
<td></td>
</tr>
<tr>
<td>class definition 85</td>
<td></td>
</tr>
<tr>
<td>comparison rules 260</td>
<td></td>
</tr>
<tr>
<td>conformance rules</td>
<td></td>
</tr>
<tr>
<td>SET...USAGE OBJECT REFERENCE 407</td>
<td></td>
</tr>
<tr>
<td>effect of VALUE clause 147</td>
<td></td>
</tr>
<tr>
<td>factory definition 87</td>
<td></td>
</tr>
<tr>
<td>IDENTIFICATION clause 93</td>
<td></td>
</tr>
<tr>
<td>IDENTIFICATION DIVISION (class and method) 93</td>
<td></td>
</tr>
<tr>
<td>INHERITS clause 98</td>
<td></td>
</tr>
<tr>
<td>INVOKE statement 335</td>
<td></td>
</tr>
</tbody>
</table>
object-oriented COBOL (continued)
method definition 89
method-name 60
object definition 87
OBJECT REFERENCE phrase in USAGE clause 222
OO class name 60
procedure division (classes and methods) 237
REPOSITORY paragraph 116
SELF and SUPER special object identifiers 13
specifying configuration section 105
subclasses and methods 99
objects in EVALUATE statement 314
obsolete language elements xxi

OCCURS clause
ASCENDING/DESCENDING KEY phrase 183
description 182
INDEXED BY phrase 184
restrictions 183
UNBOUNDED 185
variable-length tables format 185

OCCURS DEPENDING ON (ODO) clause
complex 188
description 186
object of 186
RECORD clause 168
REDEFINES clause and 182
SEARCH statement and 182
subject and object of 186
subject of 182, 186
subscripting 68
OFF phrase, SET statement 403
OMITTED phrase 296, 297
ON EXCEPTION phrase
CALL statement 299
INVOKE statement 338
JSON GENERATE statement 346
JSON PARSE statement 354
XML GENERATE statement 447
XML PARSE statement 453
ON OVERFLOW phrase
CALL statement 299
STRING statement 421, 429
ON phrase, SET statement 403
ON SIZE ERROR phrase
ADD statement 288
arithmetic statements 273
COMPUTE statement 305
DIVIDE statement 312
MULTIPLY statement 370
SUBTRACT statement 425
OPEN statement
for new/existing files 372
format and description 371
I-O phrase 372
phrases 371
programming notes 373
system dependencies 374
operands
comparison of alphanumeric 253
comparison of DBCS 254
comparison of dynamic-length elementary item 255
comparison of group 257
comparison of national 255
comparison of numeric 257
operands (continued)
comparison of UTF-8 256
composite of 274
overlapping 275, 276
operation of JSON GENERATE statement 347
operation of JSON PARSE statement 355
operation of XML GENERATE statement 447
operational sign
algebraic, description of 159
SIGN clause and 159
USAGE clause and 159
operational signs 159
optional words, syntax notation xix
ORD function 495
ORD-MAX function 495
ORD-MIN function 496
order of entries
clauses in FILE-CONTROL paragraph 122
I-O-CONTROL paragraph 138
order of evaluation in combined conditions 264
ORGANIZATION clause
description 130
format 122
INDEXED phrase 130
LINE SEQUENTIAL phrase 130
RELATIVE phrase 130
SEQUENTIAL phrase 130
out-of-line PERFORM statement 377
outermost programs, debugging 543
OUTPUT phrase 372
OUTPUT PROCEDURE phrase
MERGE statement 362
RETURN statement 392
SORT statement 414
OVERFLOW phrase
CALL statement 299
STRING statement 421, 429
overlapping operands invalid in
arithmetic statements 275
data manipulation statements 276
overloading 99
overriding 100

P
P symbol in PICTURE clause 190, 192
PACKED-DECIMAL phrase in USAGE clause 220
PADDING CHARACTER clause 132
PAGE
WRITE statement 435
page eject 56
paragraph
header, specification of 52
termination, EXIT statement 316
paragraph-name
description 243
specification of 52
paragraphs
description 49, 243
syntactical hierarchy 49
parent class 85
parentheses
combined conditions, use 263
in arithmetic expressions 244

694 Enterprise COBOL for z/OS: Enterprise COBOL for z/OS, V6.3 Language Reference
Index 695

- parsing XML documents with validation restrictions 452
- partial listings 522
- PASSWORD clause description 136 system dependencies 137
- PERFORM statement branching 377 conditional 379 END-PERFORM phrase 376 EVALUATE statement 313 execution sequences 377 EXIT statement 316 format and description 376 in-line 377 out-of-line 377 TIMES phrase 378 VARYING phrase 380, 382 period (.) actual decimal point 202 PGMNAME compiler option CANCEL statement 300 phrases definition 50 syntactical hierarchy 49 physical record BLOCK CONTAINS clause 167 definition 149 file data 149 file description entry and 149 RECORDS phrase 168 PI function 496 PICTURE character-strings 44 PICTURE clause and class condition 246 computational items and 219 CURRENCY SIGN clause 113 data categories 194 DECIMAL-POINT IS COMMA clause 115, 189 description 188 editing 201 format 188 symbols used in 189 PICTURE SYMBOL phrase 114 picture symbols - 191 , 191 . 191 * 191 / 191 + 191 $ (currency symbol) 192 0 191 9 191 A 190 asterisk 191 B 190 comma 191 CR 191 currency symbol (cs) 192, 193 DB 191 E 190 G 190 picture symbols (continued) minus 191 N 190 P 190, 192 period 191 plus 191 S 190 sequence of 192 slash 191 U 191 V 191 X 191 Z 191 plus (+) fixed insertion symbol 203 floating insertion symbol 204, 205 insertion character 205 SIGN clause 212 pointer data items relation condition 258 SET statement 404 USAGE clause 223 POINTER phrase STRING statement 419 UNSTRING statement 428 POINTER phrase in USAGE clause 223, 224 pointer-32 data items USAGE clause 224 POSITIVE in sign condition 260 PRESENT-VALUE function 496 print files, WRITE statement 438 priority-number 107, 242 procedure branching GO TO statement 320 statements, executed sequentially 283 procedure branching statements 283 procedure division description 237 format (programs, methods, classes) 237 PROCEDURE DIVISION ASCII considerations 611 declarative procedures 241 header 238 statements 283 PROCEDURE DIVISION header RETURNING phrase 241 USING phrase 239 procedure division names 64 procedure pointer data items relation condition 259 procedure-name GO TO statement 320 MERGE statement 362 perform statement 376 SORT statement 413 procedure-pointer data items SET statement 405 USAGE clause 225 procedure-pointer phrase in USAGE clause 225 procedures, description 242 PROCESS (CBL) statement 522 PROCESSING PROCEDURE phrase, in XML PARSE 452 PROCESSING-INSTRUCTION-DATA XML event 25 PROCESSING-INSTRUCTION-TARGET XML event 25
rules for syntax notation xix
run unit
description 79
termination with CANCEL statement 300
runtime options
DEBUG 588
NODEBUG 588

S

S symbol in PICTURE clause 190
SAME clause 140
SAME RECORD AREA clause
description 141
format 138
SAME SORT AREA clause
description 141
format 138
SAME SORT-MERGE AREA clause
description 142
format 138
scope of names 59
scope terminator
explicit 270
implicit 271
SD (sort file description) entry
data division 166
DATA RECORDS clause 171
description 166
level indicator 150
SD (Sort File Description) entry
description 161
SEARCH statement
AT END phrase 396, 397
binary search 399
description and format 395
serial search 397
SET statement 397
VARYING phrase 398
WHEN phrase 400
SEARCH STATEMENT
NEXT SENTENCE phrase 396
searching
order for COPY statement 533
section header
description 242
specification of 52
section-name
description 242
in EXCEPTION/ERROR declarative 541
sections 49, 242
SECURITY paragraph
description 100
format 93
SEGMENT-LIMIT clause 107
segmentation 243
segmentation considerations 293, 362, 415
SELECT clause
ASSIGN clause and 126
format 122
specifying a file name 126
SELECT OPTIONAL clause
CLOSE statement 302
description 126
SELECT OPTIONAL clause (continued)
format 122
specification for sequential I-O files 126
selection objects in EVALUATE statement 314
selection subjects in EVALUATE statement 314
SELF 260
SELF special object identifier 13, 336
sending field
MOVE statement 363
SET statement 402
STRING statement 419
UNSTRING statement 426
sentences
definition 50
description 243
syntactical hierarchy 49
SEPARATE CHARACTER phrase of SIGN clause 212
separately compiled program 79
separators 45, 230
separators, rules for 46
sequence number area (cols. 1-6) 51
sequential access mode
data organization and 134
DELETE statement 306
description 133
READ statement 388
REWRITE statement 394
sequential files
access mode allowed 134
CLOSE statement 301, 302
description 130
file description entry 161
FILE-CONTROL paragraph format 122
LINAGE clause 171
OPEN statement 371
PASSWORD clause valid with 136
permissible statements for 374
READ statement 388
REWRITE statement 394
SELECT OPTIONAL clause 126
serial search
PERFORM statement 380
SERVICE LABEL statement 539
SERVICE RELOAD statement 540
SET statement
description and format 401
DOWN BY phrase 403
dynamic-length elementary items 407
function-pointer data items 221, 405
index data item 222
object reference data items 407
OFF phrase 403
ON phrase 403
overlapping operands, unpredictable results 275
pointer data items 404
procedure-pointer data items 405
requirement for indexed items 184
SEARCH statement 403
TO FALSE phrase 404
TO phrase 402
TO TRUE phrase 404
UP BY phrase 403
sharing data 179
sharing files 167
SHIFT-IN special register 21
SHIFT-OUT special register 21
sibling program 79
SIGN clause 211
sign condition 260
SIGN function 499
SIGN IS SEPARATE clause 212
signed
  numeric item, definition 199
  operational signs 159
simple condition
  combined 262
  description and types 246
  negated 262
simple data reference 65
simple insertion editing 202
SIN function 500
single-byte ASCII 5
single-byte EBCDIC 5
size-error condition 273
skip to next page 56
SKIP1 statement 540
SKIP2 statement 540
SKIP3 statement 540
slack bytes
  between 217
  within 214
slash (/)
  comment lines 56
  insertion character 202
  symbol in PICTURE clause 191
SORT statement
  ASCENDING KEY phrase 409, 410
  COLLATING SEQUENCE phrase 412
  DESCENDING KEY phrase 409, 410
  description and format 407
  DUPLICATES phrase 412
  GIVING phrase 414
  INPUT PROCEDURE phrase 413
  OUTPUT PROCEDURE phrase 414
  segmentation considerations 415
  USING phrase 413
SORT-CONTROL special register 22, 415
SORT-CORE-SIZE special register 22, 415
SORT-FILE-SIZE special register 22, 415
SORT-MESSAGE special register 23, 415
SORT-MODE-SIZE special register 23, 415
SORT-RETURN special register 23, 415
Sort/Merge feature
  I-O-CONTROL paragraph format 138
  MERGE statement 358
  RELEASE statement 391
  RERUN clause 140
  RETURN statement 392
  SAME SORT AREA clause 141
  SAME SORT-MERGE AREA clause 142
  SORT statement 407
Sort/Merge file statement phrases
  ASCENDING/DESCENDING KEY phrase 359
  COLLATING SEQUENCE phrase 360
  GIVING phrase 361
  OUTPUT PROCEDURE phrase 362
  USING phrase 361
source code
  (continued)
    library, programming notes 529
    listing 523
source code format 51
source language debugging 587
source program
  standard COBOL reference format 51
SOURCE-COMPUTER paragraph 106
SPACE figurative constant 13
SPACES figurative constant 13
special insertion editing 202
special object identifiers
  SELF 13
  SUPER 13
special registers
  ADDRESS OF 17
  DEBUG-ITEM 17
  JNIENVPTR 18
  JSON-CODE 18
  JSON-STATUS 19
  LENGTH OF 19
  LINAGE-COUNTER 20
  RETURN-CODE 20
  SHIFT-OUT, SHIFT-IN 21
  SORT-CONTROL 22
  SORT-CORE-SIZE 22
  SORT-FILE-SIZE 22
  SORT-MESSAGE 23
  SORT-MODE-SIZE 23
  SORT-RETURN 23
  TALLY 24
  WHEN-compiled 24
XML-CODE 25
XML-EVENT 25
XML-INFORMATION 30
XML-NAMESPACE 25, 31
XML-NAMESPACE-PREFIX 25, 32
XML-NCNAME 31
XML-NCNAME-PREFIX 32
XML-NTEXT 25, 33
XML-TEXT 25, 34
SPECIAL-NAMES paragraph
  ACCEPT statement 283
  ALPHABET clause 111
  ASCII considerations 609
  ASCII-encoded file specification 174
  CLASS clause 113
  CODE-SET clause and 174
  CURRENCY SIGN clause 113
  DECIMAL-POINT IS COMMA clause 115
  description 108
  format 108
  mnemonic-name 111
  XML-SCHEMA clause 115
SQRT function 500
STANDALONE-DECLARATION XML event 25
standard alignment
  JUSTIFIED clause 180
standard alignment rules
  UTF-8 data items 158
STANDARD-1
  RECORD DELIMITER clause 132
STANDARD-1 phrase 112
STANDARD-2 phrase 112

Index 699
STANDARD-DEVIATION function 500
standards 613
START statement
description and format 416
indexed file 417
INVALID KEY phrase 280, 417
relative files 417
status key considerations 416
START-OF-CDATA-SECTION XML event 25
START-OF-DOCUMENT XML event 25
START-OF-ELEMENT XML event 25
statement operations
common phrases 271
file position indicator 282
INTO and FROM phrases 281
statements
categories of 267
data manipulation 276
definition 50
delimited scope 270
description 243
imperative 267
input-output 276
procedure branching 283
syntactical hierarchy 49
types of 50
static data 85
static method 85
status key
common processing facility 276
file processing 542
STOP RUN statement 418
STOP statement 418
storage
map listing 523
MEMORY SIZE clause 107
REDEFINES clause 206
storage manipulation statements
ALLOCATE 289
FREE 319
STRING statement
description and format 418
execution of 421
overlapping operands, unpredictable results 275
structure of the COBOL language 3
structured programming
DO-WHILE and DO-UNTIL 379
subclass 85
subclasses and methods 99
subjects in EVALUATE statement 314
subprogram linkage
CALL statement 293
CANCEL statement 300
ENTRY statement 312
subprogram termination
CANCEL statement 300
EXIT PROGRAM statement 317
GOBACK statement 319
subscripting
definition and format 68
INDEXED BY phrase of OCCURS clause 184
MOVE statement evaluation 364
OCCURS clause specification 182
subscribing (continued)
table references 68
using data-names 70
using index-names (indexing) 68
using integers 70
substitution characters
DISPLAY-OF 479
NATIONAL-OF 491
substitution field of INSPECT REPLACING 329
substrings, specifying (reference-modification) 71
SUBTRACT statement
common phrases 271
description and format 423
SUM function 501
SUPER special object identifier 13, 336
superclass 85
support 675
SUPPRESS option, COPY 526
suppress output 522
SUPPRESS phrase
JSON GENERATE statement 344
JSON PARSE statement 353
XML GENERATE statement 446
suppression editing 205
switch-status condition 261
SYMBOLIC CHARACTERS clause 115
symbolic-character 11, 61
symbolic-character figurative constant 15
symbols in PICTURE clause 189
SYNCHRONIZED clause
effect on other language elements 213
under LP(64) xxiv
VALUE clause and 227
syntax notation, rules for xix
system considerations, subprogram linkage
CALL statement 293
CANCEL statement 300
system information transfer, ACCEPT statement 284
system input device, ACCEPT statement 283
system-names
computer-name 106
SOURCE-COMPUTER paragraph 106
T
table references
indexing 68
subscribing 68
TALLY special register 24
TALLYING phrase
INSPECT statement 329
UNSTRING statement 429
TAN function 501
termination of execution
EXIT METHOD statement 317
EXIT PARAGRAPH statement 318
EXIT PERFORM statement 318
EXIT PROGRAM statement 317
EXIT SECTION statement 319
GOBACK statement 319
STOP RUN statement 418
terminators, scope 270
TEST-NUMVAL function 502
TEST-NUMVAL-C function 502
TEST-NUMVAL-F function 504

text words 526
text-name
  literal-1 525
THREAD compiler option
  requirement for indexed items 184
THROUGH (THRU) phrase
  ALPHABET clause 111
  CLASS clause 113
  EVALUATE statement 314
  PERFORM statement 376
  RENAMES clause 210
  VALUE clause 229
TIME 286
  TIMES phrase of PERFORM statement 378
TITLE statement 540
  TO FALSE phrase, SET statement 404
  TO phrase, SET statement 402
  TO TRUE phrase, SET statement 404
  transfer of control
    ALTER statement 292, 293
      basic PERFORM statement 376
      explicit 75
    GO TO statement 320
      IF statement 322
      implicit 75
    JSON PARSE statement 350
    PERFORM statement 376
    XML PARSE statement 450
  transfer of data
    ACCEPT statement 283
    MOVE statement 363
    STRING statement 418
    UNSTRING statement 426
TRIM function 505
TRIM function arguments 463
  trimming of generated JSON data 349
  trimming of generated XML data 449
TRUNC compiler option 159
  truncation of data
    arithmetic item 159
    JUSTIFIED clause 180
    ROUNDED phrase 273
    TRUNC compiler option 159
truth value
  complex conditions 261
  EVALUATE statement 315
    IF statement 322
      of complex condition 261
      sign condition 260
      with conditional statement 269
  type conformance
    SET...USAGE OBJECT REFERENCE 407
TYPE phrase
  XML GENERATE statement 445
  types of functions 460

U

U symbol in PICTURE clause 191
ULENGTH function 506
  unary operator 244
  unconditional GO TO statement 320
Unicode 3, 5
  unique names 152
  uniqueness of reference 63
  unit file, definition 302
UNIT phrase 301
  universal object reference 222
  UNKNOWN-REFERENCE-IN-ATTRIBUTE XML event 25
  UNKNOWN-REFERENCE-IN-CONTENT XML event 25
  UNRESOLVED-REFERENCE XML event 25
  unsigned numeric item, definition 199
UNSTRING statement
  description and format 426
  execution 430
  overlapping operands, unpredictable results 275
  receiving field 428
  sending field 426
UP BY phrase, SET statement 403
UPON phrase, DISPLAY 307
UPPER-CASE function 507
UPS I-0 through UPS I-7, program switches
  and switch-status condition 261
  condition-name 111
  processing special conditions 111
  SPECIAL-NAMES paragraph 111
USAGE clause
  BINARY phrase 219
  CODE-SET clause and 174
  COMPUTATIONAL phrases 220
  description 218
  DISPLAY phrase 221
  DISPLAY-1 phrase 221
  format 218
  FUNCTION-POINTER phrase 221
  INDEX phrase 222
  NATIONAL phrase 180, 222
  operational signs and 159
  PACKED-DECIMAL phrase 220
  POINTER phrase 223, 224
  PROCEDURE-POINTER phrase 225
  VALUE clause and 227
USAGE COMP-1
  size of items 159
USAGE COMP-2
  size of items 159
USAGE DISPLAY
  size of items 159
  STRING statement and 420
USAGE DISPLAY-1
  size of items 159
  STRING statement and 420
USAGE NATIONAL
  size of items 159
  STRING statement and 420
USAGE OBJECT REFERENCE phrase 335
USAGE UTF-8
  size of items 159
USE FOR DEBUGGING declarative 588
USE statement
  format and description 541
  user labels
    DEBUGGING declarative 543
  user-defined words 10
  using
    intrinsic functions under LP(64) xxiv
using (continued)
SYNCHRONIZED clause under LP(64) xxiv
Using
POINTER and POINTER-32 xxiv
USING phrase
ASSIGN clause 126
CALL statement 295
in PROCEDURE DIVISION header 238
INVOKE statement 336
MERGE statement 361
PROCEDURE DIVISION header 239
SORT statement 413
subprogram linkage 241
USUBSTR function 509
USUPPLEMENTARY function 510, 511
UTF-16 3, 5
UTF-8 category 157
UTF-8 comparisons 256
UTF-8 data types
alignment rules 158
UTF-8 function arguments 462
UTF-8 functions 460
UTF-8 groups
CORRESPONDING phrase 181
description 181
INITIALIZE statement 181
qualification of data-names 181
RENAMES clause 181
where processed as group 181
UTF-8 items
how to define 200
UTF-8 literals 39
UTF-8 phrase 226
UVALID function 512
UWIDTH function 514

V
V symbol in PICTURE clause 191
VALIDATING phrase, in XML PARSE 451
validating XML documents
restrictions 452
VALUE clause
condition-name 228
effect on object-oriented programs 147
format 226, 228
level 88 item 152
NULL/NULS figurative constant 222, 232
rules for condition-name entries 230
rules for literal values 227
VALUE OF clause
description 171
format 161
variable-length tables 185, 186
VARIANCE function 515
VARYING phrase
PERFORM statement 380
SEARCH statement 398
VERSION-INFORMATION XML event 25
VOLATILE clause
format 232

W
WHEN phrase
EVALUATE statement 314
SEARCH statement (binary search) 400
SEARCH statement (serial search) 398
WHEN SET TO FALSE phrase
VALUE clause 229
WHEN-Compiled function 516
WHEN-Compiled special register 24
WITH DEBUGGING MODE clause 106, 543, 587
WITH DETAIL phrase
JSON PARSE statement 351
WITH DUPLICATES phrase, SORT statement 412
WITH FOOTING phrase 171
WITH NO ADVANCING phrase 308
WITH NO REWIND phrase, CLOSE statement 302
WITH POINTER phrase
STRING statement 419
UNSTRING statement 428
WORKING-STORAGE SECTION 147
WRITE statement
AFTER ADVANCING 435, 438
ALTERNATE RECORD KEY 439
BEFORE ADVANCING 435, 438
description 432
END-OF-PAGE phrase 436
format 432
FROM identifier phrase 281
indexed files 439
NOT END-OF-PAGE phrase 436
relative files 439
sequential files 437

X
X symbol in PICTURE clause 191
XML document
 parsing with validation
restrictions 452
XML event
ATTRIBUTE-CHARACTER 25
ATTRIBUTE-CHARACTERS 25
ATTRIBUTE-NAME 25
ATTRIBUTE-NATIONAL-CHARACTER 25
COMMENT 25
CONTENT-CHARACTER 25
CONTENT-CHARACTERS 25
CONTENT-NATIONAL-CHARACTER 25
DOCUMENT-TYPE-DECLARATION 25
ENCODING-DECLARATION 25
END-OF-CDATA-SECTION 25
END-OF-DOCUMENT 25
END-OF-INPUT 25
END-OF-ELEMENT 25
EXCEPTION 25
NAMESPACE-DECLARATION 25
PROCESSING-INSTRUCTION-DATA 25
PROCESSING-INSTRUCTION-TARGET 25
STANDALONE-DECLARATION 25
START-OF-CDATA-SECTION 25
START-OF-DOCUMENT 25
START-OF-ELEMENT 25
UNKNOWN-REFERENCE-IN-ATTRIBUTE 25
XML event (continued)
   UNKNOWN-REFERENCE-IN-CONTENT 25
   UNRESOLVED-REFERENCE 25
   VERSION-INFORMATION 25
XML GENERATE statement
   ATTRIBUTES phrase 444
   COUNT IN phrase 443
   description 441
   element name formation 449
   ENCODING phrase 443
   END-XML phrase 447
   exception event 447
   format 441
   format conversion 448
   NAME phrase 445
   NAMESPACE phrase 444
   NAMESPACE-PREFIX phrase 444
   NOT ON EXCEPTION phrase 447
   ON EXCEPTION phrase 447
   operation 447
   SUPPRESS phrase 446
   trimming 449
   TYPE phrase 445
   XML-DECLARATION phrase 444
XML PARSE statement
   control flow 454
   description 450
   exception event 453
   format 450
   nested XML GENERATE 454
   nested XML PARSE 454
   ON EXCEPTION phrase 453
   PROCESSING PROCEDURE phrase 452
XML parsing
   with validation
      restrictions 452
XML processing
   ENCODING phrase, in XML GENERATE 443
   ENCODING phrase, in XML PARSE 452
   PROCESSING PROCEDURE phrase, in XML PARSE 452
   RETURNING NATIONAL phrase, in XML PARSE 451
   VALIDATING phrase, in XML PARSE 451
   XML-CODE special register 25, 450
   XML-EVENT special register 25
   XML-INFORMATION special register 30
   XML-NAMESPACE special register 25, 31, 450
   XML-NAMESPACE-PREFIX special register 25, 32, 450
   XML-NAMESPACE special register 25, 31
   XML-NNAMESPACE special register 25, 31
   XML-NNAMESPACE-PREFIX special register 25, 33
   XML-NTEXT special register 25, 33, 455
   XML-SCHEMA clause
      specified in SPECIAL-NAMES paragraph 115
   xml-schema-name
      description 115
      specifying in SPECIAL-NAMES paragraph 115
   XML-SCHEMA clause 115
   XML-TEXT special register 25, 34, 455

Y
   YEAR-TO-YYYY function 517

Z
   Z
      insertion character 205
      null-terminated literals 37
      symbol in PICTURE clause 191
   Z literals 37
   zero
      filling, elementary moves 364
      suppression and replacement editing 205
   ZERO figurative constant 13
   ZERO in sign condition 260
   ZEROES figurative constant 13
   ZEROS figurative constant 13

XML schema file
   environment variable 116
   XML-CODE special register
      use in XML GENERATE 447
      use in XML PARSE 453
   XML-DECLARATION phrase 444
   XML-EVENT special register 25, 455
   XML-INFORMATION special register 30
   XML-NAMESPACE special register 25, 31
   XML-NAMESPACE special register, in XML PARSE 455
   XML-NAMESPACE-PREFIX special register 25, 32
   XML-NAMESPACE special register 25, 31
   XML-NNAMESPACE special register 25, 33
   XML-NNAMESPACE-PREFIX special register 25, 33
Product Number: 5655-EC6