

# OpenCOBOL 1.1

Programmer's Guide  
1st Edition, 23 January 2010

Gary Cutler

[CutlerGL@gmail.com](mailto:CutlerGL@gmail.com)

OpenCOBOL Copyright © 2001-2009 Keisuke Nishida / Roger While  
Under the terms of the GNU General Public License

Document Copyright © 2009,2010 Gary Cutler

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License [FDL], Version 1.3 or any later version published by the Free Software Foundation; with Invariant Section "What is OpenCOBOL?", no Front-Cover Texts, and no Back-Cover Texts.

A copy of the FDL is included in the section entitled "GNU Free Documentation License".

OpenCOBOL is an evolving tool.

While all reasonable attempts will be made to maintain the currency of the information in this document, neither the author of this document nor the authors of the OpenCOBOL software, extend any warranties of any kind by this document or for the information contained therein.

## Summary of Changes

Version	Publication Date	Change Description
Pre-publication review	July-September 2009	Initial version – documents the 06 Feb 2009 build of OpenCOBOL 1.1
1 <sup>st</sup> Edition	30 September 2009	Documented changes introduced with the nnXXXnn release of OpenCOBOL 1.1 – specifically: <ul style="list-style-type: none"><li>• The COPY statement now supports the COBOL2002 standard LEADING and TRAILING options (section <a href="#">1.7</a>)</li><li>• SOURCE-COMPUTER WITH DEBUGGING MODE is now supported (section <a href="#">4.1.1</a>)</li><li>• Mnemonic names are now optional for SWITCH declarations in SPECIAL-NAMES (section <a href="#">4.1.4</a>)</li><li>• SYMBOLIC CHARACTERS are now supported (section <a href="#">4.1.4</a>)</li><li>• USE FOR DEBUGGING is now supported (section <a href="#">6.3</a>)</li><li>• Floating-point literals of the form [+]<i>nn.nnE</i>[+]<i>nn</i> are now supported (section <a href="#">1.8</a>)</li><li>• Z"xxxxx" null-delimited alphanumeric literals are now supported (section <a href="#">1.8</a>)</li><li>• The ALTER verb (section <a href="#">6.7</a>) is now supported [Editorial Comment: this change was made only because NIST tests need it and NOT because you should be using it!]</li></ul>
	23 January 2010	Corrected an error with the description of reference modifiers lacking a <i>length</i> specification.

# Table of Contents

<b>FIGURES</b>	<b>6</b>
<b>1. INTRODUCTION</b>	<b>1-1</b>
1.1. What is OpenCOBOL?	1-1
1.2. Additional References and Documents	1-1
1.3. Introducing COBOL	1-1
1.3.1. "I Heard COBOL is a Dead Language!"	1-2
1.3.2. Programmer Productivity – The "Holy Grail"	1-3
1.3.3. Notable COBOL/OpenCOBOL Features	1-4
1.3.3.1. Basic Program Readability	1-4
1.3.3.2. COBOL Program Structure	1-5
1.3.3.3. Copybooks	1-5
1.3.3.4. Structured Data	1-5
1.3.3.5. Files	1-5
1.3.3.6. Table Handling	1-8
1.3.3.7. Sorting and Merging Data	1-8
1.3.3.8. String Manipulation	1-8
1.3.3.9. Textual-User Interface (TUI) Features	1-10
1.4. Syntax Description Conventions	1-10
1.5. Source Program Format	1-11
1.6. Use of Commas and Semicolons	1-11
1.7. Using COPY	1-12
1.8. Use of Literals	1-12
1.8.1. Numeric Literals	1-12
1.8.2. Alphanumeric Literals	1-13
1.9. Use of Figurative Constants	1-14
1.10. User-Defined Names	1-14
1.11. Use of LENGTH OF	1-15
<b>2. GENERAL OPENCOBOL PROGRAM FORMAT</b>	<b>2-1</b>
2.1. General Format for Nested Source Programs	2-2
2.2. General Format for Nested Source Functions	2-2
<b>3. IDENTIFICATION DIVISION</b>	<b>3-1</b>
<b>4. ENVIRONMENT DIVISION</b>	<b>4-1</b>
4.1. CONFIGURATION SECTION	4-1
4.1.1. SOURCE-COMPUTER Paragraph	4-1
4.1.2. OBJECT-COMPUTER Paragraph	4-1
4.1.3. REPOSITORY Paragraph	4-2
4.1.4. SPECIAL-NAMES Paragraph	4-3
4.2. INPUT-OUTPUT SECTION	4-5
4.2.1. FILE-CONTROL Paragraph	4-6
4.2.1.1. ORGANIZATION SEQUENTIAL Files	4-8
4.2.1.2. ORGANIZATION RELATIVE Files	4-8
4.2.1.3. ORGANIZATION INDEXED Files	4-9
4.2.2. I-O-CONTROL Paragraph	4-10
<b>5. DATA DIVISION</b>	<b>5-1</b>
5.1. FD - File Description	5-2
5.2. SD - SORT Description	5-3
5.3. General Format for Data Descriptions	5-4
5.4. Condition Names	5-15
5.5. Constant Descriptions	5-16

<b>5.6. Screen Descriptions</b>	<b>5-17</b>
<b>6. PROCEDURE DIVISION</b>	<b>6-1</b>
<b>6.1. General PROCEDURE DIVISION Components</b>	<b>6-1</b>
6.1.1. Table References	6-1
6.1.2. Qualification of Data Names	6-1
6.1.3. Reference Modifiers	6-2
6.1.4. Expressions	6-2
6.1.4.1. Arithmetic Expressions	6-3
6.1.4.2. Conditional Expressions	6-5
6.1.5. Use of Periods (.)	6-9
6.1.6. Use of “VERB” / “END-VERB” Constructs	6-10
6.1.7. Intrinsic Functions	6-11
6.1.7.1. ABS( <i>number</i> )	6-11
6.1.7.2. ACOS( <i>angle</i> )	6-12
6.1.7.3. ANNUITY( <i>interest-rate</i> , <i>number-of-periods</i> )	6-12
6.1.7.4. ASIN( <i>number</i> )	6-12
6.1.7.5. ATAN( <i>number</i> )	6-12
6.1.7.6. BYTE-LENGTH( <i>string</i> )	6-12
6.1.7.7. CHAR( <i>integer</i> )	6-12
6.1.7.8. COMBINED-DATETIME( <i>days</i> , <i>seconds</i> )	6-13
6.1.7.9. CONCATENATE( <i>string-1</i> [, <i>string-2</i> ] ...)	6-13
6.1.7.10. COS( <i>number</i> )	6-13
6.1.7.11. CURRENT-DATE	6-13
6.1.7.12. DATE-OF-INTEGERS( <i>integer</i> )	6-13
6.1.7.13. DATE-TO-YYYYMMDD( <i>yymmdd</i> [, <i>yy-cutoff</i> ] )	6-13
6.1.7.14. DAY-OF-INTEGERS( <i>integer</i> )	6-14
6.1.7.15. DAY-TO-YYYYDDD( <i>yyddd</i> [, <i>yy-cutoff</i> ])	6-14
6.1.7.16. E	6-14
6.1.7.17. EXCEPTION-FILE	6-14
6.1.7.18. EXCEPTION-LOCATION	6-14
6.1.7.19. EXCEPTION-STATEMENT	6-14
6.1.7.20. EXCEPTION-STATUS	6-15
6.1.7.21. EXP( <i>number</i> )	6-15
6.1.7.22. EXP10( <i>number</i> )	6-15
6.1.7.23. FRACTION-PART( <i>number</i> )	6-15
6.1.7.24. FACTORIAL( <i>number</i> )	6-15
6.1.7.25. INTEGER( <i>number</i> )	6-15
6.1.7.26. INTEGER-OF-DATE( <i>date</i> )	6-15
6.1.7.27. INTEGER-OF-DAY( <i>date</i> )	6-15
6.1.7.28. INTEGER-PART( <i>number</i> )	6-16
6.1.7.29. LENGTH( <i>string</i> )	6-16
6.1.7.30. LOCALE-DATE( <i>date</i> [, <i>locale</i> ] )	6-16
6.1.7.31. LOCALE-TIME( <i>time</i> [, <i>locale</i> ] )	6-16
6.1.7.32. LOCALE-TIME-FROM-SECS( <i>seconds</i> [, <i>locale</i> ] )	6-16
6.1.7.33. LOG( <i>number</i> )	6-16
6.1.7.34. LOG10( <i>number</i> )	6-16
6.1.7.35. LOWER-CASE( <i>string</i> )	6-16
6.1.7.36. MAX( <i>number-1</i> [, <i>number-2</i> ] ...)	6-17
6.1.7.37. MIN( <i>number-1</i> [, <i>number-2</i> ] ...)	6-17
6.1.7.38. MEAN( <i>number-1</i> [, <i>number-2</i> ] ...)	6-17
6.1.7.39. MEDIAN( <i>number-1</i> [, <i>number-2</i> ] ...)	6-17
6.1.7.40. MIDRANGE( <i>number-1</i> [, <i>number-2</i> ] ...)	6-17
6.1.7.41. MOD( <i>value</i> , <i>modulus</i> )	6-17
6.1.7.42. NUMVAL( <i>string</i> )	6-17
6.1.7.43. NUMVAL-C( <i>string</i> [, <i>symbol</i> ] )	6-17
6.1.7.44. ORD( <i>char</i> )	6-18

6.1.7.45. ORD-MAX( <i>char-1</i> [, <i>char-2</i> ] ... )	6-18
6.1.7.46. ORD-MIN( <i>char-1</i> [, <i>char-2</i> ] ... )	6-18
6.1.7.47. PI	6-18
6.1.7.48. PRESENT-VALUE( <i>rate,value-1</i> [, <i>value-2</i> ] )	6-18
6.1.7.49. RANDOM [ ( <i>seed</i> ) ]	6-18
6.1.7.50. RANGE( <i>number-1</i> [, <i>number-2</i> ] ...)	6-19
6.1.7.51. REM( <i>number</i> , <i>divisor</i> )	6-19
6.1.7.52. REVERSE( <i>string</i> )	6-19
6.1.7.53. SECONDS-FROM-FORMATTED-TIME( <i>format,time</i> )	6-19
6.1.7.54. SECONDS-PAST-MIDNIGHT	6-19
6.1.7.55. SIGN( <i>number</i> )	6-19
6.1.7.56. SIN( <i>angle</i> )	6-19
6.1.7.57. SQRT( <i>number</i> )	6-19
6.1.7.58. MEAN( <i>number-1</i> [, <i>number-2</i> ] ...)	6-20
6.1.7.59. STORED-CHAR-LENGTH( <i>string</i> )	6-20
6.1.7.60. SUBSTITUTE( <i>string,from-1,to-1</i> [, <i>from-n,to-n</i> ] )	6-20
6.1.7.61. SUBSTITUTE-CASE( <i>string,from-1,to-1</i> [, <i>from-n,to-n</i> ] )	6-20
6.1.7.62. SUM( <i>number-1</i> [, <i>number-2</i> ] ...)	6-20
6.1.7.63. TAN( <i>angle</i> )	6-20
6.1.7.64. TEST-DATE-YYYYMMDD( <i>date</i> )	6-20
6.1.7.65. TEST-DAY-YYYYDDD( <i>date</i> )	6-20
6.1.7.66. TRIM( <i>string</i> [ , LEADING TRAILING ] )	6-20
6.1.7.67. UPPER-CASE( <i>string</i> )	6-21
6.1.7.68. VARIANCE( <i>number-1</i> [, <i>number-2</i> ] ...)	6-21
6.1.7.69. WHEN-COMPILED	6-21
6.1.7.70. YEAR-TO-YYYY ( <i>yy</i> [, <i>yy-cutoff</i> ])	6-21
6.1.8. Special Registers	6-21
6.1.9. Controlling Concurrent Access to Files	6-22
6.1.9.1. File Sharing	6-22
6.1.9.2. Record Locking	6-23
<b>6.2. General Format of the PROCEDURE DIVISION</b>	<b>6-24</b>
<b>6.3. General Format for DECLARATIVES Entries</b>	<b>6-25</b>
<b>6.4. ACCEPT</b>	<b>6-26</b>
6.4.1. ACCEPT Format 1 – Read from Console	6-26
6.4.2. ACCEPT Format 2 – Retrieve Command-Line Arguments	6-26
6.4.3. ACCEPT Format 3 – Retrieve Environment Variable Values	6-27
6.4.4. ACCEPT Format 4 – Retrieve Screen Data	6-28
6.4.5. ACCEPT Format 5 – Retrieve Date/Time	6-29
6.4.6. ACCEPT Format 6 - Retrieve Screen Size Data	6-29
6.4.7. ACCEPT Exception Handling	6-30
<b>6.5. ADD</b>	<b>6-31</b>
6.5.1. ADD Format 1 – ADD TO	6-31
6.5.2. ADD Format 2 – ADD GIVING	6-32
6.5.3. ADD Format 3 – ADD CORRESPONDING	6-32
<b>6.6. ALLOCATE</b>	<b>6-33</b>
<b>6.7. ALTER</b>	<b>6-34</b>
<b>6.8. CALL</b>	<b>6-35</b>
<b>6.9. CANCEL</b>	<b>6-37</b>
<b>6.10. CLOSE</b>	<b>6-38</b>
<b>6.11. COMMIT</b>	<b>6-39</b>
<b>6.12. COMPUTE</b>	<b>6-40</b>
<b>6.13. CONTINUE</b>	<b>6-41</b>
<b>6.14. DELETE</b>	<b>6-42</b>
<b>6.15. DISPLAY</b>	<b>6-43</b>
6.15.1. DISPLAY Format 1 – Upon Console	6-43
6.15.2. DISPLAY Format 2 – Access Command-Line Arguments	6-43
6.15.3. DISPLAY Format 3 – Access or Set Environment Variables	6-43

6.15.4. DISPLAY Format 4 – Screen Data	6-44
6.15.5. DISPLAY Exception Handling	6-45
<b>6.16. DIVIDE</b>	<b>6-46</b>
6.16.1. DIVIDE Format 1 – DIVIDE INTO	6-46
6.16.2. DIVIDE Format 2 – DIVIDE INTO GIVING	6-46
6.16.3. DIVIDE Format 3 – DIVIDE BY GIVING	6-47
6.16.4. DIVIDE Format 4 – DIVIDE INTO REMAINDER	6-47
6.16.5. DIVIDE Format 5 – DIVIDE BY REMAINDER	6-48
<b>6.17. ENTRY</b>	<b>6-49</b>
<b>6.18. EVALUATE</b>	<b>6-50</b>
<b>6.19. EXIT</b>	<b>6-52</b>
<b>6.20. FREE</b>	<b>6-54</b>
<b>6.21. GENERATE</b>	<b>6-55</b>
<b>6.22. GOBACK</b>	<b>6-56</b>
<b>6.23. GO TO</b>	<b>6-57</b>
6.23.1. GO TO Format 1 – Simple GO TO	6-57
6.23.2. GO TO Format 2 – GO TO DEPENDING ON	6-57
<b>6.24. IF</b>	<b>6-58</b>
<b>6.25. INITIALIZE</b>	<b>6-59</b>
<b>6.26. INITIATE</b>	<b>6-60</b>
<b>6.27. INSPECT</b>	<b>6-61</b>
<b>6.28. MERGE</b>	<b>6-64</b>
<b>6.29. MOVE</b>	<b>6-66</b>
6.29.1. MOVE Format 1 – Simple MOVE	6-66
6.29.2. MOVE Format 2 – MOVE CORRESPONDING	6-66
<b>6.30. MULTIPLY</b>	<b>6-68</b>
6.30.1. MULTIPLY Format 1 – MULTIPLY BY	6-68
6.30.2. MULTIPLY Format 2 – MULTIPLY GIVING	6-68
<b>6.31. NEXT SENTENCE</b>	<b>6-69</b>
<b>6.32. OPEN</b>	<b>6-70</b>
<b>6.33. PERFORM</b>	<b>6-71</b>
6.33.1. PERFORM Format 1 – Procedural	6-71
6.33.2. PERFORM Format 2 – Inline	6-72
<b>6.34. READ</b>	<b>6-73</b>
6.34.1. READ Format 1 – Sequential READ	6-73
6.34.2. READ Format 2 – Random Read	6-74
<b>6.35. RELEASE</b>	<b>6-76</b>
<b>6.36. RETURN</b>	<b>6-77</b>
<b>6.37. REWRITE</b>	<b>6-78</b>
<b>6.38. ROLLBACK</b>	<b>6-79</b>
<b>6.39. SEARCH</b>	<b>6-80</b>
6.39.1. SEARCH Format 1 –Sequential Search	6-80
6.39.2. SEARCH Format 2 –Binary, or Half-interval Search (SEARCH ALL)	6-81
<b>6.40. SET</b>	<b>6-83</b>
6.40.1. SET Format 1 –SET ENVIRONMENT	6-83
6.40.2. SET Format 2 –SET Program-Pointer	6-83
6.40.3. SET Format 3 –SET ADDRESS	6-83
6.40.4. SET Format 4 –SET Index	6-84
6.40.5. SET Format 5 –SET UP/DOWN	6-84
6.40.6. SET Format 6 –SET Condition Name	6-84
6.40.7. SET Format 7 –SET Switch	6-85
<b>6.41. SORT</b>	<b>6-86</b>
6.41.1. SORT Format 1 – File-based Sort	6-86
6.41.2. SORT Format 2 – Table Sort	6-88
<b>6.42. START</b>	<b>6-89</b>
<b>6.43. STOP</b>	<b>6-91</b>
<b>6.44. STRING</b>	<b>6-92</b>

<b>6.45. SUBTRACT</b>	<b>6-93</b>
6.45.1. SUBTRACT Format 1 – SUBTRACT FROM	6-93
6.45.2. SUBTRACT Format 2 – SUBTRACT GIVING	6-93
6.45.3. SUBTRACT Format 3 – SUBTRACT CORRESPONDING	6-94
<b>6.46. SUPPRESS</b>	<b>6-95</b>
<b>6.47. TERMINATE</b>	<b>6-96</b>
<b>6.48. TRANSFORM</b>	<b>6-97</b>
<b>6.49. UNLOCK</b>	<b>6-98</b>
<b>6.50. UNSTRING</b>	<b>6-99</b>
<b>6.51. WRITE</b>	<b>6-101</b>
<b>7. THE OPENCOBOL SYSTEM INTERFACE</b>	<b>7-1</b>
<b>7.1. Using the OpenCOBOL Compiler (cobc)</b>	<b>7-1</b>
7.1.1. Introduction	7-1
7.1.2. Syntax and Options	7-1
7.1.3. Compiling Executable Programs	7-2
7.1.4. Dynamically-Loadable Subprograms	7-2
7.1.5. Static Subroutines	7-3
7.1.6. Combining COBOL and C Programs	7-3
7.1.6.1. OpenCOBOL Run-Time Library Requirements	7-3
7.1.6.2. String Allocation Differences Between OpenCOBOL and C	7-4
7.1.6.3. Matching C Data Types with OpenCOBOL USAGES	7-4
7.1.6.4. OpenCOBOL Main Programs CALLing C Subprograms	7-6
7.1.6.5. C Main Programs CALLing OpenCOBOL Subprograms	7-7
7.1.7. Important Environment Variables	7-9
7.1.8. Locating Copybooks at Compilation Time	7-10
7.1.9. Using Compiler Configuration Files	7-10
<b>7.2. Running OpenCOBOL Programs</b>	<b>7-12</b>
7.2.1. Executing Programs Directly	7-12
7.2.2. Using the “cobcrun” Utility	7-12
7.2.3. Program Arguments	7-13
7.2.4. Important Environment Variables	7-13
<b>7.3. Built-In Subroutines</b>	<b>7-14</b>
7.3.1. “Call by Name” Routines	7-14
7.3.1.1. CALL “C\$CHDIR” USING <i>directory-path, result</i>	7-15
7.3.1.2. CALL “C\$COPY” USING <i>src-file-path, dest-file-path, 0</i>	7-15
7.3.1.3. CALL “C\$DELETE” USING <i>file-path, 0</i>	7-15
7.3.1.4. CALL “C\$FILEINFO” USING <i>file-path, file-info</i>	7-16
7.3.1.5. CALL “C\$JUSTIFY” USING <i>data-item, “justification-type”</i>	7-16
7.3.1.6. CALL “C\$MAKEDIR” USING <i>dir-path</i>	7-16
7.3.1.7. CALL “C\$NARG” USING <i>arg-count-result</i>	7-16
7.3.1.8. CALL “C\$PARAMSIZE” USING <i>argument-number</i>	7-17
7.3.1.9. CALL “C\$SLEEP” USING <i>seconds-to-sleep</i>	7-17
7.3.1.10. CALL “C\$TOLOWER” USING <i>data-item, BY VALUE convert-length</i>	7-17
7.3.1.11. CALL “C\$TOUPPER” USING <i>data-item, BY VALUE convert-length</i>	7-17
7.3.1.12. CALL “CBL_AND” USING <i>item-1, item-2, BY VALUE byte-length</i>	7-17
7.3.1.13. CALL “CBL_CHANGE_DIR” USING <i>directory-path</i>	7-18
7.3.1.14. CALL “CBL_CHECK_FILE_EXIST” USING <i>file-path, file-info</i>	7-18
7.3.1.15. CALL “CBL_CLOSE_FILE” USING <i>file-handle</i>	7-18
7.3.1.16. CALL “CBL_COPY_FILE” USING <i>src-file-path, dest-file-path</i>	7-18
7.3.1.17. CALL “CBL_CREATE_DIR” USING <i>dir-path</i>	7-19
7.3.1.18. CALL “CBL_CREATE_FILE” USING <i>file-path, 2, 0, 0, file-handle</i>	7-19
7.3.1.19. CALL “CBL_DELETE_DIR” USING <i>dir-path</i>	7-19
7.3.1.20. CALL “CBL_DELETE_FILE” USING <i>file-path</i>	7-19
7.3.1.21. CALL “CBL_ERROR_PROC” USING <i>function, program-pointer</i>	7-19
7.3.1.22. CALL “CBL_EXIT_PROC” USING <i>function, program-pointer</i>	7-21
7.3.1.23. CALL “CBL_EQ” USING <i>item-1, item-2, BY VALUE byte-length</i>	7-22

7.3.1.24. CALL "CBL_FLUSH_FILE" USING <i>file-handle</i>	7-22
7.3.1.25. CALL "CBL_GET_CURRENT_DIR" USING BY VALUE 0, BY VALUE <i>length</i> , BY REFERENCE <i>buffer</i>	7-22
7.3.1.26. CALL "CBL_IMP" USING <i>item-1</i> , <i>item-2</i> , BY VALUE <i>byte-length</i>	7-22
7.3.1.27. CALL "CBL_NIMP" USING <i>item-1</i> , <i>item-2</i> , BY VALUE <i>byte-length</i>	7-23
7.3.1.28. CALL "CBL_NOR" USING <i>item-1</i> , <i>item-2</i> , BY VALUE <i>byte-length</i>	7-23
7.3.1.29. CALL "CBL_NOT" USING <i>item-1</i> , BY VALUE <i>byte-length</i>	7-24
7.3.1.30. CALL "CBL_OC_NANOSLEEP" USING <i>nanoseconds-to-sleep</i>	7-24
7.3.1.31. CALL "CBL_OPEN_FILE" <i>file-path</i> , <i>access-mode</i> , 0, 0, <i>handle</i>	7-24
7.3.1.32. CALL "CBL_OR" USING <i>item-1</i> , <i>item-2</i> , BY VALUE <i>byte-length</i>	7-24
7.3.1.33. CALL "CBL_READ_FILE" USING <i>handle</i> , <i>offset</i> , <i>nbytes</i> , <i>flag</i> , <i>buffer</i>	7-25
7.3.1.34. CALL "CBL_RENAME_FILE" USING <i>old-file-path</i> , <i>new-file-path</i>	7-25
7.3.1.35. CALL "CBL_TOLOWER" USING <i>data-item</i> , BY VALUE <i>convert-length</i>	7-25
7.3.1.36. CALL "CBL_Toupper" USING <i>data-item</i> , BY VALUE <i>convert-length</i>	7-26
7.3.1.37. CALL "CBL_WRITE_FILE" USING <i>handle</i> , <i>offset</i> , <i>nbytes</i> , 0, <i>buffer</i>	7-26
7.3.1.38. CALL "CBL_XOR" USING <i>item-1</i> , <i>item-2</i> , BY VALUE <i>byte-length</i>	7-26
7.3.1.39. CALL "SYSTEM" USING <i>command</i>	7-26
7.3.2. "Call by Number" Subroutines	7-27
7.3.2.1. CALL X"91" USING <i>return-code</i> , <i>function-code</i> , <i>binary-variable-arg</i>	7-27
7.3.2.2. CALL X"F4" USING <i>byte</i> , <i>table</i>	7-28
7.3.2.3. CALL X"F5" USING <i>byte</i> , <i>table</i>	7-28
7.3.2.4. Binary Truncation	7-29
<b>8. SAMPLE PROGRAMS</b>	<b>8-1</b>
8.1. FileStat-Msgs.cpy – File Status Values	8-1
8.2. cobdump – A Hex/Char Data Dump Subroutine	8-1
8.3. OCic – an OpenCOBOL Full-Screen Compiler Front-End	8-4
8.4. WINSYSTEM – Execute Windows Shell Commands (For Cygwin)	8-21
<b>9. GLOSSARY OF TERMS</b>	<b>9-1</b>
<b>INDEX</b>	<b>I</b>
<b>GNU FREE DOCUMENTATION LICENSE</b>	<b>IX</b>

## Figures

Figure 1-1 - A Sample TUI Screen.....	1-10
Figure 1-2 - COPY Syntax.....	1-12
Figure 1-3 - Figurative Constants.....	1-14
Figure 2-1 - General OpenCOBOL Program Format .....	2-1
Figure 2-2 - General Format for Nested Source Programs .....	2-2
Figure 2-3 - General Format for Nested Source Functions .....	2-2
Figure 3-1 - IDENTIFICATION DIVISION Syntax .....	3-1
Figure 4-1 - ENVIRONMENT DIVISION Syntax.....	4-1
Figure 4-2 - CONFIGURATION SECTION Syntax.....	4-1
Figure 4-3 - SOURCE-COMPUTER Paragraph Syntax .....	4-1
Figure 4-4 - OBJECT-COMPUTER Paragraph Syntax .....	4-1
Figure 4-5 - REPOSITORY Paragraph Syntax.....	4-2
Figure 4-6 - SPECIAL-NAMES Paragraph Syntax.....	4-3
Figure 4-7 - Locale Codes .....	4-4
Figure 4-8 - Screen ACCEPT Key Codes.....	4-5
Figure 4-9 - INPUT-OUTPUT SECTION Syntax .....	4-5
Figure 4-10 - FILE-CONTROL Paragraph Syntax .....	4-6
Figure 4-11 - FILE-STATUS Values .....	4-7
Figure 4-12 - Additional FILE-CONTROL Syntax for SEQUENTIAL Files.....	4-8
Figure 4-13 - Additional FILE-CONTROL Syntax for RELATIVE Files .....	4-8
Figure 4-14 - Additional FILE-CONTROL Syntax for INDEXED Files.....	4-9



Figure 4-15 - I-O-CONTROL Paragraph Syntax.....	4-10
Figure 5-1 - General DATA DIVISION Format.....	5-1
Figure 5-2 - FD Syntax.....	5-2
Figure 5-3- LINAGE-specified Page Structure.....	5-3
Figure 5-4 - SD Syntax.....	5-3
Figure 5-5 - General Data Description Format.....	5-4
Figure 5-6 - Data Class-Specification PICTURE Symbols (A/X/9) .....	5-5
Figure 5-7 - Numeric Option PICTURE Symbols (P/S/V).....	5-6
Figure 5-8 - Sign-Encoding Characters.....	5-6
Figure 5-9 - Numeric Editing PICTURE Symbols.....	5-7
Figure 5-10 - Summary of USAGE Specifications.....	5-12
Figure 5-11 - Effect of the SYNCHRONIZED Clause.....	5-14
Figure 5-12 - Level-88 Condition Name Description Syntax.....	5-15
Figure 5-13 - Level-78 Constant Description Syntax .....	5-16
Figure 5-14 - SCREEN SECTION Data Item Description Syntax .....	5-17
Figure 5-15 - Screen Color Numbers .....	5-19
Figure 5-16 - LOWLIGHT / HIGHLIGHT Effect on Screen Colors.....	5-19
Figure 6-1 - Reference Modifier Syntax .....	6-2
Figure 6-2 – Unary - Operator Syntax.....	6-3
Figure 6-3 – Unary + Operator Syntax.....	6-3
Figure 6-4 - Exponentiation Operator Syntax.....	6-3
Figure 6-5 - Exponentiation Operator Syntax.....	6-3
Figure 6-6 - Division Operator Syntax.....	6-3
Figure 6-7 - Addition Operator Syntax .....	6-4
Figure 6-8 - Subtraction Operator Syntax .....	6-4
Figure 6-9 - Class Condition Syntax .....	6-6
Figure 6-10 - Sign Condition Syntax.....	6-6
Figure 6-11 - Using Switch Conditions.....	6-7
Figure 6-12 - Relation Condition Syntax.....	6-8
Figure 6-13 - Combined Condition Syntax .....	6-8
Figure 6-14 - Negated Condition Syntax .....	6-9
Figure 6-15 - Special Registers .....	6-21
Figure 6-16 - General PROCEDURE DIVISION Syntax.....	6-24
Figure 6-17 - General DECLARATIVES Syntax.....	6-25
Figure 6-18 - ACCEPT (Read from Console) Syntax.....	6-26
Figure 6-19 - ACCEPT (Command Line Arguments) Syntax.....	6-26
Figure 6-20 - ACCEPT (Environment Variable Values) Syntax.....	6-27
Figure 6-21 - ACCEPT (Retrieve Screen Data) Syntax.....	6-28
Figure 6-22 - ACCEPT (Retrieve Date/Time) Syntax.....	6-29
Figure 6-23 - ACCEPT Options for DATE/TIME Retrieval .....	6-29
Figure 6-24 - ACCEPT (Retrieve Screen Size Data) Syntax.....	6-29
Figure 6-25 - ACCEPT Exception Handling.....	6-30
Figure 6-26 - ADD (TO) Syntax .....	6-31
Figure 6-27 - A Sample Program Using ON SIZE ERROR.....	6-31
Figure 6-28 - ADD (GIVING) Syntax.....	6-32
Figure 6-29 - ADD (CORRESPONDING) Syntax .....	6-32
Figure 6-30 - ALLOCATE Syntax.....	6-33
Figure 6-31 - ALTER Syntax.....	6-34
Figure 6-32 - CALL Syntax.....	6-35
Figure 6-33 - CALL BY REFERENCE Can Sometimes have Unwanted Effects! .....	6-36
Figure 6-34 - CALL BY VALUE.....	6-36
Figure 6-35 - CANCEL Syntax.....	6-37
Figure 6-36 - CLOSE Syntax.....	6-38
Figure 6-37 - COMMIT Syntax.....	6-39
Figure 6-38 - COMPUTE Syntax.....	6-40
Figure 6-39 - CONTINUE Syntax.....	6-41
Figure 6-40 - DELETE Syntax.....	6-42

Figure 6-41 - DISPLAY (Upon Console) Syntax .....	6-43
Figure 6-42 - DISPLAY (Access Command-line Arguments) Syntax .....	6-43
Figure 6-43 - DISPLAY (Access / Set Environment Variables) Syntax .....	6-43
Figure 6-44 - DISPLAY (Screen Data) Syntax .....	6-44
Figure 6-45 - Exception Handling (DISPLAY) Syntax .....	6-45
Figure 6-46 - DIVIDE INTO Syntax .....	6-46
Figure 6-47 - DIVIDE INTO GIVING Syntax.....	6-46
Figure 6-48 - DIVIDE BY GIVING Syntax.....	6-47
Figure 6-49 - DIVIDE INTO REMAINDER Syntax .....	6-47
Figure 6-50 - DIVIDE BY REMAINDER Syntax.....	6-48
Figure 6-51 - ENTRY Syntax .....	6-49
Figure 6-52 - EVALUATE Syntax .....	6-50
Figure 6-53 - An EVALUATE Demo Program .....	6-51
Figure 6-54 - EXIT Syntax.....	6-52
Figure 6-55 - Using the EXIT Statement .....	6-52
Figure 6-56 - Using EXIT PARAGRAPH .....	6-52
Figure 6-57 - Using the EXIT PERFORM Statement .....	6-52
Figure 6-58 - FREE Syntax.....	6-54
Figure 6-59 - GENERATE Syntax.....	6-55
Figure 6-60 - GOBACK Syntax.....	6-56
Figure 6-61 - Simple GOTO Syntax.....	6-57
Figure 6-62 - GOTO DEPENDING ON Syntax.....	6-57
Figure 6-63 - GOTO DEPENDING ON vs IF vs EVALUATE .....	6-57
Figure 6-64 - IF Syntax.....	6-58
Figure 6-65 - INITIALIZE Syntax.....	6-59
Figure 6-66 - INITIATE Syntax.....	6-60
Figure 6-67 - INSPECT Syntax .....	6-61
Figure 6-68 - An INSPECT TALLYING Example .....	6-62
Figure 6-69 - MERGE Syntax.....	6-64
Figure 6-70 - Simple MOVE Syntax .....	6-66
Figure 6-71 - MOVE CORRESPONDING Syntax.....	6-66
Figure 6-72 - MULTIPLY BY Syntax.....	6-68
Figure 6-73 - MULTIPLY GIVING Syntax.....	6-68
Figure 6-74 - NEXT SENTENCE Syntax.....	6-69
Figure 6-75 - OPEN Syntax.....	6-70
Figure 6-76 - Procedural PERFORM Syntax.....	6-71
Figure 6-77 - Inline PERFORM Syntax.....	6-72
Figure 6-78 - READ (Sequential) Syntax.....	6-73
Figure 6-79 - READ (Random) Syntax .....	6-74
Figure 6-80 - RELEASE Syntax.....	6-76
Figure 6-81 - RETURN Syntax .....	6-77
Figure 6-82 - REWRITE Syntax.....	6-78
Figure 6-83 - ROLLBACK Syntax.....	6-79
Figure 6-84 - Sequential SEARCH Syntax.....	6-80
Figure 6-85 - Binary SEARCH (ALL) Syntax.....	6-81
Figure 6-86 - SET ENVIRONMENT Syntax .....	6-83
Figure 6-87 - SET Program Pointer Syntax .....	6-83
Figure 6-88 - SET ADDRESS Syntax.....	6-83
Figure 6-89 - SET Index Syntax.....	6-84
Figure 6-90 - SET UP/DOWN Syntax .....	6-84
Figure 6-91 - SET Condition Name Syntax.....	6-84
Figure 6-92 - SET Switch Syntax.....	6-85
Figure 6-93 - File-Based SORT Syntax.....	6-86
Figure 6-94 - Table SORT Syntax.....	6-88
Figure 6-95 - START Syntax.....	6-89
Figure 6-96 - STOP Syntax .....	6-91
Figure 6-97 - STRING Syntax.....	6-92

Figure 6-98 - SUBTRACT FROM Syntax.....	6-93
Figure 6-99 - SUBTRACT GIVING Syntax.....	6-93
Figure 6-100 - SUBTRACT CORRESPONDING Syntax .....	6-94
Figure 6-101 - SUPPRESS Syntax.....	6-95
Figure 6-102 - TERMINATE Syntax.....	6-96
Figure 6-103 - TRANSFORM Syntax.....	6-97
Figure 6-104 - The TRANSFORM Statement at Work.....	6-97
Figure 6-105 - UNLOCK Syntax.....	6-98
Figure 6-106 - UNSTRING Syntax.....	6-99
Figure 6-107 - An UNSTRING Example .....	6-99
Figure 6-108 - WRITE Syntax.....	6-101
Figure 7-1 - C/OpenCOBOL Data Type Matches .....	7-4
Figure 7-2 - OpenCOBOL CALLing C.....	7-6
Figure 7-3 - C CALLing OpenCOBOL.....	7-7
Figure 7-4 - Compiler Environment Variables.....	7-9
Figure 7-5 - Run-Time Environment Variables.....	7-13
Figure 7-6 - A Binary Truncation Demo Program .....	7-29
Figure 7-7 - A Non-Scientific Comparison of Numeric Data Item USAGE Performance .....	7-31



# 1. Introduction

## 1.1. What is OpenCOBOL?

This document describes the syntax, semantics and usage of the COBOL programming language as implemented by the current version of OpenCOBOL.

OpenCOBOL is an open-source COBOL compiler and runtime environment. The OpenCOBOL compiler generates C code which is automatically compiled and linked. While originally developed for UNIX operating systems, OpenCOBOL can also be built for MacOS computers or Windows computers utilizing the UNIX-emulation features of such tools as Cygwin and MinGW<sup>1</sup>. It has also been built as a truly native Windows application utilizing Microsoft's freely-downloadable Visual Studio Express package to provide the C compiler and linker/loader.

The principal developers of OpenCOBOL are Keisuke Nishida and Roger While. They may be contacted at the OpenCOBOL website - [www.opencobol.org](http://www.opencobol.org).

This document was intended to serve as a full-function reference and user's guide suitable for both those readers learning COBOL for the first time as well as those already familiar with some dialect of the COBOL language. The author of this document is Gary Cutler, who may be reached via postings at the [www.opencobol.org](http://www.opencobol.org) forum, or by email at [CutlerGL@gmail.com](mailto:CutlerGL@gmail.com).

## 1.2. Additional References and Documents

For those wishing to learn COBOL for the first time, I can strongly recommend the following resources.

If you like to hold a book in your hands, I strongly recommend "Murach's Structured COBOL", by Mike Murach, Anne Prince and Raul Menendez (2000) - ISBN 9781890774059. Mike Murach and his various writing partners have been writing outstanding COBOL textbooks for several decades, and this text is no exception. It's an excellent book for those familiar with the concepts of programming in other languages, but unfamiliar with COBOL.

Would you prefer a web-based tutorial? Try the University of Limerick (Ireland) COBOL web site - <http://www.csis.ul.ie/cobol/>.

## 1.3. Introducing COBOL

If you already know a programming language, and that language isn't COBOL, chances are that language is Java, C or C++. You will find COBOL a much different programming language than those – sometimes those differences are a good thing and sometimes they aren't. The thing to remember about COBOL is this – it was designed to solve business problems. It was designed to do that in the 1950s.

COBOL was the first programming language to become standardized such that a COBOL program written on computer "A" made by company "X" would be able to be compiled and executed on computer "B" made by company "Y". This may not seem like such a big deal today, but it was a radical departure from all programming languages that came before it and even many that came after it.

The name "COBOL" actually says it all – COBOL is an acronym that stands for "COMmon Business Oriented Language". Note the fact that the word "common" comes before all others. The word "business" is a close second. Therein lies the key to COBOL's success.

---

<sup>1</sup> The MinGW approach is a personal favorite with the author of this manual because it creates an OpenCOBOL compiler and runtime that require only a single MinGW DLL to be available to OpenCOBOL tools and user programs. That DLL is freely distributable under the terms of the GNU General Public License. A MinGW build of OpenCOBOL fits easily on and runs from a 128MB flash drive with no need to install any software onto the Windows computer that will be using it. Some functionality of the language, dealing with the sharing of files between concurrently executing OpenCOBOL programs and record locking on certain types of files, is sacrificed however.

### 1.3.1. “I Heard COBOL is a Dead Language!”

Phoenician is a dead language. Mayan is a dead language. Latin is a dead language. What makes these languages dead is the fact that no one speaks them anymore. COBOL is NOT a dead language, and despite pontifications that come down to us from the ivory towers of academia, it isn't even on life support.

What made those other languages die is the fact that they became obsolete. As the peoples that spoke them were overrun or superseded by other populations that eventually replaced them, no one saw any need to speak their languages. There was no good reason to keep on speaking a language whose creators had become irrelevant to history.

COBOL is different. Certainly, there were more people that “spoke” COBOL back in the 1980s than there are now. Remember, however, the second word in COBOL's acronym – business. Businesses are complex social and economic organisms that exist for but a single purpose – to make money. One of the approaches businesses take to satisfy that all-important survival trait is that they want to avoid expenses.

This avoidance of expense turns out to have been key to the survival of COBOL because those programmers of the 1980s (give or take a decade) were very busy programmers. Estimates are that as many as a several hundred billion lines of COBOL code were written for businesses world-wide. Because of the first word in COBOL's name (“Common”), as businesses replaced their older, slower and less-reliable computer systems with newer, faster and more-reliable ones, they found that the massive investment they had in their COBOL software inventory paid dividends by remaining functional on those new systems - many times with no changes needed whatsoever!

Unwilling to endorse change merely for the sake of change, businesses replaced these billions and billions of lines of COBOL code only when absolutely necessary and only when financially justifiable. That justification appeared to have come as the 20<sup>th</sup> century was nearing the end.

Written long before the end of the century was near, many of those COBOL applications used 2-digit years instead of four digit years because, when the programs were written, computer storage of any kind was expensive. Why should millions and millions of bytes of storage be wasted by all those “19” sequences when the software can simply assume them? Since their software would suddenly think the current year was “1900” after the stroke of midnight, December 31<sup>st</sup> 1999, businesses knew they were going to have to do something about the “Y2K” (programmer “geek speak” for “Year 2000”) problem.

At last! Y2K was going to be the massive asteroid strike that finally killed off the COBOL dinosaur.

Unfortunately for those seeking the extinction of COBOL, that proved to be wishful thinking.

Always concerned with the bottom line, businesses actually analyzed the problems with their programs. Many applications were replaced with newer and “better” versions that used more appropriate (translation: more politically correct) languages and computer systems. BUT ... many applications were not replaced. These were the absolutely essential applications whose replacement would cripple the business if everything didn't go absolutely perfectly. These COBOL applications were modified to use 4-digit years instead of 2-digit ones. At the same time, many of them received cosmetic “face lifts” to make their computer/human interfaces more acceptable, frequently with the help of modules developed in the newer languages.

The result is that even today, after the Y2K “extinction event”, there are, by industry estimates, over 40 billion lines of COBOL code still running the businesses of the 21<sup>st</sup> century. A fact that is disturbing to some is that – just as tiny little furry mammals evolved to cope with the original “extinction event” holocaust – COBOL has also evolved into a leaner and meaner “animal” capable of competing in niches and providing services unthought-of back in 1968. That fact is confirmed by the fact that those lines of COBOL code being tracked by industry analysts are actually *growing* at the rate of about 4 billion a year.

Evolution, you see, is in COBOLs DNA. Over time, COBOL evolved in form and function, first via work done by the American National Standards Institute (ANSI) and eventually through the efforts of the International Standards Organization (ISO).

The first widely-adopted standard for COBOL was published by ANSI in 1968<sup>2</sup>. Named the ANS68 standard, this version of COBOL was originally standardized for use primarily as the business programming tool of the US Defense Department; it quickly was adopted by other Government agencies and private businesses alike.

Subsequent standards published in 1974 and 1985 (ANS74 and ANS85, respectively) added new features and evolved the language toward adoption of the programmer-productivity tool of the time – “Structured Programming”.

As the 21<sup>st</sup> century dawned, programming had moved out of the board room and into the Game Room, the Living Room and even the Kitchen; as computers became more and more inexpensive they appeared in games, entertainment devices and appliances. Even the automobile became home to computers galore. These computers need software, and that software is written in the so-called “modern” languages.

Combined with Y2K, these trends became the impetus for COBOL to evolve even newer features and capabilities. The COBOL2002 standard<sup>3</sup> introduced object-oriented features and syntax that make the language more programmer-friendly to those trained by today’s programming curricula. The COBOL20xx standard, currently under development, carries the evolution forward to the point where a COBOL20xx implementation will be fully as “modern” as any other programming language.

Through all this evolution, however, care was taken with each new standard to protect the investment businesses (or anyone, for that matter) had in COBOL software. Generally, a new COBOL standard – once implemented and adopted by a business - required minimal, if any, changes to upgrade existing applications. When changes were necessary, those changes could frequently be made using tools that mechanically upgraded entire libraries of source code with little or no need for human intervention.

The OpenCOBOL implementation of the COBOL language supports virtually the entire ANS85 standard as well as some significant features of the COBOL2002 standard, although the truly object-oriented features are not there (yet).

### 1.3.2. Programmer Productivity – The “Holy Grail”

Throughout the history of computer programming, the search for new ways to improve of the productivity of programmers has been the all-important consideration. Sometimes this search has taken the form of introducing new features in programming languages, or even new languages altogether, and sometimes it has evolved new ways of using the existing languages. Other than hobbyists, programming is an activity performed for money. Businesses abhor spending anything more than is absolutely necessary. Even government agencies try to spend as little money on projects as is absolutely necessary<sup>4</sup>.

The amount of programming necessary to accomplish a given task – including rework needed by any errors found during testing (*testing*: “that time during which an application is actually in production use attempting to serve the purpose for which it was designed” ☺) is the measure of *programmer productivity*. Anything that reduces that effort will therefore reduce the time spent in such activities therefore reducing the expense of same. When the expense of programming is reduced, programmer productivity is increased.

While many technological and procedural developments have made evolutionary improvements to programmer productivity, each of the following has been responsible for revolutionary improvements:

- The development of so-called “higher-level” programming languages that enable a programmer to specify in a single statement of the language an action that would have required many more separate statements in a prior programming language. The standardization of such languages, making them usable on a wide variety

---

<sup>2</sup> To that point, in 1968 the US Government made it a requirement that any computer system sold to them must run a version of COBOL that adhered to the ANS68 standard. The requirement that computers sold to the US Government had to support the current COBOL standard remained for many, many years.

<sup>3</sup> “Popular” names for COBOL standards no longer include an organization’s name, and now use Y2K-compliant 4-digit years.

<sup>4</sup> This is a religious issue because it is an assertion that – sadly – must be taken purely on faith; there is, unfortunately, all too little real-world evidence to support it. It makes sense, so one can only hope it is true.

of computers and operating systems, is a COBOL was a pioneering development in this area, being one of the first higher-level languages.

- The establishment of programming techniques that make programs easier to read and therefore easier to understand. Not only do such techniques reduce the amount of rework necessary simply to make a program work as designed, but they also reduce the amount of time a programmer needs to study an existing program in order how to best adapt it to changing business requirements. The foremost development in this area was *structured programming*. Introduced in the late 1970s, this approach to programming spawned new programming languages (PASCAL, ALGOL, PL/1) designed around it. With the ANSI85 standard, COBOL embraced the principles espoused by structured programming mavens as well as any of the languages designed strictly around it.
- The establishment of programming techniques AND the introduction of programming language capabilities to facilitate the reusability of program code. Anything that supports *code reusability* can have a profound impact to the amount of time it takes to develop new applications or to make significant changes to existing ones. In recent years, object-oriented programming has been the industry “poster child” for code reusability. By enabling program logic and the data structures that logic manipulates encapsulated into easily stored and retrieved (and therefore “reusable”) modules called *classes*, the object-oriented languages such as Java, C++ and C# have become the favorites of academia. Since students are being trained in these technologies and only these, by and large, it’s no surprise that – today - object-oriented programming languages are the darlings of the industry.

The reality is, however, that good programmers have been practicing code reusability for more than a half-century. Up until recently, COBOL programmers have had some of the best code reusability tools available - they’ve been doing it with copybooks (section [1.7](#)) and subroutines (sections [6.8](#), [7.1.4](#) and [7.1.5](#)) rather than classes, methods and attributes but the net results have been similar. With the COBOL2002 standard and the improvements made by the COBOL20xx standard, the playing field is leveled in this regard.

### 1.3.3. Notable COBOL/OpenCOBOL Features

#### 1.3.3.1. Basic Program Readability

When it first was developed, COBOL’s easily-readable syntax made it profoundly different to anything that had been seen before. For the first time, it was possible to specify logic in a manner that was – at least to some extent – comprehensible even to non-programmers. Take for example, the following code written in FORTRAN – a language developed only a year before COBOL:

```
E = P * Q
I = I + E
```

With its original limitation on the length of variable names (one letter or a letter followed by a number), and its use of algebraic notation to express actions being taken, FORTRAN wasn’t a particularly readable language, even by programmers. Compare this with the equivalent COBOL code:

```
MULTIPLY PRICE BY QUANTITY GIVING EXTENDED-AMOUNT
ADD EXTENDED-AMOUNT TO INVOICE-TOTAL
```

Clearly, even a non-programmer could at least conceptually understand what was going on! Over time, languages like FORTRAN evolved more robust variable names, but FORTRAN was never as readable as COBOL.

The inherent readability of COBOL code was a blessing at first, but eventually it became considered as a curse. As more and more people became at least informed about programming if not downright skilled, the syntax of COBOL became one of the reasons the ivory-tower types wanted to see it eradicated.

I would MUCH rather be handed an assignment to make significant changes to a COBOL program about which I know nothing than to be asked to do the same with a C, C++ or Java program.

Those that argue that it is too boring/wasteful/time-consuming/insulting (choose the word you prefer) to have to code a COBOL program “from scratch” are clearly ignorant of the following facts:



- Many systems have program-development tools available to ease the task of coding programs; those tools that concentrate on COBOL are capable of providing templates for much of the “overhead” verbiage of any program
- Good programmers have – for decades – maintained their own skeleton “template” programs for a variety of program types; simply load a template into a text edit and you’ve got a good start to the program
- Legend has it that there’s actually only ever been ONE program ever written in COBOL – all programs ever written after that sprang from that one!

### 1.3.3.2. COBOL Program Structure

COBOL programs are structured into four major areas of coding, each with it’s own purpose. These four areas are known as DIVISIONS.

Each DIVISION may consist of a variety of SECTIONS and each SECTION consists of one or more PARAGRAPHS. A PARARAPH consists of SENTENCES, each of which consists of one or more STATEMENTS.

This hierarchical structure of program components standardizes the composition of all COBOL programs. Much of this manual describes the various divisions, sections, paragraphs and statements that may comprise any COBOL program.

The four divisions, and their function, are described in section [2](#). Each division has its own chapter (sections [3](#), [4](#), [5](#) and [6](#)) and each of those chapters will describe the sections, et. al. available to programmers in each of those divisions.

### 1.3.3.3. Copybooks

A “copybook” is a segment of program code may be utilized by multiple programs simply by having that program use the COPY statement (section [1.7](#)) to import that code into the program. This code may define files, data structures or procedural code.

Today’s current programming languages have a statement (usually, this statement is named “include” or “#include”) that performs this same function. What makes the COBOL copybook feature different than the “include” facility in current languages, however, is the fact that the COBOL COPY statement can edit the imported source code as it is being copied. This capability enables copybook libraries extremely valuable to making code reusable.

### 1.3.3.4. Structured Data

COBOL introduced the concept of structured data back in the 1960s. Structured data is data which may be accessed as a single item or may be broken down into sub-items based upon their character position of occurrence within the structure. These structures called *group items* (page [9-2](#)). At the bottom of any structure are data items that aren’t broken down into sub-items. COBOL refers to these as *elementary items* (page [9-1](#)).

### 1.3.3.5. Files

One of COBOLs main strengths is the wide variety of files it is capable of accessing. OpenCOBOL, like other COBOL implementations, needs to have the structure of any files that it will be reading and/or writing described to it. The highest-level characteristic of a file’s structure is defined by specifying the ORGANIZATION (section [4.2.1](#)) of the file, as follows:

ORGANIZATION IS LINE SEQUENTIAL	These are files with the simplest of all internal structures. Their contents are structured simply as a series of data records, each terminated by a special end-of-record delimiter character. An ASCII line-feed character (hexadecimal 0A) is the end-of-record delimiter character used by any UNIX or pseudo-UNIX (MinGW, Cygwin, MacOS) OpenCOBOL build. A truly native Windows build would use a carriage-return, line-feed (hexadecimal 0D0A) sequence.
------------------------------------	---

Records in this type of file need not be the same length.

Records must be read from or written to these files in a purely sequential manner. The only

way to read (or write) record number 100 would be to have read (or written) records number 1 thru 99 first.

When the file is written by an OpenCOBOL program, the delimiter sequence will be automatically appended to each data record as it is written to the file.

When the file is read, the OpenCOBOL runtime system will strip the trailing delimiter sequence from each record and pad the data (to the right) with SPACES, if necessary, if the data just read is shorter than the area described for data records in the program. If the data is too long, it will be truncated and the excess will be lost.

These files should not be defined to contain any exact binary data fields because the contents of those fields could inadvertently have the end-of-record sequence as part of their values – this would confuse the runtime system when reading the file, and it would interpret that value as an actual end-of-record sequence.

#### ORGANIZATION IS RECORD BINARY SEQUENTIAL

These files also have a simple internal structure. Their contents are structured simply as a series of fixed-length data records with no special end-of-record delimiter.

Records in this type of file are all the same physical length. If variable-length logical records are defined to the program (section 5.3), the space occupied by each physical record in the file will occupy the maximum possible space.

Records must be read from or written to these files in a purely sequential manner. The only way to read (or write) record number 100 would be to have read (or written) records number 1 thru 99 first.

When the file is written by an OpenCOBOL program, no delimiter sequence is appended to the data.

When the file is read, the data is transferred into the program exactly as it exists in the file. In the event that a short record is read as the very last record, that record will be SPACE padded.

Care must be taken that programs reading such a file describe records whose length is exactly the same as that used by the programs that created the file. For example, the following shows the contents of a RECORD BINARY SEQUENTIAL file created by a program that wrote five 6-character records to it. The “A”, “B”, ... values and the background colors reflect the records that were written to the file:

```
AAAAAABBBBBBCCCCCDDDDDEEEEE
```

Now, assume that another program reads this file, but described 10-character records rather than 6. Here are the records that program will read:

```
AAAAAABBBB  
BBCCCCCDD  
DDDEEEEE
```

There may be times where this is exactly what you were looking for. More often than not, however, this is not desirable behavior. *Suggestion:* use a copybook to describe the record layouts of any file; this guarantees that multiple programs accessing that file will “see” the same record sizes and layouts.

These files can contain exact binary data fields. The contents of record fields are irrelevant to the reading process as there is no end-of-record delimiter.

#### ORGANIZATION IS RELATIVE

The contents of these files consist of a series of fixed-length data records prefixed with a four-byte USAGE COMP-5 (Figure 5-10) record header. The record header contains the length of the data, in bytes. The byte-count does not include the four-byte record header.

Records in this type of file are all the same physical length. If variable-length logical records are defined to the program (section 5.3), the space occupied by each physical record in the file will occupy the maximum possible space.

This file organization was defined to accommodate either sequential or random processing. With a RELATIVE file, it is possible to read or write record 100 directly, without having to have first read or written records 1-99. The OpenCOBOL runtime system uses the program-defined maximum record size to calculate a relative byte position in the file where the record header and data begin, and then transfers the necessary data to or from the program.

When the file is written by an OpenCOBOL program, no delimiter sequence is appended to the data, but a record-length field is added to the beginning of each physical record.

When the file is read, the data is transferred into the program exactly as it exists in the file.

Care must be taken that programs reading such a file describe records whose length is exactly the same as that used by the programs that created the file. It won't be a pretty site when the OpenCOBOL runtime library ends up interpreting a four-byte ASCII character string as a record length when it transfers data from the file into the program!

*Suggestion:* use a copybook to describe the record layouts of any file; this guarantees that multiple programs accessing that file will "see" the same record sizes and layouts.

These files can contain exact binary data fields. The contents of record fields are irrelevant to the reading process as there is no end-of-record delimiter.

#### ORGANIZATION IS INDEXED

This is the most advanced file structure available to OpenCOBOL programs. It's not possible to describe the physical structure of such files because that structure will vary depending upon which advanced file-management facility was included into the OpenCOBOL build you will be using (Berkeley Database [BDB], VBISAM, etc.). We will – instead – discuss the logical structure of the file.

There will be multiple structures stored for an INDEXED file. The first will be a data component, which may be thought of as being similar to the internal structure of a RELATIVE file. Data records may not, however, be directly accessed by their record number as would be the case with a RELATIVE file, nor may they be processed sequentially by their physical sequence in the file.

The remaining structures will be one or more index components. An index component is a data structure that (somehow) enables the contents of a field, called a *primary key*, within each data record (a customer number, an employee number, a product code, a name, etc.) to be converted to a record number so that the data record for any given primary key value can be directly read, written and/or deleted. Additionally, the index data structure is defined in such a manner as to allow the file to be processed sequentially, record-by-record, in ascending sequence of the primary key field values. Whether this index structure exists as a binary-searchable tree structure (btree), an elaborate hash structure or something else is pretty much irrelevant to the programmer – the behavior of the structure will be as it was just described. The runtime system will not allow two records to be written to an indexed file with the same primary key value.

The capability exists for an additional field to be defined as what is known as an *alternate key*. Alternate key fields behave just like primary keys, allowing both direct and sequential access to record data based upon the alternate key field values, with one exception. That exception is the fact that alternate keys may be allowed to have duplicate values, depending upon how the alternate key field is described to the OpenCOBOL compiler (section [4.2.1.3](#)).

There may be any number of alternate keys, but each key field comes with a disk space penalty as well as an execution time penalty. As the number of alternate key fields increases, it will take longer and longer to write and/or modify records in the file.

These files can contain exact binary data fields. The contents of record fields are irrelevant to the reading process as there is no end-of-record delimiter.

All files are initially described to an OpenCOBOL program using a SELECT statement (section [4.2.1](#)) coded in the FILE-CONTROL paragraph of the INPUT-OUTPUT SECTION of the ENVIRONMENT DIVISION. In addition to defining a name

by which the file will be referenced within the program, the SELECT statement will specify the name and path by which the file will be known to the operating system along with its ORGANIZATION, locking (section [6.1.9.2](#)) and sharing (section [6.1.9.1](#)) attributes.

A file description (section [5.1](#)) in the FILE SECTION of the WORKING-STORAGE SECTION of the DATA DIVISION will define the structure of records within the file, including whether or not variable-length records are possible and – if so – what the minimum and maximum length might be. In addition, the file description entry can specify file I/O block sizes.

### 1.3.3.6. Table Handling

Other programming languages have arrays, COBOL has tables. They're basically the same thing. What makes COBOL tables special are two special statements that exist in the COBOL language – SEARCH (section [6.39.1](#)) and SEARCH ALL (section [6.39.2](#)).

The first can search a table sequentially, stopping only when either a table entry matching one of any number of search conditions is found, or when all table entries have been checked against the search criteria and none matched any of those criteria.

The second can perform an extremely fast search against a table sorted by and searched against a “key” field contained in each table entry. The algorithm used for such a search is a binary search (also known as a half-interval search). This algorithm ensures that only a small number of entries in the table need to be checked in order to find a desired entry or to determine that the desired entry doesn't exist in the table. The larger the table, the more effective this search becomes. For example, a table containing 32,768 entries will be able to locate a particular entry or will determine the entry doesn't exist by looking at no more than fifteen (15) entries! The algorithm is explained in detail in the SEARCH ALL documentation (section [6.39.2](#)).

### 1.3.3.7. Sorting and Merging Data

The COBOL language includes a powerful SORT statement (section [6.41.1](#)) that can sort large amounts of data according to arbitrarily complex key structures. This data may originate from within the program or may be contained in one or more external files. The sorted data may be written automatically to one or more output files or may be processed, record-by-record in the sorted sequence.

A special form of the SORT statement (section [6.41.2](#)) also exists just to sort the data that resides in a table. This is particularly useful if you wish to use SEARCH ALL against the table.

A companion statement – MERGE (section [6.28](#)) – can combine the contents of multiple files together, provided those files are all sorted in a similar manner according to the same key structure(s). The resulting output will consist of the contents of all of the input files, merged together and sequenced according to the common key structure(s). The output of a MERGE may be written automatically to one or more output files or may be processed internally by the program.

### 1.3.3.8. String Manipulation

There have been programming languages designed specifically for the processing of text strings, and there have been programming languages designed for the sole purpose of performing high-powered numerical computations. Most programming languages fall somewhere in the middle, between these two extremes. COBOL is no exception, although it does include some very powerful string manipulation capabilities; OpenCOBOL actually has even more string-manipulation capabilities than many other COBOL implementations. The following chart illustrates the capabilities of OpenCOBOL with regard to strings:

Capability	OpenCOBOL Feature Supporting that Capability
Concatenate two or more strings	CONCATENATE Intrinsic Function (section <a href="#">6.1.7.9</a> ) STRING Statement (section <a href="#">6.44</a> )
Conversion of a numeric time or date to a formatted character string	LOCALE-TIME or LOCALE-DATE Intrinsic Functions (sections <a href="#">6.1.7.31</a> and <a href="#">6.1.7.30</a> ), respectively

Capability	OpenCOBOL Feature Supporting that Capability
Convert a binary value to its corresponding character in the program's characterset	CHAR Intrinsic Function (section <a href="#">6.1.7.7</a> ); add 1 to argument before invoking the function; The description of the CHAR function shows a technique that utilizes the MOVE statement that will accomplish the same thing without the need of adding 1 to the numeric argument value first
Convert a character string to lower-case	LOWER-CASE Intrinsic Function (section <a href="#">6.1.7.35</a> ) C\$TOLOWER Built-in Subroutine (section <a href="#">7.3.1.10</a> ) CBL_TOLOWER Built-in Subroutine (section <a href="#">7.3.1.35</a> )
Convert a character string to upper-case	UPPER-CASE Intrinsic Function (section <a href="#">6.1.7.67</a> ) C\$TOUPPER Built-in Subroutine (section <a href="#">7.3.1.11</a> ) CBL_TOUPPER Built-in Subroutine (section <a href="#">7.3.1.36</a> )
Convert a character to its numeric value in the program's characterset	ORD Intrinsic Function (section <a href="#">6.1.7</a> ); subtract 1 from the result; The description of the ORD function shows a technique that utilizes the MOVE statement that will accomplish the same thing without the need of adding 1 to the numeric argument value first
Count occurrences of substrings in a larger string	INSPECT Statement with TALLYING Option (section <a href="#">6.27</a> )
Decode a formatted numeric string back to a numeric value (for example, decode "\$12,342.19-" to a -12342.19 value)	NUMVAL and NUMVAL-C Intrinsic Functions (sections <a href="#">6.1.7.42</a> and <a href="#">6.1.7.43</a> )
Determine the length of a string or data-item capable of storing strings	LENGTH or BYTE-LENGTH Intrinsic Functions (sections <a href="#">6.1.7.29</a> and <a href="#">6.1.7.6</a> )
Extract a substring of a string based on its starting character position and length	MOVE Statement (section <a href="#">6.29.1</a> ) with a reference modifier on the "sending" field
Format a numeric item for output, including thousands-separators ("," in the USA), currency symbols (" \$" in the USA), decimal points, credit/debit symbols, leading or trailing sign characters	MOVE Statement (section <a href="#">6.29</a> ) with picture-symbol editing applied to the receiving field (section <a href="#">5.3</a> )
Justification (Left, Right or Centered) of a string field	C\$JUSTIFY built-in subroutine (section <a href="#">7.3.1.5</a> )
Monoalphabetic substitution of one or more characters in a string with different characters	INSPECT Statement with CONVERTING Option (section <a href="#">6.27</a> ) TRANSFORM Statement (section <a href="#">6.48</a> ) SUBSTITUTE and SUBSTITUTE-CASE Intrinsic Functions (sections <a href="#">6.1.7.60</a> and <a href="#">6.1.7.61</a> )
Parse a string, breaking it up into substrings based upon one or more delimiting character sequences; these delimiters may be single characters, multiple-character strings or multiple consecutive occurrences of either	UNSTRING Statement (section <a href="#">6.50</a> )
Removal of leading or trailing spaces from a string	TRIM Intrinsic Function (section <a href="#">6.1.7.66</a> )
Substitution of a single substring with another <u>of the same length</u> , based upon the substrings starting character position and length	MOVE Statement (section <a href="#">6.29.1</a> ) with a reference modifier on the "receiving" field
Substitution of one or more substrings in a string with replacement substrings <u>of the same length</u> , regardless of where they occur	INSPECT Statement with REPLACING Option (section <a href="#">6.27</a> ) SUBSTITUTE and SUBSTITUTE-CASE Intrinsic Functions (sections <a href="#">6.1.7.60</a> and <a href="#">6.1.7.61</a> )
Substitution of one or more substrings in a string with replacement substrings <u>of a</u>	SUBSTITUTE and SUBSTITUTE-CASE Intrinsic Functions (sections <a href="#">6.1.7.60</a> and <a href="#">6.1.7.61</a> )

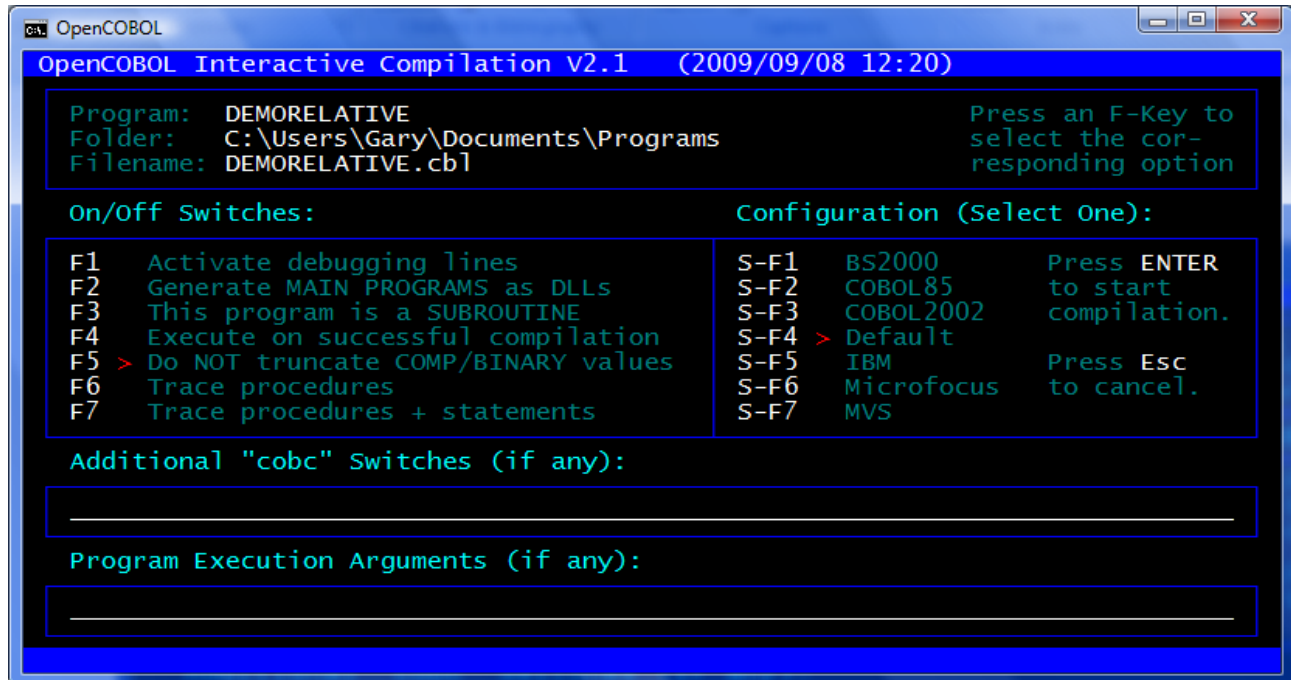
Capability	OpenCOBOL Feature Supporting that Capability
different <u>length</u> , regardless of where they occur	

### 1.3.3.9. Textual-User Interface (TUI) Features

The COBOL2002 standard formalizes extensions to the COBOL language that allow for the definition and processing of text-based screens. OpenCOBOL implements virtually all the screen-handling features described by COBOL2002.

Here is an example of such a screen as it might appear in the console window of a Windows computer:

Figure 1-1 - A Sample TUI Screen



Screens such as this<sup>5</sup> are defined in the SCREEN SECTION of the DATA DIVISION (section 0). Once defined, screens are used at run-time via the ACCEPT (section 6.4.4) and DISPLAY (section 6.15.4) statements.

The COBOL2002 standard only covers textual-user interface (TUI) screens and not the more-advanced graphical-user interface (GUI) screen design and processing capabilities built into most modern operating systems. There are subroutine-based packages available that can do full GUI development, but none are open-source.

## 1.4. Syntax Description Conventions

Syntax of the OpenCOBOL language will be described in this manual with conventions familiar to COBOL programmers. The following is a description of those syntactical-description techniques:

<b>UPPERCASE</b>	COBOL language keywords and implementation-dependent names (the so-called “reserved words” of the COBOL language) will appear in uppercase.
<b><u>UNDERLINING</u></b>	reserved words that are underlined are required in whatever syntactical context they are shown. If a reserved word is NOT underlined, it is optional and its presence or absence has no effect on the program.
<i>lowercase</i>	Generic terms representing substitutable arguments will be shown in lowercase.

<sup>5</sup> This screen comes from the program named OCic – a full-screen front-end to the OpenCOBOL compiler – the source code of which is included as a sample in this manual. See section 8.3 for the listing of the program.

[ brackets ]	Square brackets are used to enclose optional clauses. Any clauses not enclosed in square brackets are mandatory.
<i>choice-1   choice-2</i>	Simple choices may be indicated with a vertical bar separating them. Although not typically used in COBOL syntactical diagrams, this convention is an effective alternative that may be used when square brackets would make a syntax diagram too complicated.
{ braces }	Braces are used to enclose alternatives. Exactly one of the alternatives contained within the braces must be selected.
{  selector  }	Choice indicators are used to enclose alternatives where one <u>or more</u> of the enclosed selections may be selected.
...	A three-dot sequence (called an “ellipsis”) may appear following brackets, braces, selectors or lowercase entries to indicate that the syntax element preceding the ellipsis may occur multiple times.
<b>Shaded Areas</b>	Shaded areas are used to highlight syntax elements that are <u>recognized</u> by the OpenCOBOL compiler but will either have no effect on the generated code or will be rejected as being unsupported. Such elements are either present in the OpenCOBOL language to facilitate the porting of programs from other COBOL environments, reflect syntax elements that are not yet fully implemented or syntax elements that have become obsolete.

## 1.5. Source Program Format

Traditional COBOL program source format allows programs to be coded using 80-character (maximum) lines with a fixed format. As of the ANSI 2002 standard, a free-format source code format is defined where source code lines can be up to 256 characters long with no fixed meanings assigned to specific column ranges.

OpenCOBOL provides the following four methods for specifying the format of source code input files:

-fixed	This OpenCOBOL compiler switch specifies that all source input will be in traditional (80-column) fixed format. THIS IS THE DEFAULT MODE.
-free	This OpenCOBOL compiler switch specifies that all source input will be in ANSI2002 free (256 column) format.
>> <u>SOURCE</u> FORMAT IS <u>FREE</u>	This source line, when encountered by the OpenCOBOL compiler, will switch the compiler’s expectations into free format mode. The “>>” characters MUST begin in column 8 or beyond. Directives such as this and the next one may be used to switch the compiler back and forth between free and fixed mode at will.
>> <u>SOURCE</u> FORMAT IS <u>FIXED</u>	This source line, when encountered by the OpenCOBOL compiler, will switch the compiler’s expectations into fixed format mode. Directives such as this and the prior one may be used to switch the compiler back and forth between free and fixed mode at will.

The following are special directives or characters that may be used in OpenCOBOL programs to signify various things.

“*” in column 7	Signifies the source line is a comment. This is valid only when in FIXED mode.
“D” in column 7	Signifies the source line is a valid OpenCOBOL statement that will be treated as a comment <u>unless</u> the “-fdebugging-line” switch is specified to the OpenCOBOL compiler (in that instance, the lines will be compiled). This is valid only when in FIXED mode.
“*>” in any column	Denotes the remainder of the source line is a comment. This may be used in either FREE or FIXED mode, but if it is used in FIXED mode, the “*” should be in column 7 or beyond.
“>>D” in any column	Signifies the source line is a valid OpenCOBOL statement that will be treated as a comment <u>unless</u> the “-fdebugging-line” switch is specified to the OpenCOBOL compiler (in that instance, the lines will be compiled). This is valid when in FIXED or FREE mode, and must be the first non-blank sequence on the source line. In FREE mode, this sequence may begin in any column. In FIXED mode, this sequence must begin in column 8 or beyond.

## 1.6. Use of Commas and Semicolons



A comma character (“,”) or a semicolon (“;”) may be inserted into an OpenCOBOL program to improve readability at any spot where white space would be legal (except, of course, within alphanumeric literals). These characters are always optional. COBOL standards require that commas be followed by at least one space, when they’re used. Many modern COBOL compilers (OpenCOBOL included) relax this rule, allowing the space to be omitted in most instances. This can cause “confusion” to the compiler if the DECIMAL POINT IS COMMA clause is used (see section [4.1.4](#)).

The following statement, which calls a subroutine passing it two arguments (the numeric constants 1 and 2):

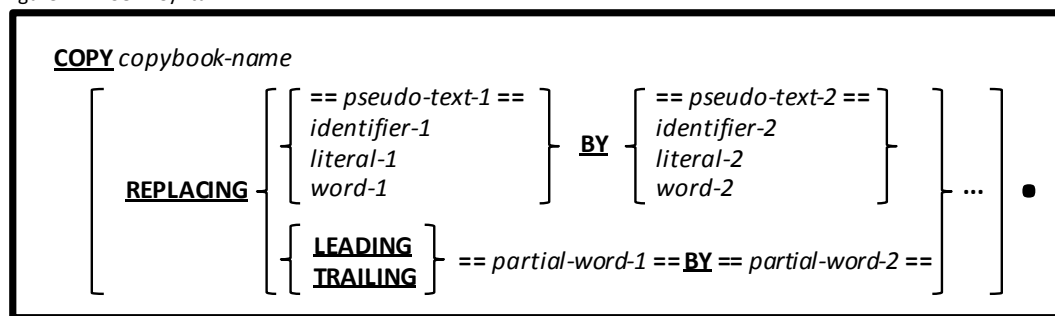
```
CALL “SUBROUTINE” USING 1,2
```

would – with DECIMAL POINT IS COMMA – actually be interpreted as a subroutine call with ONE arguments (the data non-integer numeric constant 1.2).

If you don’t already have it – develop the habit of coding a space after a comma used as punctuation! As an alternative, consider using a semicolon as there is no possibility for “confusion”.

## 1.7. Using COPY

Figure 1-2 - COPY Syntax



COPY statements are used to import copybooks (section [1.3.3.3](#)) into a program.

OpenCOBOL completely supports the use of copybooks. These are separate source files containing **ANY COBOL SYNTAX WHATSOEVER**, including other COPY statements.

COPY statements may be used anywhere within a COBOL program where the code contained within the copybook would be syntactically valid.

The syntax diagram above places great emphasis on a period at the end of the COPY statement and any REPLACING clauses it may have. A period is absolutely mandatory at the end of every COPY statement, even if – to the eye of an experienced COBOL programmer – it doesn’t seem like there should be a period.

All COPY statements are resolved and the contents of the corresponding copybooks inserted into the program source code before the actual compilation process begins.

The optional “REPLACING” clause allows any reserved words (*word-1*, *word-2*), data items (*identifier-1*, *identifier-2*), literals (*literal-1*, *literal-2*) or whitespace-delimited phrases to be replaced. Any number of such substitutions may be made as a copybook is included into a program.

See section [7.1.8](#) - Locating Copybooks at Compilation Time – for the details as to exactly how the OpenCOBOL compiler locates copybooks when programs are being compiled.

## 1.8. Use of Literals

Literals are constant values that will not change during the execution of a program. There are two fundamental types of literals – numeric and alphanumeric.

### 1.8.1. Numeric Literals

Numeric literals are numeric constants which may be used as array subscripts, as values in arithmetic expressions, or in any procedural statement where a numeric value may be used. Numeric literals may take any of the following forms:

- Integers such as 1, 56, 2192 or -54.



- Non-integer fixed point values such as 1.12 or -2.95.
- Floating-point values using “Enn” notation such as 9.92E25 (representing  $9.92 \times 10^{25}$ ) or 5.7E-14 (representing  $5.7 \times 10^{-14}$ ). Both the mantissa (the number before the “E”) and the exponent (the number after the E) may be explicitly specified as positive (with a +), negative or unsigned (and therefore implicitly positive). A floating-point literal’s value must be within the range  $-1.7 \times 10^{308}$  to  $+1.7 \times 10^{308}$ .with 15 decimal digits of precision.
- Hexadecimal numeric literals such as H’1F’ ( $1F_{16} = 31_{10}$ ), h’22’ ( $22_{16} = 34_{10}$ ) or H’DEAD’ ( $DEAD_{16} = 57005_{10}$ ). The “H” character may either be upper- or lower-case and either single quote (‘) or double-quote (“) characters may be used. Hexadecimal numeric literals are limited to a maximum value of H’FFFFFFFFFFFFFF’ (a 64-bit value).

## 1.8.2. Alphanumeric Literals

Alphanumeric literals are character strings suitable for display on a computer screen, printing on a report, transmission through a communications connection or storage in PIC X or PIC A data items (section [5.3](#)). These are NOT valid for use in arithmetic expressions unless they can first be converted to their numeric computational equivalent (see the NUMVAL and NUMVAL-C intrinsic functions in section [6.1.7](#)).

Alphanumeric literals may take any of the following forms:

- Any sequence of characters enclosed by a pair of single-quote (‘) characters or a pair of double-quote (“) characters constitutes a *string literal*. The double-quote character (“) may be used as a data character within such a literal. If a single-quote character must be included as a data character, express that character as two consecutive single-quotes (‘’). The single-quote character (‘) may be used as a data character within such a literal. If a double-quote character must be included as a data character, express that character as two consecutive double-quotes (“”).
- A literal formed according to the same rules as for a string literal (above), but prefixed with the letter “Z” (upper- or lower-case) constitutes a *zero-delimited string literal*. These literals differ from ordinary string literals in that they will be explicitly terminated with a byte of hexadecimal value 00. This facilitates the “sharing” of such literals with C programs<sup>6</sup>.
- A hexadecimal literal such as X’4A4B4C’ ( $4A4B4C_{16}$  = the ASCII string ‘JKL’), x’20’ ( $20_{16}$  = a space) or X’30313233’ ( $30313233_{16}$  = the ASCII string ‘0123’). The “X” character may either be upper- or lower-case and either single quote (‘) or double-quote (“) characters may be used. These hexadecimal alphanumeric literals should always consist of an even number of hexadecimal digits, because each character is represented by eight bits worth of data (2 hex digits). Hexadecimal alphanumeric literals may be of almost unlimited length.

Alphanumeric literals too long to fit on a single line may be continued to the next line in one of two ways:

- If you are using SOURCE FORMAT FIXED mode (section [1.5](#)), the alphanumeric literal can be run right up to and including column 72. The literal may then be continued on the next line anywhere after column 11 by coding another quote or apostrophe (whichever was used to begin the literal originally). The continuation line must also have a hyphen (-) coded in column 7. Here is an example:

```

1 2 3 4 5 6 7 8
1234567890123456789012345678901234567890123456789012345678901234567890
01 LONG-LITERAL-VALUE-DEMO PIC X(60) VALUE "This is a long l
- "iteral that must
- " be continued."
```

- Regardless of the current SOURCE FORMAT, OpenCOBOL allows alphanumeric literals to be broken up into separate fragments. These fragments have their own beginning and ending quote/apostrophe characters and are “glued together” using “&” characters. No continuation indicator is needed. Here’s an example:

```

1 2 3 4 5 6 7 8
1234567890123456789012345678901234567890123456789012345678901234567890
01 LONG-LITERAL-VALUE-DEMO PIC X(60) VALUE "This is a" &
```

<sup>6</sup> In the C programming language, strings must be terminated with a null byte (X’00’).

“ long literal that must “ &  
“be continued.”.

If your program is using free-form format, there’s less need to continue long alphanumeric literals because statements may be as long as 255 characters.

Numeric literals may be split across lines just as alphanumeric literals are, using either of the above techniques and reserved words can be split across lines too (using the first technique). Numeric literals and reserved words don’t get split very often though – it just makes for ugly-looking programs.

## 1.9. Use of Figurative Constants

Figurative constants are reserved words that may be used in lieu of certain literals. In general, a figurative constant may be freely used anywhere its corresponding value could have been used; when used, their value is interpreted as if it were prefixed with “ALL” (see section [5.3](#) for a discussion of “ALL”).

The following chart lists the OpenCOBOL figurative constants and their respective equivalent values.

Figure 1-3 - Figurative Constants

Figurative Constant	Type of Literal	Equivalent Value
ZERO, ZEROS, ZEROES	Numeric	0
SPACE, SPACES	Alphanumeric	Blank
QUOTE, QUOTES	Alphanumeric	Double-quote character(s)
LOW-VALUE, LOW-VALUES	Alphanumeric	The character whose value in the programs collating sequence is lowest. If a program is using the ASCII collating sequence, this will represent a sequence of characters comprised entirely of 0-bits.
HIGH-VALUE, HIGH-VALUES	Alphanumeric	The character whose value in the programs collating sequence is highest. If a program is using the ASCII collating sequence, this will represent a sequence of characters comprised entirely of 1-bits.
NULL	Alphanumeric	A character comprised entirely of zero-bits (regardless of the programs collating sequence).

## 1.10. User-Defined Names

When you write OpenCOBOL programs, you’ll need to create a variety of names to represent various aspects of the program, the programs data and the external environment in which the program is running.

User-defined names may be composed from the characters “A” through “Z” (upper- and/or lower-case), “0” through “9”, dash (“-”) and underscore (“\_”). User-defined names may neither start nor end with hyphen or underscore characters.

With the exception of [procedure names](#), user-defined names must contain at least one letter.

When user-defined names are created as names for data, they will be referenced in this document under the term [identifier](#).

## 1.11. Use of LENGTH OF

Alphanumeric literals and identifiers may optionally be prefixed with the clause “LENGTH OF”. In such cases, the literal actually is a numeric literal with a value equal to the number of bytes in the alphanumeric literal. For example, the following two OpenCOBOL statements both display the same result (27):

```
01 Demo-Identifier          PIC X(27).  *> This is a 27-character data-item
:
:
:
DISPLAY LENGTH OF "This is a LENGTH OF Example"
DISPLAY LENGTH OF Demo-Identifier
DISPLAY 27
```

The LENGTH OF clause on a *literal* or *identifier* reference may generally be used anywhere a numeric literal might be specified, with the following exceptions:

1. In place of a literal on a DISPLAY statement.
2. As part of a WRITE or RELEASE statement's FROM clause.
3. As part of the TIMES clause of a PERFORM.

## 2. General OpenCOBOL Program Format

Figure 2-1 - General OpenCOBOL Program Format

```
{ [ IDENTIFICATION DIVISION . ]
  PROGRAM-ID. program-name-1 [ IS INITIAL PROGRAM ] .
  [ ENVIRONMENT DIVISION . environment-division-content ]
  [ DATA DIVISION . data-division-content ]
  [ PROCEDURE DIVISION . procedure-division-content ]
  [ nested-source-program | nested-source-function ] ...
  [ END PROGRAM program-name-1 . ] } ...
```

COBOL programs are organized into DIVISIONS – major groupings of language statements that all relate to a common purpose.

Not all divisions are needed in every program, but they must be specified in the order shown when they are used.

1. The OpenCOBOL compiler will compile the source code provided to it (a compilation unit) into a single executable program. This source code can be provided as a single *program unit* (a source code sequence defined by those DIVISIONs required by the program unit, followed by an optional END PROGRAM clause) or as multiple program units EACH consisting of the necessary DIVISIONs and a mandatory END PROGRAM clause. When multiple program units are being compiled in a single compilation unit, the last program unit need not contain an END PROGRAM clause – all others, however, must have one.
2. Specifying multiple input files to the OpenCOBOL compiler defines a compilation unit that consists of the contents of the specified files, compiled in the sequence in which the files are specified. The effect is the same as if a single source file containing multiple program units were compiled, except that the individual source files need not contain END PROGRAM clauses unless they contain multiple program units.
3. Regardless of how many program units comprise a single compilation unit, only a single output executable program will be generated. The first program unit encountered in the compilation unit will serve as the main program – all others must serve as subprograms, called by the main program or by one of the other program units in the sequence.
4. Here is a brief summary of the purpose of each DIVISION:

DIVISION	Purpose
IDENTIFICATION	The IDENTIFICATION DIVISION (section <a href="#">3</a> ) provides basic identification of the program by giving it a program id (a name).
ENVIRONMENT	The ENVIRONMENT DIVISION (section <a href="#">4</a> ) defines the external computer environment in which the program will be operating. This includes defining any files that the program may be accessing.
DATA	The DATA DIVISION (section <a href="#">5</a> ) is used to define all data that will be processed by a program.
PROCEDURE	The PROCEDURE DIVISION (section <a href="#">6</a> ) contains all executable program code.

## 2.1. General Format for Nested Source Programs

Figure 2-2 - General Format for Nested Source Programs

```
[ IDENTIFICATION DIVISION . ]
PROGRAM-ID. program-name-1 [ IS { | INITIAL
                           | COMMON | } PROGRAM ] .
[ ENVIRONMENT DIVISION . environment-division-content ]
[ DATA DIVISION . data-division-content ]
[ PROCEDURE DIVISION . procedure-division-content ]
[ nested-source-program | nested-source-function ] ...
[ END PROGRAM program-name-1 . ]
```

Nested source programs are program units imbedded inside other program units (they follow the PROCEDURE DIVISION of their “parent” program unit and there is no intervening END PROGRAM between the two). As such they serve as subprograms available ONLY to the parent program unit in which they are imbedded<sup>7</sup>.

1. Nested source programs may themselves contain other nested programs. Care should be taken to include END PROGRAM clauses between nested subprograms that should be considered at “equal levels” in the nesting structure.

## 2.2. General Format for Nested Source Functions

Figure 2-3 - General Format for Nested Source Functions

```
FUNCTION-ID. function-name-1 [ IS { | INITIAL
                              | COMMON | } PROGRAM ] .
[ ENVIRONMENT DIVISION . environment-division-content ]
DATA DIVISION. data-division-content
PROCEDURE DIVISION
    [ USING data-item-1 ... ]
    [ RETURNING data-item-n ].
    procedure-division-content
[ nested-source-program | nested-source-function ] ...
[ END FUNCTION function-name-1 . ]
```

User-defined functions are defined in the OpenCOBOL syntax but are not currently supported.

1. Attempts to compile a user-defined function will be rejected with a message such as the following:

```
name:line: Error: FUNCTION-ID is not yet implemented
```

<sup>7</sup> Of course, there are always exceptions to every rule. See the discussion of the COMMON clause to the PROGRAM-ID paragraph on page 8.

### 3. IDENTIFICATION DIVISION

Figure 3-1 - IDENTIFICATION DIVISION Syntax

[ **IDENTIFICATION DIVISION** . ]

**PROGRAM-ID**. *program-name-1* [ IS { **INITIAL** | **COMMON** } ] **PROGRAM** ] .

The IDENTIFICATION DIVISION provides basic identification of the program by giving it a *program id* (a name).

1. While the actual IDENTIFICATION DIVISION header is optional, the PROGRAM-ID clause is not.
2. The PROGRAM-ID clause defines the name (*program-name*) by which other programs may refer to this one (i.e. CALL "*program-name*").
3. Program names ARE case-sensitive. If the compilation unit is being created as a dynamically-loadable library file (by using the "-m" option on the OpenCOBOL compiler command), then the library filename created by the compiler will exactly match the *program-name*. If the compilation unit is being created as an *executable file* (by using the "-x" option on the OpenCOBOL compiler command) then the program-id may be any valid COBOL identifier name because the executable filename will be the same as the source program filename without the "cbl" or "cob" extension.
4. The INITIAL and COMMON clauses are used within subprograms. The COMMON clause should be used only within subprograms that are nested source programs.
5. The INITIAL clause, if specified, guarantees the subprogram will be in its initial (i.e. compiled) state each and every time it is executed, not just the first time.
6. The COMMON clause, if any, makes a nested source program (subprogram) unit available to the parent program unit as well as to other nested source program units of that parent.
7. Obsolete IDENTIFICATION DIVISION entries such as DATE-WRITTEN, DATE-COMPILED, AUTHOR, INSTALLATION, SECURITY and REMARKS are normally ignored unless the "-Wobsolete" compilation switch is used; in such a case, warning messages will be generated but compilation will continue.



## 4. ENVIRONMENT DIVISION

Figure 4-1 - ENVIRONMENT DIVISION Syntax

```
ENVIRONMENT DIVISION.
[ CONFIGURATION SECTION. ]
[ INPUT-OUTPUT SECTION. ]
```

The ENVIRONMENT DIVISION defines the external computer environment in which the program will be operating. This includes defining any files that the program may be accessing.

1. If none of the features provided by the ENVIRONMENT DIVISION are required by a program, the ENVIRONMENT DIVISION need not be specified.

### 4.1. CONFIGURATION SECTION

Figure 4-2 - CONFIGURATION SECTION Syntax

```
CONFIGURATION SECTION.
[ SOURCE-COMPUTER. source-computer-contents ]
[ OBJECT-COMPUTER. object-computer-contents ]
[ REPOSITORY. repository-contents ]
[ SPECIAL-NAMES. special-names-contents ]
```

The CONFIGURATION DIVISION defines the computer system upon which the program is being compiled and executed and also specifies any special environmental configuration or compatibility characteristics.

1. The sequence in which the CONFIGURATION SECTION paragraphs are specified is irrelevant.

#### 4.1.1. SOURCE-COMPUTER Paragraph

Figure 4-3 - SOURCE-COMPUTER Paragraph Syntax

```
SOURCE-COMPUTER. computer-name-1
[ WITH DEBUGGING MODE ] .
```

The SOURCE-COMPUTER paragraph defines the computer upon which the program is being compiled and provides one way in which debugging code imbedded within the program may be activated.

1. The value specified for *computer-name-1* is irrelevant, provided it is a valid COBOL word that does not match any OpenCOBOL reserved word.
2. The WITH DEBUGGING MODE clause, if present, will signal the compiler that debugging lines are to be compiled. Section [1.5](#) discusses how debugging lines are specified in an OpenCOBOL program.
3. Even without the WITH DEBUGGING MODE clause, it is still possible to compile debugging lines. Debugging lines may also be compiled by specifying the “**-fdebugging-1 line**” switch to the OpenCOBOL compiler.

#### 4.1.2. OBJECT-COMPUTER Paragraph

Figure 4-4 - OBJECT-COMPUTER Paragraph Syntax

```
OBJECT-COMPUTER. computer-name-2
MEMORY SIZE IS integer-1 { WORDS
CHARACTERS }
[ PROGRAM COLLATING SEQUENCE IS alphabet-name-1 ]
[ SEGMENT-LIMIT IS integer-2 ] .
```

The OBJECT-COMPUTER paragraph describes the computer upon which the program will execute. This paragraph is not merely documentation.

1. The value specified for *computer-name-2* is irrelevant, provided it is a valid COBOL word that does not match any OpenCOBOL reserved word.



2. The MEMORY SIZE and SEGMENT-LIMIT clauses are supported for compatibility purposes, but are non-functional in OpenCOBOL.
3. The PROGRAM COLLATING SEQUENCE clause allows you to specify a customized character collating sequence to be used when alphanumeric values are compared to one another. Data will still be stored in the character set native to the computer, but the logical sequence in which characters are ordered for comparison purposes can be altered from that inherent to the computer's native character set. The *alphabet-name-1* you specify needs to be defined in the SPECIAL-NAMES section ([4.1.4](#)).
4. If no PROGRAM COLLATING SEQUENCE clause is specified, the collating sequence implied by the character set native to the computer (usually ASCII) will be used.

### 4.1.3. REPOSITORY Paragraph

Figure 4-5 - REPOSITORY Paragraph Syntax

```

REPOSITORY.
FUNCTION { function-name-1... } INTRINSIC .
          ALL

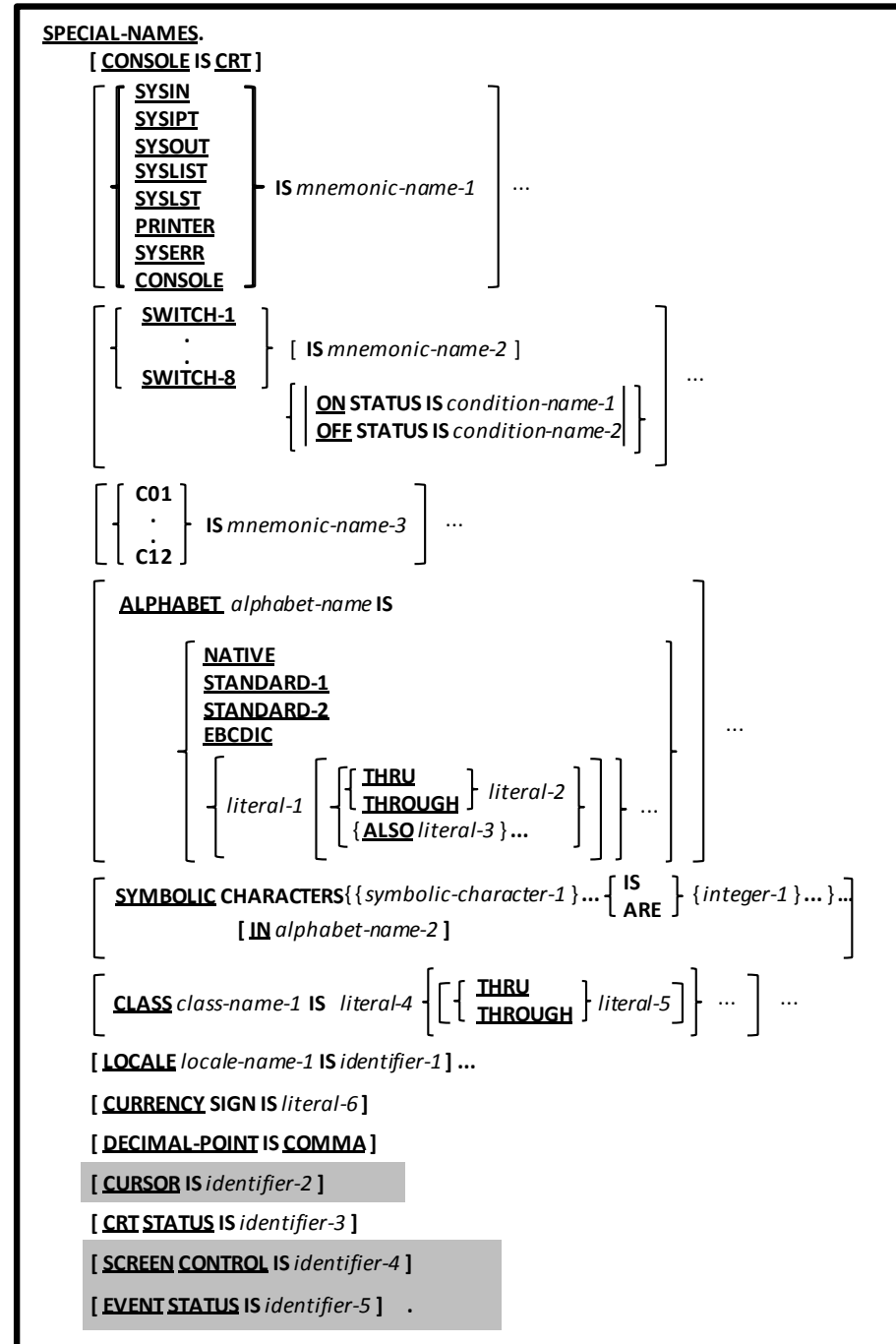
```

The REPOSITORY paragraph provides a mechanism for controlling access to the various built-in intrinsic functions.

1. You may flag one or more (or ALL) intrinsic functions as being usable without the need to code the keyword "FUNCTION" in front of the function names. See section [6.1.7](#) for more information about intrinsic functions.
2. As an alternative to using the REPOSITORY paragraph, you may instead compile your OpenCOBOL programs using the "**-functions-all**" switch.

## 4.1.4. SPECIAL-NAMES Paragraph

Figure 4-6 - SPECIAL-NAMES Paragraph Syntax



The SPECIAL-NAMES paragraph provides a means for specifying the currency sign, choosing the decimal point, [specifying symbolic-characters,] relating implementer-names to user-specified mnemonic names, relating alphabet names to character sets or collating sequences, and relating class names to sets of characters.

In short, this paragraph provides a means of easily “configuring” a COBOL program created in another computing environment so that it will compile with minimal changes in an OpenCOBOL environment.

1. The CONSOLE IS CRT clause exists to provide source code compatibility with other versions of OpenCOBOL. It allows the devices “CRT” and “CONSOLE” to be used interchangeably on DISPLAY (section [6.15.1](#)) and ACCEPT (section [6.4.1](#)) statements. This isn’t needed when coding OpenCOBOL programs “from scratch” because OpenCOBOL already considers those two devices to be synonymous.
2. The IS mnemonic-name-1 clause allows you to specify an alternate name for one of the built-in OpenCOBOL device names specified before the “IS”.

3. The external values of SWITCH-1 through SWITCH-8 are specified to a program using the environment variables COB\_SWITCH\_1 through COB\_SWITCH\_8, respectively. A value of "ON" turns the switch on. Any other value (including the environment variable being undefined) turns the switch off. The ON STATUS and/or OFF STATUS clauses define condition names that may be used to test whether a switch is set or not at run-time. See sections [6.1.4.2.1](#) and [6.1.4.2.4](#) for more information.
4. The ALPHABET clause provides a means for relating a name to a specified character code set or collating sequence, including those you define yourself using the "literal-1" option. You may specify an alphanumeric literal for any of the *literal-1*, *literal-2* or *literal-3* specifications. You may also specify any of the figurative constants SPACE[S], ZERO[[E]S], QUOTE[S], HIGH-VALUE[S] or LOW-VALUE[S].
5. The SYMBOLIC CHARACTERS clause may be used to define your own figurative constants. Take for example the following code which defines figurative constant names for a variety of ASCII characters:

```

SPECIAL-NAMES.
  SYMBOLIC CHARACTERS SOH IS 2
                      BEL IS 8
                      DC1 IS 18
                      DC2 IS 19

```

Values are specified not by their decimal or hexadecimal value by their ordinal position in the program's character set. Remember that the NUL character (all zero bits) is character 1 – this made the values for the ASCII characters shown in the example actually one greater than their actual decimal value.

6. User-defined classes are defined using the CLASS clause. The literal(s) specified on that clause define the possible characters that may be found in a data item's value in order to be considered part of the class. For example, the following defines a class called "Hexadecimal", the definition of which specifies the only characters that may be present in a data item if that data item is to be part of the "Hexadecimal" class:

```

CLASS Hexadecimal IS '0' THRU '9', 'A' THRU 'F', 'a' THRU 'f'

```

See section [6.1.4.2.2](#) for an example of how this user-defined class might be used.

The LOCALE clause may be used to associate UNIX-standard locale names with an identifier defined in the DATA DIVISION. Locale names may be any of the following:

Figure 4-7 - Locale Codes

af_ZA	dv_MV	fi_FI	lt_LT	sma_NO
am_ET	el_GR	fil_PH	lv_LV	sma_SE
ar_AE	en_029	fo_FO	mi_NZ	smj_NO
ar_BH	en_AU	fr_BE	mk_MK	smj_SE
ar_DZ	en_BZ	fr_CA	ml_IN	smn_FI
ar_EG	en_CA	fr_CH	mn_Cyrl_MN	sms_FI
ar_IQ	en_GB	fr_FR	mn_Mong_CN	sq_AL
ar_JO	en_IE	fr_LU	moh_CA	sr_Cyrl_BA
ar_KW	en_IN	fr_MC	mr_IN	sr_Cyrl_CS
ar_LB	en_JM	fy_NL	ms_BN	sr_Latn_BA
ar_LY	en_MY	ga_IE	ms_MY	sr_Latn_CS
ar_MA	en_NZ	gbz_AF	mt_MT	sv_FI
ar_OM	en_PH	gl_ES	nb_NO	sv_SE
ar_QA	en_SG	gsw_FR	ne_NP	sw_KE
ar_SA	en_TT	gu_IN	nl_BE	syr_SY
ar_SY	en_US	ha_Latn_NG	nl_NL	ta_IN
ar_TN	en_ZA	he_IL	nn_NO	te_IN
ar_YE	en_ZW	hi_IN	ns_ZA	tg_Cyrl_TJ
arn_CL	es_AR	hr_BA	oc_FR	th_TH
as_IN	es_BO	hr_HR	or_IN	tk_TM
az_Cyrl_AZ	es_CL	hu_HU	pa_IN	tmz_Latn_DZ
az_Latn_AZ	es_CO	hy_AM	pl_PL	tn_ZA
ba_R	es_CR	id_ID	ps_AF	tr_IN
be_BY	es_DO	ig_NG	pt_BR	tr_TR
bg_BG	es_EC	ii_CN	pt_PT	tt_RU
bn_IN	es_ES	is_IS	qut_GT	ug_CN
bo_BT	es_GT	it_CH	quz_BO	uk_UA
bo_CN	es_HN	it_IT	quz_EC	ur_PK
br_FR	es_MX	iu_Cans_CA	quz_PE	uz_Cyrl_UZ
bs_Cyrl_BA	es_NI	iu_Latn_CA	rm_CH	uz_Latn_UZ
bs_Latn_BA	es_PA	ja_JP	ro_RO	vi_VN
ca_ES	es_PE	ka_GE	ru_RU	wen_DE
cs_CZ	es_PR	kh_KH	rw_RW	wo_SN
cy_GB	es_PY	kk_KZ	sa_IN	xh_ZA

da_DK	es_SV	kl_GL	sah_RU	yo_NG
de_AT	es_US	kn_IN	se_FI	zh_CN
de_CH	es_UY	ko_KR	se_NO	zh_HK
de_DE	es_VE	kok_IN	se_SE	zh_MO
de_LI	et_EE	ky_KG	si_LK	zh_SG
de_LU	eu_ES	lb_LU	sk_SK	zh_TW
dsb_DE	fa_IR	lo_LA	sl_SI	zu_ZA

- The CURRENCY SIGN clause may be used to define any single character as the currency sign used in PICTURE symbol editing (see [Figure 5-9](#)). The default currency sign is a dollar-sign (\$).
- The DECIMAL POINT IS COMMA clause reverses the definition of the “,” and “.” characters when they are used as PICTURE editing symbols (see [Figure 5-9](#)) and numeric literals. This can have unwanted side-effects – see section [1.6](#).
- The PICTURE of *identifier-3* (CRT-STATUS) should be 9(4). This field will receive a 4-digit value indicating the runtime status of a screen ACCEPT. These status codes are as follows:

Figure 4-8 - Screen ACCEPT Key Codes

Code	Meaning	Code	Meaning
0000	ENTER key pressed	2004	Down Arrow <sup>5</sup>
1001 - 1064	F1 – F64	2005	Esc <sup>10</sup>
2001	PgUp <sup>8</sup>	2006	Print Screen <sup>5</sup>
2002	PgDn <sup>4</sup>	8000	No data is available on screen ACCEPT
2003	Up Arrow <sup>9</sup>	9000	Fatal screen I/O error

The actual key pressed to generate a function key (Fn) will depend on the type of terminal device you’re using (PC, Macintosh, VT100, etc.) and what type of enhanced display driver was configured with the version of OpenCOBOL you’re using. For example, on an OpenCOBOL built for a Windows PC using MinGW and PDCurses, F1-F12 are the actual F-keys on the PC keyboard, F13-F24 are entered by shifting the F-keys, F25-F36 are entered by holding Ctrl while pressing an F-key and F37-F48 are entered by holding Alt while pressing an F-key. On the other hand, an OpenCOBOL implementation built for Windows using Cygwin and NCurses treats the PCs F1-F12 keys as the actual F1-F12, while shifted F-keys will enter F11-F20. With Cygwin/NCurses, Ctrl- and Alt-modified F-keys aren’t recognized. Neither are Shift-F11 or Shift-F12.

- If the CRT STATUS clause is not specified, an implicit COB-CRT-STATUS identifier (with a PICTURE of 9(4)) will be allocated for the purpose of receiving screen ACCEPT statuses.
- While the SCREEN CONTROL and EVENT STATUS clauses are clearly noted at compilation time as being unsupported, the CURSOR IS clause is not; currently, however, it appears to be non-functional at runtime.

## 4.2. INPUT-OUTPUT SECTION

Figure 4-9 - INPUT-OUTPUT SECTION Syntax

```

INPUT-OUTPUT SECTION.
[ FILE-CONTROL. file-control-contents ]
[ I-O-CONTROL. io-control-contents ]
    
```

The INPUT-OUTPUT section provides for the detailed definition of any files the program will be accessing.

- If the compiler “config” file you are using has “relaxed-syntax-check” set to “yes”, the FILE-CONTROL and I-O-CONTROL paragraphs may be specified without the INPUT-OUTPUT SECTION header having been specified. See section [7.1.8](#) for more information on config files and their effect on programs.

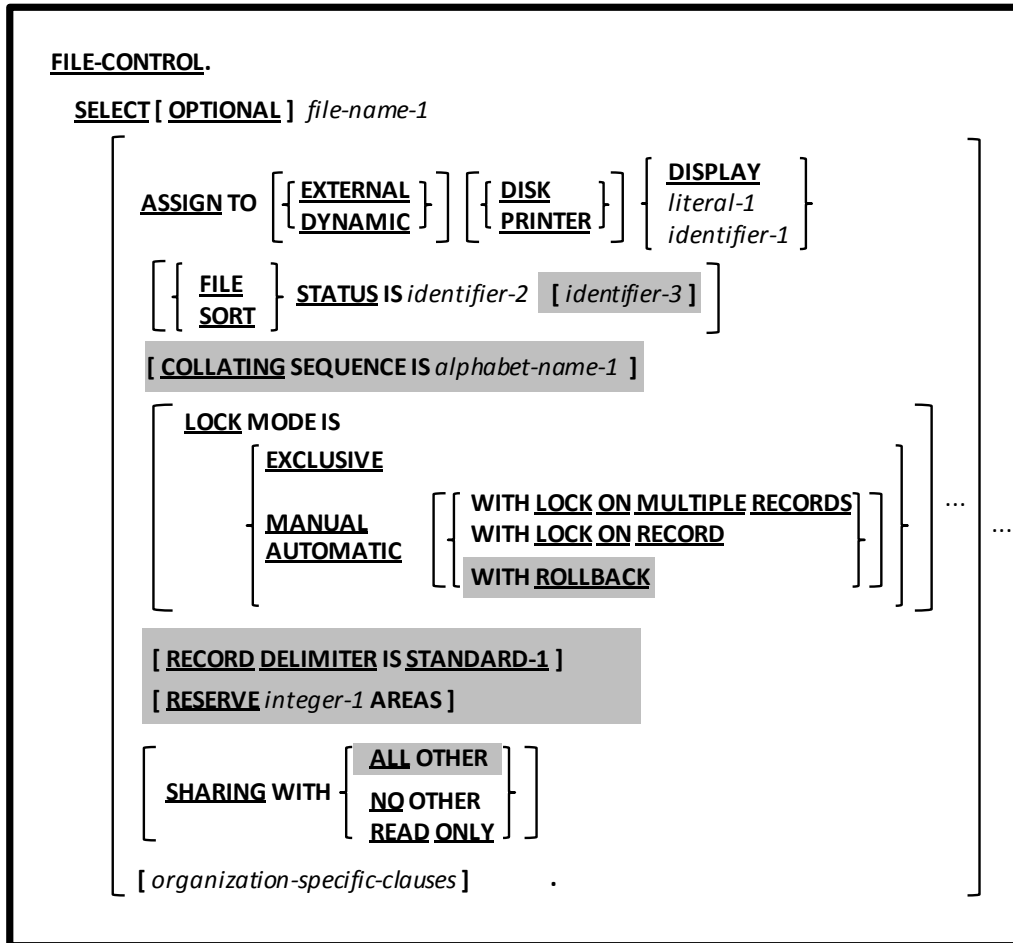
<sup>8</sup> These keys are available ONLY if the environment variable COB\_SCREEN\_EXCEPTIONS is set to any non-blank value at runtime.

<sup>9</sup> These keys are not detectable on Windows systems

<sup>10</sup> This key is available ONLY if the environment variable COB\_SCREEN\_ESC is set to any non-blank value at runtime (this is in addition to setting COB\_SCREEN\_EXCEPTIONS)

## 4.2.1. FILE-CONTROL Paragraph

Figure 4-10 - FILE-CONTROL Paragraph Syntax



The SELECT statement of the FILE-CONTROL paragraph creates a definition of a file and links that COBOL definition to the external operating system environment.

What is shown here are those clauses of the SELECT statement that are common to all types of files.

Upcoming sections will discuss special SELECT clauses that only pertain to certain types of files.

- The COLLATING SEQUENCE, RECORD DELIMITER, RESERVE and SHARING WITH ALL OTHER clauses, as well as the specification of a secondary FILE-STATUS field and LOCK MODE ... WITH ROLLBACK, while syntactically recognized, are not currently supported by OpenCOBOL.
- The OPTIONAL clause, to be used only for files that will be used to provide input data to the program, indicates the file may or may not actually be available at run-time. Attempts to OPEN (section [6.32](#)) an OPTIONAL file when the file does not exist will receive a special non-fatal file status value (see status 05 in [#0](#) below) indicating the file is not available; a subsequent attempt to READ that file (section [6.34](#)) will return an end-of-file condition.
- The OpenCOBOL compiler parser tables actually allow the somewhat nonsensical statement:
 

```
SELECT My-File ASSIGN TO DISK DISPLAY.
```

 ...to be coded and successfully parsed. The effect will be the same as if this were coded:
 

```
SELECT My-File ASSIGN TO DISPLAY.
```

 ...which will be to create a file assigned to the PC screen.
- If the "literal-1" option is used on the ASSIGN clause, it defines the external link from the COBOL file to an operating system file as follows:
  - If an environment variable named "DD\_literal-1" exists, its value will be treated as the full path/filename of the file. If not, then ...

- If an environment variable named “*dd\_literal-1*” exists, its value will be treated as the full path/filename of the file. If not, then ...
- If an environment variable named “*literal-1*” exists, its value will be treated as the full path/filename of the file. If not, then...
- The literal itself will be treated as the full path/filename to the file.

This behavior will be influenced by the “filename-mapping” setting in the config file you are using when compiling your programs. The behavior stated above applies only if “filename-mapping: yes” is in-effect. If “filename-mapping: no” is used, only the last option (treating the literal itself as the full name of the file) is possible. See section [7.1.8](#) for more information on config files and their effect on programs.

The PICTURE of *identifier-2* (the FILE STATUS clause) should be 9(2). An I/O status code will be saved to this identifier after every I/O verb that is executed against the file. Possible status codes are as follows:

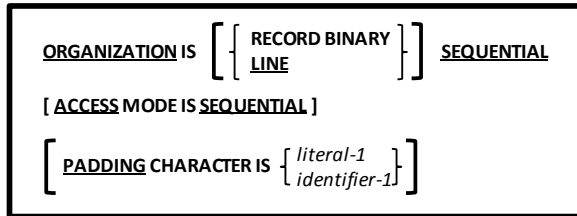
Figure 4-11 - FILE-STATUS Values

Status Value	Meaning
00	Success
02	Success (Duplicate Record Key Written)
05	Success (Optional File Not Found)
07	Success (No Unit)
10	End of file
14	Out of key range
21	Key invalid
22	Attempt to duplicate key value
23	Key not found
30	Permanent I/O error
31	Inconsistent filename
34	Boundary violation
35	File not found
37	Permission denied
38	Closed with lock
39	Conflicting attribute
41	File already OPEN
42	File not OPEN
43	Read not done
44	Record overflow
46	READ error
47	OPEN INPUT denied
48	OPEN OUTPUT denied
49	OPEN I-O denied
51	Record locked
52	End of page
57	LINAGE specifications invalid
61	File sharing failure
91	File not available

6. The LOCK and SHARING clauses define the conditions under which this file will be usable by other programs executing concurrently with this one. File locking and sharing is covered in section [6.1.9](#).

### 4.2.1.1. ORGANIZATION SEQUENTIAL Files

Figure 4-12 - Additional FILE-CONTROL Syntax for SEQUENTIAL Files

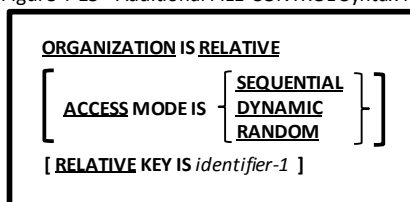


SEQUENTIAL files are those whose internal structure (in COBOL, this is referred to as *organization*) is such that the data in those files can only be processed in a sequential manner; in order to read the 100<sup>th</sup> record in such a file, you first must read records 1 through 99.

- Files declared as ORGANIZATION RECORD BINARY SEQUENTIAL will consist of records with no explicit end-of-record delimiter character sequences; records in such files are “delineated” by a calculated byte-offset (based on record length) into the file. Such files cannot be prepared with any standard text-editing or word processing software as all such programs will imbed delimiter characters. Such files may contain either USAGE DISPLAY or USAGE COMPUTATIONAL (of any variety) data since no character sequence will be interpreted as an end-of-record delimiter.
- Specifying ORGANIZATION IS RECORD BINARY SEQUENTIAL is the same as specifying ORGANIZATION SEQUENTIAL.
- Files declared as ORGANIZATION LINE SEQUENTIAL will consist of records terminated by an ASCII line-feed character (X"10"). When reading a LINE SEQUENTIAL file, records in excess of the size implied by the file's FD will be truncated while records shorter than that size will be padded to the right with the PADDING CHARACTER value.
- The default PADDING CHARACTER value is SPACE.
- While the PADDING CHARACTER clause is syntactically acceptable for all file ORGANIZATIONS, it only makes sense for LINE SEQUENTIAL files as these are the only files where incoming records can ever be padded.
- Both fixed- and variable-length record formats are supported.
- Files ASSIGNED to PRINTER or CONSOLE should be specified as ORGANIZATION LINE SEQUENTIAL.
- See the discussion of the CLOSE(section 0), COMMIT (section 6.11), DELETE (section 6.14), MERGE (section 6.28), OPEN (section 6.32), READ(section 6.34), REWRITE(section 6.37), SORT (section 6.41.1), UNLOCK (section 6.49) and WRITE(section 1), verbs for information on how SEQUENTIAL files are processed.

### 4.2.1.2. ORGANIZATION RELATIVE Files

Figure 4-13 - Additional FILE-CONTROL Syntax for RELATIVE Files



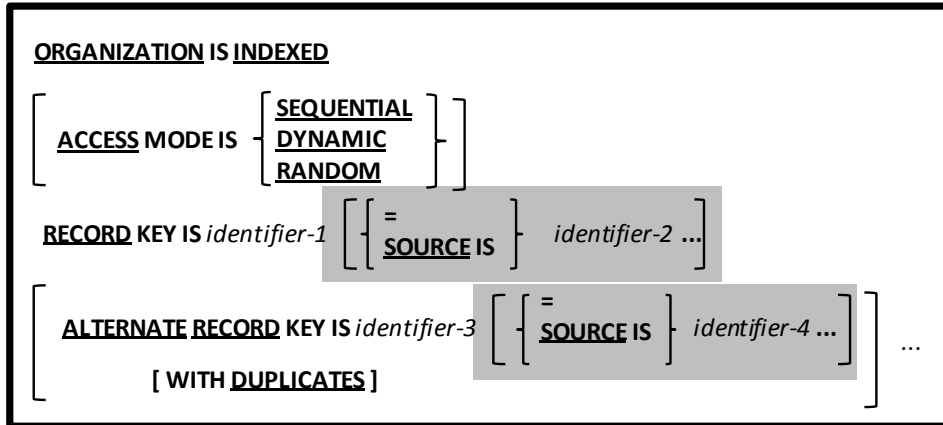
RELATIVE files are files with an internal organization such that records may be processed in a sequential manner or in a random manner, where records may be read, written and updated by specifying the relative record number in the file.

- ORGANIZATION RELATIVE files cannot be assigned to CONSOLE or PRINTER.
- The RELATIVE KEY clause is optional only if ACCESS MODE SEQUENTIAL is specified.
- While records in a ORGANIZATION RELATIVE file may be defined as having variable-length records, the file will be structured in such a manner as to reserve the maximum possible space for each record.
- An ACCESS MODE of SEQUENTIAL indicates that the records of the file will be processed in a sequential manner, while an ACCESS MODE of RANDOM means that records will be processed in random sequence. The DYNAMIC ACCESS MODE indicates the file will be processed either in RANDOM or SEQUENTIAL mode, and may switch back and forth between the two when the program executes (see the START verb in section 6.42).
- The default ACCESS MODE is SEQUENTIAL.

6. The RELATIVE KEY data item is a numeric data item that cannot be a field within records of this file. Its purpose is to return the current relative record number of a RELATIVE file that is being processed in SEQUENTIAL access mode and to be a retrieval key that specifies the relative record number to be read or written when processing a RELATIVE file in RANDOM access mode.
7. See the discussion of the CLOSE(section 0), COMMIT (section 6.11), DELETE (section 6.14), MERGE (section 6.28), OPEN (section 6.32), READ(section 6.34), REWRITE(section 6.37), SORT (section 6.41.1), START (section 6.42), UNLOCK (section 6.49) and WRITE(section 1), verbs for information on how RELATIVE files are processed.

### 4.2.1.3. ORGANIZATION INDEXED Files

Figure 4-14 - Additional FILE-CONTROL Syntax for INDEXED Files



INDEXED files, like RELATIVE files, may have their records processed either sequentially or in a random manner. Unlike RELATIVE files, however, the actual location of a record in an INDEXED file is based upon the value(s) of one or more alphanumeric fields within records of the file.

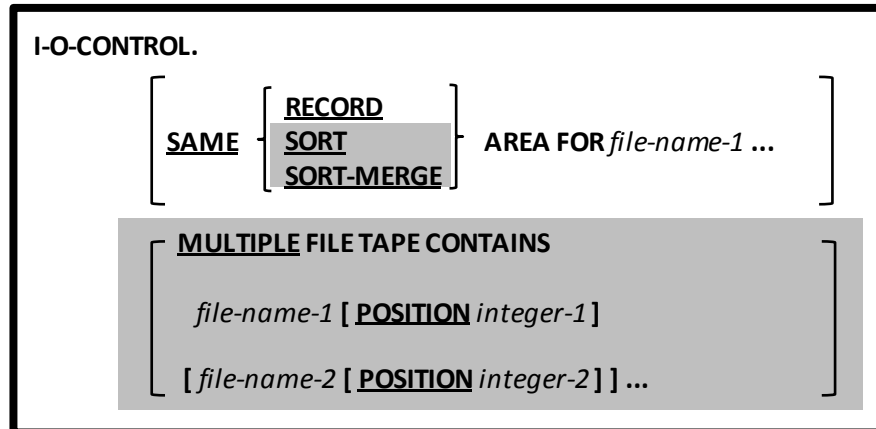
For example, an INDEXED file containing product data might use the product identification code as a key. This means you may read, write or update the “A6G4328”th record or the “Z8X7723”th record directly, based upon the product id value of those records!

1. The specification of so-called “split keys”, while syntactically recognized, are not currently supported by OpenCOBOL.
2. An ACCESS MODE of SEQUENTIAL indicates that the records of the file will be processed in a sequential manner with respect to the values of the RECORD KEY or an ALTERNATE RECORD KEY, while an ACCESS MODE of RANDOM means that records will be processed in random sequence of a key field. The DYNAMIC ACCESS MODE indicates the file will be processed either in RANDOM or SEQUENTIAL mode, and may switch back and forth between the two when the program executes (see the START verb in section 6.42).
3. The default ACCESS MODE is SEQUENTIAL.
4. The PRIMARY KEY clause defines the field(s) within the record used to provide the primary access to records within the file.
5. The ALTERNATE RECORD KEY clause, if used, defines an additional field within the record that provides an alternate means of directly accessing records or an additional field by which the file’s contents may be processed sequentially. You have the choice of allowing records to have duplicate alternate key values, if necessary.
6. There may be multiple ALTERNATE RECORD KEY clauses, each defining an additional alternate key for the file.
7. PRIMARY KEY values must be unique for all records within the value. ALTERNATE RECORD KEY values for records within the file may have duplicate values if and only if the WITH DUPLICATES clause is specified for the alternate key.
8. See the discussion of the CLOSE(section 0), COMMIT (section 6.11), DELETE (section 6.14), MERGE (section 6.28), OPEN (section 6.32), READ(section 6.34), REWRITE(section 6.37), SORT (section 6.41.1), START (section 6.42), UNLOCK (section 6.49) and WRITE(section 1), verbs for information on how INDEXED files are processed.



## 4.2.2. I-O-CONTROL Paragraph

Figure 4-15 - I-O-CONTROL Paragraph Syntax

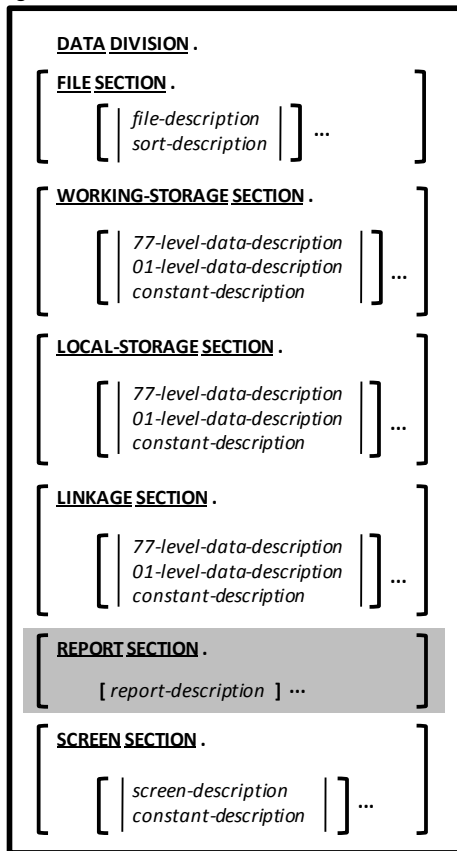


The I-O-CONTROL Paragraph can be used to optimize certain aspects of file processing.

1. The SAME SORT AREA and SAME SORT-MERGE AREA clauses are non-functional. The SAME RECORD AREA is functional, however.
2. The SAME RECORD AREA clause allows you to specify that multiple files should share the same input and output memory buffers. These buffers can sometimes get quite large, and by having multiple files share the same buffer memory you get significantly cut down the amount of memory the program is using (thus making “room” for more procedural code or data). If you do use this feature, take care to ensure that no more than one of the specified files are ever open simultaneously.
3. The MULTIPLE FILE TAPE clause is obsolete and is therefore recognized but not otherwise supported.

## 5. DATA DIVISION

Figure 5-1 - General DATA DIVISION Format

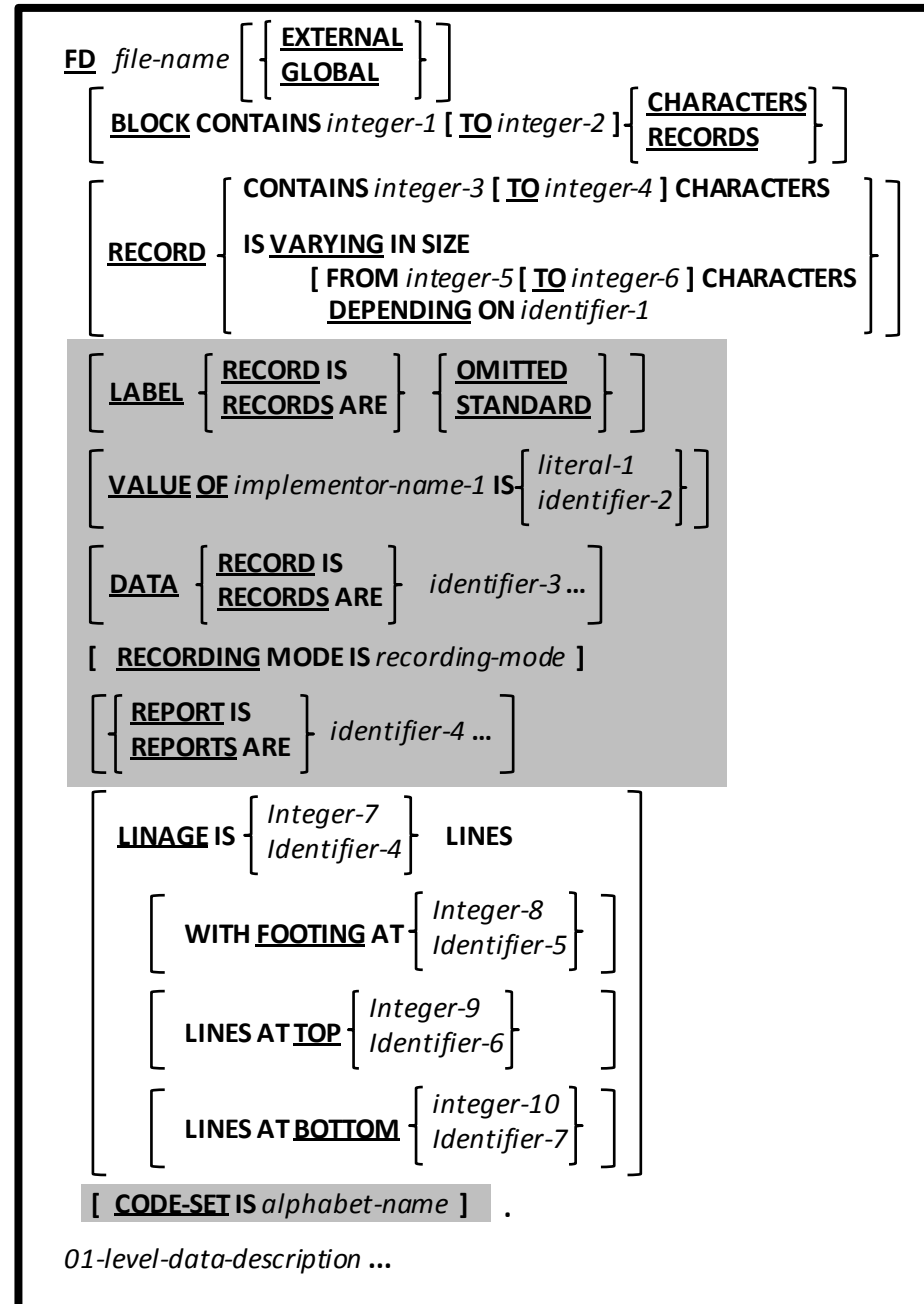


The DATA DIVISION is used to define all data that will be processed by a program. Depending upon the type of data and/or the manner in which the data will be used, its definition will be specified in one of the sections shown in the syntax skeleton to the left.

1. Any SECTIONS that are declared must be specified in the order shown. If no DATA DIVISION sections are needed, the DATA DIVISION header itself may be omitted.
2. The REPORT SECTION is syntactically recognized but will – if used – be rejected as unsupported. OpenCOBOL does not support the RWCS (it does support the LINAGE clause in an FD, however).
3. The LOCAL-STORAGE SECTION is used in a manner identical to the WORKING-STORAGE SECTION with one exception: data defined in the LOCAL-STORAGE SECTION is [re]initialized to its initial state every time the program (usually a subprogram) is executed while WORKING-STORAGE SECTION data is static—it remains in its *last-used state* until the program is CANCELED or the execution of the main program is terminated.
4. LOCAL-STORAGE cannot be used in nested programs.
5. The SCREEN SECTION allows you do define text-based screen layouts using conventions and syntax similar to what you might expect to use if you were using the REPORT SECTION to lay out the structure of a report.
6. Note that there is no COMMON-STORAGE SECTION in OpenCOBOL. This feature has actually been removed from the COBOL standard. Its functionality, however, has been replaced by the EXTERNAL and GLOBAL data item attributes.

## 5.1. FD - File Description

Figure 5-2 - FD Syntax



There must be a detailed description for every file SELECTed in your program. These detailed descriptions will be coded in the FILE SECTION.

There are two types of such descriptions – FDs and SDs, used to describe regular data files and sort work files, respectively.

The FD provides a detailed description of the record format(s) used with the file as well as how those records are “bundled” into physical blocks for processing efficiency.

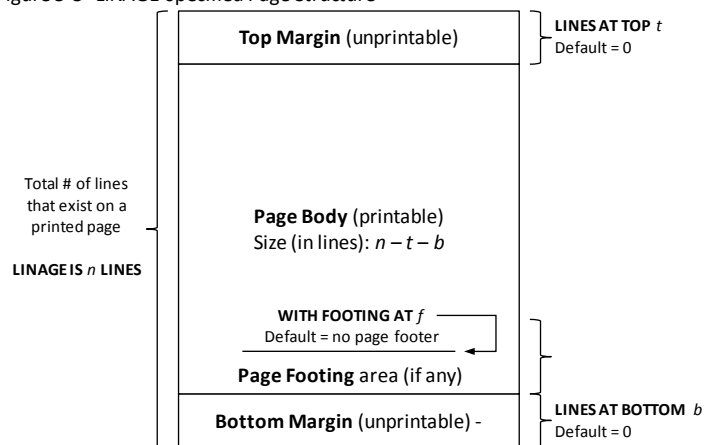
1. The CODE-SET clause, while syntactically recognized, is not currently supported by OpenCOBOL.
2. The LABEL RECORD, DATA RECORD, RECORDING MODE and VALUE OF clauses are obsolete. If used, they will have no impact on the generated code. The identifiers specified on the DATA RECORD clause will be verified as being defined within the program, but the compiler won't care whether they are actually specified as records of the file or not.
3. The COBOL programming language allows for the “blocking” of multiple logical data records into a single physical data record; an actual physical write to an output file being processed sequentially will occur when a memory block is filled with new records (see the COMMIT verb in section [6.11](#)). Similarly, when reading a file sequentially, the first READ issued against the file will retrieve the first physical record (block), from which the first logical

record will be retrieved and delivered to the program. Subsequent READ statements will retrieve successive logical records from the buffer until it is exhausted, in which case another physical read is performed to acquire the next physical record. The BLOCK CONTAINS clause in the FD allows all this processing to be performed in a manner that is completely transparent to the programmer.

4. The RECORD CONTAINS and RECORD IS VARYING clauses are ignored (with a warning message issued) when used with LINE SEQUENTIAL files. With other file organizations these mutually-exclusive clauses define the length of data records within the file. These sizes are used by the BLOCK CONTAINS ... RECORDS clause to calculate a block size.
5. The REPORT IS clause is syntactically recognized but will cause an error since the RWCS is not currently supported by OpenCOBOL.
6. The LINAGE clause can only be specified for ORGANIZATION RECORD BINARY SEQUENTIAL or ORGANIZATION LINE SEQUENTIAL files. If used on an ORGANIZATION RECORD SEQUENTIAL file, the definition of that file will be implicitly changed to LINE SEQUENTIAL.
7. The LINAGE clause is used to specify the logical boundaries (in terms of numbers of lines) of various areas on a printed page, as shown in Figure 5-3.

The manner in which this page structure will be utilized is discussed in section [6.51](#) (the WRITE statement).

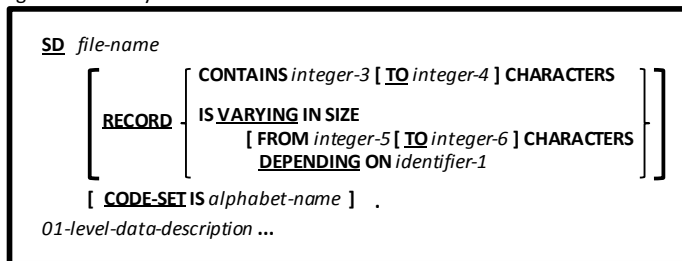
Figure 5-3- LINAGE-specified Page Structure



8. By specifying the EXTERNAL clause, the FD is capable of being shared between all program units (either separately compiled or compiled in the same compilation unit) in a given execution thread, provided the FD is described (with an EXTERNAL clause) in each compilation unit requiring it. This sharing allows the file to be OPENed, read and/or written and CLOSEd in different program units.
9. By specifying the GLOBAL clause, the FD is capable of being shared between all program units in the same compilation unit in a given execution thread, provided the FD is described (with a GLOBAL clause) in each program unit requiring it. This sharing allows the file to be OPENed, read and/or written and CLOSEd in different program units. Separately compiled programs cannot share a GLOBAL FD (but they can share an EXTERNAL FD).

## 5.2. SD - SORT Description

Figure 5-4 - SD Syntax



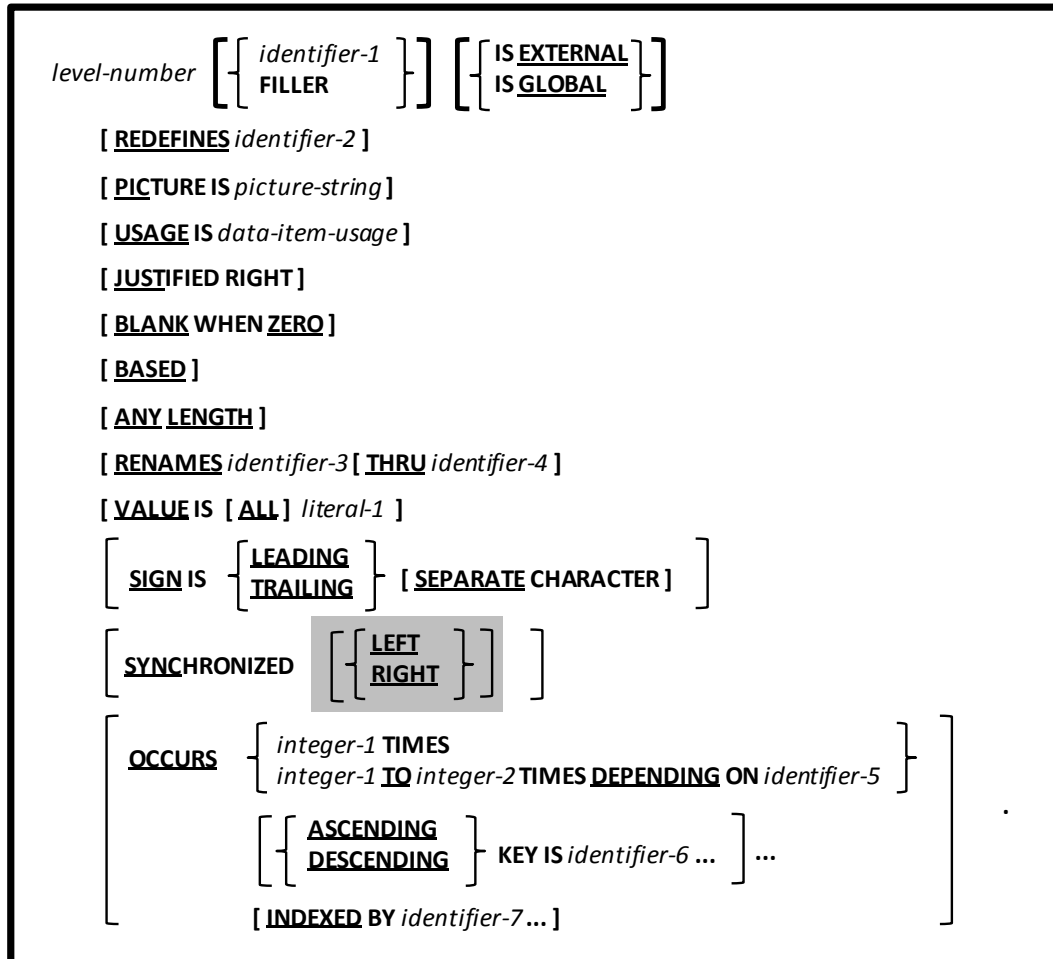
Sort work files (see sections [6.28](#) and [6.41.1](#)) are described using an SD, not an FD.

1. The full "FD" syntax is actually available for a sort description, but only those syntax elements shown here are meaningful.
2. Sort files should be assigned to DISK.

3. Sorts will be performed in memory, if the amount of data being sorted allows.
4. Should disk work files be necessary due to the amount of data being sorted, they will be automatically allocated to disk in a folder defined by the TMPDIR, TMP or TEMP environment variables (see section [7.2.4](#)). These disk files WILL NOT be automatically purged upon program execution termination (normal or otherwise). Temporary sort work files will be named "cob\*.tmp", in case you want to delete them yourself or from within your program upon sort termination.
5. If you specify a specific filename in the sort file's SELECT, it will be ignored.

### 5.3. General Format for Data Descriptions

Figure 5-5 - General Data Description Format



The syntax skeleton shown here describes the manner in which data items are defined in all DATA DIVISION sections except the SCREEN SECTION.

1. Not specifying an identifier name or FILLER immediately after the level number has the same effect as if FILLER were specified.
2. As with other COBOL implementations, level numbers are restricted to the following values, with the meanings shown:
  - 01 – a highest-level data item that may be complete in and of itself (also called an elementary item) or may be broken down into sub-items (also called a group item). 01-level data items are also frequently referred to as *records* or *record descriptions*.
  - 02-49 – these level numbers are used to define data items that are subcomponents of a higher-level data item (the numerically lower the level number, the higher the data item is in the overall hierarchy of the data structure being defined – all structured data must begin with a single 01-level item). All levels 02-49 may be elementary items. All levels 02-48 may also be group items.

- 66 – a regrouping of level 02-49 data – the RENAMES clause is the only one allowed for such items.
- 77 – a data item that is not broken down into sub-items and is not a sub-item of any other data (this is a somewhat obsolete convention as the same can be accomplished using level 01)

There are two additional level numbers (78 and 88) that have very special uses. These are described separately in sections [5.5](#) (78) and [5.4](#) (88).

3. Level-66 data items are merely re-groupings of consecutive data items in a structure that are re-grouped in such a way as to define a group-item name (*identifier-1*) by which they all may be referenced.
4. The PICTURE clause defines the class (numeric, alphabetic or alphanumeric) of the data that may be contained by the data item being defined. A PICTURE also (sometimes in conjunction with USAGE) defines the amount of storage reserved for the data item. The three basic class-specification PICTURE symbols have the following uses:

Figure 5-6 - Data Class-Specification PICTURE Symbols (A/X/9)

Basic Symbol	Meaning and Usage
9	Defines a spot reserved for a single decimal digit. The actual amount of storage occupied will depend on the specified USAGE.
A	Defines a place reserved for a single alphabetic character (“A”-“Z”, “a”-“z”). Each “A” represents a single byte of storage.
X	Defines a place reserved for a single character of storage. Each “X” represents a single byte of storage.

These three symbols are used repeatedly in a PICTURE clause to define how many of each class of data may be contained within the field. For example:

- PIC 9999      Allocates a data item that can store four-digit positive numbers (we’ll see shortly how negative values can be accounted for). If the USAGE of the field is DISPLAY (the default), four bytes of storage will be allocated and each byte may contain the character “0”, “1”, “2”, ... , “8” or “9”. There is no run-time enforcement of the fact that only digits are allowed. A compilation-time WARNING will be issued if literal value that violates the digits-only rule is MOVED to the field. A run-time violation is detectable using a class condition test (see section [6.1.4.2.2](#)).
- PIC 9(4)      Identical to the above – a repeat count enclosed within parenthesis can be used with any PICTURE symbols that allows repetition.
- PIC X(10)     This data item can hold a string of any ten characters.
- PIC A(10)     This data item can hold a string of any ten letters. There is no enforcement of the fact that only letters are allowed, but a violation is detectable via a class condition test (see section [6.1.4.2.2](#)).
- PIC AA9(3)A   This is exactly the same as specifying X(6), but it documents the fact that values should be two letters followed by 3 digits followed by a single letter. There is no enforcement and no capability of detecting violations other than a “brute force” check by character position.

Data items containing “A” or “X” PICTURE symbols cannot be used in arithmetic calculations.

In addition to the above [Figure 5-7](#) shows the numeric option PICTURE symbols that may be used with “PIC 9” data items:

Figure 5-7 - Numeric Option PICTURE Symbols (P/S/V)

Numeric Option Symbol	Meaning and Usage
P	<p>Defines an implied digit position that will be considered to be a 0 when the data item is referenced at run-time. This symbol is used to allow data items that will contain very large values to be allocated using less storage by assuming a certain number of trailing zeros (one per "P") to exist at the end of values.</p> <p>All computations and other operations performed against such a data item will behave as if the zeros were actually there.</p> <p>When values are stored into such a field they will have the digit positions defined by the "P" symbols stripped from the values as they are stored.</p> <p>For example, let's say you need to allocate a data item that contains however many millions of dollars of revenue your company has in gross revenues this year:</p> <p><b>01 Gross-Revenue PIC 9(9).</b></p> <p>In which case 9 bytes of storage will be reserved. The values 000000000 thru 999999999 will represent the gross-revenues. But, if only the <u>millions</u> are tracked (meaning the last six digits are always going to be 0), you could define the field as:</p> <p><b>01 Gross-revenue PIC 9(3)P(6).</b></p> <p>Whenever Gross-Revenue is referenced in the program, the actual value in storage will be treated as if each P symbol (6 of them, in this case) were a zero.</p> <p>If you wanted to store the value 128 million into that field, you would do so as if the "P"s were "9"s:</p> <p><b>MOVE 128000000 TO Gross-Revenue.</b></p>
S	<p>This symbol, which if used must be the very first symbol in the PICTURE value, indicates that negative values are possible for this data item. Without an "S", any negative values stored into this data item via a MOVE or arithmetic statement will have the negative sign stripped from it (in effect becoming the absolute value).</p>
V	<p>This symbol is used to define where an implied decimal-point (if any) is located in a numeric item. Just as there may only be a single decimal point in a number so may there be no more than one "V" in a PICTURE. Implied decimal points occupy no space in storage – they just specify how values are used. For example, if the value "1234" is in storage in a field defined as PIC 999V9, that value would be treated as 123.4 in any statements that referenced it.</p>

5. The SIGN clause, allowable only for USAGE DISPLAY numeric data items, specifies how an "S" symbol will be interpreted. Without the SEPARATE CHARACTER option, the sign of the data item's value will be encoded by transforming the last (TRAILING) or first (LEADING) digit as follows:

Figure 5-8 - Sign-Encoding Characters

First/Last Digit	Encoded Value For POSITIVE	Encoded Value For NEGATIVE
0	0	p
1	1	q
2	2	r
3	3	s
4	4	t
5	5	u
6	6	v
7	7	w
8	8	x
9	9	y

If the SEPARATE CHARACTER clause is used, then an actual “+” or “-” sign will be inserted into the field’s value as the first (LEADING) or last (TRAILING) character.

- 6. OpenCOBOL supports all standard COBOL PICTURE editing symbols, namely “\$”, comma, asterisk (\*), decimal-point, CR, DB, + (plus), - (minus), “B”, “0” (zero) and “/”, as follows:

Figure 5-9 - Numeric Editing PICTURE Symbols

Editing Symbol	Meaning and Usage																					
- (minus)	<p>This symbol must be used either at the very beginning of a PICTURE or at the very end. If “-” is used, none of “+”, “CR” or “DB” may be used. It is used to edit numeric values.</p> <p>Multiple consecutive “-” symbols are allowed only at the very beginning of the field. This is called a <i>floating minus sign</i>.</p> <p>Each “-” symbol will count as one character position in the size of the data item.</p> <p>If only a single “-” symbol is specified, that symbol will be “replaced” by a “-” if the value moved to the field is negative, or a SPACE otherwise.</p> <p>If a floating minus sign is used, think of the editing process as if it worked like this:</p> <ol style="list-style-type: none"> <li>1. Determine what the edited value would be if each “-” were actually a “9”.</li> <li>2. Locate the digit in the edited result that corresponds to the right-most “-” and scan the edited value back to the left from that point until you come to a “0” that has nothing but “0” characters to the left of it.</li> <li>3. Replace that “0” with a “-” if the value moved to the field is negative or a SPACE otherwise.</li> <li>4. Replace all remaining “0” characters to the left of that position by SPACES.</li> </ol> <p>Some examples (the symbol ␣ denotes a space):</p> <table border="1" data-bbox="370 919 1312 1163"> <thead> <tr> <th>If this value...</th> <th>...is moved to a field with this PICTURE...</th> <th>... this value in storage will result:</th> </tr> </thead> <tbody> <tr> <td>17</td> <td>-999</td> <td>␣017</td> </tr> <tr> <td>-17</td> <td>-999</td> <td>-017</td> </tr> <tr> <td>265</td> <td>----99</td> <td>␣␣␣265</td> </tr> <tr> <td>-265</td> <td>----99</td> <td>␣␣-265</td> </tr> <tr> <td>51</td> <td>999-</td> <td>051␣</td> </tr> <tr> <td>-51</td> <td>999-</td> <td>051-</td> </tr> </tbody> </table>	If this value...	...is moved to a field with this PICTURE...	... this value in storage will result:	17	-999	␣017	-17	-999	-017	265	----99	␣␣␣265	-265	----99	␣␣-265	51	999-	051␣	-51	999-	051-
If this value...	...is moved to a field with this PICTURE...	... this value in storage will result:																				
17	-999	␣017																				
-17	-999	-017																				
265	----99	␣␣␣265																				
-265	----99	␣␣-265																				
51	999-	051␣																				
-51	999-	051-																				
\$ <sup>11</sup>	<p>This symbol must be only be used at the very beginning of a PICTURE except that a “+” or “-” may appear to the left of it. It is used to edit numeric values.</p> <p>Multiple consecutive “\$” symbols are allowed. This is called a <i>floating currency symbol</i>.</p> <p>Each “\$” symbol will count as one character position in the size of the data item.</p> <p>If only a single “\$” symbol is specified, that symbol will be inserted into the edited value at that position unless there are so many significant digits to the field value that the position occupied by the “\$” is needed to represent a leading non-zero digit. In such cases, the “\$” will be treated as a “9”.</p> <p>If a floating currency sign is used, think of the editing process as if it worked like this:</p> <ol style="list-style-type: none"> <li>1. Determine what the edited value would be if each “\$” were actually a “9”.</li> <li>2. Locate the digit in the edited result that corresponds to the right-most “\$” and scan the edited value back to the left from that point until you come to a “0” that has nothing but “0” characters to the left of it.</li> <li>3. Replace that “0” with a “\$”.</li> <li>4. Replace all remaining “0” characters to the left of that position by SPACES.</li> </ol> <p>Some examples (the symbol ␣ denotes a space):</p> <table border="1" data-bbox="370 1667 1312 1789"> <thead> <tr> <th>If this value...</th> <th>...is moved to a field with this PICTURE...</th> <th>... this value in storage will result:</th> </tr> </thead> <tbody> <tr> <td>17</td> <td>\$999</td> <td>\$017</td> </tr> <tr> <td>265</td> <td>\$\$\$\$99</td> <td>␣␣\$265</td> </tr> </tbody> </table>	If this value...	...is moved to a field with this PICTURE...	... this value in storage will result:	17	\$999	\$017	265	\$\$\$\$99	␣␣\$265												
If this value...	...is moved to a field with this PICTURE...	... this value in storage will result:																				
17	\$999	\$017																				
265	\$\$\$\$99	␣␣\$265																				

<sup>11</sup> The default currency sign used is “\$”. Other countries use different currency signs. The SPECIAL-NAMES paragraph (see section 4.1.4) allows any symbol to be defined as a currency symbol. If the currency sign is defined to the character “#”, for example, then you would use the “#” character as a PICTURE editing symbol.



Editing Symbol	Meaning and Usage												
* (asterisk)	<p>This symbol must be only be used at the very beginning of a PICTURE except that a “+” or “-” may appear to the left of it. It is used to edit numeric values.</p> <p>Multiple consecutive “*” symbols are not only allowed, but are the typical usage. This is called a <i>floating check protection symbol</i>.</p> <p>Each “*” symbol will count as one character position in the size of the data item.</p> <p>Think of the editing process as if it worked like this:</p> <ol style="list-style-type: none"> <li>1. Determine what the edited value would be if each “*” were actually a “9”.</li> <li>2. Locate the digit in the edited result that corresponds to the right-most “*” and scan the edited value back to the left from that point until you come to a “0” that has nothing but “0” characters to the left of it.</li> <li>3. Replace that “0” with a “*”.</li> <li>4. Replace all remaining “0” characters to the left of that position by “*” also.</li> </ol> <p>An example:</p> <table border="1" data-bbox="375 659 1312 751"> <thead> <tr> <th>If this value...</th> <th>...is moved to a field with this PICTURE...</th> <th>... this value in storage will result:</th> </tr> </thead> <tbody> <tr> <td>265</td> <td>*****99</td> <td>****265</td> </tr> </tbody> </table>	If this value...	...is moved to a field with this PICTURE...	... this value in storage will result:	265	*****99	****265						
If this value...	...is moved to a field with this PICTURE...	... this value in storage will result:											
265	*****99	****265											
, (comma) <sup>12</sup>	<p>Each comma (,) in the PICTURE string represents a character position into which the character “,” will be inserted. This character position is counted in the size of the item. The “,” symbol is a “smart symbol” capable of masquerading as the <u>floating</u> symbol to its left and right should there be insufficient digits of precision to the numeric value being edited to require the insertion of a “,” character.</p> <p>For example (the symbol ␣ denotes a space):</p> <table border="1" data-bbox="375 919 1312 1073"> <thead> <tr> <th>If this value...</th> <th>...is moved to a field with this PICTURE...</th> <th>... this value in storage will result:</th> </tr> </thead> <tbody> <tr> <td>17</td> <td>\$\$,\$\$\$,\$99</td> <td><del>\$\$\$\$\$\$</del>\$17</td> </tr> <tr> <td>265</td> <td>\$\$,\$\$\$,\$99</td> <td><del>\$\$\$\$\$\$</del>\$265</td> </tr> <tr> <td>1456</td> <td>\$\$,\$\$\$,\$99</td> <td><del>\$\$\$\$</del>\$1,456</td> </tr> </tbody> </table>	If this value...	...is moved to a field with this PICTURE...	... this value in storage will result:	17	\$\$,\$\$\$,\$99	<del>\$\$\$\$\$\$</del> \$17	265	\$\$,\$\$\$,\$99	<del>\$\$\$\$\$\$</del> \$265	1456	\$\$,\$\$\$,\$99	<del>\$\$\$\$</del> \$1,456
If this value...	...is moved to a field with this PICTURE...	... this value in storage will result:											
17	\$\$,\$\$\$,\$99	<del>\$\$\$\$\$\$</del> \$17											
265	\$\$,\$\$\$,\$99	<del>\$\$\$\$\$\$</del> \$265											
1456	\$\$,\$\$\$,\$99	<del>\$\$\$\$</del> \$1,456											
. (period) <sup>12</sup>	<p>This symbol inserts a decimal point into the edited value at the point where an implied decimal point exists in the value. It is used to edit numeric values. Note that the period specified at the end of every data item definition IS NOT treated as an editing symbol!</p> <p>An example:</p> <pre>01 Edited-Value      PIC 9(3).99. 01 Payment          PIC 9(3)V99  VALUE 152.19. ... MOVE Payment TO Edited-Value. DISPLAY Edited-Value.</pre> <p>Will display 152.19</p>												
/ (slash)	<p>This symbol – usually used when editing dates for printing – inserts a “/” character into the edited value. The inserted “/” character will occupy a byte of storage in the edited result.</p> <p>An example:</p> <pre>01 Edited-Date      PIC 99/99/9999. ... MOVE 08182009 TO Edited-Date. DISPLAY Edited-Date.</pre> <p>The displayed value will be 08/18/2009.</p>												
+ (plus)	<p>This symbol must be used either at the very beginning of a PICTURE or at the very end. If “+” is used, none of “-”, “CR” or “DB” may be used. It is used to edit numeric values.</p> <p>Multiple consecutive “+” symbols are allowed only at the very beginning of the field. This is called a <i>floating plus sign</i>.</p> <p>Each “+” symbol will count as one character position in the size of the data item. If only a single “+” symbol is specified, that symbol will be replaced by a “-” if the value moved to the field is negative, or a “+”</p>												

<sup>12</sup> If DECIMAL-POINT IS COMMA is specified in the SPECIAL-NAMES paragraph, the meanings and usages of the “.” and “,” characters will be reversed

Editing Symbol	Meaning and Usage																					
	<p>otherwise.</p> <p>If a floating plus sign is used, think of the editing process as if it worked like this:</p> <ol style="list-style-type: none"> <li>Determine what the edited value would be if each "+" were actually a "9".</li> <li>Locate the digit in the edited result that corresponds to the right-most "+" and scan the edited value back to the left from that point until you come to a "0" that has nothing but "0" characters to the left of it.</li> <li>Replace that "0" with a "-" if the value moved to the field is negative or a "+" otherwise.</li> <li>Replace all remaining "0" characters to the left of that position by SPACES.</li> </ol> <p>Some examples (the symbol ␣ denotes a space):</p> <table border="1" data-bbox="375 499 1308 743"> <thead> <tr> <th>If this value...</th> <th>...is moved to a field with this PICTURE...</th> <th>... this value in storage will result:</th> </tr> </thead> <tbody> <tr> <td>17</td> <td>+999</td> <td>+017</td> </tr> <tr> <td>-17</td> <td>+999</td> <td>-017</td> </tr> <tr> <td>265</td> <td>++++99</td> <td>␣␣␣+265</td> </tr> <tr> <td>-265</td> <td>++++99</td> <td>␣␣␣-265</td> </tr> <tr> <td>51</td> <td>999+</td> <td>051+</td> </tr> <tr> <td>-51</td> <td>999-</td> <td>051-</td> </tr> </tbody> </table>	If this value...	...is moved to a field with this PICTURE...	... this value in storage will result:	17	+999	+017	-17	+999	-017	265	++++99	␣␣␣+265	-265	++++99	␣␣␣-265	51	999+	051+	-51	999-	051-
If this value...	...is moved to a field with this PICTURE...	... this value in storage will result:																				
17	+999	+017																				
-17	+999	-017																				
265	++++99	␣␣␣+265																				
-265	++++99	␣␣␣-265																				
51	999+	051+																				
-51	999-	051-																				
0 (zero)	<p>This symbol inserts a "0" character into the edited value. The inserted "0" character will occupy a byte of storage in the edited result.</p> <p>An example:</p> <p><b>01 Edited-Phone-Number PIC 9(3)B9(3)B9(4).</b></p> <p>...</p> <p><b>MOVE 5185551212 TO Edited-Phone-Number.</b></p> <p><b>DISPLAY Edited-Phone-Number.</b></p> <p>The displayed value will be <b>518 555 1212</b>.</p>																					
B	<p>This symbol inserts a SPACE character into the edited value. The inserted SPACE character will occupy a byte of storage in the edited result.</p> <p>An example:</p> <p><b>01 Edited-Phone-Number PIC 9(3)B9(3)B9(4).</b></p> <p>...</p> <p><b>MOVE 5185551212 TO Edited-Phone-Number.</b></p> <p><b>DISPLAY Edited-Phone-Number.</b></p> <p>The displayed value will be <b>518 555 1212</b>.</p>																					
CR	<p>This symbol must be used only at the very end of a PICTURE. If "CR" is used, none of "-", "+" or "DB" may be used. It is used to edit numeric values.</p> <p>Multiple "CR" symbols are not allowed in one PICTURE clause.</p> <p>A "CR" symbol will count as two character positions in the size of the data item.</p> <p>If the value moved into the field is negative, the characters "CR" will be inserted into the edited value, otherwise two SPACES will be inserted.</p> <p>Some examples (the symbol ␣ denotes a space):</p> <table border="1" data-bbox="375 1524 1308 1646"> <thead> <tr> <th>This value...</th> <th>...is moved to a field with this PICTURE...</th> <th>...resulting in this value in storage:</th> </tr> </thead> <tbody> <tr> <td>17</td> <td>99CR</td> <td>17␣␣</td> </tr> <tr> <td>-17</td> <td>99CR</td> <td>17CR</td> </tr> </tbody> </table>	This value...	...is moved to a field with this PICTURE...	...resulting in this value in storage:	17	99CR	17␣␣	-17	99CR	17CR												
This value...	...is moved to a field with this PICTURE...	...resulting in this value in storage:																				
17	99CR	17␣␣																				
-17	99CR	17CR																				
DB	<p>This symbol must be used only at the very end of a PICTURE. If "DB" is used, none of "-", "+" or "CR" may be used. It is used to edit numeric values.</p> <p>Multiple "DB" symbols are not allowed in one PICTURE clause.</p> <p>A "DB" symbol will count as two character positions in the size of the data item.</p> <p>If the value moved into the field is negative, the characters "DB" will be inserted into the edited value, otherwise two SPACES will be inserted.</p> <p>Some examples (the symbol b denotes a space):</p> <table border="1" data-bbox="375 1927 1308 1959"> <thead> <tr> <th>This value...</th> <th>...is moved to a field with</th> <th>...resulting in this value in</th> </tr> </thead> <tbody> <tr> <td></td> <td></td> <td></td> </tr> </tbody> </table>	This value...	...is moved to a field with	...resulting in this value in																		
This value...	...is moved to a field with	...resulting in this value in																				

Editing Symbol	Meaning and Usage											
		this PICTURE...	storage:									
	17	99DB	17bb									
	-17	99DB	17DB									
Z	<p>This symbol must be only be used at the very beginning of a PICTURE except that a “+” or “-” may appear to the left of it. It is used to edit numeric values.</p> <p>Multiple consecutive “Z” symbols are not only allowed, but are the typical manner in which this editing symbol is used. This is called a <i>floating zero suppression</i>.</p> <p>Each “Z” symbol will count as one character position in the size of the data item.</p> <p>Think of the editing process as if it worked like this:</p> <ol style="list-style-type: none"> <li>Determine what the edited value would be if each “Z” were actually a “9”.</li> <li>Locate the digit in the edited result that corresponds to the right-most “Z” and scan the edited value back to the left from that point until you come to a “0” that has nothing but “0” characters to the left of it.</li> <li>Replace that “0” with a SPACE.</li> <li>Replace all remaining “0” characters to the left of that position by SPACES.</li> </ol> <p>Some examples (the symbol ␣ denotes a space):</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th style="background-color: black; color: white;">This value...</th> <th style="background-color: black; color: white;">...is moved to a field with this PICTURE...</th> <th style="background-color: black; color: white;">...resulting in this value in storage:</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">17</td> <td style="text-align: center;">Z999</td> <td style="text-align: center;">␣017</td> </tr> <tr> <td style="text-align: center;">265</td> <td style="text-align: center;">ZZZZZ99</td> <td style="text-align: center;">␣␣␣265</td> </tr> </tbody> </table>			This value...	...is moved to a field with this PICTURE...	...resulting in this value in storage:	17	Z999	␣017	265	ZZZZZ99	␣␣␣265
This value...	...is moved to a field with this PICTURE...	...resulting in this value in storage:										
17	Z999	␣017										
265	ZZZZZ99	␣␣␣265										

No more than one editing symbol may be used in a floating manner in the same PICTURE clause.

- Numeric data items containing editing symbols are referred to as numeric edited fields. Such data items may receive values in the various arithmetic statements but may not be used as sources of data in those same statements. The statements in question are ADD (section 6.5), COMPUTE(section 6.12), DIVIDE(section 0), MULTIPLY (section 6.30) and SUBTRACT (section 6.45).
- By specifying the EXTERNAL clause, the data item being defined is capable of being shared between all program units (either separately compiled or compiled in the same compilation unit) in a given execution thread, provided the data item is described (with an EXTERNAL clause) in each compilation unit requiring it.
- By specifying the GLOBAL clause, the data item is capable of being shared between all program units in the same compilation unit in a given execution thread, provided the data item is described (with an GLOBAL clause) in each program unit requiring it.
- The EXTERNAL clause may only be specified at the 77 or 01 level.
- An EXTERNAL item must have a data name (i.e. *identifier-1*) and that name cannot be FILLER.
- EXTERNAL cannot be combined with GLOBAL, REDEFINES or BASED.
- The VALUE clause is ignored on EXTERNAL data items or on any data items defines as subordinate to an EXTERNAL data item.
- The OCCURS clause is used to create a data structure called a *table*<sup>13</sup> that repeats multiple times. For example:  
**05 QUARTLY-REVENUE OCCURS 4 TIMES PIC 9(7)V99.**

Will allocate the following:

QUARTLY-REVENUE (1)	QUARTLY-REVENUE (2)	QUARTLY-REVENUE (3)	QUARTLY-REVENUE (4)
---------------------	---------------------	---------------------	---------------------

Each occurrence is referenced using the subscript syntax (a numeric literal, arithmetic expression or numeric identifier enclosed within parenthesis) shown in the diagram. The OCCURS clause may be used at the group level too, in which case the entire group structure repeats, as follows:

**05 X OCCURS 3 TIMES.**

<sup>13</sup> Other programming languages with which you might be familiar refer to this sort of structure as an *array*.

```

10 A    PIC X(1).
10 B    PIC X(1).
10 C    PIC X(1).

```

X (1)			X (2)			X (3)		
A (1)	B (1)	C (1)	A (2)	B (2)	C (2)	A (3)	B (3)	C (3)

See sections [6.1.1](#) (Table References), [6.39](#) (SEARCH), [6.41](#) (SORT) as well as item [#28](#) below for more information about tables.

13. The optional DEPENDING ON clause can be added to an OCCURS to create a variable-length table. Such tables will be allocated out to the maximum size specified as *integer-2*. At execution time the value of *identifier-5* will determine how many of the table elements are accessible.
14. The OCCURS clause cannot be specified in a data description entry that has a level number of 01, 66, 77, or 88.
15. VALUE specifies an initial compilation-time value that will be assigned to the storage occupied by the data item in the program object code generated by the compiler. If the optional "ALL" clause is used, it may only be used with an alphanumeric literal value; the value will be repeated as needed to completely fill the data item. Here are some examples with and without ALL:
 

```

PIC X(5) VALUE "A" - will have the value "A",SPACE,SPACE,SPACE,SPACE
PIC X(5) VALUE ALL "A" - will have the value "A","A","A","A","A"
PIC 9(3) VALUE 1 - will have the value 001
PIC 9(3) VALUE ALL "1" - will have the value 111

```
16. The ASCENDING KEY, DESCENDING KEY and INDEXED BY clauses will be discussed in section [6.39](#) (SEARCH).
17. The BASED and ANY LENGTH clauses cannot be used together.
18. The JUSTIFIED RIGHT clause, valid only on an alphabetic (PIC A) or alphanumeric (PIC X) item, will cause values shorter than the length of the data item to be right-justified and space-filled when they are MOVED into the data item.
19. Data items declared with BASED are allocated no storage at compilation time. At run-time, the ALLOCATE verb is used to allocate space for and (optionally) initialize such items.
20. Data items declared with the ANY LENGTH attribute have no fixed compile-time length. Such items may only be defined in the LINKAGE SECTION as they may only serve as subroutine argument descriptions. ANY LENGTH items must have a PICTURE clause that specifies exactly one A, X or 9 symbol.
21. The BLANK WHEN ZERO clause, when used on a numeric item, will cause that item's value to be automatically transformed into SPACES if a value of 0 is ever MOVED to the item.
22. The REDEFINES clause causes *identifier-1* to occupy the same physical storage space as *identifier-2*, so that storage may be defined in a different manner with a (probably) different structure. The following must all be true in order to use REDEFINES:
  - a. The level number of *identifier-2* must be the same as that of *identifier-1*.
  - b. The level number of *identifier-2* (and *identifier-1*) cannot be 66, 77, 78 or 88.
  - c. If "n" represents the level number of *identifier-2* (and *identifier-1*), then no other data items with level number "n" may be defined between *identifier-1* and *identifier-2*.
  - d. The total allocated size of *identifier-1* must be the same as the total allocated size of *identifier-2*.
  - e. No OCCURS clause may be defined on *identifier-2*. There may – however – be items defined with OCCURS clauses subordinate to *identifier-2*.
  - f. No VALUE clause may be defined on *identifier-2*. No data items subordinate to *identifier-2* may have VALUE clauses, with the exception of level-88 condition names.

23. The following table summarizes the various possible USAGE specifications:

Figure 5-10 - Summary of USAGE Specifications

USAGE	Allocated Space (Bytes)	Storage Format	Allows negative Values?	Used w/ PIC?	Identical To
<u>BINARY</u>	Depends on number of "9"s in PICTURE and the "binary-size" setting of the configuration file (section <a href="#">7.1.8</a> ) used to compile the program	Most-compatible – see <a href="#">#24</a>	If PICTURE contains "S"	Yes	COMPUTATIONAL, COMPUTATIONAL-4
<u>BINARY-CHAR</u> or <u>BINARY-CHAR SIGNED</u>	One byte	Native – see <a href="#">#24</a>	Yes	No	
<u>BINARY-CHAR UNSIGNED</u>	One byte	Native – see <a href="#">#24</a>	No – see <a href="#">#25</a>	No	
<u>BINARY-C-LONG</u> or <u>BINARY-C-LONG SIGNED</u>	Allocates the same amount of storage as does the C language "long" data type on that computer; typically this is 32 bits but it could be 64 bits	Native – see <a href="#">#24</a>	Yes	No	
<u>BINARY-C-LONG UNSIGNED</u>	Allocates the same amount of storage as does the C language "long" data type on that computer; typically this is 32 bits but it could be 64 bits	Native – see <a href="#">#24</a>	No – see <a href="#">#25</a>	No	
<u>BINARY-DOUBLE</u> or <u>BINARY-DOUBLE SIGNED</u>	Allocates a "traditional" double-word of storage (64 bits)	Native – see <a href="#">#24</a>	Yes	No	
<u>BINARY-DOUBLE UNSIGNED</u>	Allocates a "traditional" double-word of storage (64 bits)	Native – see <a href="#">#24</a>	No – see <a href="#">#25</a>	No	
<u>BINARY-LONG</u> or <u>BINARY-LONG SIGNED</u>	Allocates a word of storage (32 bits)	Native – see <a href="#">#24</a>	Yes	No	SIGNED-LONG, SIGNED-INT
<u>BINARY-LONG UNSIGNED</u>	Allocates a word of storage (32 bits)	Native – see <a href="#">#24</a>	No – see <a href="#">#25</a>	No	UNSIGNED-LONG, UNSIGNED-INT
<u>BINARY-SHORT</u> or <u>BINARY-SHORT SIGNED</u>	Allocates a half-word of storage (16 bits)	Native – see <a href="#">#24</a>	Yes	No	SIGNED-SHORT
<u>BINARY-SHORT UNSIGNED</u>	Allocates a half-word of storage (16 bits)	Native – see <a href="#">#24</a>	No – see <a href="#">#25</a>	No	UNSIGNED-SHORT
<u>COMPUTATIONAL</u>	Depends on number of "9"s in PICTURE and the "binary-size" setting of the configuration file (section <a href="#">7.1.8</a> ) used to compile the program	Most-compatible – see <a href="#">#24</a>	If PICTURE contains "S"	Yes	BINARY, COMPUTATIONAL-4
<u>COMPUTATIONAL_1</u>	Allocates a word of storage (32 bits)	Single-precision floating-point	Yes	No	
<u>COMPUTATIONAL_2</u>	Allocates a double-word of storage (64 bits)	Double-precision floating-point	Yes	No	
<u>COMPUTATIONAL_3</u>	Allocates 4 bits per "9" in the PICTURE plus a (trailing) 4-byte field for the sign, rounded up to the nearest byte, SYNCHRONIZED RIGHT (see <a href="#">#27</a> )	Packed decimal – see <a href="#">#26</a>	If PICTURE contains "S"	No	PACKED-DECIMAL
<u>COMPUTATIONAL_4</u>	Depends on number of "9"s in PICTURE and the "binary-size" setting of the configuration file (section <a href="#">7.1.8</a> ) used to compile the program	Most-compatible – see <a href="#">#24</a>	If PICTURE contains "S"	Yes	BINARY, COMPUTATIONAL
<u>COMPUTATIONAL_5</u>	Depends on number of "9"s in PICTURE and the "binary-size" setting of the configuration file (section <a href="#">7.1.8</a> ) used to compile the program		If PICTURE contains "S"	Yes	

USAGE	Allocated Space (Bytes)	Storage Format	Allows negative Values?	Used w/ PIC?	Identical To
<u>COMPUTATIONAL-X</u>	Allocates bytes based upon the number of “9”s in the PICTURE according to the “binary-size” setting of “1–8” in the configuration file used to compile the program. See section <a href="#">7.1.8</a> for an illustration of how a value of “1–8” for “binary-size” would work.	Most-compatible – see <a href="#">#24</a>	If PICTURE contains “S”	Yes	
<u>DISPLAY</u>	Depends on PICTURE – One character <sup>14</sup> per X, A, 9, period, \$, Z, O, *, S (if SEPARATE CHARACTER specified), +, - or B symbol in PICTURE; Add 2 more bytes if DB or CR symbol used	Characters	If PICTURE contains “S”	Yes	
<u>INDEX</u>	Allocates a word of storage (32 bits)	Native – see <a href="#">#24</a>	No	No	
<u>NATIONAL</u>	USAGE NATIONAL, while syntactically recognized, is not supported by OpenCOBOL				
<u>PACKED-DECIMAL</u>	Allocates 4 bits per “9” in the PICTURE plus a (trailing) 4-byte field for the sign, rounded up to the nearest byte, SYNCHRONIZED RIGHT (see <a href="#">#27</a> )	Packed decimal – see <a href="#">#26</a>	If PICTURE contains “S”	No	COMPUTATIONAL-3
<u>POINTER</u>	Allocates a word of storage (32 bits)	Native – see <a href="#">#24</a>	No	No	
<u>PROGRAM-POINTER</u>	Allocates a word of storage (32 bits)	Native – see <a href="#">#24</a>	No	No	
<u>SIGNED-INT</u>	Allocates a word of storage (32 bits)	Native – see <a href="#">#24</a>	Yes	No	BINARY-LONG-SIGNED, SIGNED-LONG
<u>SIGNED-LONG</u>	Allocates a word of storage (32 bits)	Native – see <a href="#">#24</a>	Yes	No	BINARY-LONG SIGNED, SIGNED-INT
<u>SIGNED-SHORT</u>	Allocates a half-word of storage (16 bits)	Native – see <a href="#">#24</a>	Yes	No	BINARY SHORT SIGNED
<u>UNSIGNED-INT</u>	Allocates a word of storage (32 bits)	Native – see <a href="#">#24</a>	No – see <a href="#">#25</a>	No	BINARY-LONG UNSIGNED, UNSIGNED-LONG
<u>UNSIGNED-LONG</u>	Allocates a word of storage (32 bits)	Native – see <a href="#">#24</a>	No – see <a href="#">#25</a>	No	BINARY-LONG UNSIGNED, UNSIGNED-INT
<u>UNSIGNED-SHORT</u>	Allocates a half-word of storage (16 bits)	Native – see <a href="#">#24</a>	No – see <a href="#">#25</a>	No	BINARY-SHORT UNSIGNED

#### 24. Binary data can be stored in either a “Big-Endian” or “Little-Endian” form.

Big-endian data allocation calls for the bytes that comprise a binary item to be allocated such that the least-significant byte is the right-most byte. For example, a four-byte binary item having a value of decimal 20 would be big-endian allocated as 00000014 (shown in hexadecimal notation).

Little-endian data allocation calls for the bytes that comprise a binary item to be allocated such that the least-significant byte is the left-most byte. For example, a four-byte binary item having a value of decimal 20 would be little-endian allocated as 14000000 (shown in hexadecimal notation).

All CPUs are capable of “understanding” big-endian format, which makes it the “most-compatible” form of binary storage across computer systems.

Some CPUs – such as the Intel/AMD i386/x64 architecture processors such as those used in most Windows PCs – prefer to process binary data stored in a little-endian format. Since that format is more efficient on those systems, it is referred to as the “native” binary format.

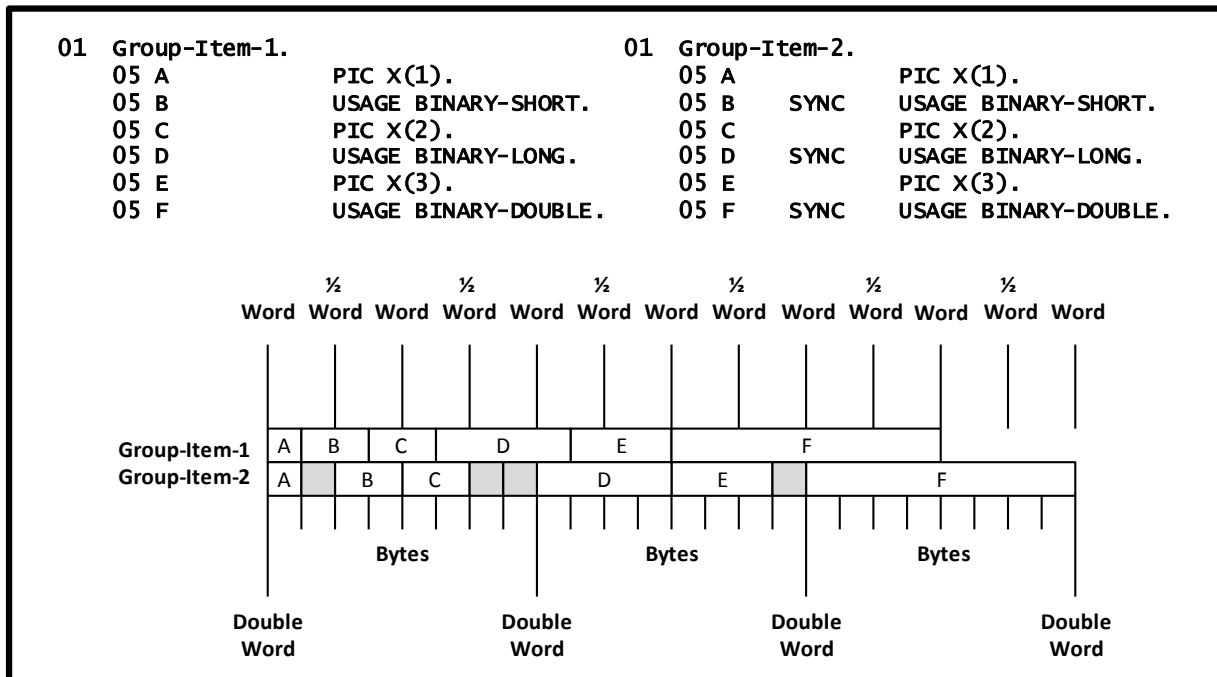
<sup>14</sup> In this context, one character is the same as one byte, unless you’ve built yourself an OpenCOBOL system that uses Unicode (unlikely), in which case 1 character = two bytes.

On a system supporting only one format of binary storage (generally, that would be big-endian), the terms “most-efficient” format and “native format” are synonymous.

25. Binary data items that have the UNSIGNED attribute explicitly coded, or that do not have an “S” symbol in their PICTURE clause cannot preserve negative values that may be stored into them. Attempts to store a negative value into such a field will actually result in the binary representation of the negative number actually being interpreted as if it were a positive number. For example, on a computer running an Intel or AMD processor, the value of -3 expressed as a binary value would be 1111101<sub>2</sub>. If that value is moved into a USAGE BINARY-CHAR UNSIGNED field, it would actually be interpreted as 01111101<sub>2</sub>, or 253.
26. Packed-decimal (i.e. USAGE COMP-3 or USAGE PACKED-DECIMAL) data is stored as a series of bytes such that each byte contains two 4-bit fields with each field representing a “9” in the PICTURE and storing a single decimal digit. The last byte will always contain a single 4-bit digit (corresponding to a “9” and a 4-bit sign indicator (always present, even if no “S” symbol is used). The first byte may contain an unused left-most 4-bit field, depending on how many “9” symbols were used in the PICTURE. The sign indicator will have a value of a hexadecimal A thru F, with values of A, C, E and F indicating a positive sign and B or D representing a negative value. Therefore, a PIC S9(3) COMP-3 packed-decimal field with a value of -15 would be stored internally as a hexadecimal 015D (or perhaps a 015B). If you attempt to store a negative number into a packed decimal field that has no “S” in its PICTURE, the absolute value of the negative number will actually be stored.
27. The SYNCHRONIZED clause (which may be abbreviated as SYNC) optimizes the storage of binary numeric items to store them in such a manner as to make it as fast as possible for the CPU to fetch them. This synchronization is performed as follows:
  - a. If the binary item occupies one byte of storage, no synchronization is performed.
  - b. If the binary item occupies two bytes of storage, the binary item is allocated at the next half-word boundary.
  - c. If the binary item occupies four bytes of storage, the binary item is allocated at the next word boundary.
  - d. If the binary item occupies four bytes of storage, the binary item is allocated at the next word boundary.

Here’s an example of a group item’s storage allocation with and without using SYNCHRONIZED:

Figure 5-11 - Effect of the SYNCHRONIZED Clause



The grey blocks represent the unused “slack” bytes that are allocated in the **Group-Item-2** structure because of the SYNC clauses.

The LEFT and RIGHT options to the SYNCHRONIZED clause are recognized for syntactical compatibility with other COBOL implementations, but are otherwise non-functional.

28. Initializing a table is one of the trickier aspects of COBOL data definition. There are basically three standard techniques and a fourth that people familiar with other COBOL implementations but new to OpenCOBOL may find interesting. So, here are the three “standard” approaches:

a. Don’t bother worrying about it at compile-time. Use the INITIALIZE verb to initialize all data item occurrences in a table (at run-time) to their data-type-specific default values (numerics: 0, alphabetic and alphanumeric: SPACES).

b. Initialize small tables at compile time by including a VALUE clause on the group item that serves as a “parent” to the table, as follows:

```
05 SHIRT-SIZES          VALUE "S 14M 15L 16XL17".
   10 SHIRT-SIZE-TBL   OCCURS 4 TIMES.
       15 SST-SIZE     PIC X(2).
       15 SST-NECK     PIC 9(2).
```

c. Initialize tables of almost any size at compilation time by utilizing the REDEFINES clause:

```
05 SHIRT-SIZE-VALUES.
   10 PIC X(4)          VALUE "S 14".
   10 PIC X(4)          VALUE "M 15".
   10 PIC X(4)          VALUE "L 16".
   10 PIC X(4)          VALUE "XL17".
05 SHIRT-SIZES         REDEFINES SHIRT-SIZE-VALUES.
   10 SHIRT-SIZE-TBL   OCCURS 4 TIMES.
       15 SST-SIZE     PIC X(2).
       15 SST-NECK     PIC 9(2).
```

Admittedly, the table shown in #28c is much more verbose than #28b. What is good about #28c, however, is that you can have as many FILLER/VALUE items as you need for a larger table (and those values can be as long as necessary!

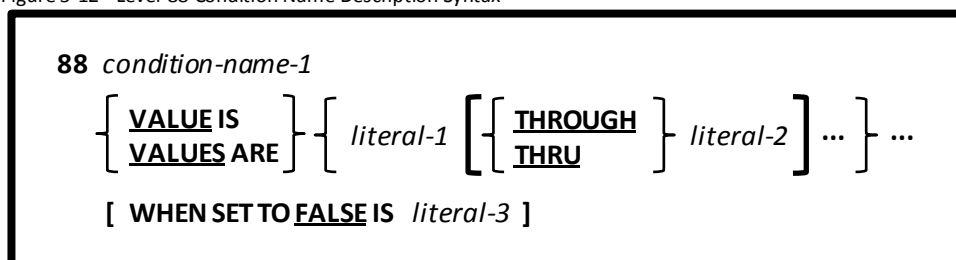
Many COBOL compilers do not allow the use of VALUE and OCCURS on the same data item; additionally, they don’t allow a VALUE clause on a data item subordinate to an OCCURS. OpenCOBOL, however, has neither of these restrictions! Observe the following example, which illustrates the fourth manner in which tables may be initialized in OpenCOBOL:

```
05 X                   OCCURS 6 TIMES.
   10 A                 PIC X(1) VALUE '?'.
   10 B                 PIC X(1) VALUE '%'.
   10 N                 PIC 9(2) VALUE 10.
```

In this example, all six “A” items will be initialized to “?”, all six “B” items will be initialized to “%” and all six “N” items will be initialized to 10. It’s not clear exactly how many times this sort of initialization will be useful, but it’s there if you need it.

## 5.4. Condition Names

Figure 5-12 - Level-88 Condition Name Description Syntax



Condition names are Boolean (i.e. “TRUE” / “FALSE”) data items.

1. Condition names are always defined subordinate to another data item. That data item must be an elementary item.
2. Condition names do not occupy any storage.
3. The VALUE(s) specified for the condition name specify the specific values and/or ranges of values of the parent elementary data item that will cause the condition name to have a value of TRUE.



4. The optional FALSE clause defines an explicit value that will be assigned to the parent elementary data item should the SET statement ever be used to set the condition-name-1 to FALSE. See section [6.40.6](#) for more information on how the SET statement can be used to specify the TRUE/FALSE value of a condition name.
5. See section [6.1.4.2.1](#) for a general discussion of condition names.

## 5.5. Constant Descriptions

Figure 5-13 - Level-78 Constant Description Syntax

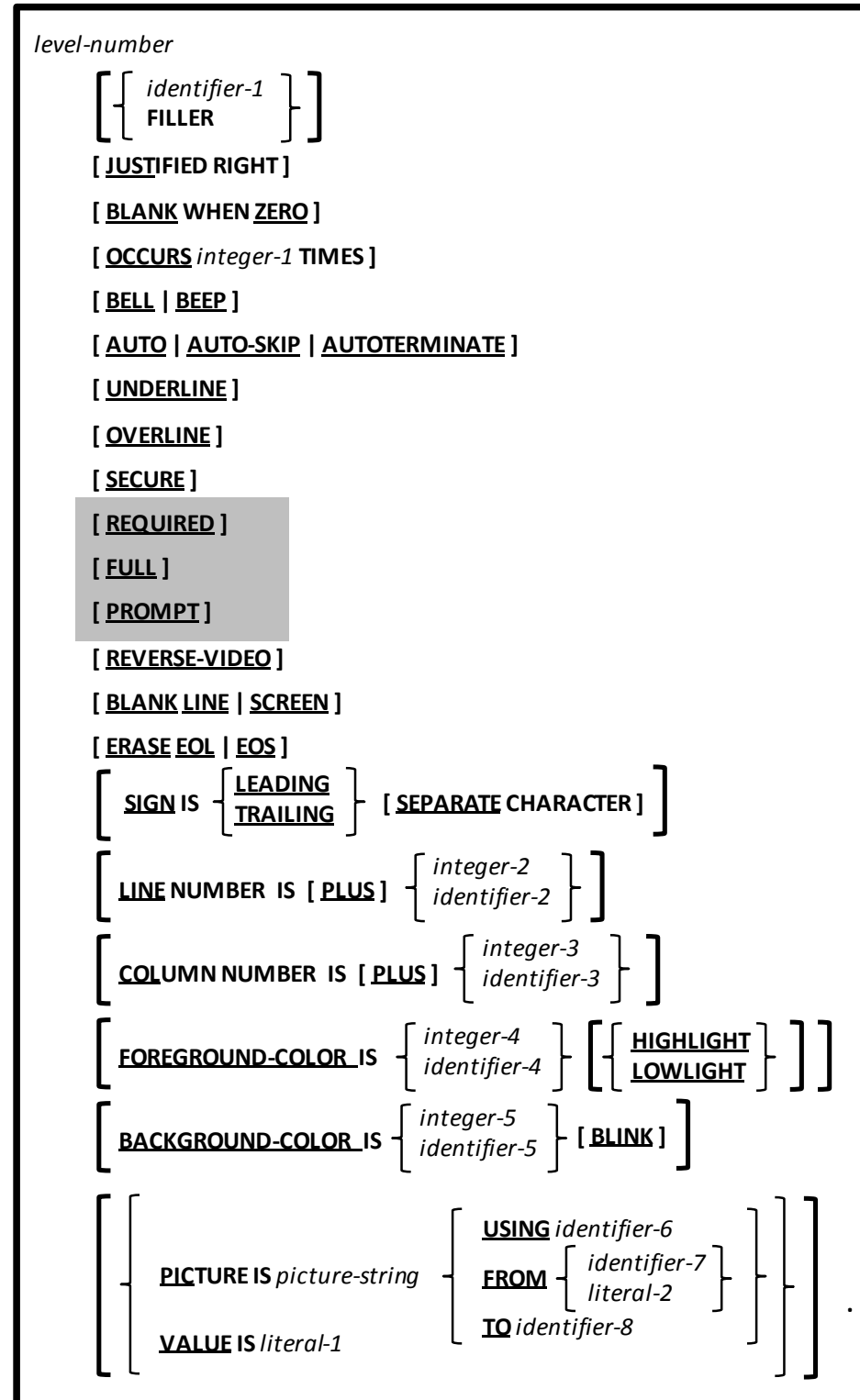
<pre> 78 identifier-1 <b>VALUE IS</b> literal-1 .  01 identifier-2 <b>CONSTANT</b> [ <b>IS GLOBAL</b> ] AS {     literal-2     <b>LENGTH OF</b> identifier-3     <b>BYTE-LENGTH OF</b> identifier-4 }</pre>
---

Data descriptions of these forms do not actually allocate any storage, but instead are a means of associating a name with an alphanumeric or numeric literal.

1. The two forms are essentially identical when defining a value of a literal. Defining a constant whose value is the length of another item is – as you can see – possible only using the “01 CONSTANT” form.
2. The GLOBAL clause, while recognized syntactically, is currently unsupported by OpenCOBOL and should generate a compiler warning if used. As of the Feb 06 2009 packaging of OpenCOBOL 1.1, however, its use will actually abort the compiler.

## 5.6. Screen Descriptions

Figure 5-14 - SCREEN SECTION Data Item Description Syntax



The syntax skeleton shown here describes how data items are defined in the SCREEN SECTION.

These data items are used via special forms of the ACCEPT (section 0) and DISPLAY (section 6.15.4) verbs to create TUI (that is “Textual User Interface” programs).

1. Data items with level numbers 66, 78 and 88 may be used in the SCREEN SECTION; they have the same syntax, rules and usage as they do in the other DATA DIVISION sections.
2. Use the BELL or BEEP clauses (they are synonymous) to cause an audible tone to occur when the screen item is DISPLAYed.

3. The AUTO clause (the three forms are all equivalent) will cause the cursor to automatically advance to the next input-enabled field if the field having the AUTO clause is completely filled.
4. The UNDERLINE and OVERLINE clauses are essentially non-functional on Windows systems as those video attributes are not currently supported by the Windows console window API. The UNDERLINE clause will have an effect, however, as it will make the foreground color of the field blue regardless of the value specified (or implied) by the FOREGROUND-COLOR attribute. Whether or not these clauses operate on UNIX systems will depend upon the video attributes of the terminal output device being used.
5. The SECURE attribute may only be used on a field allowing data entry (USING or TO). This attribute will cause all data entered into the field to appear as asterisks.
6. The REQUIRED and FULL attributes, while syntactically acceptable, are non-functional.
7. The PROMPT attribute is superfluous in OpenCOBOL as its behavior is assumed on all input fields.<sup>15</sup>
8. The REVERSE-VIDEO attribute reverses the meaning of the specified or implied FOREGROUND-COLOR and BACKGROUND-COLOR attributes.
9. The BLANK clause will blank-out the screen or line from the point indicated by any LINE and/or COLUMN clause on the data item. In addition, the console window foreground and background colors will be set to whatever is specified on the item. Use of this clause ANYWHERE within an O1-level item (or its subordinate items, if any) will cause ALL displayed fields ANYWHERE within that O1-level item (or its subordinate items) to be invisible.
10. The ERASE clause will erase the remainder of the console window's current line (EOL) or the console window screen (EOS) starting at the end of the field having the ERASE clause to be erased and to have its colors set to the foreground and background colors in effect for the field containing the ERASE clause.
11. Without LINE or COLUMN clauses, SCREEN SECTION fields will display on the console window beginning at whatever line/column coordinate is stated or implied by the ACCEPT or DISPLAY statement that presents the screen item. After a field is presented to the console window, the next field will be presented immediately following that field.

The LINE and COLUMN clauses provide a means of explicitly stating where a field should be presented on the console window. Coordinates may be stated on an absolute basis (i.e. "LINE 1 COLUMN 5") or on a relative basis based upon the end of the previously-presented field (i.e. "LINE PLUS 2 COLUMN PLUS 1"). Identifiers or literals may be used to define the absolute or relative position. If identifiers are used, they must be PIC 9 items without editing symbols (any numeric USAGE is allowed except for COMPUTATIONAL-1 or COMPUTATIONAL-2. Note that either of these floating-point USAGE specifications will be accepted, but will produce unpredictable results.

Fields do not need to be defined in LINE/COLUMN sequence of their presentation, unless of course you're relying on the implicit positioning of screen items by not using LINE and COLUMN.

The TAB and BACK-TAB (Shift-TAB) keys will position the cursor from field to field in the line/column sequence in which the fields occur on the console window, regardless of the sequence in which they were defined in the SCREEN SECTION.

You may abbreviate COLUMN as COL, if you wish.

---

<sup>15</sup> The PROMPT attribute is used to specify that empty input fields will be marked by a non-blank character to ensure they're visible. This functionality is ALWAYS on for all editable screen fields in OpenCOBOL (an underscore character is used).

12. The FOREGROUND-COLOR and BACKGROUND-COLOR clauses are used to specify the color of text (foreground) or the screen (background). You specify colors by number (0-7) according to the following:

Figure 5-15 - Screen Color Numbers

Integer	Color
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Yellow
7	White

13. The HIGHLIGHT and LOWLIGHT options control the intensity of text (foreground). This is intended to provide a three-level intensity scheme (LOWLIGHT, nothing-specified, HIGHLIGHT), but the Windows console only supports two-levels, so LOWLIGHT is the same as leaving this clause off altogether. Using this modifier to the FOREGROUND-COLOR attribute, you can actually have sixteen text colors, not just eight, as follows:

Figure 5-16 - LOWLIGHT / HIGHLIGHT Effect on Screen Colors

FOREGROUND-COLOR <i>integer</i>	LOWLIGHT	HIGHLIGHT
0	Black	Dark Grey
1	Dark Blue / Indigo	Bright Blue
2	Dark Green	Bright Green
3	Dark Cyan	Bright Cyan
4	Dark Red	Bright Red
5	Dark Magenta	Bright Magenta
6	Gold / Brown	Yellow
7	Light Grey	White

14. The BLINK attribute modifies the visual appearance of the BACKGROUND-COLOR specification. The Windows console does not support blinking, so the visual effect of BLINK in the Windows version of OpenCOBOL is to provide the same sixteen colors to the BACKGROUND-COLOR palette as are possible with FOREGROUND-COLOR combined with LOWLIGHT/HIGHLIGHT.
15. Foreground and background color attributes are inheritable from other fields. They are not inherited from the prior field encountered but rather from parent data items (data items with numerically lower level numbers). Observe the following...

```

78 B|ack          VALUE 0.
78 B|ue          VALUE 1.
78 G|reen        VALUE 2.
78 W|hite        VALUE 7.
...
02 XYZ BACKGROUND-COLOR B|ack FOREGROUND-COLOR G|reen ...
05 ABC BACKGROUND-COLOR B|ue FOREGROUND-COLOR W|hite ...
05 DEF (no BACKGROUND-COLOR or FOREGROUND-COLOR specified) ...

```

The color of field DEF will be Green-on-White (inherited from XYZ)

16. The VALUE clause is used to define fixed text that cannot be changed.
17. The FROM clause is used to define a field whose contents should come from the specified literal or identifier.
18. The TO clause is used to define a data-entry field with no initial value; when a value is entered, it will be saved to the specified identifier.
19. The USING clause is a combination of "FROM *identifier*" and "TO *identifier*".



## 6. PROCEDURE DIVISION

### 6.1. General PROCEDURE DIVISION Components

#### 6.1.1. Table References

COBOL uses parenthesis to specify the subscripts used to reference table entries (tables in COBOL are what other programming languages refer to as arrays).

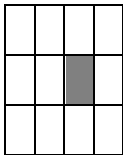
For example, observe the following data structure which simulates a 4 column by 3 row grid of characters:

```

01 GRID.
   05 GRID-ROW OCCURS 3 TIMES.
      10 GRID-COLUMN OCCURS 4 TIMES.
         15 GRID-CHARACTER          PIC X(1).

```

A reference to the GRID-CHARACTER shaded in the following diagram:



Would be coded as:

```
GRID-CHARACTER (2, 3)
```

Subscripts may be specified as numeric (integer) literals, PIC 9 (integer) data items, USAGE INDEX data items or arithmetic expressions resulting in an integer value involving any combination of these. The ability to use full arithmetic expressions as table (array) subscripts, while common in many languages, is rare in the COBOL universe. See section [6.1.4.1](#) for a discussion of arithmetic expressions.

#### 6.1.2. Qualification of Data Names

COBOL allows data names to be duplicated within a program, provided references to those data names may be made in such a manner as to make those references unique through a process known as *qualification*.

To see qualification at work, observe the following segments of two data records defined in a COBOL program:

```

01 EMPLOYEE.
   05 MAILING-ADDRESS.
      10 STREET          PIC X(35).
      10 CITY            PIC X(15).
      10 STATE          PIC X(2).
      10 ZIP-CODE.
         15 ZIP-CODE-5  PIC 9(5).
         15 FILLER      PIC X(4).
01 CUSTOMER.
   05 MAILING-ADDRESS.
      10 STREET          PIC X(35).
      10 CITY            PIC X(15).
      10 STATE          PIC X(2).
      10 ZIP-CODE.
         15 ZIP-CODE-5  PIC 9(5).
         15 FILLER      PIC X(4).

```

Now, let's deal with the problem of setting the CITY portion of an EMPLOYEE's MAILING-ADDRESS to "Philadelphia". Clearly, the following cannot work because the compiler will be unable to determine which of the two CITY fields you are referring to:

```
MOVE "Philadelphia" TO CITY.
```

We could qualify the reference to CITY as follows, in an attempt to correct the problem:

```
MOVE "Philadelphia" TO CITY OF MAILING-ADDRESS.
```

Unfortunately that too is insufficient because it is still insufficient to identify specifically which CITY is being referenced. To truly identify which specific CITY you want, you'd have to code the following:

```
MOVE "Philadelphia" TO CITY OF MAILING-ADDRESS OF EMPLOYEE.
```

Now there can be no confusion as to which CITY is being changed. Fortunately, you don't need to be so specific; COBOL allows intermediate qualification levels to be omitted. This allows you to specify:

```
MOVE "Philadelphia" TO CITY OF EMPLOYEE.
```

If you need to qualify a reference to a table, do so as follows:

```
identifier-1 OF identifier-2 ( subscript ... )
```

The reserved word "IN" may be used in lieu of "OF".

### 6.1.3. Reference Modifiers

Figure 6-1 - Reference Modifier Syntax

$\left[ \begin{array}{l} \text{identifier-1 [ OF identifier-2 ] [ ( subscript ... ) ] } \\ \text{intrinsic-function-reference} \end{array} \right] (\text{start} : [ \text{length} ])$
--

The COBOL '85 standard introduced the concept of a *reference modifier* to facilitate references to only a portion of a data item; OpenCOBOL fully supports reference modification.

The *start* value indicates the starting character position being referenced (character position values start with 1, not 0 as is the case in some programming languages) and *length* specifies how many characters are wanted. If no *length* is specified, a value equivalent to the remaining character positions from *start* to the end will be assumed.

Here are a few examples:

<b>CUSTOMER-LAST-NAME (1:3)</b>	references the first three characters of CUSTOMER-LAST-NAME
<b>CUSTOMER-LAST-NAME (4:)</b>	references all character positions of CUSTOMER-LAST-NAME from the fourth onward.
<b>FUNCTION CURRENT-DATE (5:2)</b>	references the current month (see the documentation of the CURRENT-DATE intrinsic function on page <a href="#">6-13</a> for details)
<b>Hex-Digits (Nibble + 1:1)</b>	Assuming that "Nibble" is a numeric data item with a value in the range 0-15, and Hex-Digits is a PIC X(16) item with a value of "0123456789ABCDEF", this converts that numeric value to a hexadecimal digit.
<b>Hex-Digits (Nibble + 1:)</b>	Does the same as the above – if you leave out the length, 1 is assumed; YOU STILL NEED THE ":" CHARACTER THOUGH.
<b>Array-Element (6) (7:5)</b>	References 5 characters in the 6 <sup>th</sup> occurrence of Array-Element, starting at character position 7.

Reference modification may be used anywhere an identifier is legal, including serving as the receiving field of statements like MOVE, STRING and ACCEPT, to name a few.

### 6.1.4. Expressions

OpenCOBOL, like other COBOL implementations, supports two basic types of expressions:

- *Arithmetic expressions*, which calculate a numeric result
- *Conditional Expressions*, which calculate a TRUE or FALSE value

Unlike other programming languages, which allow arithmetic values such as 0 and -1 to represent FALSE and TRUE, respectively, COBOL treats logical TRUE/FALSE values as something different from 0/-1. OpenCOBOL adheres to this policy.

### 6.1.4.1. Arithmetic Expressions

Arithmetic expressions are formed using following operators. In complex expressions composed of multiple operators, a precedence of operation applies whereby those operations having a higher precedence are computed first before operations with a lower precedence.

Precedence / Operation	Discussion
<p>Figure 6-2 – Unary - Operator Syntax</p> <p>Precedence: 1<sup>ST</sup> (Highest)</p> $- \left[ \begin{array}{l} \textit{numeric-literal-1} \\ \textit{identifier-1} \\ (\textit{arith-expr-1}) \end{array} \right]$	<p>The unary - operator returns the arithmetic negation of its single argument, effectively returning as its value the product of its argument and -1.</p>
<p>Figure 6-3 – Unary + Operator Syntax</p> <p>Precedence: 1<sup>ST</sup> (Highest)</p> $+ \left[ \begin{array}{l} \textit{numeric-literal-1} \\ \textit{identifier-1} \\ (\textit{arith-expr-1}) \end{array} \right]$	<p>The unary + operator returns the value of its single argument, effectively returning as its value the product of its argument and +1.</p>
<p>Figure 6-4 - Exponentiation Operator Syntax</p> <p>Precedence: 2<sup>nd</sup></p> $\left[ \begin{array}{l} \textit{numeric-literal-1} \\ \textit{identifier-1} \\ (\textit{arith-expr-1}) \end{array} \right] ** \left[ \begin{array}{l} \textit{numeric-literal-2} \\ \textit{identifier-2} \\ (\textit{arith-expr-2}) \end{array} \right]$	<p>The value of the left-hand argument raised to the power indicated by the right-hand argument is computed. OpenCOBOL allows the “^” symbol to be used in lieu of the “**” symbol.</p>
<p>Figure 6-5 - Exponentiation Operator Syntax</p> <p>Precedence: 3<sup>rd</sup></p> $\left[ \begin{array}{l} \textit{numeric-literal-1} \\ \textit{identifier-1} \\ (\textit{arith-expr-1}) \end{array} \right] * \left[ \begin{array}{l} \textit{numeric-literal-2} \\ \textit{identifier-2} \\ (\textit{arith-expr-2}) \end{array} \right]$	<p>The product of the left-hand argument and the right-hand argument is computed.</p>
<p>Figure 6-6 - Division Operator Syntax</p> <p>Precedence: 3<sup>rd</sup></p> $\left[ \begin{array}{l} \textit{numeric-literal-1} \\ \textit{identifier-1} \\ (\textit{arith-expr-1}) \end{array} \right] / \left[ \begin{array}{l} \textit{numeric-literal-2} \\ \textit{identifier-2} \\ (\textit{arith-expr-2}) \end{array} \right]$	<p>The value of the left-hand argument divided by the right-hand argument is computed.</p>



Precedence / Operation	Discussion
<p data-bbox="354 216 683 237">Figure 6-7 - Addition Operator Syntax</p> <p data-bbox="383 254 654 285">Precedence: 4<sup>th</sup> (Lowest)</p> <div data-bbox="177 310 857 478" style="border: 2px solid black; padding: 10px; display: inline-block;"> <math display="block">\left[ \begin{array}{l} \textit{numeric-literal-1} \\ \textit{identifier-1} \\ (\textit{arith-expr-1}) \end{array} \right] + \left[ \begin{array}{l} \textit{numeric-literal-2} \\ \textit{identifier-2} \\ (\textit{arith-expr-2}) \end{array} \right]</math> </div>	<p data-bbox="891 327 1430 390">The sum of the left-hand argument and the right-hand argument is computed.</p>
<p data-bbox="342 489 695 510">Figure 6-8 - Subtraction Operator Syntax</p> <p data-bbox="383 527 654 558">Precedence: 4<sup>th</sup> (Lowest)</p> <div data-bbox="177 583 857 751" style="border: 2px solid black; padding: 10px; display: inline-block;"> <math display="block">\left[ \begin{array}{l} \textit{numeric-literal-1} \\ \textit{identifier-1} \\ (\textit{arith-expr-1}) \end{array} \right] - \left[ \begin{array}{l} \textit{numeric-literal-2} \\ \textit{identifier-2} \\ (\textit{arith-expr-2}) \end{array} \right]</math> </div>	<p data-bbox="891 600 1424 663">The value of the right-hand argument subtracted from the left-hand argument is computed.</p>

The COBOL standards require the use of at least one space before and after the exponentiation, multiplication, division, addition and subtraction operators. This is the best policy to follow when coding expressions as it ensures compatibility with other COBOL implementations and avoids the need to deal with the following special rules which define the circumstances under which leading and/or trailing spaces may be omitted:

1. OpenCOBOL does not actually require leading or trailing spaces around the exponentiation, multiplication or division operators.
2. The ADDITION operator must be followed by a space if it is followed by an unsigned numeric literal. Failure to do so (for example: "4+3") will result in an "Invalid Expression" error because the compiler will treat the "+" as an attempt to specify a signed numeric literal, leaving the expression with no operator. In any other circumstances, the leading and trailing spaces around the ADDITION operator are optional.
3. The SUBTRACTION operator must be followed by a space if it is followed by an unsigned numeric literal. Failure to do so (for example: "4-3") will result in an "Invalid Expression" error because the compiler will treat the "-" as an attempt to specify a signed numeric literal, leaving the expression with no operator.
4. The SUBTRACTION operator must have a leading and/or trailing space if neither argument is a parenthesized expression. Failure to include one or the other space ("3-Arg", "Arga-Argb", ...) will cause the compiler to look for a defined reserved word or user-defined name that (hopefully) doesn't exist – generating a "'identifier' Undefined" error. If you are really unlucky, it actually WILL find such an identifier, which is almost certain to cause runtime problems!
5. If the argument of a UNARY PLUS operator is an unsigned numeric literal, the unary plus operator must be followed by a space to avoid being treated as part of the numeric literal (thus making it a signed positive numeric literal).
6. If the argument of a UNARY NEGATION operator is an unsigned numeric literal, the unary negation operator must be followed by a space to avoid being treated as part of the numeric literal (thus making it a signed negative numeric literal).

Here are some examples of arithmetic expressions (all of which involve numeric literals, to simplify the discussion).

Expression	Result	Notes
$3 * 4 + 1$	13	* has precedence over +
$2 \wedge 3 * 4 - 10$	22	$2^3$ is 8, times 4 is 32, minus 10 is 22.
$2 ** 3 * 4 - 10$	22	Same as the above – OpenCOBOL allows either “^” or “**” to be used as the exponentiation operator.
$3 * (4 + 1)$	15	Parenthesis provide for a recursive application of the arithmetic expression rules, allowing arithmetic expressions to become components within other, more complex, arithmetic expressions.
$5 / 2.5 + 7 * 2 - 1.15$	15.35	Integer and non-integer operands may be freely intermixed

Of course, arithmetic expression operands may be numeric data items (any USAGE except DISPLAY, POINTER or PROGRAM POINTER) as well as numeric literals.

### 6.1.4.2. Conditional Expressions

Conditional expressions are expressions which identify the conditions under which a program may make a decision about processing to be performed. As such, conditional expressions produce a value of TRUE or FALSE.

There are seven types of conditional expressions, as follows, in increasing order of complexity.

#### 6.1.4.2.1. Condition Names (Level-88 Items)

These are the simplest of all conditions. Observe the following code:

```

05  SHIRT-SIZE                PIC 99V9.
   88  LILLIPUTIAN            VALUE 0 THRU 12.5
   88  XS                     VALUE 13 THRU 13.5.
   88  S                      VALUE 14, 14.5.
   88  M                      VALUE 15, 15.5.
   88  L                      VALUE 16, 16.5.
   88  XL                    VALUE 17, 17.5.
   88  XXL                   VALUE 18, 18.5.
   88  HUMUNGOUS            VALUE 19 THRU 99.9.

```

The condition names “LILLIPUTIAN”, “XS”, “S”, “M”, “L”, “XL”, “XXL” and “HUMONGOUS” will have TRUE or FALSE values based upon the values within their parent data item (SHIRT-SIZE). So, a program wanting to test whether or not the current SHIRT-SIZE value can be classified as “XL” could have that decision coded as a combined condition (the most complex type of conditional expression), as follows:

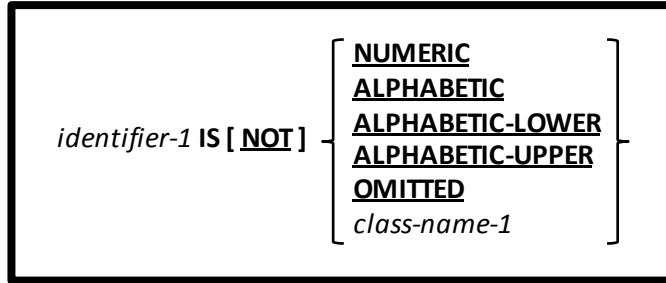
```
IF SHIRT-SIZE = 17 OR SHIRT-SIZE = 17.5
```

Or it could utilize the condition name XL as follows:

```
IF XL
```

#### 6.1.4.2.2. Class Conditions

Figure 6-9 - Class Condition Syntax



Class conditions evaluate the type of data that is currently stored in a data item.

1. The NUMERIC class test considers only the characters “0”, “1”, ..., “9” to be numeric; only a data item containing nothing but digits will pass a IS NUMERIC class test. Spaces, decimal points, commas, currency signs, plus signs, minus signs and any other characters except the digit characters will all fail “IS NUMERIC” class tests.
2. The ALPHABETIC class test considers only upper-case letters, lower-case letters and SPACES to be alphabetic in nature.
3. The ALPHABETIC-LOWER and ALPHABETIC-UPPER class conditions consider only spaces and the respective type of letters to be acceptable in order to pass such a class test.
4. Only data items whose USAGE is either explicitly or implicitly defined as DISPLAY may be used in NUMERIC or any of the ALPHABETIC class conditions.
5. Some COBOL implementations disallow the use of group items or PIC A items with NUMERIC class conditions and the use of PIC 9 items with ALPHABETIC class conditions. OpenCOBOL has no such restrictions.
6. The OMITTED class condition is used when it is necessary for a subroutine to determine whether or not a particular argument was passed to the subroutine. In such class conditions, *identifier-1* must be a LINKAGE SECTION item defined on the USING clause of the subprograms “PROCEDURE DIVISION” header. See section [6.8](#) for the method to use when omitting arguments from a CALL to a subprogram.
7. The *class-name-1* option allows you to test for a user-defined class. Here’s an example. First, assume the following SPECIAL-NAMES definition of the user-defined class “Hexadecimal”:

**SPECIAL-NAMES.**

**CLASS Hexadecimal IS ‘0’ THRU ‘9’, ‘A’ THRU ‘F’, ‘a’ THRU ‘f’.**

Now observe the following code, which will execute the **150-Process-Hex-Value** procedure if **Entered-Value** contains nothing but valid hexadecimal digits:

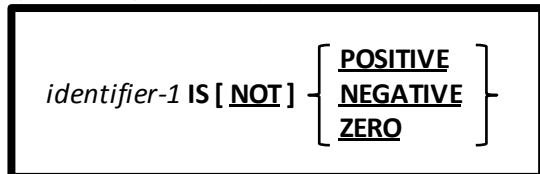
```

IF Entered-Value IS Hexadecimal
  PERFORM 150-Process-Hex-Value
END-IF

```

### 6.1.4.2.3. Sign Conditions

Figure 6-10 - Sign Condition Syntax

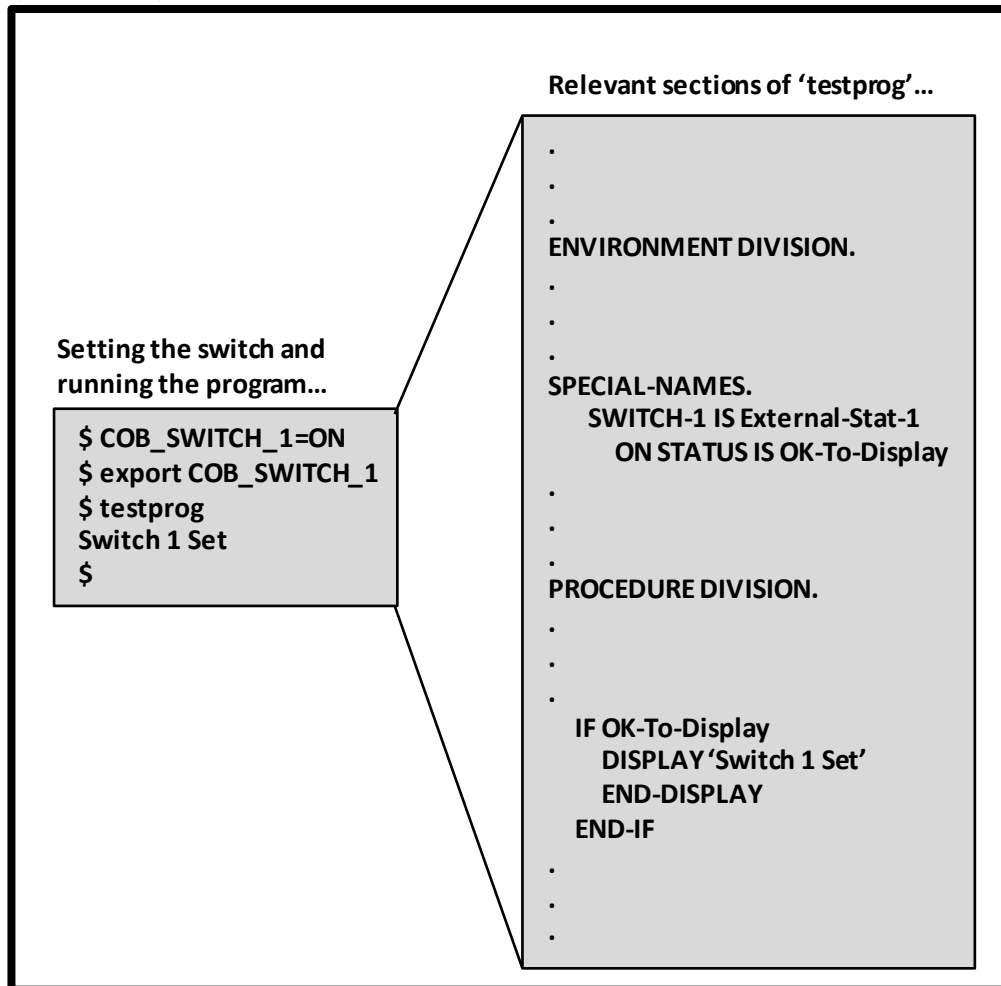


Sign conditions evaluate the numeric state of a PIC 9 data item.

1. Only data items defined with some sort of numeric USAGE/PICTURE can be used for this type of class condition.
2. A POSITIVE or NEGATIVE class condition will be TRUE only if the value of *identifier-1* is strictly greater than or less than zero, respectively. A ZERO class condition can be passed only if the value of *identifier-1* is exactly zero.

## 6.1.4.2.4. Switch-Status Conditions

Figure 6-11 - Using Switch Conditions

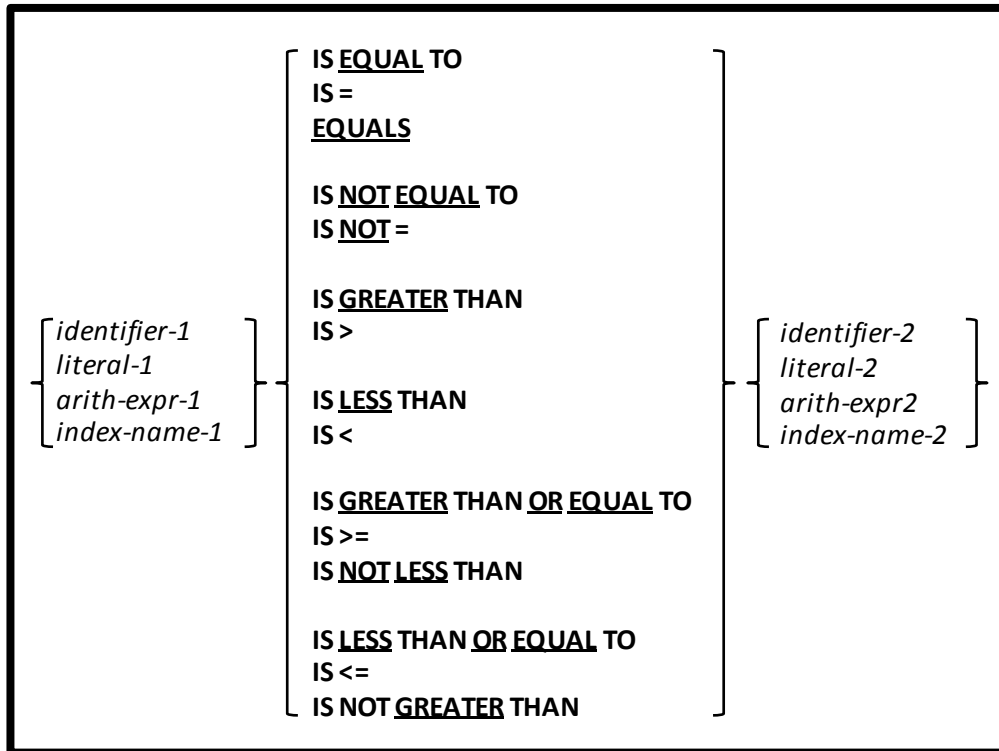


In the SPECIAL-NAMES paragraph (see section [4.1.4](#)), an external switch name can be associated with one or more condition names. These condition names may then be used to test the ON/OFF status of the external switch.

An example is shown to the left.

### 6.1.4.2.5. Relation Conditions

Figure 6-12 - Relation Condition Syntax

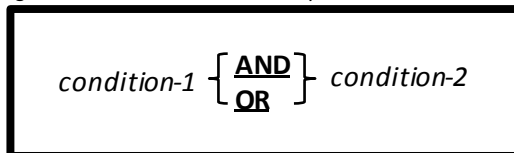


These conditions evaluate how two different values “relate” to each other.

1. When comparing one numeric value to another, the USAGE and number of significant digits in either value are irrelevant as the comparison is performed using the actual algebraic values.
2. When comparing strings, the comparison is made based upon the program’s collating sequence (see section [4.1.2](#)). When the two string arguments are of unequal length, the shorter is assumed to be padded (on the right) with a sufficient number of SPACES as to make the two strings of equal length. String comparisons take place on a corresponding character-by-character basis until an unequal pair of characters is found. At that point, the relative position of where each character in the pair falls in the collating sequence will determine which is greater (or less) than the other.

### 6.1.4.2.6. Combined Conditions

Figure 6-13 - Combined Condition Syntax



A combined condition is one that computes a TRUE/FALSE value from the TRUE/FALSE values of two other conditions (which could – themselves – be combined conditions).

1. If either condition has a value of TRUE, the result of ORing the two together will result in a value of TRUE. Only when ORing two FALSE conditions will a result of FALSE occur.
2. In order for AND to yield a value of TRUE, both conditions must have a value of TRUE. In all other circumstances, AND produces a FALSE value.
3. When chaining multiple, similar conditions together with the same operator (OR/AND), and left or right arguments having common operators and subjects, it is possible to abbreviate the program code. For example:

```
IF ACCOUNT-STATUS = 1 OR ACCOUNT-STATUS = 2 OR ACCOUNT-STATUS = 7
```

Could be abbreviated as:

```
IF ACCOUNT-STATUS = 1 OR 2 OR 7
```

4. Just as multiplication takes precedence over addition in arithmetic expressions, so does AND take precedence over OR in combined conditions. Use parenthesis to change this precedence, if necessary. For example:

**FALSE OR TRUE AND TRUE** evaluates to TRUE  
**(FALSE OR FALSE) AND TRUE** evaluates to FALSE  
**FALSE OR (FALSE AND TRUE)** evaluates to TRUE

### 6.1.4.2.7. Negated Conditions

Figure 6-14 - Negated Condition Syntax

**NOT** *condition*

A condition may be negated by prefixing it with the NOT operator.

1. The NOT operator has the highest precedence of all logical operators, just as a unary minus sign (which negates a numeric value) is the highest precedence arithmetic operator.
2. Parenthesis must be used to explicitly signify the sequence in which conditions are evaluated and processed if the default precedence isn't desired. For example:

**NOT TRUE AND FALSE AND NOT FALSE** evaluates to **FALSE AND FALSE AND TRUE** which evaluates to **FALSE**  
**NOT (TRUE AND FALSE AND NOT FALSE)** evaluates to **NOT (FALSE)** which evaluates to **TRUE**  
**NOT TRUE AND (FALSE AND NOT FALSE)** evaluates to **FALSE AND (FALSE AND TRUE)** which evaluates to **FALSE**

### 6.1.5. Use of Periods (.)

All COBOL implementations distinguish between sentences and statements in the PROCEDURE DIVISION. A statement is a single executable COBOL instruction. For example, these are all statements:

```
MOVE SPACES TO Employee-Address
ADD 1 TO Record-Counter
DISPLAY "Record-Counter=" Record-Counter
```

Some COBOL statements have a "scope of applicability" associated with them where one or more other statements can be considered to be part of or related to the statement in question. An example of such a situation might be the following, where the interest on a loan is being calculated and displayed - 4% interest if the loan balance is under \$10000 and 4.5% otherwise:

```
IF Loan-Balance < 10000
  MULTIPLY Loan-Balance BY 0.04 GIVING Interest
ELSE
  MULTIPLY Loan-Balance BY 0.045 GIVING Interest
DISPLAY "Interest Amount = " Interest
```

In this example, the "IF" statement actually has a scope that can include two sets of associated statements – one set to be executed when the "IF" condition is TRUE and another if it is FALSE.

Unfortunately, there's a problem with the above. A human being looking at that code will probably understand that the DISPLAY statement, because of its lack of indentation, is to be executed regardless of the TRUE/FALSE value of the "IF" condition. Unfortunately, the OpenCOBOL compiler (or any other COBOL compiler for that matter) won't see it that way because it really couldn't care less what sort of indentation, if any, is used. In fact, the OpenCOBOL compiler would be just as happy to see the code written like this:

```
IF Loan-Balance < 10000 MULTIPLY Loan-balance BY 0.04
  GIVING Interest ELSE MULTIPLY Loan-Balance BY 0.045
  GIVING Interest DISPLAY "Interest Amount = " Interest
```

So how then do we inform the compiler that the DISPLAY statement is outside the scope of the "IF"?

That's where *sentences* come in.

A COBOL *sentence* is defined as any arbitrarily long sequence of statements, followed by a period (.) character. The period character is what terminates the scope of a set of statements. Therefore, our example needs to be coded like this:

```
IF Loan-Balance < 10000
    MULTIPLY Loan-Balance BY 0.04 GIVING Interest
ELSE
    MULTIPLY Loan-Balance BY 0.045 GIVING Interest.
DISPLAY "Interest Amount = " Interest
```

See the period at the end of the second MULTIPLY? That is what terminates the scope of the “IF”, thus making the DISPLAY something that will be executed regardless of how the “Loan-Balance < 10000” test evaluated.

### 6.1.6. Use of “VERB” / “END-VERB” Constructs

Prior to the 1985 COBOL standard, using a period character was the only way to signal the end of a statement’s scope. Unfortunately, this caused some problems.

Take a look at this code:

```
IF A = 1
    IF B = 1
        DISPLAY "A & B = 1"
ELSE
    IF B = 1
        DISPLAY "A NOT = 1 BUT B = 1"
    ELSE
        DISPLAY "NEITHER A NOR B = 1".
```

The problem with this code is the fact that the ELSE will be associated with the “IF B = 1” statement, not the “IF A = 1” statement (remember – no COBOL compiler cares about how you indent your code). This sort of problem led to the following band-aid solution<sup>16</sup> added to the COBOL language:

```
IF A = 1
    IF B = 1
        DISPLAY "A & B = 1"
    ELSE
        NEXT SENTENCE
ELSE
    IF B = 1
        DISPLAY "A NOT = 1 BUT B = 1"
    ELSE
        DISPLAY "NEITHER A NOR B = 1".
```

The NEXT SENTENCE statement (see section [6.31](#)) informs COBOL that if the “B = 1” condition is false, control should fall into the first statement that follows the next period.

With the 1985 standard for COBOL, a much more elegant solution was introduced. Those COBOL verbs (statements) that needed such a thing were allowed to use an “END-verb” construct to end their scope without disrupting the scope of any statements whose scope they might have been in. Any COBOL 85 compiler would have allowed the following solution to our problem:

```
IF A = 1
    IF B = 1
        DISPLAY "A & B = 1"
    END-IF
ELSE
    IF B = 1
        DISPLAY "A NOT = 1 BUT B = 1"
    ELSE
        DISPLAY "NEITHER A NOR B = 1".
```

<sup>16</sup> Yes, I realize you could have changed the code to “IF A = 1 AND B = 1”, but that wouldn’t have allowed me to make my case here

This new facility made the period almost obsolete, as our program segment would probably be coded like this today:

```

IF A = 1
  IF B = 1
    DISPLAY "A & B = 1"
  END-IF
ELSE
  IF B = 1
    DISPLAY "A NOT = 1 BUT B = 1"
  ELSE
    DISPLAY "NEITHER A NOR B = 1"
  END-IF
END-IF

```

COBOL (OpenCOBOL included) still requires that each PROCEDURE DIVISION paragraph contain at least once *sentence* if there is any executable code in that paragraph, but a popular coding standard is now to simply code a single period right before the end of each paragraph. Check out the “OCic” sample program in section [8.3](#) and you’ll see how that would be done.

The standard for the COBOL language shows the various “END-verb” specifications to be optional because using a period as a scope-terminator remains legal. Some statements have an “END-verb” scope-terminator defined for them that they don’t appear to need.<sup>17</sup>

If you will be porting existing code over to OpenCOBOL, you’ll find it an accommodating facility capable of conforming to language and coding standards that code is likely to use. If you are creating new OpenCOBOL programs, however, I would strongly counsel you to use the “END-verb” structures religiously in those programs.

### 6.1.7. Intrinsic Functions

OpenCOBOL supports a variety of “intrinsic functions” that may be used anywhere in the PROCEDURE DIVISION where a literal is allowed. For example:

```
MOVE FUNCTION LENGTH(Employee-Last-Name) TO Employee-LN-Len.
```

Note how the word “FUNCTION” is part of the syntax when you use an intrinsic function. You can use intrinsic functions without having to include the reserved word FUNCTION via settings in the REPOSITORY paragraph of the CONFIGURATION SECTION. See section [4.1.3](#) for more information.

The following intrinsic functions, known to other “dialects” of COBOL, are defined to OpenCOBOL as reserved words but are not otherwise implemented currently. Any attempts to use these functions will result in a compile-time error message.

BOOLEAN-OF-INTEGERS	HIGHEST-ALGEBRAIC	NUMVAL-F
CHAR-NATIONAL	INTEGER-OF-BOOLEAN	STANDARD-COMPARE
DISPLAY-OF	LOCALE-COMPARE	TEST-NUMVAL
EXCEPTION-FILE-N	LOWEST-ALGEBRAIC	TEST-NUMVAL-C
EXCEPTION-LOCATION-N	NATIONAL-OF	TEST-NUMVAL-F

The supported intrinsic functions are listed in the following table, along with their syntax and usage notes. Remember that the keyword FUNCTION is required to be specified immediately before the function name in order to use the function, unless otherwise indicated via the REPOSITORY paragraph of the CONFIGURATION SECTION.

#### 6.1.7.1. ABS(number)

Determines and returns the absolute value of the *number* (a PIC 9 item or numeric literal ) supplied as an argument.

<sup>17</sup> STRING (section [6.44](#)) and UNSTRING (section [6.50](#)), for example – could it be there are plans in the works for a future standard to introduce an option to such statements that would need a scope-terminator?



### 6.1.7.2. ACOS(*angle*)

The ACOS function determines and returns the trigonometric arc-cosine, or inverse cosine, of the *angle* (a PIC 9 item or numeric literal) supplied as an argument.

### 6.1.7.3. ANNUITY(*interest-rate, number-of-periods*)

This function returns a numeric value approximating the ratio of an annuity paid at the specified *interest-rate* (PIC 9 items or numeric literal) for each of the specified *number-of-periods* (PIC 9 items or numeric literal).

The interest-rate is the rate of interest paid at each payment. If you only have an annual interest rate and you wish to compute annuity payments for monthly payments, divide the annual interest rate by 12 and use that value for *interest-rate* on this function.

Multiply this result times the desired principal amount to determine the amount of each period's payment.

A note for the financially challenged: an annuity is basically a reverse loan; an accountant would take the result of this function multiplied by -1 to compute a loan payment you are making.

### 6.1.7.4. ASIN(*number*)

The ASIN function determines and returns the trigonometric arc-sine, or inverse sine, of the *angle* (a PIC 9 item or numeric literal) supplied as an argument.

### 6.1.7.5. ATAN(*number*)

Use this function to determine and return the trigonometric arc-tangent, or inverse tangent, of the *angle* (a PIC 9 item or numeric literal) supplied as an argument.

### 6.1.7.6. BYTE-LENGTH(*string*)

BYTE-length returns the length – in bytes – of the specified string (a group item, USAGE DISPLAY elementary item or alphanumeric literal). This intrinsic function is identical to the LENGTH function.

### 6.1.7.7. CHAR(*integer*)

This function returns the character in the ordinal position specified by the *integer* argument (a PIC 9 item or numeric literal; no fractional part is allowed) from the collating sequence being used by the program.

For example, if the program is using the (default) ASCII character set, CHAR(34) returns the 34<sup>th</sup> character in the ASCII character set – an exclamation-point ("!"). If you are using this function to convert a numeric value to its corresponding ASCII character, you must use an argument value one greater than the numeric value.

If an argument whose value is less than 1 or greater than 256 is specified, the character in the program collating sequence corresponding to a value of all zero bits is returned.

The following code is an alternative approach when you just wish to convert a number to its ASCII equivalent:

```
01 Char-Value.
   05 Numeric-Value          USAGE BINARY-CHAR.
   .
   .
   .
   MOVE numeric-character-value TO Numeric-Value
   The Char-Value item now has the corresponding ASCII character value
```

### 6.1.7.8. COMBINED-DATETIME(*days, seconds*)

This function returns a 12-digit result, the first seven digits of which are the value of the *days* argument (a PIC 9 item or a numeric literal) and the last five of which are the value of the *seconds* argument (a PIC 9 item or a numeric literal).

If a days value less than 1 or greater than 3067671 is specified, or if a seconds value less than 1 or greater than 86400 is specified, a value of 0 is returned and a runtime error will result.

### 6.1.7.9. CONCATENATE(*string-1* [, *string-2* ] ...)

This function concatenates the specified *strings* (group items, USAGE DISPLAY elementary items and/or alphanumeric literals) together into a single string result.

If a numeric literal or PIC 9 identifier is specified as a *string*, decimal points will be removed and negative signs in PIC S9 fields will be inserted as defined by the SIGN clause (or absence thereof) of the field. Numeric literals are processed as if SIGN IS TRAILING were in effect.

### 6.1.7.10. COS(*number*)

The COS function determines and returns the trigonometric cosine of the *angle* (a PIC 9 item or numeric literal) supplied as an argument.

### 6.1.7.11. CURRENT-DATE

Returns the current date and time as the following 21-character structure:

```
01 CURRENT-DATE-AND-TIME.
   05 CDT-Year           PIC 9(4).
   05 CDT-Month          PIC 9(2). *> 01-12
   05 CDT-Day            PIC 9(2). *> 01-31
   05 CDT-Hour           PIC 9(2). *> 00-23
   05 CDT-Minutes        PIC 9(2). *> 00-59
   05 CDT-Seconds        PIC 9(2). *> 00-59
   05 CDT-Hundredths-Of-Secs PIC 9(2). *> 00-99
   05 CDT-GMT-Diff-Hours PIC S9(2)
                        SIGN LEADING SEPARATE.
   05 CDT-GMT-Diff-Minutes PIC 9(2). *> 00 or 30
```

Since the CURRENT-DATE function has no arguments, no parenthesis should be specified.

### 6.1.7.12. DATE-OF-INTEGGER(*integer*)

This function returns a calendar date in *yyyymmdd* format. The date is determined by adding the number of days specified as *integer* (a PIC 9 item or numeric literal; cannot contain a fractional part) to December 31, 1600. For example, DATE-OF-INTEGGER(1) returns 16010101.

A value less than 1 or greater than 3067671 (9999/12/31) will return a result of 0.

### 6.1.7.13. DATE-TO-YYYYMMDD(*yymmdd* [, *yy-cutoff* ] )

You can use this function to convert the six-digit date specified as *yymmdd* (a PIC 9 data item or a numeric literal) to an eight-digit format (*yyyymmdd*). The optional *yy-cutoff* (a PIC 9 data item or a numeric literal) argument is the year cutoff used to delineate centuries; if the year component of the date meets or exceeds this cutoff value, the result will be 19*yyyymmdd*; if the year component of the date is less than the cutoff value, the result will be 20*yyyymmdd*. The default cutoff value if no second argument is given will be 50.

### 6.1.7.14. DAY-OF-INTEGGER(*integer*)

This function returns a calendar date in yyyyddd (i.e. Julian) format. The date is determined by adding the number of days specified as *integer* (a PIC 9 item or numeric literal; cannot contain a fractional part) to December 31, 1600. For example, DATE-OF-INTEGGER(1) returns 1601001.

A value less than 1 or greater than 3067671 (9999/12/31) will return a result of 0.

### 6.1.7.15. DAY-TO-YYYYDDD(*yyddd* [, *yy-cutoff*])

You can use this function to convert the five-digit date specified as *yyddd* (a PIC 9 data item or a numeric literal) to a seven-digit format (yyyyddd). The optional *yy-cutoff* argument (a PIC 9 data item or a numeric literal) is the year cutoff used to delineate centuries; if the year component of the date meets or exceeds this cutoff value, the result will be 19yyddd; if the year component of the date is less than the cutoff, the result will be 20yyddd. The default cutoff value if no second argument is given will be 50.

### 6.1.7.16. E

This function returns the mathematical constant “E” (the base of natural logarithms). The maximum precision with which this value may be returned is 2.7182818284590452353602874713526625.

Since the E function has no arguments, no parenthesis should be specified.

### 6.1.7.17. EXCEPTION-FILE

This function returns I/O exception information from the most-recently executed input or output statement. The information is returned to a structure resembling the following:

```
01 INPUT-OUTPUT-EXCEPTION.
   05 IOE-FILE-STATUS          PIC 9(2).
   05 IOE-FILE-SELECT-NAME     PIC X(32).
```

See

[Figure 4-11](#) for information about possible file-status values.

The name returned after the file status information will be the “SELECT” name of the file, and it will be returned ONLY if the returned file status value is not 00.

Since the EXCEPTION-FILE function has no arguments, no parenthesis should be specified.

### 6.1.7.18. EXCEPTION-LOCATION

This function returns exception information from the most-recently failing statement. The information is returned to a 1023 character string in one of the following formats, depending on the nature of the failure:

```
program-id; paragraph OF section; statement-number
program-id; section; statement-number
program-id; paragraph; statement-number
program-id; statement-number
```

Since the EXCEPTION-LOCATION function has no arguments, no parenthesis should be specified.

The program must be compiled with the “-g” option for this function to return any meaningful information.

See section [6.5.1](#) for an example of this function at work.

### 6.1.7.19. EXCEPTION-STATEMENT

This function returns the most-recent COBOL statement that generated an exception condition.

Since the EXCEPTION-STATEMENT function has no arguments, no parenthesis should be specified.

The program must be compiled with the “-g” option for this function to return any meaningful information.

See section [6.5.1](#) for an example of this function at work.

### 6.1.7.20. EXCEPTION-STATUS

This function returns the error type (as a text string) from the most-recent COBOL statement that generated an exception condition.

Since the EXCEPTION-STATUS function has no arguments, no parenthesis should be specified.

See section [6.5.1](#) for an example of this function at work.

### 6.1.7.21. EXP(number)

Computes and returns the value of the mathematical constant “e” raised to the power specified by *number* (a PIC 9 item or numeric literal).

### 6.1.7.22. EXP10(number)

Computes and returns the value of 10 raised to the power specified by *number* (a PIC 9 item or numeric literal).

### 6.1.7.23. FRACTION-PART(number)

This function returns that portion of *number* that occurs to the right of the decimal point. *Number* must be a numeric data item or a numeric literal. FRACTION-PART(3.1415), for example, returns a value of 0.1415. This function is equivalent to the expression:

$$number - \text{FUNCTION INTEGER-PART}(number)$$

### 6.1.7.24. FACTORIAL(number)

This function computes and returns the factorial value of *number* (a PIC 9 item or numeric literal).

### 6.1.7.25. INTEGER(number)

The INTEGER function returns the greatest integer value that is less than or equal to *number* (a PIC 9 item or numeric literal).

### 6.1.7.26. INTEGER-OF-DATE(date)

This function converts *date* (a PIC 9 item or numeric literal; cannot contain a fractional part) – presumed to be a Gregorian calendar form standard date (YYYYMMDD) - to integer date form – that is, the number of days that have transpired since 1600/12/31.

### 6.1.7.27. INTEGER-OF-DAY(date)

This function converts *date* (a PIC 9 item or numeric literal; cannot contain a fractional part) – presumed to be a Julian calendar form standard date (YYYYDDD) to integer date form – that is, the number of days that have transpired since 1600/12/31.

### 6.1.7.28. INTEGER-PART(*number*)

Returns the integer portion of the value of *number* (a PIC 9 item or numeric literal).

### 6.1.7.29. LENGTH(*string*)

Returns the length – in bytes – of *string* (a group item, USAGE DISPLAY elementary item or alphanumeric literal). This intrinsic function is identical to the BYTE-LENGTH function.

### 6.1.7.30. LOCALE-DATE(*date* [, *locale* ] )

Converts the eight-digit date (a PIC 9 item or numeric literal; cannot contain a fractional part) from YYYYMMDD format to the format appropriate to the current locale. On a Windows system, this will be the “short date” format as set using Control Panel.

You may include an optional second argument to specify the *locale* name (group item or PIC X identifier) you’d like to use for date formatting. If used, this second argument MUST be an identifier. Locale names are specified using UNIX-standard names. The complete list of supported locale names is shown in [Figure 4-7](#).

### 6.1.7.31. LOCALE-TIME(*time* [, *locale* ] )

Converts the four- (HHMM) or six-digit (HHMMSS) *time* (a PIC 9 item or numeric literal; cannot contain a fractional part) to a format appropriate to the current locale. On a Windows system, this will be the “time” format as set using Control Panel.

You may include an optional *locale* name (a group item or PIC X identifier) you’d like to use for time formatting. If used, this second argument MUST be an identifier. Locale names are specified using UNIX-standard names. The complete list of supported locale names is shown in [Figure 4-7](#).

### 6.1.7.32. LOCALE-TIME-FROM-SECS(*seconds* [, *locale* ] )

Converts the number of *seconds* since midnight (a PIC 9 item or numeric literal; cannot contain a fractional part) to a format appropriate to the current locale. On a Windows system, this will be the “time” format as set using Control Panel.

You may include an optional *locale* name (a group item or PIC X identifier) you’d like to use for time formatting. If used, this second argument MUST be an identifier. Locale names are specified using UNIX-standard names. The complete list of supported locale names is shown in [Figure 4-7](#).

### 6.1.7.33. LOG(*number*)

Computes and returns the natural logarithm (base “e”) of *number* (a PIC 9 item or numeric literal).

### 6.1.7.34. LOG10(*number*)

Computes and returns the base 10 logarithm of *number* (a PIC 9 item or numeric literal).

### 6.1.7.35. LOWER-CASE(*string*)

This function returns the value of *string* (a group item, USAGE DISPLAY elementary item or alphanumeric literal), converted entirely to lower case.

### 6.1.7.36. MAX(*number-1* [, *number-2* ] ...)

This function returns the maximum value from the specified list *numbers* (PIC 9 items and/or numeric literals).

### 6.1.7.37. MIN(*number-1* [, *number-2* ] ...)

This function returns the minimum value from the specified list *numbers* (PIC 9 items and/or numeric literals).

### 6.1.7.38. MEAN(*number-1* [, *number-2* ] ...)

This function returns the statistical mean value of the specified list *numbers* (PIC 9 items and/or numeric literals).

### 6.1.7.39. MEDIAN(*number-1* [, *number-2* ] ...)

This function returns the statistical median value of the specified list *numbers* (PIC 9 items and/or numeric literals).

### 6.1.7.40. MIDRANGE(*number-1* [, *number-2* ] ...)

The MIDRANGE (middle range) function returns a numeric value that is the arithmetic mean (average) of the values of the minimum and maximum *numbers* (PIC 9 items and/or numeric literals).

### 6.1.7.41. MOD(*value*, *modulus*)

Returns *value* modulo *modulus*. Both arguments may be PIC 9 data items or numeric literals. Either (or both) may have a non-integer value.

The result is determined according to the following formula:

$$value - (modulus * FUNCTION INTEGER (value / modulus))$$

### 6.1.7.42. NUMVAL(*string*)

The NUMVAL function converts a *string* (a group item, USAGE DISPLAY elementary item or alphanumeric literal) to its corresponding numeric value by parsing that string according to the rules for COBOL PICTURE editing. For example, the string "12,345.55" will be converted to a numeric value of 12345.55.<sup>18</sup>

### 6.1.7.43. NUMVAL-C(*string* [, *symbol* ])

This function performs a function similar to that of the NUMVAL function, but provides for the specification of a converts a *string* (a group item, USAGE DISPLAY elementary item or alphanumeric literal) to its corresponding numeric value by parsing that string according to the rules for COBOL PICTURE editing. The optional symbol character represents the currency *symbol* (a group item, USAGE DISPLAY elementary item or alphanumeric literal) that occurs within string. For example, the string "\$12,345.55" will be converted to a numeric value of 12345.55 (" \$" is the assumed default currency symbol).<sup>18</sup>

---

<sup>18</sup> The string parsing rules for NUMVAL and NUMVAL-C are quite loose, essentially ignoring any non-numeric or invalid characters. Thus, both routines will generate a result of 1234 for "\$1,234", "\*\*\*\*\*1234", "xxxxx12xxx34" and so on.

### 6.1.7.44. ORD(*char*)

This function returns the ordinal position in the program character set (usually ASCII) corresponding to the 1<sup>st</sup> character of the *char* argument (a group item, USAGE DISPLAY elementary item or alphanumeric literal). For example, assuming the program is using the standard ASCII collating sequence, ORD("!") returns 34 because "!" is the 34<sup>th</sup> ASCII character. If you are using this function to convert an ASCII character to its numeric value, you must subtract one from the result.

The following code is an alternative approach when you just wish to convert an ASCII character to its numeric equivalent:

```
01 Char-Value.
   05 Numeric-Value          USAGE BINARY-CHAR.
.
.
.
MOVE "character" TO Char-Value
The Numeric-Value item now has the corresponding numeric value
```

### 6.1.7.45. ORD-MAX( *char-1* [, *char-2* ] ... )

This function returns the ordinal position in the argument list corresponding to the argument whose 1<sup>st</sup> character has the highest position in the program collating sequence (usually ASCII). For example, assuming the program is using the standard ASCII collating sequence, ORD-MAX("Z", "z", "!") returns 2 because the ASCII character "z" occurs after "Z" and "!" in the program collating sequence. Each *char* argument is a group item, USAGE DISPLAY elementary item or alphanumeric literal

### 6.1.7.46. ORD-MIN( *char-1* [, *char-2* ] ... )

This function returns the ordinal position in the argument list corresponding to the argument whose 1<sup>st</sup> character has the lowest position in the program collating sequence (usually ASCII). For example, assuming the program is using the standard ASCII collating sequence, ORD-MIN("Z", "z", "!") returns 3 because the ASCII character "!" occurs before "Z" and "z" in the program's collating sequence. Each *char* argument is a group item, USAGE DISPLAY elementary item or alphanumeric literal

### 6.1.7.47. PI

This function returns the mathematical constant "PI". The maximum precision with which this value may be returned is 3.1415926535897932384626433832795029.

Since the PI function has no arguments, no parenthesis should be specified.

### 6.1.7.48. PRESENT-VALUE(*rate,value-1* [, *value-2* ] )

The PRESENT-VALUE function returns a value that approximates the present value of a series of future period-end amounts specified by the various *value* arguments at a discount rate specified by the *rate* argument. All arguments are PIC 9 items and/or numeric literals.

The following formula summarizes the functions operation:  $result = \sum_{n=1}^{\#\_of\_values} \frac{value_n}{(1+rate)^n}$

### 6.1.7.49. RANDOM [ ( *seed* ) ]

The RANDOM function returns a non-integer value in the range 0 to 1 (for example, 0.123456789).

If *seed* is specified, it must be zero or a positive integer (specified as a PIC 9 item and/or numeric literal). It is used as the seed value to generate a sequence of pseudo-random numbers.

If a subsequent reference specifies *seed*, a new sequence of pseudo-random numbers is started.

If the first executed reference to this function does not specify a *seed*, the seed will be supplied by the compiler.

In each case, subsequent references without specifying a *seed* return the next number in the current sequence.

### 6.1.7.50. RANGE(*number-1* [, *number-2* ] ...)

The RANGE function returns a value that is equal to the value of the maximum *number* in the argument list minus the value of the minimum *number* argument. All arguments are PIC 9 items and/or numeric literals.

### 6.1.7.51. REM(*number*, *divisor*)

This function returns a numeric value that is the remainder of *number* divided by *divisor*. Both arguments may be PIC 9 items and/or numeric literals.

The result is determined according to the following formula:

$$number - (divisor * FUNCTION INTEGER-PART (number / divisor))$$

### 6.1.7.52. REVERSE(*string*)

This function returns the byte-by-byte reversed value of the specified *string* (a group item, USAGE DISPLAY elementary item or alphanumeric literal).

### 6.1.7.53. SECONDS-FROM-FORMATTED-TIME(*format*,*time*)

This function decodes a string whose value represents a formatted time and returns the total number of seconds that string represents. The time string must contain hours, minutes and seconds. The time argument may be specified as a group item, USAGE DISPLAY elementary item or an alphanumeric literal.

The *format* argument is a string (a group item, USAGE DISPLAY elementary item or an alphanumeric literal) documenting the format of *time* using "hh", "mm" and "ss" to denote where the respective time information can be found. Any other characters found in *format* represent character positions that will be ignored. For example, a *format* of "hhmmss" indicates that *time* will be treated as a six-digit value where the first two characters are the number of hours, the next two represent minutes and the last two represent seconds. Similarly, a *format* of "hh:mm:ss" states that *time* will be an eight-character string where characters 3 and 6 will be ignored.

### 6.1.7.54. SECONDS-PAST-MIDNIGHT

This function returns the current time of day expressed as the total number of elapsed seconds since midnight.

### 6.1.7.55. SIGN(*number*)

The SIGN function returns a -1 if the value of *number* (a PIC 9 item or numeric literal) is negative, a zero if the value of *number* is exactly zero and a 1 if the value of *number* is greater than 0.

### 6.1.7.56. SIN(*angle*)

Determines and returns the trigonometric sine of the specified *angle* (a PIC 9 item or numeric literal).

### 6.1.7.57. SQRT(*number*)

The SQRT function returns a numeric value that approximates the square root of number (a PIC 9 item or numeric literal with a non-negative value).



### 6.1.7.58. MEAN(number-1 [, number-2 ] ...)

This function returns the statistical standard deviation of the specified list *numbers* (PIC 9 items and/or numeric literals).

### 6.1.7.59. STORED-CHAR-LENGTH(*string*)

Returns the length – in bytes – of the specified *string* (a group item, USAGE DISPLAY elementary item or alphanumeric literal) MINUS the total number of trailing spaces, if any.

### 6.1.7.60. SUBSTITUTE(*string*,*from-1*,*to-1* [, *from-n*,*to-n* ] )

This function parses the specified *string*, replacing all occurrences of the *from-n* strings with the corresponding *to-n* strings. The *from* strings must match exactly with regard to value and case. The *from* strings do not have to be the same length as the *to* strings. All arguments are group items, USAGE DISPLAY elementary items or alphanumeric literals.

A null *to* string will be treated as a single SPACE.

### 6.1.7.61. SUBSTITUTE-CASE(*string*,*from-1*,*to-1* [, *from-n*,*to-n* ] )

The SUBSTITUTE-CASE function operates the same as the SUBSTITUTE function, except that *from* string matching is performed without regard for case. All arguments are group items, USAGE DISPLAY elementary items or alphanumeric literals.

### 6.1.7.62. SUM(number-1 [, number-2 ] ...)

The SUM function returns a value that is the sum of the *number* arguments (PIC 9 items and/or numeric literals).

### 6.1.7.63. TAN(*angle*)

Determines and returns the trigonometric tangent of the specified angle (a PIC 9 item or numeric literal).

### 6.1.7.64. TEST-DATE-YYYYMMDD(*date*)

Determines if the supplied *date* (a PIC 9 item or numeric literal; cannot contain a fractional part) is a valid date of the form *yyyymmdd* and that the date is in the range 1601/01/01 to 9999/12/31. If it is value, a 0 value is returned. If it isn't, a value of 1 is returned.

### 6.1.7.65. TEST-DAY-YYYYDDD(*date*)

Determines if the supplied date (a PIC 9 item or numeric literal (cannot contain a fractional part) is a valid date of the form *yyyddd* and that the date is in the range 1601001 to 9999365. If it is value, a 0 value is returned. If it isn't, a value of 1 is returned.

### 6.1.7.66. TRIM(*string*[ , LEADING|TRAILING ] )

This function removes leading or trailing spaces from the specified *string* (a group item, USAGE DISPLAY elementary item or alphanumeric literal). The second argument is specified as a keyword, not a quoted string or identifier. If no second argument is specified, both leading and trailing spaces will be removed.

### 6.1.7.67. UPPER-CASE(*string*)

This function returns the value of *string* (a group item, USAGE DISPLAY elementary item or alphanumeric literal), converted entirely to upper case.

### 6.1.7.68. VARIANCE(*number-1* [, *number-2* ] ...)

This function returns the statistical variance of the specified list *numbers* (PIC 9 items and/or numeric literals).

### 6.1.7.69. WHEN-COMPILED

This function returns date and time the program was compiled in the same format as the result returned by the CURRENT-DATE function.

Since the WHEN-COMPILED function has no arguments, no parenthesis should be specified.

### 6.1.7.70. YEAR-TO-YYYY (*yy* [, *yy-cutoff*])

YEAR-TO-YYYY converts *yy* (a) - a two-digit year - to a four-digit format (yyyy). The optional *yy-cutoff* argument (also a PIC 9 data item or numeric literal) is the year cutoff used to delineate centuries; if *yy* meets or exceeds this cutoff value, the result will be 19*yy*; if *yy* is less than the cutoff, the result will be 20*yy*. The default cutoff value if no second argument is given will be 50.

## 6.1.8. Special Registers

OpenCOBOL, like other COBOL dialects, includes a number of data items that are automatically available to a programmer without the need to actually define them in the DATA DIVISION. COBOL refers to such items as registers or special registers. The special registers available to an OpenCOBOL program are as follows:

Figure 6-15 - Special Registers

Register Name	Implied COBOL PIC/USAGE <sup>19</sup>	Usage
LINAGE-COUNTER	BINARY-LONG SIGNED	<p>An occurrence of this register exists for each SELECTed file having a LINAGE clause (see section <a href="#">5.1</a>). If there are multiple files whose FDs have a LINAGE clause, any explicit references to this register will require qualification (using "OF <i>file-name</i>").</p> <p>The value of this register will be the current logical line number within the page body (see section <a href="#">5.1</a> for a discussion of how the LINAGE clause structures logical pages).</p> <p><b>DO NOT MODIFY THE CONTENTS OF THIS REGISTER.</b></p>
NUMBER-OF-CALL-PARAMETERS	BINARY-LONG SIGNED	<p>This register contains the number of arguments passed to a subprogram. Its value will be zero when referenced in a main program.</p> <p>See the C\$NARG built-in subroutine documentation in section <a href="#">7.3.1.7</a> for another way of retrieving the same data.</p>

<sup>19</sup> See section [5.3](#) for a description of the PICTURE / USAGE specifications

Register Name	Implied COBOL PIC/USAGE <sup>19</sup>	Usage
RETURN-CODE	BINARY-LONG SIGNED	This register provides a numeric data item into which a subroutine may MOVE a value prior to transferring control back to the program that CALLED it, or into which a main program may MOVE a value before returning control to the operating system.  Many built-in subroutines (section 7.3) will return a value using this register.  These values are – by convention – used to signify success (usually with a value of 0) or failure (usually with a non-zero value) of the process the program setting the RETURN-CODE value was attempting to perform.
SORT-RETURN	BINARY-LONG SIGNED	This register is used to report the success/fail status of a RELEASE or RETURN statement. A value of 0 is reported on success. A value of 16 denotes failure. An “AT END” condition on a RETURN is <u>not</u> considered a failure.
WHEN-COMPILED	See “Usage”	This register contains the date and time the program was compiled in the format “mm/dd/yyhh.mm.ss”. Note that only a two-digit year is provided.

## 6.1.9. Controlling Concurrent Access to Files

The manipulation of data files is one of the COBOL language’s great strengths. There are features built-in to the COBOL language to deal with the possibility that multiple programs may be attempting to access the same file concurrently. Multiple program concurrent access is dealt with in two ways – *file sharing* and *record locking*.

Not all OpenCOBOL implementations support file sharing and record-locking options. Whether they do or not depends upon the operating system they were built for and the build options that were used when the specific OpenCOBOL implementation was generated.

### 6.1.9.1. File Sharing

OpenCOBOL controls concurrent-file access at the highest level through the concept of file sharing, enforced when a program attempts to OPEN a file (see section 6.32). This is accomplished via a UNIX operating-system routine called “`fcntl()`”. That module is not currently supported by Windows<sup>20</sup> and is not present in the MinGW Unix-emulation package. OpenCOBOL builds created using a MinGW environment will be incapable of supporting file-sharing controls – files will always be shared in such environments. An OpenCOBOL build created using the Cygwin environment on Windows would have access to “`fcntl()`” and therefore will support file sharing. Of course, actual Unix builds of OpenCOBOL, as well as MacOS builds<sup>21</sup>, will have no issues using BDB because “`fcntl()`” is built-in to Unix.

Any limitations you impose on a successful OPEN will remain in place until your program either issues a CLOSE against the file or terminates.

There are three ways in which concurrent access to a file may be controlled at the file level:

<sup>20</sup> Windows has other means of providing equivalent functionality to “`fcntl()`”, but the BDB package was not coded to utilize them. The use of other advanced file I/O packages that support both the UNIX and Windows concurrent-access routines (such as VBISAM) are currently under investigation by the author.

<sup>21</sup> Apple Computer’s MacOS X operating system is based on an open-source version of UNIX and therefore includes support of “`fcntl()`”.

Sharing Option	Effect
ALL OTHER	When your program opens a file in this manner, no restrictions will be placed on other programs attempting to OPEN the file after your program did. This is the default sharing mode.
NO OTHER	When your program opens a file in this manner, your program announces that it is unwilling to allow <u>any</u> other program to have <u>any</u> access to the file as long as you are using that file; OPEN attempts made in other programs will fail with a file status of 37 (“PERMISSION DENIED”) until such time as you CLOSE the file (see section <a href="#">6.10</a> ).
READ ONLY	Opening a file in this manner indicates you are willing to allow other programs to OPEN the file for INPUT while you have it OPEN. If they attempt any other OPEN, their OPEN will fail with a file status of 37.

Of course, your program may fail if someone else got to the file first and OPENed it with a sharing option that imposed file-sharing limitations.

### 6.1.9.2. Record Locking

Record-locking is supported by advanced file-management software that provides a single point-of-control for access to files (usually ORGANIZATION INDEXED files). One such runtime package capable of doing this is the Berkely Database (BDB) package. The various I/O statements are capable of imposing limitations on the access – by other concurrently-executing programs – to the file record they just accessed. These limitations are syntactically imposed by placing a lock on the record. Other records in the file remain available, assuming that file-sharing limitations imposed at OPEN-time didn’t prevent access to the entire file.

Locks remain in-effect until a program holding the lock terminates, Issues a CLOSE (section [6.10](#)) against the file, issues an UNLOCK (section [6.49](#)) against the file, executes a COMMIT (section [6.11](#)) or executes a ROLLBACK (section [6.38](#)).

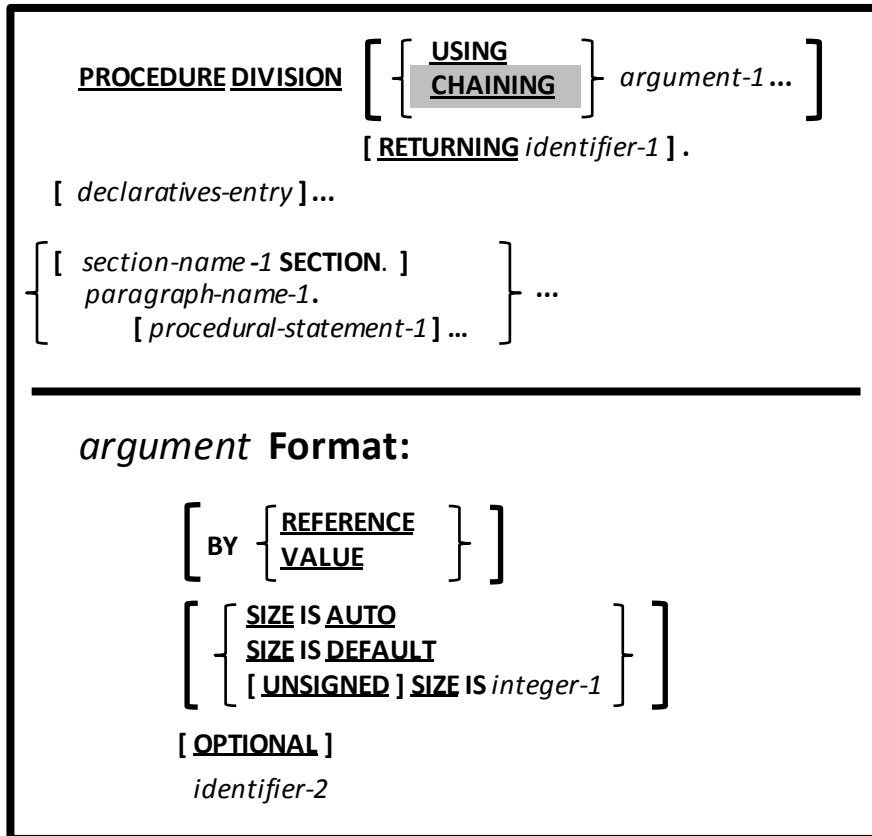
The record locking options (not all options are available to all statements) are as shown in the following table.

Record Locking Option	Effect
WITH LOCK	Access to the record by other programs will be denied.
WITH NO LOCK	The record will not be locked. This is the default locking option in effect for all statements.
IGNORING LOCK WITH IGNORE LOCK	This option is possible only when reading records – it informs OpenCOBOL that any locks held by other programs should be ignored.  The two options shown are synonymous.
WITH WAIT	This option is possible only when reading records – it informs OpenCOBOL that the program is willing to wait for a lock held on the record being read to be released.  Without this option, an attempt to read a locked record will be immediately aborted and a file status of 47 will be returned.  With this option, the program will wait for a pre-configured time for the lock to be released. If the lock is released within the preconfigured wait time, the read will be successful. If the pre-configured wait time expires before the lock is released, the read attempt will be aborted and a 47 file status will be issued.

If the OpenCOBOL build you are using was configured to use BDB, record locking will be available by using the execution-time environment variable DB\_HOME (see section [7.2.4](#)).

## 6.2. General Format of the PROCEDURE DIVISION

Figure 6-16 - General PROCEDURE DIVISION Syntax



The first (optional) segment of the PROCEDURE DIVISION is a special area known as “declaratives”. In this area, you may define processing routines that are to be used as special “trap” routines executed only when certain events occur. These will be described in section [6.3](#).

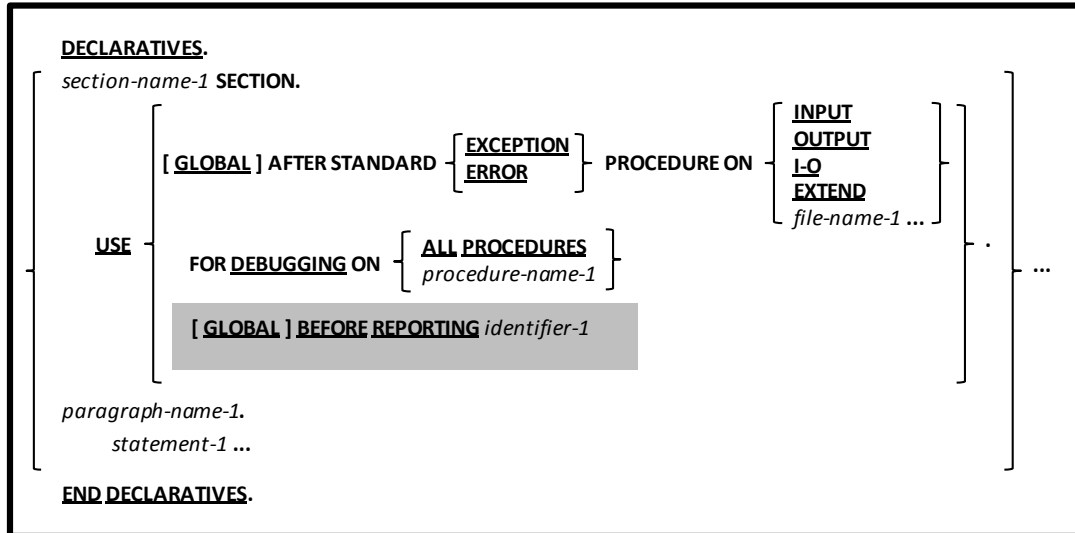
The various sections and paragraphs in which the procedural logic of your program will be coded will follow any “declaratives”. The PROCEDURE DIVISION is the only one of the COBOL divisions that allows you to create your own sections and paragraphs.

1. The USING and RETURNING clauses define arguments to a program serving as a subroutine. All identifiers specified on these clauses must be defined in the LINKAGE SECTION of the program in which the USING and/or RETURNING clauses appear.
2. The CHAINING clause should be used only by a program that will be invoked by another program via the CHAIN verb. The identifiers specified with CHAINING clauses must be defined in the LINKAGE SECTION of the program in which the CHAINING clauses appear. Chaining, however, is currently syntactically supported by OpenCOBOL but is otherwise non-functional. Attempts to use the CHAIN verb will be rejected.
3. While intended for use with user-defined FUNCTIONS (which are not currently supported by OpenCOBOL), the RETURNING clause can be used as a means of specifying and documenting an argument to a subprogram in which a value is returned.
4. The BY REFERENCE clause indicates that the program will be passed the address of the data item corresponding to a program argument; this program will be allowed to modify the contents of any BY REFERENCE argument. BY REFERENCE is the assumed default for all USING/CHAINING arguments (CHAINING arguments must be BY REFERENCE).
5. The BY VALUE clause indicates the program will be passed a read-only copy of the data item from the calling program that corresponds to the argument. The contents of BY VALUE arguments cannot be changed by the subprograms receiving them.
6. The USING mechanism is NOT how OpenCOBOL programs should retrieve their command-line arguments as is the case in some mainframe implementations of COBOL. See the ACCEPT verb for information on how program command line arguments should be retrieved.
7. The various SIZE clauses specify the size (in bytes) of received arguments. The SIZE IS AUTO clause (the default) indicates that argument size will be determined automatically based upon the size of the item in the calling

program. The remaining SIZE options allow you to force a specific size to be assumed, with SIZE IS DEFAULT being the same as UNSIGNED SIZE IS 4.

### 6.3. General Format for DECLARATIVES Entries

Figure 6-17 - General DECLARATIVES Syntax



The DECLARATIVES area of the PROCEDURE DIVISION allows the programmer to define a series of “trap” routines capable of intercepting certain events that may occur at program execution time.

1. Since the RWCS is not currently supported by OpenCOBOL, the USE BEFORE REPORTING clause will be syntactically recognized but rejected as being unsupported.
2. The USE FOR DEBUGGING clause allows you to define a routine that will be invoked immediately before the named procedure is executed (or immediately before any procedure is executed if the ALL PROCEDURES option is used).
3. The USE AFTER STANDARD ERROR PROCEDURE clause defines a routine invoked any time a failure is encountered with the specified I/O type (or against the specified file(s)).
4. The GLOBAL option, if used, allows a declaratives procedure to be used across all program units in the same compilation unit.
5. DECLARATIVES routines (of any type) may not reference any procedures outside the scope of the DECLARATIVES area except for referencing them via the PERFORM statement.

## 6.4. ACCEPT

### 6.4.1. ACCEPT Format 1 – Read from Console

Figure 6-18 - ACCEPT (Read from Console) Syntax

```
ACCEPT identifier
  [ FROM mnemonic-name ]
[ END-ACCEPT ]
```

This format of the ACCEPT verb is used to read a value from the console window and store it into a data item (*identifier*).

1. If the FROM clause is used, the specified *mnemonic-name* must be either SYSIN or CONSOLE, or a user-defined mnemonic-name assigned to one of those two devices via the SPECIAL-NAMES paragraph. The devices SYSIN and CONSOLE may be used interchangeably and both reference the console window.
2. If no FROM clause is specified, FROM CONSOLE is assumed.

### 6.4.2. ACCEPT Format 2 – Retrieve Command-Line Arguments

Figure 6-19 - ACCEPT (Command Line Arguments) Syntax

```
ACCEPT identifier
  FROM { COMMAND-LINE
          ARGUMENT-NUMBER
          ARGUMENT-VALUE [ exception-handler ] }
[ END-ACCEPT ]
```

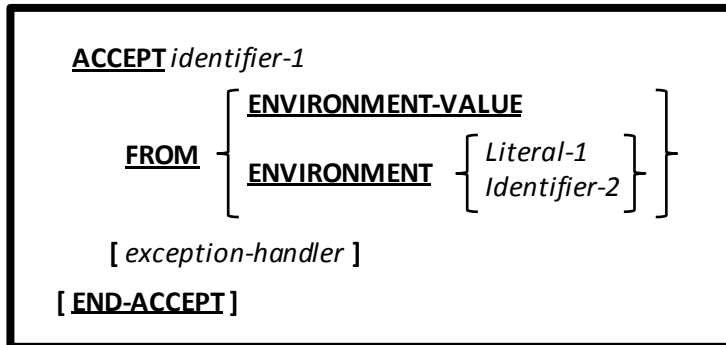
This format of the ACCEPT verb is used to retrieve arguments from the programs command-line.

1. When you accept from the COMMAND-LINE option, you will retrieve the entire set of arguments entered on the command line that executed the program, exactly as they were specified. Parsing that returned data into its meaningful information will be your responsibility.
2. By accepting from ARGUMENT-NUMBER, you will be asking the OpenCOBOL run-time system to parse the arguments from the command-line and return the number of arguments found. Parsing will be conducted according to the operating system's rules, as follows:
  - Arguments will be separated by treating SPACES between characters as the delineators between arguments. The number of spaces separating two non-blank values is irrelevant.
  - Strings enclosed in double-quote characters (") will be treated as a single argument, regardless of how many spaces (if any) might be imbedded within those quotation characters.
  - On Windows systems, single-quote, or apostrophe characters (') will be treated just like any other data character and will NOT delineate strings.
3. By accepting from ARGUMENT-VALUE, you will be asking the OpenCOBOL run-time system to parse the arguments from the command-line and return the arguments whose number is currently in the ARGUMENT-NUMBER register<sup>22</sup>. Parsing will be conducted according to the rules set forth in #2 above.
4. The syntax and usage of the optional exception-handler is discussed in section [6.4.7](#).

<sup>22</sup> Use format #2 of the DISPLAY statement to set ARGUMENT-NUMBER to the desired value

### 6.4.3. ACCEPT Format 3 – Retrieve Environment Variable Values

Figure 6-20 - ACCEPT (Environment Variable Values) Syntax



This format of the ACCEPT verb is used to retrieve environment variable values.

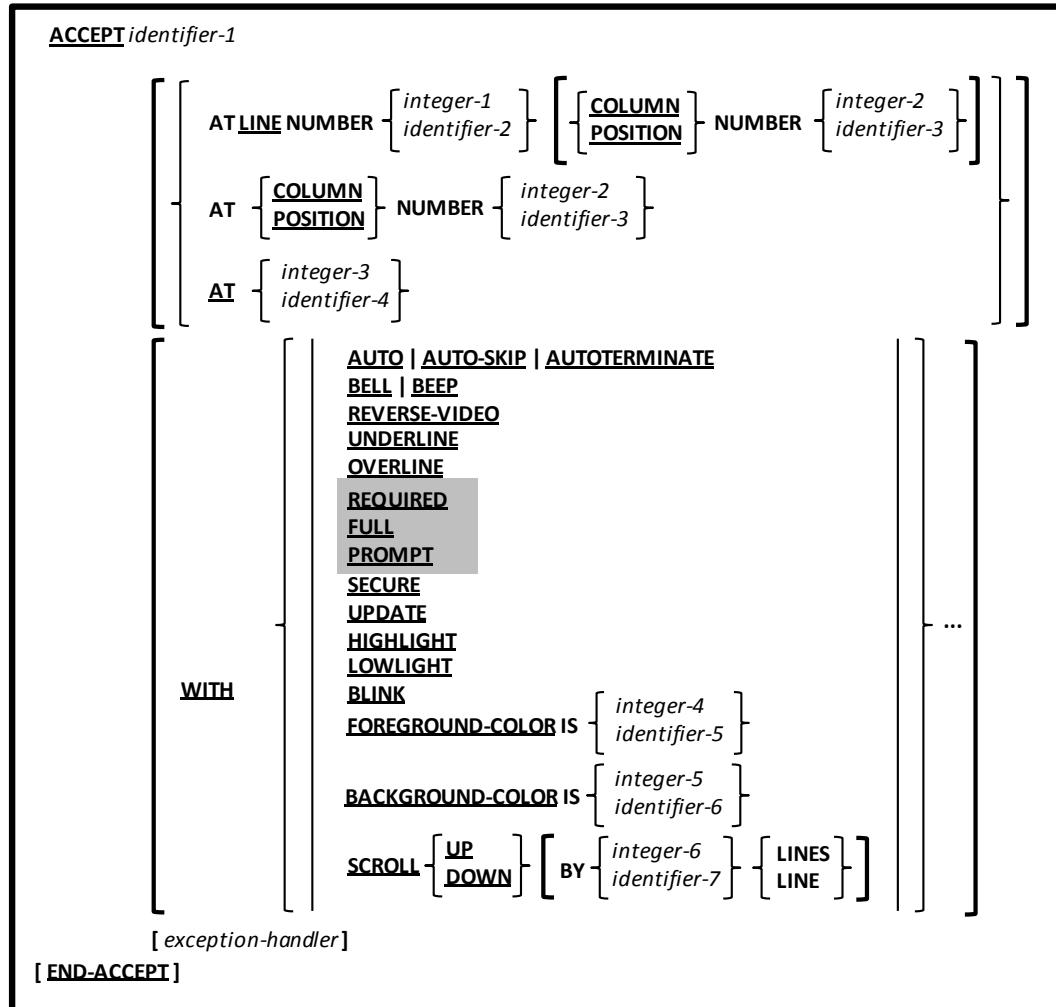
1. By accepting from ENVIRONMENT-VALUE, you will be asking the OpenCOBOL run-time system to retrieve the value of the environment variable whose name is currently in the ENVIRONMENT-NAME register<sup>23</sup>.
2. A simpler approach to retrieving an environment variables value is to use "ACCEPT ... FROM ENVIRONMENT". Using that form, you specify the environment variable to be retrieved right on the ACCEPT command itself.
3. The syntax and usage of the optional exception-handler is discussed in section [6.4.7](#).

<sup>23</sup> Use format #3 of the DISPLAY statement to set ENVIRONMENT-NAME to the desired environment variable name



## 6.4.4. ACCEPT Format 4 – Retrieve Screen Data

Figure 6-21 - ACCEPT (Retrieve Screen Data) Syntax

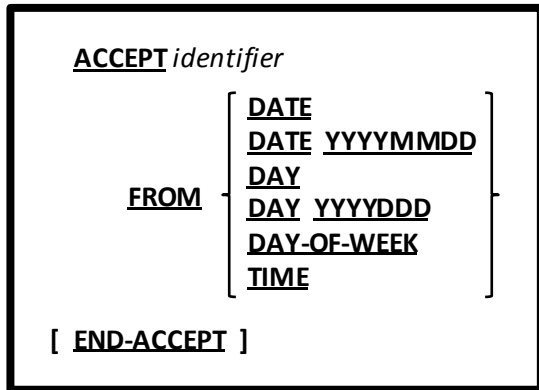


This format of the ACCEPT verb is used to retrieve data from a formatted console window screen using a data item defined in the SCREEN SECTION.

1. If *identifier-1* is defined in the SCREEN SECTION, all cursor positioning (AT) and attribute specifications (WITH) are taken from the SCREEN SECTION definition and anything specified on the ACCEPT will be ignored. Use the AT and WITH options only when ACCEPTING a data item that was not defined in the SCREEN SECTION.
2. The various AT clauses provide a means of positioning the cursor to a specific spot on the screen before the screen is read. The *literal-3* / *identifier-4* value must be a four-digit value with the 1<sup>st</sup> two digits indicating the line where the cursor should be positioned and the last two digits being the column.
3. Most of the WITH options were described in section [0](#), with the exception of the UPDATE and SCROLL options. With the exception of SCROLL, WITH options should be specified only once.
4. The UPDATE option which causes the current contents of *identifier-1* to be displayed before a new value is accepted.
5. The SCROLL option will cause the entire contents of the screen to be scrolled UP or DOWN by the specified number of lines before any value is displayed on the screen. It is possible to specify a SCROLL UP clause as well as a SCROLL DOWN clause. If no LINES specification is made, "1 LINE" will be assumed.
6. The syntax and usage of the optional exception-handler is discussed in section [6.4.7](#).

### 6.4.5. ACCEPT Format 5 - Retrieve Date/Time

Figure 6-22 - ACCEPT (Retrieve Date/Time) Syntax



This format of the ACCEPT verb is used to retrieve the current system date and/or time and store it into a data item.

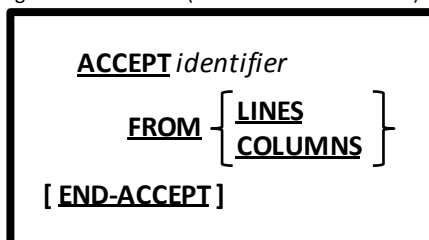
1. The data retrieved from the system, and the format in which it is structured, will vary according to the following chart:

ACCEPT Option	Data Returned	identifier-1 Format
DATE	Current date in Gregorian form	01 CURRENT-DATE. 05 CD-YEAR PIC 9(2). 05 CD-MONTH PIC 9(2). 05 CD-DAY-OF-MONTH PIC 9(2).
DATE YYYYMMDD	Current date in Gregorian form	01 CURRENT-DATE. 05 CD-YEAR PIC 9(4). 05 CD-MONTH PIC 9(2). 05 CD-DAY-OF-MONTH PIC 9(2).
DAY	Current date in Julian form	01 CURRENT-DATE. 05 CD-YEAR PIC 9(2). 05 CD-DAY-OF-YEAR PIC 9(3).
DAY YYYYDDD	Current date in Julian form	01 CURRENT-DATE. 05 CD-YEAR PIC 9(4). 05 CD-DAY-OF-YEAR PIC 9(3).
DAY-OF-WEEK	Current day of the week	01 CURRENT-DATE. 05 CD-DAY-OF-WEEK PIC 9(1). 88 MONDAY VALUE 1. 88 TUESDAY VALUE 2. 88 WEDNESDAY VALUE 3. 88 THURSDAY VALUE 4. 88 FRIDAY VALUE 5. 88 SATURDAY VALUE 6. 88 SUNDAY VALUE 7.
TIME	Current time	01 CURRENT-TIME. 05 CT-HOURS PIC 9(2). 05 CT-MINUTES PIC 9(2). 05 CT-SECONDS PIC 9(2). 06 CT-HUNDREDTHS-OF-SECS PIC 9(2).

Figure 6-23 - ACCEPT Options for DATE/TIME Retrieval

### 6.4.6. ACCEPT Format 6 - Retrieve Screen Size Data

Figure 6-24 - ACCEPT (Retrieve Screen Size Data) Syntax




This format of the ACCEPT verb is used to retrieve the displayable size (in character positions) of the console window in which the program is executing.

1. In environments such as a Windows console window, where the logical size of the window may far exceed that of the physical console window, the size returned will be that of the physical console window.

### 6.4.7. ACCEPT Exception Handling

Figure 6-25 - ACCEPT Exception Handling



```
[ ON EXCEPTION
  imperative-statement-1 ]
[ NOT ON EXCEPTION
  imperative-statement-2 ]
```

The EXCEPTION and NOT EXCEPTION clauses available on some formats of the ACCEPT verb allow you to specify code to be executed specifically upon the failure or success (respectively) of the ACCEPT. Since ACCEPT does not set any sort of return code or status flag, this is the only way to detect success and failure.

## 6.5. ADD

### 6.5.1. ADD Format 1 – ADD TO

Figure 6-26 - ADD (TO) Syntax

```

ADD { [ LENGTH OF ] { literal-1 } } ...
      TO { identifier-2 [ ROUNDED ] } ...
      [ ON SIZE ERROR imperative-statement-1 ]
      [ NOT ON SIZE ERROR imperative-statement-2 ]
      [ END-ADD ]

```

This format of the ADD statement generates the arithmetic sum of all arguments that appear before the TO (*identifier-1* or *literal-1*) and then adds that sum to each of the identifiers listed after the TO (*identifier-2*).

1. *Identifier-1* and *identifier-2* must be numeric unedited data items.
2. *Literal-1* must be a numeric literal.
3. Should a non-integer result be generated and assigned to any of the *identifier-2* data items that have the optional ROUNDED keyword, the result saved into *identifier-2* will be rounded according to the standard mathematical rules for rounding values to the least-significant digit of precision. For example, if the PICTURE is 99V99 and the result to be saved is 12.152, the saved value will be 12.15 whereas a result of 76.165 would save a value of 76.17.
4. If the optional LENGTH OF clause is used on any *literal-1* or *identifier-1*, the arithmetic value used during the computation process will be the length – in bytes – of the data item or literal, not the actual value of that data item or literal.
5. The optional ON SIZE ERROR clause allows you to specify code that will be executed if the result to be saved into an *identifier-2* item exceeds the capacity of that item. For example, if the PICTURE is 99V99 and the result to be saved is 101.43, a SIZE ERROR condition would exist. Without an ON SIZE ERROR clause, OpenCOBOL will store a value of 01.43 into the field. With an ON SIZE ERROR clause, the value of the *identifier-2* item will be unchanged and *imperative-statement-1* will be executed. As an illustration, observe the small demo program and output shown here. It also illustrates some of the “EXCEPTION” intrinsic functions (see section [6.1.7](#)).

Figure 6-27 - A Sample Program Using ON SIZE ERROR

```

1.  IDENTIFICATION DIVISION.
2.  PROGRAM-ID. corrdemo.
3.  DATA DIVISION.
4.  WORKING-STORAGE SECTION.
5.  01 Item-1      VALUE 1          PIC 99V99.
6.  PROCEDURE DIVISION.
7.  100-Main SECTION.
8.  P1.
9.      ADD 19 81.43 TO Item-1
10.     ON SIZE ERROR
11.         DISPLAY 'Item-1:' Item-1
12.         DISPLAY 'Error:'  FUNCTION EXCEPTION-STATUS
13.         DISPLAY 'where:'  FUNCTION EXCEPTION-LOCATION
14.         DISPLAY 'What:'   FUNCTION EXCEPTION-STATEMENT
15.     END-ADD.
16.     STOP RUN.

```

When executed, the program produces the following output:

```

Item-1:0100
Error: EC-SIZE-OVERFLOW
where: corrdemo; P1 OF 100-Main; 9
What: ADD

```

6. The NOT ON SIZE ERROR clause, if specified, will execute an imperative statement if the ADD statement did not encounter a field size overflow condition.

## 6.5.2. ADD Format 2 – ADD GIVING

Figure 6-28 - ADD (GIVING) Syntax

```

ADD { [ LENGTH OF ] { literal-1 } } ...
      [ TO identifier-2 ]
      GIVING { identifier-3 [ ROUNDED ] } ...
      [ ON SIZE ERROR imperative-statement-1 ]
      [ NOT ON SIZE ERROR imperative-statement-2 ]
      [ END-ADD ]

```

This format of the ADD statement generates the arithmetic sum of all arguments that appear before the TO (*identifier-1* or *literal-1*), adds that sum to the contents of *identifier-2* (if any) and then replaces the contents of the identifiers listed after the GIVING (*identifier-3*) with that sum.

1. *Identifier-1* and *identifier-2* must be numeric unedited data items.
2. *Identifier-3* must be a numeric (edited or unedited) data item.
3. *Literal-1* must be a numeric literal.
4. The contents of *identifier-2* are not altered.
5. The use and behavior of the ROUNDED, LENGTH OF, ON SIZE ERROR and NOT ON SIZE ERROR clauses is as described in section [6.5.1](#) for Format 1 of the ADD statement.

## 6.5.3. ADD Format 3 – ADD CORRESPONDING

Figure 6-29 - ADD (CORRESPONDING) Syntax

```

ADD CORRESPONDING identifier-1 TO identifier-2 [ ROUNDED ]
      [ ON SIZE ERROR imperative-statement-1 ]
      [ NOT ON SIZE ERROR imperative-statement-2 ]
      [ END-ADD ]

```

This format of the ADD statement generates code equivalent to individual ADD TO statements for corresponding matches of data items found subordinate to the two identifiers.

1. The rules for identifying corresponding matches are as discussed in section [6.29.2](#) – MOVE CORRESPONDING.
2. The use and behavior of the ROUNDED, ON SIZE ERROR and NOT ON SIZE ERROR clauses is as described in section [6.5.1](#) for Format 1 of the ADD statement.

## 6.6. ALLOCATE

Figure 6-30 - ALLOCATE Syntax

```

ALLOCATE { expression-1 CHARACTERS }
           identifier-1
           [ INITIALIZED ]
           [ RETURNING identifier-2 ]

```

The ALLOCATE statement is used to dynamically allocate memory at run-time.

1. If used, *expression-1* must be an arithmetic expression with a non-zero positive integer value.
2. If used, *identifier-1* should be an 01-level item defined with the BASED attribute in WORKING-STORAGE or LOCAL-STORAGE. It can be an 01 item defined in the LINKAGE SECTION, but using such a data item is not recommended.
3. If used, *identifier-2* should be a USAGE POINTER data item.
4. The optional RETURNING clause will return the address of the allocated memory block into the specified USAGE POINTER item. When this option is used, OpenCOBOL will retain knowledge of the originally-requested size of the allocated memory block in case a FREE (section [6.20](#)) statement is ever issued against that USAGE POINTER item.
5. When the “*identifier-1*” option is used, INITIALIZE will initialize the allocated memory block according to the PICTURE and (if any) VALUE clauses present in the definition of *identifier-1* as if a **INITIALIZE *identifier-1* WITH FILLER ALL TO VALUE THEN TO DEFAULT** were executed once *identifier-1* was allocated. See section 6.25 for a discussion of the INITIALIZE statement.
6. When the “*expression-1* CHARACTERS” option is used, INITIALIZE will initialize the allocated memory block to binary zeros.
7. If the INITIALIZE clause is not used, the initial contents of allocated memory will be left to whatever rules of memory allocation are in effect for the operating system the program is running under.
8. There are two basic ways in which this statement is used. The simplest is:

```
ALLOCATE My-01-Item
```

With this form, a block of storage equal in size to the defined size of **My-01-Item** (which must have been defined with the BASED attribute) will be allocated. The address of that block of storage will become the base address of **My-01-Item** so that it and its subordinate data items become usable within the program.

A second (and equivalent) approach is:

```
ALLOCATE LENGTH OF My-01-Item CHARACTERS RETURNING The-Pointer.
SET ADDRESS OF My-01-Item TO The-Pointer.
```

With this form, the ALLOCATE statements allocates a block of memory exactly the size as would be needed for **My-01-Item**; that address is returned into a pointer variable. The SET statement then “bases” the address of **My-01-Item** to be the address of the memory block created by the ALLOCATE.

The only real functional difference between these two approaches is that – with the first – the INITIALIZED clause, if any, will be honored.

9. Referencing a BASED data item either before its storage has been ALLOCATED or after its storage has been FREEd will lead to unpredictable results<sup>24</sup>.

<sup>24</sup> The COBOL standards like to use the term “unpredictable results” to indicate any sort of unexpected or undesirable behavior – the results in this case probably are predictable though – the program will probably abort from attempting to access an invalid address.

## 6.7. ALTER

Figure 6-31 - ALTER Syntax

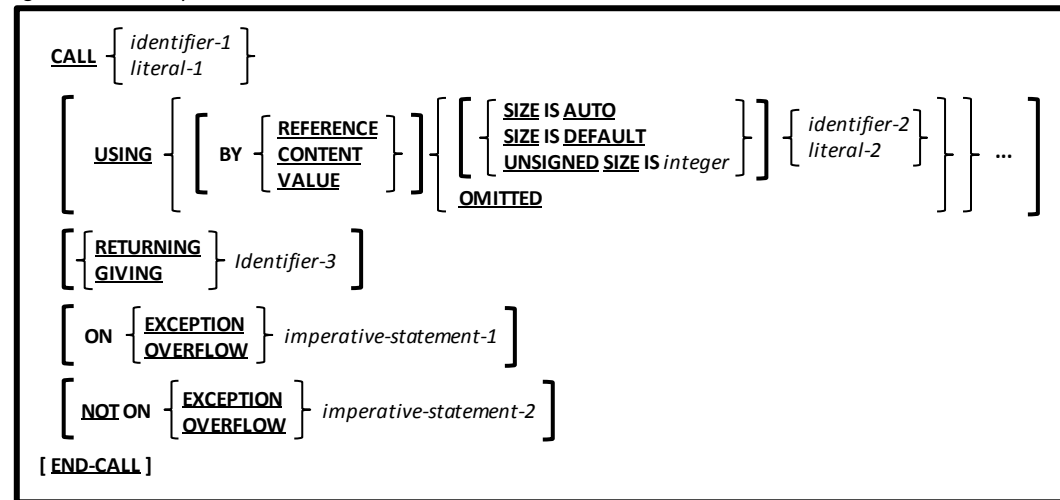
```
ALTER procedure-name-1 TO PROCEED TO procedure-name-2 .
```

The ALTER verb was used in the early years of the COBOL language to edit a program, changing a “GO TO” statement at run time to branch to a spot in the program different than where the GO TO statement was originally compiled for.

1. Support for the ALTER verb has been added to OpenCOBOL for the purpose of enabling OpenCOBOL to pass those National Institute of Standards and Technology (NIST) tests for the COBOL programming language that require support for the ALTER verb.
2. Use of this statement is STRONGLY discouraged.

## 6.8. CALL

Figure 6-32 - CALL Syntax



The CALL statement is used to transfer control to another program, called a *subprogram* or *subroutine*.

1. The expectation is that the subprogram will eventually return control back to the CALLing program, at which point the CALLing program will resume execution starting with the statement immediately following the CALL. Subprograms are not required to return to their CALLers, however, and are free to halt program execution if they wish.
2. The EXCEPTION and OVERFLOW keywords may be used interchangeably.
3. The RETURNING and GIVING keywords may be used interchangeably.
4. The value of *literal-1* or *identifier-1* is the entry-point of the subprogram you wish to CALL. See sections [7.1.4](#) and [7.1.5](#) for more information on how this entry-point is used.
5. When you CALL a subroutine using *identifier-1*, you are forcing the runtime system to call a dynamically-loadable module. See section [7.1.4](#) for information on dynamically-loadable modules.
6. The optional ON EXCEPTION clause specifies code to be executed should the loading of a dynamically-loadable module fail. By specifying ON EXCEPTION, the default behavior of generating an error message and halting the program will be overridden – replaced by whatever logic you specify.
7. The optional NOT ON EXCEPTION clause specifies code to be executed should the loading of a dynamically-loadable module succeed.
8. The USING clause defines a list of arguments that may be passed from the CALLing program to the subprogram. The manner in which the arguments are passed depends upon the BY clause.
9. If the subprogram being CALLED is an OpenCOBOL program, and if that program had the INITIAL attribute specified on its PROGRAM-ID clause, all of the subprogram's DATA DIVISION data will be restored to its initial state each time the subprogram is executed<sup>25</sup>. This [re]-initialization behavior will always apply to any data defined in the subprogram's LOCAL-STORAGE SECTION (if any), regardless of the use (or not) of INITIAL.

<sup>25</sup> This is regardless of which entry-point within the subprogram is CALLED



10. BY REFERENCE (the default) passes the address of the argument to the subprogram and will allow the subprogram to change that argument's value. This can be dangerous when passing a literal value as an argument.

11. BY CONTENT passes the address of a copy of the argument to the subprogram. If the subprogram changes the value of such an argument, the original version of it back in the CALLing program remains unchanged. Clearly, as [Figure 6-33](#) shows, this is the safest way to pass literal values to a subprogram.

12. BY VALUE passes the address of the argument as the argument. Check out the coding example in [Figure 6-34](#). Why would you want this? The answer is simple – if the subprogram is written in OpenCOBOL, you probably wouldn't! This feature exists to provide compatibility with C, C++ and other languages.

13. The RETURNING clause allows you to specify a data item into which the subroutine should return a value. If you use this clause on the CALL, the subroutine should include a RETURNING clause on its PROCEDURE DIVISION header. Of course, a subroutine may pass a value back in ANY argument passed BY REFERENCE.

14. For additional information, see sections [6.9](#) (CANCEL), [6.17](#) (ENTRY), [6.19](#) (EXIT) and [6.22](#) (GOBACK).

Figure 6-33 - CALL BY REFERENCE Can Sometimes have Unwanted Effects!

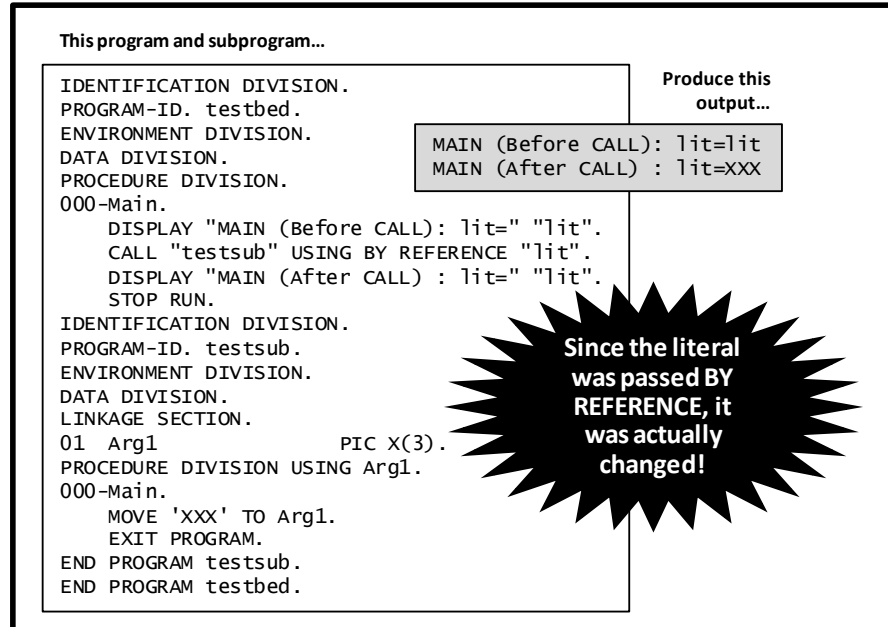
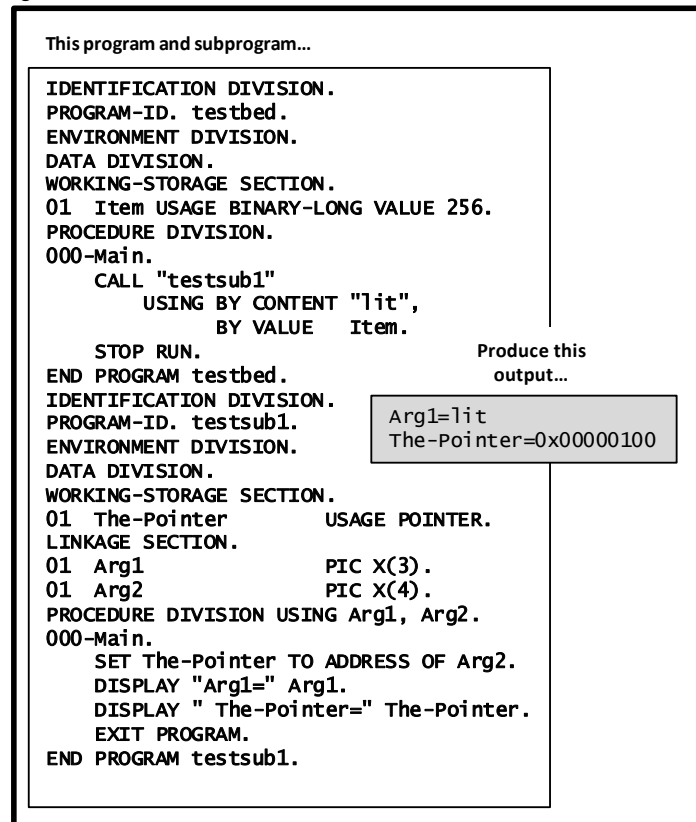
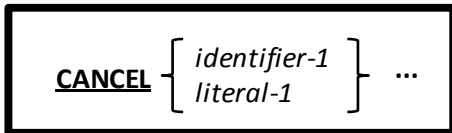


Figure 6-34 - CALL BY VALUE



## 6.9. CANCEL

Figure 6-35 - CANCEL Syntax



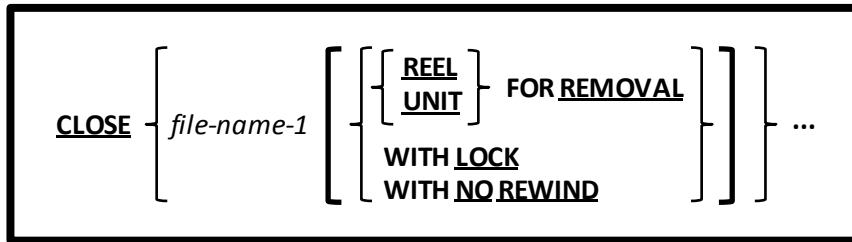
```
CANCEL { identifier-1  
          literal-1 } ...
```

The CANCEL statement unloads the dynamically-loadable module containing the entry-point specified as *literal-1* or *identifier-1* from memory.

1. If the dynamically-loadable module unloaded by the CANCEL is subsequently re-executed, all DATA DIVISION storage for that dynamically-loadable module will once again be in its initial state.

## 6.10. CLOSE

Figure 6-36 - CLOSE Syntax



The CLOSE statement terminates the programs access to the specified file(s) or to the currently mounted reel/unit of the file(s).

1. The CLOSE statement may only be executed against files that have been successfully OPENed.
2. The REEL, UNIT and NO REWIND clauses are available only for ORGANIZATION SEQUENTIAL (either LINE or RECORD BINARY) SEQUENTIAL files. The words REEL and UNIT may be used interchangeably, and reflect a file that is stored on or will be written to multiple removable tapes/disks. Not all systems support such devices, and OpenCOBOL features to manipulate such multiple-unit files may not be functional on your system.
3. The REEL and UNIT phrases are intended for use with files which have had MULTIPLE REEL or MULTIPLE UNIT specified in their SELECT clause. If the run-time system does not recognize multi-unit files, the CLOSE REEL and CLOSE UNIT statements will perform no function.
4. Once a file has been closed, it cannot be accessed again until it has been successfully re-OPENed.
5. A successful CLOSE without REEL or UNIT executed against a file that was OPENed in either OUTPUT or EXTEND mode will write any remaining unwritten record buffers to the file; regardless of OPEN mode, any record locks held for closed files will be released as well. A closed file will then be no longer available for subsequent READ, WRITE, REWRITE, START or DELETE statements until it is once again OPENed.
6. A CLOSE WITH LOCK option will prevent your program from re-opening the file again in the same program execution.
7. A successful CLOSE with REEL or UNIT will write any remaining unwritten record buffers to the closed files and will release any record locks held for those files as well. The currently mounted reel/unit of the file will be dismounted and the next reel/unit requested. The file's status remains OPEN.

## 6.11. COMMIT

Figure 6-37 - COMMIT Syntax

**COMMIT**

The COMMIT statement performs an UNLOCK against every currently-OPEN file.

1. See the UNLOCK statement (section [6.49](#)) for additional details.

## 6.12. COMPUTE

Figure 6-38 - COMPUTE Syntax

```
COMPUTE { identifier-1 [ ROUNDED ] } ... { EQUAL } arithmetic-expression  
= imperative-statement-1  
[ ON SIZE ERROR imperative-statement-1 ]  
[ NOT ON SIZE ERROR imperative-statement-2 ]  
[ END-COMPUTE ]
```

The COMPUTE statement provides a means of easily performing complex arithmetic operations with a single statement, instead of using cumbersome and possibly confusing sequences of ADD, SUBTRACT, MULTIPLY and DIVIDE statements.

1. The word EQUAL and the equals-sign (=) may be used interchangeably.
2. The ON SIZE ERROR, NOT ON SIZE ERROR and ROUNDED clauses are coded and operate the same as the clauses of the same name available to the ADD statement (see section [6.5.1](#)).

## 6.13. CONTINUE

Figure 6-39 - CONTINUE Syntax

### CONTINUE

The CONTINUE statement is a no-operation statement, performing no action whatsoever.

- The CONTINUE statement is often used with IF statements (section [6.24](#)) as a place-holder for conditionally-executed code that is either not yet needed or not yet designed. The following two sentences are equivalent. One uses CONTINUE statements to mark places where code may need to be inserted in the future.

“Minimalist” Coding (Specifying only what is necessary)	Coding With CONTINUE (Documenting where code might be needed someday)
<pre> IF A = 1   IF B = 1     DISPLAY 'A=1 &amp; B=1' END-DISPLAY   END-IF ELSE   IF A = 2     IF B = 2       DISPLAY 'A=2 &amp; B=2' END-DISPLAY     END-IF   END-IF END-IF </pre>	<pre> IF A = 1   IF B = 1     DISPLAY 'A=1 &amp; B=1' END-DISPLAY   ELSE     CONTINUE   END-IF ELSE   IF A = 2     IF B = 2       DISPLAY 'A=2 &amp; B=2' END-DISPLAY     ELSE       CONTINUE     END-IF   ELSE     CONTINUE   END-IF END-IF </pre>

Coding such as this is generally a matter of personal preference or site coding standards. There is no difference in the object code generated by the two, so there isn't a run-time efficiency issue (just one of “coding efficiency”).

- Another IF-statement usage for CONTINUE is to avoid the use of NOT in the conditional expression coded on the IF statement. This too is a personal and/or site standards issue. Here's an example:

Without CONTINUE	With CONTINUE
<pre> IF Action-Flag NOT = 'I' AND 'U'   DISPLAY 'Invalid Action-Flag'   EXIT PARAGRAPH END-IF </pre>	<pre> IF Action-Flag = 'I' OR 'U'   CONTINUE ELSE   DISPLAY 'Invalid Action-Flag'   EXIT PARAGRAPH END-IF </pre>

Because of the way COBOL (OpenCOBOL included) handles the abbreviation of conditional expressions, the conditional expression in the left-hand box is actually a short-hand version of the (not-so-intuitive):

```
IF Action-Flag NOT = 'I' AND Action-Flag NOT = 'U'
```

Many programmers would have coded the “IF” (incorrectly) as “IF Action-Flag NOT = 'I' OR 'U'” – this is sure to cause run-time problems.

This causes many programmers to consider the code in the right-hand box to be more readable, even though it is a little longer.

## 6.14. DELETE

Figure 6-40 - DELETE Syntax

```
DELETE file-name RECORD  
    [ INVALID KEY imperative-statement-1 ]  
    [ NOT INVALID KEY imperative-statement-2 ]  
[ END-DELETE ]
```

The DELETE statement logically deletes a record from an ORGANIZATION RELATIVE or ORGANIZATION INDEXED file.

1. The INVALID KEY and NOT INVALID KEY clauses cannot be specified for a file whose ACCESS MODE IS SEQUENTIAL.
2. The INVALID KEY clause provides the ability to react to a DELETE failure, while the NOT INVALID KEY clause gives the program the capability of specifying actions to be taken if the DELETE succeeds.
3. The ORGANIZATION of *file-name* must be RELATIVE or INDEXED.
4. For RELATIVE or INDEXED files in the SEQUENTIAL access mode, the last input-output statement executed for *file-name* prior to the execution of the DELETE statement must have been a successfully executed READ statement. That READ will therefore identify the record to be deleted.
5. If *file-name* is a RELATIVE file whose ACCESS MODE is either RANDOM or DYNAMIC, the record to be deleted is the one whose relative record number is currently the value of the field specified as the files RELATIVE KEY.
6. If *file-name* is an INDEXED file whose ACCESS MODE is RANDOM or DYNAMIC, the record to be deleted is the one whose primary key is currently the value of the field specified as the RECORD KEY of the file.
7. An INVALID KEY condition will exist, and can be dealt with via the INVALID KEY clause, if the record specified to be deleted by the RELATIVE KEY or RECORD KEY value does not exist in an access mode RANDOM or DYNAMIC file. This is a condition that cannot exist for ACCESS MODE SEQUENTIAL files because of rule #4. DELETE failures on ACCESS MODE SEQUENTIAL files can only be "handled" via DECLARATIVES.

## 6.15. DISPLAY

### 6.15.1. DISPLAY Format 1 – Upon Console

Figure 6-41 - DISPLAY (Upon Console) Syntax

```

DISPLAY { identifier-1 } ...
           { literal-1 }
           [ UPON mnemonic-name ]
           [ WITH NO ADVANCING ]
           [ exception-handler ]
           [ END-DISPLAY ]

```

This format of the DISPLAY statement displays the specified identifier contents and/or literal values on the shell or console window from which the program was started.

The displayed text will appear starting in column 1 of the next available line. If all screen lines have previously had text displayed to them, the screen will scroll upward one line and the text will appear on the last line.

1. If no UPON clause is specified, UPON CONSOLE will be assumed.
2. The specified mnemonic-name must be CONSOLE, CRT, PRINTER or any user-defined mnemonic name associated with one of these devices within the SPECIAL-NAMES paragraph (see section 4.1.4). All such mnemonics specify the same destination – the shell (UNIX) or console (Windows) window from which the program was run.
3. The NO ADVANCING clause, if used, will suppress the normal carriage-return / line-feed sequence that normally is added to the end of any console display.

### 6.15.2. DISPLAY Format 2 – Access Command-Line Arguments

Figure 6-42 - DISPLAY (Access Command-line Arguments) Syntax

```

DISPLAY { identifier-1 } ... UPON { ARGUMENT-NUMBER }
           { literal-1 }           { COMMAND-LINE }
           [ exception-handler ]
           [ END-DISPLAY ]

```

This form of the DISPLAY statement may be used to specify the command-line argument number to be retrieved by a subsequent ACCEPT or to specify a new value for the command-line arguments themselves.

1. Executing a DISPLAY ... UPON COMMAND-LINE will influence subsequent ACCEPT ... FROM COMMAND-LINE statements (which will then return the DISPLAYed value), but will not influence subsequent ACCEPT ... FROM ARGUMENT-VALUE statements – these will continue to return the original program execution parameters.

### 6.15.3. DISPLAY Format 3 – Access or Set Environment Variables

Figure 6-43 - DISPLAY (Access / Set Environment Variables) Syntax

```

DISPLAY { identifier-1 } ... UPON { ENVIRONMENT-VALUE }
           { literal-1 }           { ENVIRONMENT-NAME }
           [ exception-handler ]
           [ END-DISPLAY ]

```

This form of the DISPLAY statement can be used to create or modify environment variables.

1. To create or change an environment variable will require two DISPLAY statements, which must be executed in the following sequence:



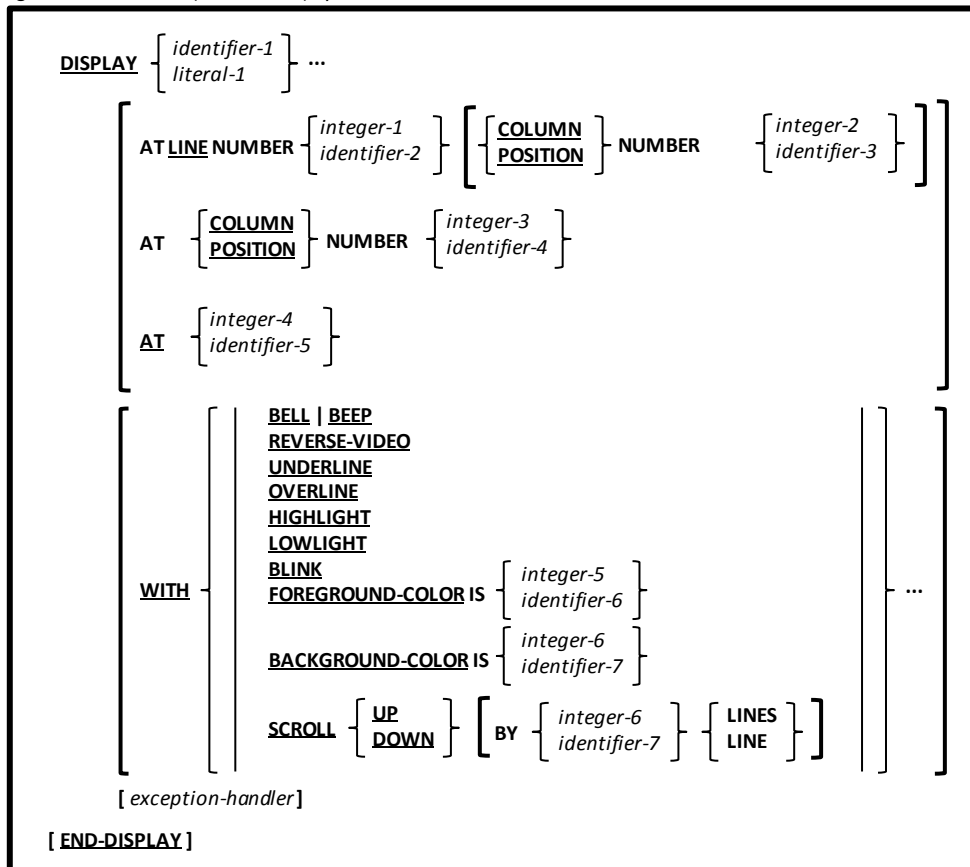
**DISPLAY**  
*environment-variable-name* UPON ENVIRONMENT-NAME  
**END-DISPLAY**

**DISPLAY**  
*environment-variable-value* UPON ENVIRONMENT-VALUE  
**END-DISPLAY**

- Environment variables created or changed from within OpenCOBOL programs will be available to any sub-shell processes spawned by that program (i.e. CALL "SYSTEM") but will not be known to the shell or console window that started the OpenCOBOL program.
- Consider using SET ENVIRONMENT (section [6.40.1](#)) in lieu of DISPLAY to set environment variables as it is much simpler.

### 6.15.4. DISPLAY Format 4 – Screen Data

Figure 6-44 - DISPLAY (Screen Data) Syntax



This format of the DISPLAY statement presents data onto a formatted screen.

- If *identifier-1* is defined in the SCREEN SECTION, all cursor positioning (AT) and attribute specifications (WITH) are taken from the SCREEN SECTION definition and anything specified on the DISPLAY will be ignored. Use the AT and WITH options only when DISPLAYing a data item that was not defined in the SCREEN SECTION.
- The various AT clauses provide a means of positioning the cursor to a specific spot on the screen before the data is presented to the screen. The *literal-3* / *identifier-4* value must be a four-digit value with the 1<sup>st</sup> two digits indicating the line where the cursor should be positioned and the last two digits being the column.
- See section [6.4.4](#) (ACCEPT ACCEPT Format 4 – Retrieve Screen Data) for a description of the SCROLL option.
- The remaining "WITH" options were described in section [0](#).

### 6.15.5. DISPLAY Exception Handling

Figure 6-45 - Exception Handling (DISPLAY) Syntax

<b><u>ON EXCEPTION</u></b> <i>imperative-statement-1</i>
<b><u>NOT ON EXCEPTION</u></b> <i>imperative-statement-2</i>

The EXCEPTION and NOT EXCEPTION clauses available on all formats of the DISPLAY verb allow you to specify code to be executed specifically upon the failure or success (respectively) of the DISPLAY. Since DISPLAY does not set any sort of return code or status flag, this is the only way to detect success and failure.

## 6.16. DIVIDE

### 6.16.1. DIVIDE Format 1 – DIVIDE INTO

Figure 6-46 - DIVIDE INTO Syntax

```

DIVIDE { identifier-1 / literal-1 } INTO { identifier-2 [ ROUNDED ] } ...
      [ ON SIZE ERROR imperative-statement-1 ]
      [ NOT ON SIZE ERROR imperative-statement-2 ]
      [ END-DIVIDE ]

```

This format of DIVIDE will divide a specified value into one or more data items, replacing each of those data items with the result of its old value divided by the *identifier-1 / literal-1* value. Any remainder calculated as a result of the division is discarded.

1. *Identifier-1* and *identifier-2* must be numeric unedited data items.
2. *Literal-1* must be a numeric literal.
3. The ON SIZE ERROR, NOT ON SIZE ERROR and ROUNDED clauses are coded and operate the same as the clauses of the same name available to the ADD statement (see section 6.5).
4. If the *identifier-1 / literal-1* value is zero, a SIZE ERROR condition will result. A SIZE ERROR will also occur if the result of the division requires more digits of precision to the left of a decimal-point than are available in any of the receiving fields.

### 6.16.2. DIVIDE Format 2 – DIVIDE INTO GIVING

Figure 6-47 - DIVIDE INTO GIVING Syntax

```

DIVIDE { identifier-1 / literal-1 } INTO { identifier-2 / literal-2 } GIVING { identifier-3 [ ROUNDED ] } ...
      [ ON SIZE ERROR imperative-statement-1 ]
      [ NOT ON SIZE ERROR imperative-statement-2 ]
      [ END-DIVIDE ]

```

This format of DIVIDE will divide a specified value (*identifier-1 / literal-1*) into another value (*identifier-2 / literal-2*) and will then replace the contents of one or more receiving data items (*identifier-3 ...*) with the results of that division. Any remainder calculated as a result of the division is discarded.

1. *Identifier-1* and *identifier-2* must be numeric unedited data items.
2. *Identifier-3* must be a numeric (edited or unedited) data item.
3. *Literal-1* and *literal-2* must be numeric literals.
4. The ON SIZE ERROR, NOT ON SIZE ERROR and ROUNDED clauses are coded and operate the same as the clauses of the same name available to the ADD statement (see section 6.5).
5. If the *identifier-1 / literal-1* value is zero, a SIZE ERROR condition will result. A SIZE ERROR will also occur if the result of the division requires more digits of precision to the left of a decimal-point than are available in any of the receiving fields.

### 6.16.3. DIVIDE Format 3 – DIVIDE BY GIVING

Figure 6-48 - DIVIDE BY GIVING Syntax

```

DIVIDE { identifier-1 / literal-1 } BY { identifier-2 / literal-2 } GIVING { identifier-3 [ ROUNDED ] } ...
      [ ON SIZE ERROR imperative-statement-1 ]
      [ NOT ON SIZE ERROR imperative-statement-2 ]
[ END-DIVIDE ]

```

This format of DIVIDE will divide a specified value (*identifier-1* / *literal-1*) by another value (*identifier-2* / *literal-2*) and will then replace the contents of one or more receiving data items (*identifier-3* ...) with the results of that division. Any remainder calculated as a result of the division is discarded.

1. *Identifier-1* and *identifier-2* must be numeric unedited data items.
2. *Identifier-3* must be a numeric (edited or unedited) data item.
3. *Literal-1* and *literal-2* must be numeric literals.
4. The ON SIZE ERROR, NOT ON SIZE ERROR and ROUNDED clauses are coded and operate the same as the clauses of the same name available to the ADD statement (see section 6.5).
5. If the *identifier-2* / *literal-2* value is zero, a SIZE ERROR condition will result. A SIZE ERROR will also occur if the result of the division requires more digits of precision to the left of a decimal-point than are available in any of the receiving fields.

### 6.16.4. DIVIDE Format 4 – DIVIDE INTO REMAINDER

Figure 6-49 - DIVIDE INTO REMAINDER Syntax

```

DIVIDE { identifier-1 / literal-1 } INTO { identifier-2 / literal-2 } GIVING identifier-3 [ ROUNDED ]
      REMAINDER identifier-4
      [ ON SIZE ERROR imperative-statement-1 ]
      [ NOT ON SIZE ERROR imperative-statement-2 ]
[ END-DIVIDE ]

```

This format of DIVIDE will divide a specified value (*identifier-1* / *literal-1*) into another value (*identifier-2* / *literal-2*) and will then replace the contents of a single data item (*identifier-3*) with the results of that division. Any remainder calculated as a result of the division is saved in *identifier-4*.

1. *Identifier-1* and *identifier-2* must be numeric unedited data items.
2. *Identifier-3* and *identifier-4* must be numeric (edited or unedited) data items.
3. *Literal-1* and *literal-2* must be numeric literals.
4. The ON SIZE ERROR, NOT ON SIZE ERROR and ROUNDED clauses are coded and operate the same as the clauses of the same name available to the ADD statement (see section 6.5).
5. If the *identifier-1* / *literal-1* value is zero, a SIZE ERROR condition will result. A SIZE ERROR will also occur if the result of the division requires more digits of precision to the left of a decimal-point than are available in any of the receiving fields.

### 6.16.5. DIVIDE Format 5 – DIVIDE BY REMAINDER

Figure 6-50 - DIVIDE BY REMAINDER Syntax

```

DIVIDE { identifier-1 } BY { identifier-2 } GIVING identifier-3 [ ROUNDED ]
      REMAINDER identifier-4
      [ ON SIZE ERROR imperative-statement-1 ]
      [ NOT ON SIZE ERROR imperative-statement-2 ]
      [ END-DIVIDE ]

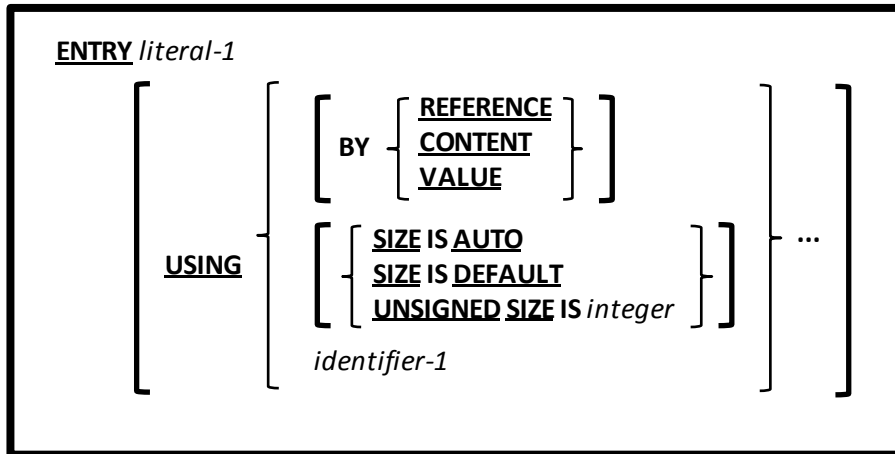
```

This format of DIVIDE will divide a specified value (*identifier-1 / literal-1*) by another value (*identifier-2 / literal-2*) and will then replace the contents of a single receiving data item (*identifier-3*) with the results of that division. Any remainder calculated as a result of the division is saved in *identifier-4*.

1. *Identifier-1* and *identifier-2* must be numeric unedited data items.
2. *Identifier-3* and *identifier-4* must be umeric (edited or unedited) data items.
3. *Literal-1* and *literal-2* must be numeric literals.
4. The ON SIZE ERROR, NOT ON SIZE ERROR and ROUNDED clauses are coded and operate the same as the clauses of the same name available to the ADD statement (see section 6.5).
5. If the *identifier-2 / literal-2* value is zero, a SIZE ERROR condition will result. A SIZE ERROR will also occur if the result of the division requires more digits of precision to the left of a decimal-point than are available in any of the receiving fields.

## 6.17. ENTRY

Figure 6-51 - ENTRY Syntax

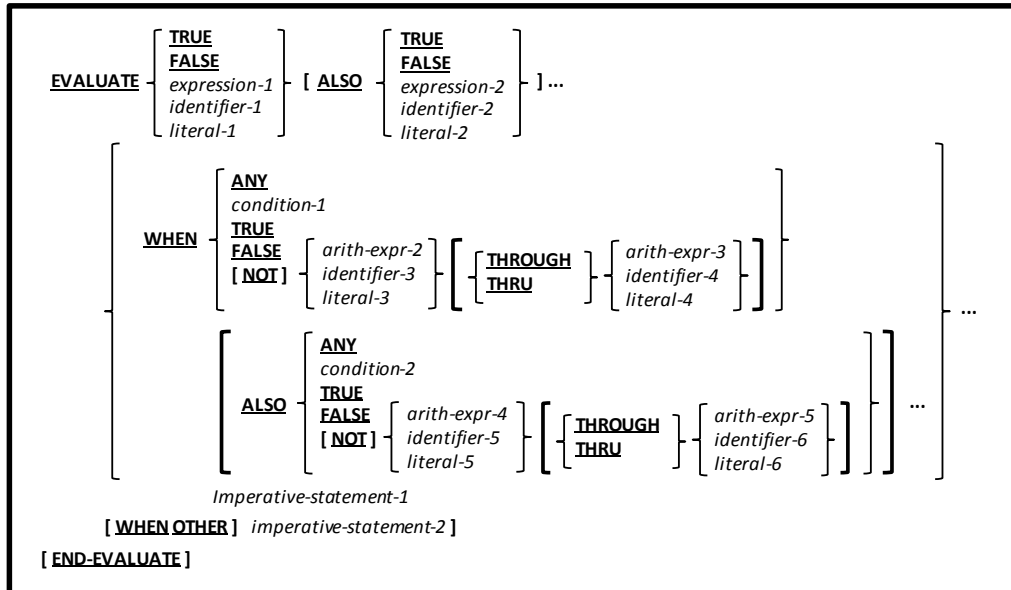


The ENTRY statement is used to define an alternate *entry-point* into a subroutine, along with the arguments that subroutine will be expecting.

1. You may not use an ENTRY statement in a nested subprogram (see section [2.1](#)).
2. The USING clause matches up against the USING clause of any CALL statements that will be invoking the subroutine.
3. The *literal-1* value will specify the entry-point name of the subroutine. It must be specified exactly on CALL statements (with regard to the use of upper- and lower-case letters) as it is specified on the ENTRY statement.

## 6.18. EVALUATE

Figure 6-52 - EVALUATE Syntax



The EVALUATE statement provides a means of defining processing that should take place under a variety of related circumstances

1. The reserved words THRU and THROUGH may be used interchangeably.
2. When using THROUGH, the values connected by a THROUGH clause (*arith-expr-n, identifier-n and/or literal-n*) must be the same class. For example:

**Legal:**

**(3 + Years-Of-Service) THROUGH 99**  
**"A" THRU "Z"**  
**X'00' THRU X'1F'**  
**15.7 THROUGH 19.4**

**Not Legal:**

**0 THRU "A"**  
**Last-Name THRU Zip-Code** (Assuming Last-Name is PIC X and Zip-Code is PIC 9)

3. The values specified after the EVALUATE verb but before the first WHEN clause are known as *selection subjects* while the values specified on each WHEN clause are known as *selection objects*.
4. Each WHEN clause must have the same number of selection objects as the EVALUATE verb has selection subjects.
5. Each EVALUATE clause's selection subject will be tested for equality to each WHEN clauses corresponding selection object.
6. The first WHEN clause found where all such equality tests described in rule #5 result in TRUE results will be the one whose imperative statement will be executed.
7. If none of the WHEN clauses have all such equality tests as described in rule #5 resulting in TRUE results then the imperative statement associated with the WHEN OTHER clause (*imperative-statement-2*) will be executed. If there is no WHEN OTHER clause, control will simply fall into the next statement following the EVALUATE statement.
8. Once a WHEN or WHEN OTHER clause's imperative statement has been executed, control will fall into the next statement following the EVALUATE statement.
9. Using a selection object of ANY will cause an automatic match with whatever selection subject the ANY was matched against.

Here's a "case study" that will illustrate the usefulness of the EVALUATE statement. A program is being developed to compute the interest to be paid on accounts based upon their average daily balance [ADB]. The business rules for this process are as follows:

1. Interest-bearing checking accounts will receive no interest if their ADB is less than \$1000. Interest-bearing checking accounts with an ADB \$1000 to \$1499.99 will receive 1% of the ADB as interest. Those with an ADB of \$1500 or more will receive 1.5% of the ADB as interest.

2. Statement savings accounts will receive 1.5% interest on an ADB up to \$10000 and 1.75% for any ADB amounts over \$10000.
3. Platinum savings accounts receive 2% interest on their ADB, regardless of average balance amounts.
4. No other types of accounts receive interest.

Here's a sample OpenCOBOL program that can be used to test an "EVALUATE" implementation of these business rules. Output from the program is shown in the inset.

Figure 6-53 - An EVALUATE Demo Program

```

>>SOURCE FORMAT FREE
IDENTIFICATION DIVISION.
PROGRAM-ID. evaldemo.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 Account-Type          PIC X(1).
   88 Interest-Bearing-Checking VALUE 'c'.
   88 Statement-Savings VALUE 's'.
   88 Platinum-Savings VALUE 'p'.
01 ADB-Char              PIC X(10).
01 Ave-Daily-Balance     PIC 9(7)V99.
01 Formatted-Amount      PIC Z(6)9.99.
01 Interest-Amount       PIC 9(7)V99.
PROCEDURE DIVISION.
000-Main.
  PERFORM FOREVER
    DISPLAY "Enter Account Type (c,s,p,other): " WITH NO ADVANCING
    ACCEPT Account-Type
    IF Account-Type = SPACES
      STOP RUN
    END-IF
    DISPLAY "Enter Ave Daily Balance (nnnnnnn.nn): " WITH NO ADVANCING
    ACCEPT ADB-Char
    MOVE FUNCTION NUMVAL(ADB-Char) TO Ave-Daily-Balance
    EVALUATE TRUE ALSO Ave-Daily-Balance
      WHEN Interest-Bearing-Checking ALSO 0.00 THRU 999.99
        MOVE 0 TO Interest-Amount
      WHEN Interest-Bearing-Checking ALSO 1000.00 THRU 1499.99
        COMPUTE Interest-Amount ROUNDED = 0.01 * Ave-Daily-Balance
      WHEN Interest-Bearing-Checking ALSO ANY
        COMPUTE Interest-Amount ROUNDED = 0.015 * Ave-Daily-Balance
      WHEN Statement-Savings ALSO 0.00 THRU 10000.00
        COMPUTE Interest-Amount ROUNDED = 0.015 * Ave-Daily-Balance
      WHEN Statement-Savings ALSO ANY
        COMPUTE Interest-Amount ROUNDED = 0.015 * Ave-Daily-Balance
          + 0.175 * (Ave-Daily-Balance - 10000)
      WHEN Platinum-Savings ALSO ANY
        COMPUTE Interest-Amount ROUNDED = 0.020 * Ave-Daily-Balance
      WHEN OTHER
        MOVE 0 TO Interest-Amount
    END-EVALUATE
    MOVE Interest-Amount TO Formatted-Amount
    DISPLAY "Accrued Interest = " Formatted-Amount
  END-PERFORM
  .

```

```

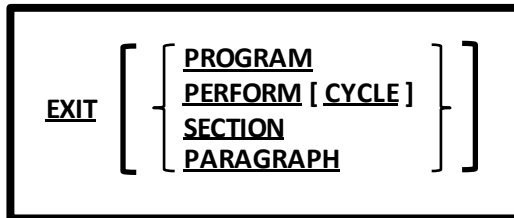
Enter Account Type (c,s,p,other): c
Enter Ave Daily Balance (nnnnnnn.nn): 250
Accrued Interest = 0.00
Enter Account Type (c,s,p,other): c
Enter Ave Daily Balance (nnnnnnn.nn): 1250
Accrued Interest = 12.50
Enter Account Type (c,s,p,other): c
Enter Ave Daily Balance (nnnnnnn.nn): 1899.99
Accrued Interest = 28.50
Enter Account Type (c,s,p,other): s
Enter Ave Daily Balance (nnnnnnn.nn): 22000.00
Accrued Interest = 2430.00
Enter Account Type (c,s,p,other): p
Enter Ave Daily Balance (nnnnnnn.nn): 1.98
Accrued Interest = 0.04

```



## 6.19. EXIT

Figure 6-54 - EXIT Syntax



1. When used without any of the optional clauses, the “EXIT” statement simply provides a common “GO TO” end point for a series of procedures. [Figure 6-55](#) illustrates the use of the EXIT statement.
2. When an EXIT statement is used, it must be the only statement in the paragraph in which it occurs.
3. The EXIT statement is a no-operation statement (much like the CONTINUE statement).

4. An EXIT PARAGRAPH statement transfers control to a point immediately past the end of the current paragraph, while an EXIT SECTION statement causes control to pass to point immediately past the last paragraph in the current section.

If the EXIT PARAGRAPH or EXIT SECTION resides in a paragraph within the scope of a procedural PERFORM (section [6.33.1](#)), control will be returned back to the PERFORM for evaluation of any TIMES, VARYING and/or UNTIL clauses. If the EXIT PARAGRAPH or EXIT SECTION resides outside the scope of a procedural PERFORM, control simply transfers to the first executable statement in the next paragraph (EXIT PARAGRAPH) or section (EXIT SECTION).

[Figure 6-56](#) shows how the example shown in [Figure 6-55](#) could have been coded without a GO TO by utilizing an EXIT PARAGRAPH statement.

5. The EXIT PERFORM and EXIT PERFORM CYCLE statements are intended to be used in conjunction with an inline PERFORM statement (section [6.33.2](#)).
6. An EXIT PERFORM CYCLE will terminate the current iteration of the inline PERFORM, giving control to any TIMES, VARYING and/or UNTIL clauses for them to determine if another cycle needs to be performed.
7. An EXIT PERFORM will terminate the inline PERFORM outright, transferring control to the first statement following the PERFORM. [Figure 6-57](#) shows the final modification to the [Figure 6-55](#) example; by using Inline PERFORM and EXIT PERFORM statements we can really streamline processing.

The EXIT statement is a multi-purpose statement; it may provide a common end point for a series of procedures, exit an inline PERFORM, a paragraph or a section or it may mark the logical end of a called program.

Figure 6-55 - Using the EXIT Statement

```

01 Switches.
   05 Input-File-Switch   PIC X(1).
   88 EOF-On-Input-File  VALUE 'Y' FALSE 'N'.
.
.
.
   SET EOF-On-Input-File TO FALSE.
   PERFORM 100-Process-A-Transaction
   UNTIL EOF-On-Input-File.
.
.
.
100-Process-A-Transaction.
   READ Input-File AT END
   SET EOF-On-Input-File TO TRUE
   EXIT PARAGRAPH.
   IF Input-Rec of Input-File = SPACES
   EXIT PARAGRAPH.  *> IGNORE BLANK RECORDS!
   process the record just read

```

Figure 6-56 - Using EXIT PARAGRAPH

```

01 Switches.
   05 Input-File-Switch   PIC X(1).
   88 EOF-On-Input-File  VALUE 'Y' FALSE 'N'.
.
.
.
   SET EOF-On-Input-File TO FALSE.
   PERFORM 100-Process-A-Transaction
   UNTIL EOF-On-Input-File.
.
.
.
100-Process-A-Transaction.
   READ Input-File AT END
   SET EOF-On-Input-File TO TRUE
   EXIT PARAGRAPH.
   IF Input-Rec of Input-File = SPACES
   EXIT PARAGRAPH.  *> IGNORE BLANK RECORDS!
   process the record just read

```

Figure 6-57 - Using the EXIT PERFORM Statement

```

PERFORM FOREVER
   READ Input-File AT END
   EXIT PERFORM
   END-READ
   IF Input-Rec of Input-File = SPACES
   EXIT PERFORM CYCLE  *> IGNORE BLANK RECORDS!
   END-IF
   process the record just read
END PERFORM

```

8. Finally, the EXIT PROGRAM statement terminates the execution of subroutine (i.e. a program that has been CALLED by another), returning to the CALLing program at the statement following the CALL. If executed by a main program, the EXIT PROGRAM statement is non-functional. The COBOL2002 standard has made a common extension to the COBOL language - the GOBACK statement (section [6.22](#)) - standard; the GOBACK statement should be strongly considered as the preferred alternative to EXIT PROGRAM for new programs.

## 6.20. FREE

Figure 6-58 - FREE Syntax

```
FREE { [ ADDRESS OF ] identifier-1 }...
```

The FREE statement releases memory previously allocated to the program by the ALLOCATE statement (section [6.6](#)).

1. *Identifier-1* must be a USAGE POINTER data item or an 01-level data item with the BASED attribute.
2. If *identifier-1* is a USAGE POINTER data item and it contains a valid address, the FREE statement will release the memory block the pointer references. In addition, any BASED data items that the pointer was used to provide an address for will become un-based and therefore un-usable. If *identifier-1* did not contain a valid address, no action will be taken.
3. If *identifier-1* is a BASED data item and that data item is currently based (meaning it currently has memory allocated for it), its memory is released and *identifier-1* will become un-based and therefore un-usable. If *identifier-1* was not based, no action will be taken.
4. The ADDRESS OF clause adds no special function to the FREE statement.

## 6.21. GENERATE

Figure 6-59 - GENERATE Syntax

```
GENERATE [ identifier-1  
          report-name-1 ]
```

Although syntactically recognized by the OpenCOBOL compiler, the GENERATE statement is non-functional because the RWCS (COBOL Report Writer) is not currently supported by OpenCOBOL.

## 6.22. GOBACK

Figure 6-60 - GOBACK Syntax

**GOBACK**

The GOBACK statement is used to logically terminate an executing program.

1. If executed within a subroutine (i.e. a CALLED program), GOBACK will transfer control back to the CALLing program, specifically to the statement following the CALL.
2. If executed within a main program, GOBACK will act as a STOP RUN statement (section [6.43](#)).

## 6.23. GO TO

### 6.23.1. GO TO Format 1 – Simple GO TO

Figure 6-61 - Simple GOTO Syntax

```
GO TO procedure-name
```

This form of the GO TO statement unconditionally transfers control in a program to the specified *procedure-name*.

1. If the specified procedure name is a SECTION, control will transfer to the first paragraph in that section.

### 6.23.2. GO TO Format 2 – GO TO DEPENDING ON

Figure 6-62 - GOTO DEPENDING ON Syntax

```
GO TO procedure-name-1 ...  
DEPENDING ON identifier-1
```

This form of the GO TO statement will transfer control to any one of a number of specified procedure names depending on the numeric value of the identifier specified on the statement.

1. The PICTURE and/or USAGE of the specified *identifier-1* must be such as to define it as a numeric, unedited, preferably unsigned integer data item.
2. If the value of *identifier-1* has the value 1, control will be transferred to the 1<sup>st</sup> specified procedure name. If the value is 2, control will transfer to the 2<sup>nd</sup> procedure name, etc.
3. If the value of *identifier-1* is less than 1 or exceeds the total number of procedure names specified on the GO TO statement, control will simply fall thru into the next statement following the GO TO.
4. The following chart shows how GO TO DEPENDING ON may be used in a real application situation, and compares it against the two alternatives – IF and EVALUATE.

Figure 6-63 - GOTO DEPENDING ON vs IF vs EVALUATE

GOTO DEPENDING ON	IF	EVALUATE
<pre>GO TO PROCESS-ACCT-TYPE-1 PROCESS-ACCT-TYPE-2 PROCESS-ACCT-TYPE-3 DEPENDING ON ACCT-TYPE. Code to handle invalid account type GO TO DONE-WITH-ACCT-TYPE. PROCESS-ACCT-TYPE-1. Code to handle account type 1 GO TO DONE-WITH-ACCT-TYPE. PROCESS-ACCT-TYPE-2. Code to handle account type 2 GO TO DONE-WITH-ACCT-TYPE. PROCESS-ACCT-TYPE-3. Code to handle account type 3 DONE-WITH-ACCT-TYPE.</pre>	<pre>IF ACCT-TYPE = 1 Code to handle account type 1 ELSE IF ACCT-TYPE = 2 Code to handle account type 2 ELSE IF ACCT-TYPE = 3 Code to handle account type 3 ELSE Code to handle invalid account type END-IF.</pre>	<pre>EVALUATE ACCT-TYPE WHEN 1 Code to handle account type 1 WHEN 2 Code to handle account type 2 WHEN 3 Code to handle account type 3 WHEN OTHER Code to handle invalid account type END-EVALUATE.</pre>

There is no question that “modern programming philosophy” would prefer the EVALUATE approach. An interesting note is that the code generated by the IF and EVALUATE approaches is virtually identical. Sometimes, NEW, while it might be considered BETTER, doesn’t always mean DIFFERENT!

## 6.24. IF

Figure 6-64 - IF Syntax

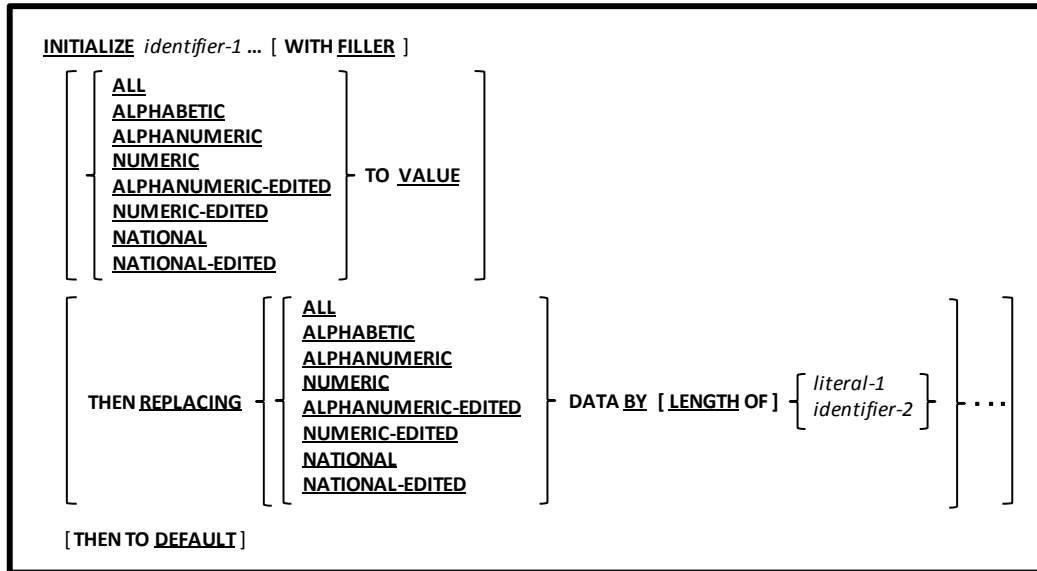
```
IF conditional-expression THEN  
    imperative-statement-1  
[ ELSE imperative-statement-2 ]  
[ END-IF ]
```

The IF statement is used to conditionally execute a single imperative statement or to select one of two different imperative statements based upon the TRUE/FALSE value of a conditional expression.

1. If *conditional-expression* evaluates to TRUE, *imperative-statement-1* will be executed regardless of whether or not an ELSE clause is present. Once *imperative-statement-1* has been executed, control falls into the first statement following the END-IF or to the first statement of the next sentence if there is no END-IF clause.
2. If the optional ELSE clause is present and *conditional-expression-1* evaluates to FALSE, then (and only then) *imperative-statement-2* will be executed. Once *imperative-statement-2* has been executed, control falls into the first statement following the END-IF or to the first statement of the next sentence if there is no END-IF clause.
3. See section [6.1.5](#) for a discussion (and examples) of how periods (.) and END-IF statements are both similar to and different from each other in the way they are capable of ending the scope of an IF statement.

## 6.25. INITIALIZE

Figure 6-65 - INITIALIZE Syntax



The INITIALIZE statement sets the elementary item(s) specified as *identifier-1*, or those elementary items subordinate to group items specified as *identifier-1* to specific values.

- The list of data items eligible to be set to new values by this statement is:
  - Every elementary item specified as *identifier-1 ...*, PLUS...
  - Every elementary item defined subordinate to every group item specified as *identifier-1 ...*, with the following exceptions:
    - USAGE INDEX items are excluded.
    - Items with a REDEFINES as part of their definition are excluded; this means that items subordinate to them are excluded as well. The *identifier-1* items themselves may have a REDEFINES and may be subordinate to an item that has a REDEFINES, however.

This list is referred to as the list of *receiving fields*.

- None of the *identifier-1* fields may have the OCCUR DEPENDING ON clause (section 5.3) in their definition nor may any items subordinate to the *identifier-1* fields have an OCCURS DEPENDING ON.
- The optional WITH FILLER clause, if present, will allow FILLER items to be retained in the list of receiving fields (otherwise they will be excluded).
- If no TO VALUE or REPLACING clauses are specified, a DEFAULT clause will be assumed.
- If the optional REPLACING clause is specified, every possible MOVE of the sending field to every possible receiving field must be legal in order for the INITIALIZE to be syntactically acceptable to the compiler.
- Initialization for each receiving field takes place by applying the first of the following rules that apply to the field:
  - If a TO VALUE clause exists, does the receiving field qualify as one of the data categories listed on the clause? If it does, the data item will be initialized to its VALUE clause value.
  - If a REPLACING clause exists, does the receiving field qualify as one of the data categories listed on the clause? If it does, the receiving field will be initialized to the specified sending field value.
  - If a DEFAULT clause exists, initialize the field to a value appropriate to its USAGE (Alphanumeric and Numeric initialized to SPACES, Pointer and Program-Pointer initialized to NULL, all numeric and numeric-edited initialized to ZERO).



## 6.26. INITIATE

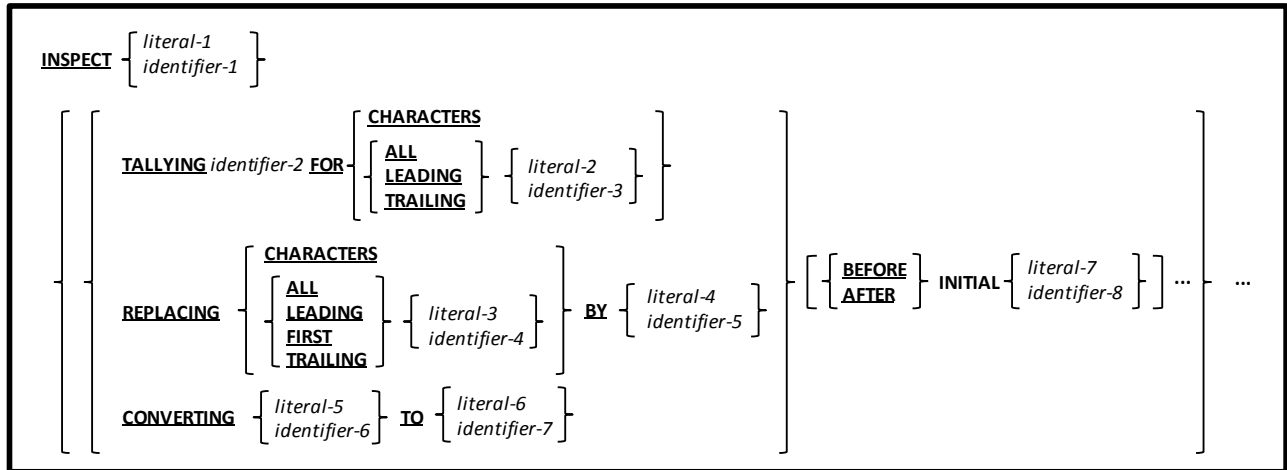
Figure 6-66 - INITIATE Syntax

```
INITIATE report-name-1 ...
```

Although syntactically recognized by the OpenCOBOL compiler, the INITIATE statement is non-functional because the RWCS (COBOL Report Writer) is not currently supported by OpenCOBOL.

## 6.27. INSPECT

Figure 6-67 - INSPECT Syntax



This statement is used to perform various counting or data-alteration operations against strings.

1. *Identifier-1* and *literal-1* must be explicitly or implicitly defined as alphanumeric USAGE DISPLAY data. *Identifier-1* may be a group item.
2. The specification of *literal-1* prevents the use of either the REPLACING or CONVERTING clauses.
3. To avoid confusion and/or conflicts, the TALLYING, REPLACING and CONVERTING clauses will be executed in the order they are coded.

Additional rules for INSPECT vary, depending upon the clause(s) specified.

### TALLYING clause rules:

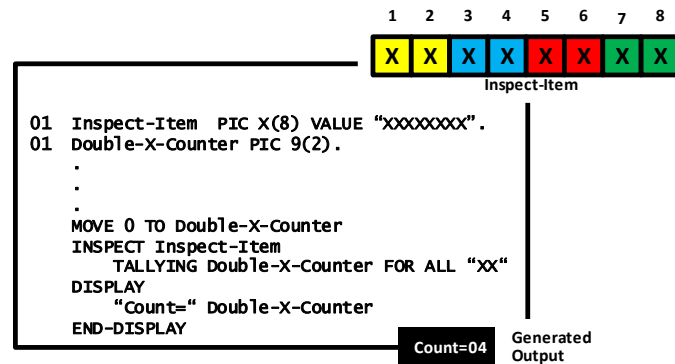
The purpose of the TALLYING clause is to count how many occurrences of a string appear within *identifier-1* or *literal-1*.

1. *Identifier-2* must be an unedited numeric item.
2. *Identifier-3* and *literal-2* must be explicitly or implicitly defined as alphanumeric USAGE DISPLAY data. *Identifier-3* may be a group item.
3. *Identifier-2* will be incremented by 1 each time the target string being searched for is found in *identifier-1*. The target string will be:
  - a. Any single character if the CHARACTERS option is used; this form basically just counts total characters
  - b. ALL, all LEADING, only the FIRST or all TRAILING occurrences of *Identifier-3* or *literal-2*.
4. Normally the entire *literal-1* or *identifier-1* string will be scanned. This behavior may be modified, however, using the optional BEFORE|AFTER clause to specify a starting and/or ending point based upon data found in the string being scanned.

- Once an occurrence of the target string is found and TALLYed, the INSPECT TALLYING process will resume from the end of the found occurrence. This prevents the possibility of counting overlapping occurrences.

The example shows an 8-character item whose value is "XXXXXXXX" used as the object of an INSPECT TALLYING that is looking for "XX" occurrences:

Figure 6-68 - An INSPECT TALLYING Example



Only four (4) "XX" occurrences were found. Character positions 2-3, 4-5 and 6-7 – even though they are "XX" occurrences – weren't counted because they overlapped other occurrences.

### REPLACING clause rules:

The purpose of the REPLACING clause is to replace occurrences of a substring within a string with a different substring of the same length. If you need to replace one or more substrings with others of a different length, consider using the SUBSTITUTE intrinsic function (section [6.1.7](#)).

- Identifier-4* and *literal-3* must be explicitly or implicitly defined as alphanumeric USAGE DISPLAY data. *Identifier-4* may be a group item.
- Identifier-5* and *literal-4* must be explicitly or implicitly defined as alphanumeric USAGE DISPLAY data. *Identifier-5* may be a group item.
- Identifier-4 / literal-3* must be the same length as *identifier-5 / literal-4*.
- The substring specified before the "BY" will be referred to as the *target string*. The substring specified after the "BY" will be referred to as the *replacement string*.
- Target strings are identified as:
  - Any sequence of characters as long as the length of the replacement string if the CHARACTERS option is used
  - ALL, all LEADING, only the FIRST or all TRAILING occurrences of *Identifier-4* or *literal-3*.
- Normally the entire *identifier-1* string will be scanned. This behavior may be modified, however, using the optional BEFORE|AFTER clause to specify a starting and/or ending point based upon data found in the string being scanned.
- Once an occurrence of the target string is found and replaced, the INSPECT REPLACING process will resume from the end of the found occurrence. This prevents the possibility of replacing overlapping occurrences. This is very similar to how TALLYING handled the possibility of overlapping occurrences.

### CONVERTING clause rules:

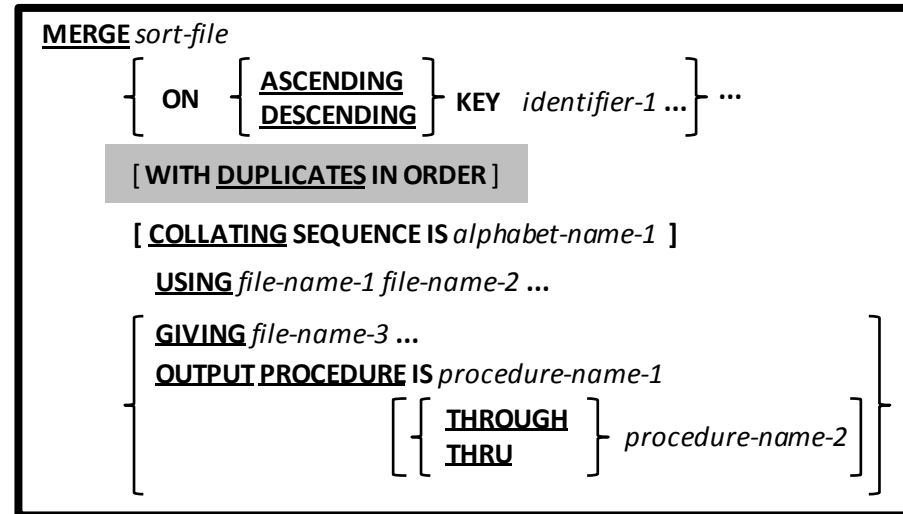
The purpose of the CONVERTING clause is to perform a series of monoalphabetic substitutions against a data item.

- Identifier-5* and *literal-6* must be explicitly or implicitly defined as alphanumeric USAGE DISPLAY data. *Identifier-5* may be a group item.
- Identifier-6* and *literal-7* must be explicitly or implicitly defined as alphanumeric USAGE DISPLAY data. *Identifier-6* may be a group item.
- Identifier-5 / literal-6* must be the same length as *identifier-6 / literal-7*.
- The substring specified before the "TO" will be referred to as the *target string*. The substring specified after the "TO" will be referred to as the *replacement string*.

5. The contents of *identifier-1* will be scanned – character by character. For each character, if that character appears in the target string, the corresponding character (by relative position) in the replacement string will then replace that character in *identifier-1*.
6. If the length of the replacement string exceeds that of the target string, the excess will be ignored.
7. If the length of the target string exceeds that of the replacement string, the replacement string will be assumed to be padded to the right with SPACES to make up the difference.
8. This INSPECT statement clause was introduced in the 1985 standard of COBOL, making the TRANSFORM verb (section [6.48](#)) obsolete.

## 6.28. MERGE

Figure 6-69 - MERGE Syntax



The MERGE statement combines two or more identically sequenced files on a set of specified keys.

1. The *sort-file* named on the MERGE statement must be defined using a sort description (SD) in the FILE SECTION of the DATA DIVISION. See section [5.2](#). This file is referred to in the remainder of this discussion as the “merge file”.
2. *File-name-1*, *file-name-2* and *file-name-3* (if specified) must reference ORGANIZATION LINE SEQUENTIAL or ORGANIZATION RECORD BINARY SEQUENTIAL files. These files must be defined using a file description (FD) in the FILE SECTION of the DATA DIVISION. See section [5.1](#). The same file may be used for *file-name-1* and *file-name-2*.
3. The *identifier-1* ... field(s) must be defined as field(s) within a record of *sort-file*.
4. The WITH DUPLICATES IN ORDER clause is supported for compatibility purposes, but is non-functional.
5. The record descriptions of *file-name-1*, *file-name-2*, *file-name-3* (if any) and *sort-file* are assumed to be identical in layout and size. While the actual data names used for fields in these files’ records may differ, the structure of records, PICTURE of fields, size of fields and USAGE of data should match field-by-field across all files.

A common programming technique when using the MERGE statement is to define the records of all files involved on the MERGE as simple elementary items of the form “**01** *record-name* **PIC X(n)**.” where *n* is the record size. The only file where records are actually described in detail would then be the *sort-file*.

6. The following rules apply to the files named on the USING clause:
  - a. None of them may be OPEN at the time the MERGE is executed.
  - b. Each of those files is assumed to be already sorted according to the specifications set forth on the MERGE statement’s KEY clause.
  - c. No two of those files may be referenced on a SAME RECORD AREA, SAME SORT AREA or SAME SORT-MERGE AREA statement<sup>26</sup>.
7. As the MERGE begins execution, the first record in each of the USING files is read.
8. As the MERGE statement executes, the current record from each of the USING files is inspected and compared to each other according to the rules set forth by the KEY clause. The record that should be “next” in sequence (according to KEY) will be written to the merge file and the USING file from which that record came will be read so that its next record is available. As end-of-file conditions are reached on USING files, those files will be excluded from further MERGE processing – processing continues with the remaining USING files. This process will continue until all USING files have been completely processed.

<sup>26</sup> See section [4.2.2](#)

9. Once the merge file has been populated, the merged data will be written to *file-name-3* if the GIVING clause was specified, or will be processed by utilizing an OUTPUT PROCEDURE defined as *procedure-name 1* or *procedure-name-1* THRU *procedure-name-2*.
10. When GIVING is specified, *file-name-3* ... must not be OPEN at the time the MERGE is executed.
11. When an OUTPUT PROCEDURE is used, merged records are manually read from the merge file – one at a time – using the RETURN statement (section [6.36](#)).
12. A STOP RUN, EXIT PROGRAM or GOBACK executed within an OUTPUT PROCEDURE will terminate the currently executing program as well as the MERGE.
13. A GO TO statement that transfers control out of the OUTPUT PROCEDURE will terminate the MERGE but allows the program to continue executing from the point where the GO TO transferred control to. Once an OUTPUT PROCEDURE has been aborted using a GO TO it cannot be resumed. You may, however, re-execute the MERGE statement itself. Any records not yet RETURNed from the merge file will be lost if a MERGE is restarted in this manner. **USING A “GO TO” TO PREMATURELY TERMINATE A SORT, OR RE-STARTING A PREVIOUSLY-CANCELLED MERGE IS NOT CONSIDERED GOOD PROGRAMMING STYLE AND SHOULD BE AVOIDED.**
14. An OUTPUT PROCEDURE is terminated either implicitly by a fall-thru of control past the last statement of *procedure-name-2* (or *procedure-name-1* if there is no *procedure-name-2*) or explicitly via an EXIT SECTION / EXIT PARAGRAPH executed in *procedure-name-2* (or *procedure-name-1* if there is no *procedure-name-2*). Once the OUTPUT PROCEDURE terminates, the output phase – and the MERGE statement itself - is complete.
15. The scope of the OUTPUT PROCEDURE must not allow a file-based SORT (section [6.41.1](#)), MERGE or RELEASE (section [6.35](#)) statement to be executed.

## 6.29. MOVE

### 6.29.1. MOVE Format 1 – Simple MOVE

Figure 6-70 - Simple MOVE Syntax

```
MOVE { literal-1  
identifier-1 } TO identifier-2 ...
```

This statement moves a specific value to one or more receiving data items.

1. The MOVE statement will replace the contents of one or more receiving data items (*identifier-2 ...*) with a new value.
2. The exact manner in which the new value is stored in each receiving data item will depend upon the PICTURE and USAGE of each *identifier-2* item.

### 6.29.2. MOVE Format 2 – MOVE CORRESPONDING

Figure 6-71 - MOVE CORRESPONDING Syntax

```
MOVE CORRESPONDING identifier-1 TO identifier-2 ...
```

This statement moves similarly-named elementary items from one group item to another.

1. The word CORRESPONDING may be abbreviated as CORR.
2. Both *identifier-1* and *identifier-2* must be group items.
3. Two data items subordinate to *identifier-1* and *identifier-2* are said to correspond if they meet the following conditions:
  - a. They both have the same name, but that name is not FILLER
  - b. If they are not immediately subordinate to *identifier-1* and *identifier-2*, then the items they ARE subordinate to have the same name, but that name is not FILLER; if those items, in turn, are not *identifier-1* and *identifier-2*, then this rule continues to apply recursively upward through the structure of *identifier-1* and *identifier-2*
  - c. They are both elementary items (ADD CORR,SUBTRACT CORR) or at least one of them is an elementary item (MOVE CORR)
  - d. Neither potential corresponding candidate is a REDEFINES or RENAMES of another data item
  - e. Neither potential corresponding candidate has an OCCURS clause (they MAY, however, contain subordinate data items that contain an OCCURS clause)
4. When corresponding matches are established, the effect of a MOVE CORRESPONDING on those matches will be as if a series of individual MOVEs were done – one for each match.

The previous rules may be best understood with an example. Observe the following code:

```

IDENTIFICATION DIVISION.
PROGRAM-ID. corrdemo.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 X.
   05 A          VALUE 'A'          PIC X(1).
   05 G1.
      10 G2.
         15 B    VALUE 'B'          PIC X(1).
   05 C.
      10 FILLER VALUE 'C'          PIC X(1).
   05 G3.
      10 G4.
         15 D    VALUE 'D'          PIC X(1).
   05 V1         VALUE 'E'          PIC X(1).
   05 E          REDEFINES V1 PIC X(1).
   05 F          VALUE 'F'          PIC X(1).
   05 G          VALUE ALL 'G'.
      10 G2      OCCURS 4 TIMES PIC X(1).
01 Y.
   02 A          VALUE ALL 'H'      PIC X(4).
   02 G1.
      03 G2.
         04 B    PIC X(1).
   02 C          PIC X(1).
   02 G3.
      03 G5.
         04 D    PIC X(1).
   02 E          PIC X(1).
   02 V2         PIC X(1).
   02 G          PIC X(4).
   02 H          OCCURS 4 TIMES PIC X(1).
   66 F          RENAMES V2.
PROCEDURE DIVISION.
100-Main.
   MOVE ALL '-' TO Y.
   DISPLAY ' Names: ' 'ABCDEFGGGGHHHH'.
   DISPLAY ' Before: ' Y.
   MOVE CORR X TO Y.
   DISPLAY ' After: ' Y.
   STOP RUN.

```

The DISPLAY statements produce the output:

Names: ABCDEFGGGGHHHH

Before: -----

After: ABC---GGGG---

- OpenCOBOL had no problem establishing a “corresponding” relationship between the “A”, “B” and “C” data items within the “X” and “Y” group items. Note that even though “X” uses a level numbering scheme of 01-05-10-15 while “Y” uses 01-02-03-04, that fact makes no difference to the establishment of corresponding matches.
- The “G” items were found to match even though G OF X was the parent of a data item that contains an OCCURS clause
- No match could be made with the “D” items because they violate rule #3b (look carefully at the four group item names).
- No match could be made with the “E” items because E OF X violates rule #3d (REDEFINES).
- No match could be made with the “F” items because E OF X violates rule #3d (RENAMES).
- No match could be made with the “H” items because H OF Y contains an OCCURS clause, therefore violating rule #3e.



## 6.30. MULTIPLY

### 6.30.1. MULTIPLY Format 1 – MULTIPLY BY

Figure 6-72 - MULTIPLY BY Syntax

```

MULTIPLY { literal-1
             identifier-1 } BY { identifier-2 [ ROUNDED ] } ...

[ ON SIZE ERROR imperative-statement-1 ]

[ NOT ON SIZE ERROR imperative-statement-2 ]

[ END-MULTIPLY ]

```

This format of the MULTIPLY statement generates arithmetic products.

1. *Identifier-1* and *identifier-2* must be numeric unedited data items.
2. *Literal-1* must be a numeric literal.
3. The value of *identifier-1* or *integer-1* multiplied by each individual *identifier-2* will be computed and each of those products in turn will be moved to the corresponding *identifier-2* data item, replacing the old contents.
4. The ON SIZE ERROR, NOT ON SIZE ERROR and ROUNDED clauses are coded and operate the same as the clauses of the same name available to the ADD statement (see section [6.5](#)).

### 6.30.2. MULTIPLY Format 2 – MULTIPLY GIVING

Figure 6-73 - MULTIPLY GIVING Syntax

```

MULTIPLY { literal-1
             identifier-1 } BY { literal-2
                                   identifier-2 }
GIVING { identifier-3 [ ROUNDED ] } ...

[ ON SIZE ERROR imperative-statement-1 ]

[ NOT ON SIZE ERROR imperative-statement-2 ]

[ END-MULTIPLY ]

```

This format of the MULTIPLY statement generates the arithmetic product of two values and then replaces the contents of the identifiers listed after the GIVING (*identifier-3* ...) with that product.

1. *Identifier-1* and *identifier-2* must be numeric unedited data items.
2. *Identifier-3* must be a numeric (edited or unedited) data item.
3. *Literal-1* and *literal-2* must be numeric literals.
4. The values of *identifier-1* and *identifier-2* are not altered.
5. The ON SIZE ERROR, NOT ON SIZE ERROR and ROUNDED clauses are coded and operate the same as the clauses of the same name available to the ADD statement (see section [6.5](#)).

## 6.31. NEXT SENTENCE

Figure 6-74 - NEXT SENTENCE Syntax

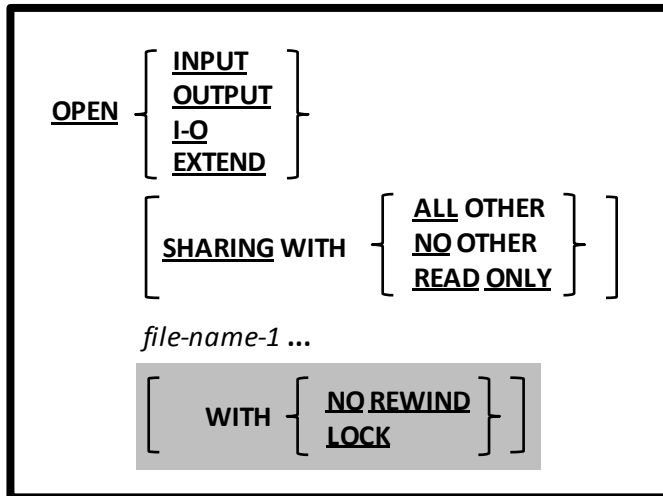
**NEXT SENTENCE**

The NEXT SENTENCE statement is a means of “breaking out” of a series of nested “IF” statements.

1. The NEXT SENTENCE statement is valid only when used within an “IF” statement.
2. As its name implies, this statement causes control to transfer to the next sentence in the program.
3. See section [6.1.5](#) for a discussion of why the NEXT SENTENCE statement is needed for COBOL programs that are coded according to pre-1985 standards. You’ll also see why programs coded for 1985 (and beyond) standards don’t need it.
4. New OpenCOBOL programs should be coded to use the END-IF scope terminator for IF statements, which invalidates the use of NEXT SENTENCE in favor of the CONTINUE statement (section [6.13](#)).

## 6.32. OPEN

Figure 6-75 - OPEN Syntax



The OPEN statement makes one or more files described in your program available for use.

- Any file defined in an OpenCOBOL program must be successfully OPENed before it may be referenced on a CLOSE (section 6.10), DELETE (section 6.14), READ (section 6.34), START (section 6.42) or UNLOCK (section 6.49) statement. Additionally, a file must be successfully OPENed for any of its record data names (or data elements subordinate to those records) to be referenced on ANY statement.
- Any attempt to OPEN a file that is already OPEN will fail with a file status of 41 ("File Already OPEN"). This is a fatal error that will terminate the program.
- Any OPEN failure (including "File Already OPEN") may be trapped using DECLARATIVES (section 6.3) or an error procedure (section 7.3.2), but when those trap routines exit the OpenCOBOL runtime system will terminate the program. Ultimately, you cannot recover from an OPEN failure.
- The INPUT, OUTPUT, I-O and EXTEND options inform OpenCOBOL of the manner in which you wish to use the file, as follows:

OPEN Mode	Effect
INPUT	You may only read the existing contents of the file - only the CLOSE, READ, START and UNLOCK statements will be allowed.
OUTPUT	You may only write new content (which will completely replace any previous file contents) to the file - only the CLOSE, UNLOCK and WRITE statements will be allowed.
I-O	You may perform any operation you wish against the file - all file I/O statements will be allowed.
EXTEND	You may only write new content (which will be appended after any previously existing file content) to the file - only the CLOSE, UNLOCK and WRITE statements will be allowed.

- The SHARING clause informs OpenCOBOL how you are willing to co-exist with any other OpenCOBOL programs that may attempt to OPEN the same file after your program does. Sharing options were discussed in section 6.1.9.1.
- The WITH NO REWIND and WITH LOCK clauses are non-functional.

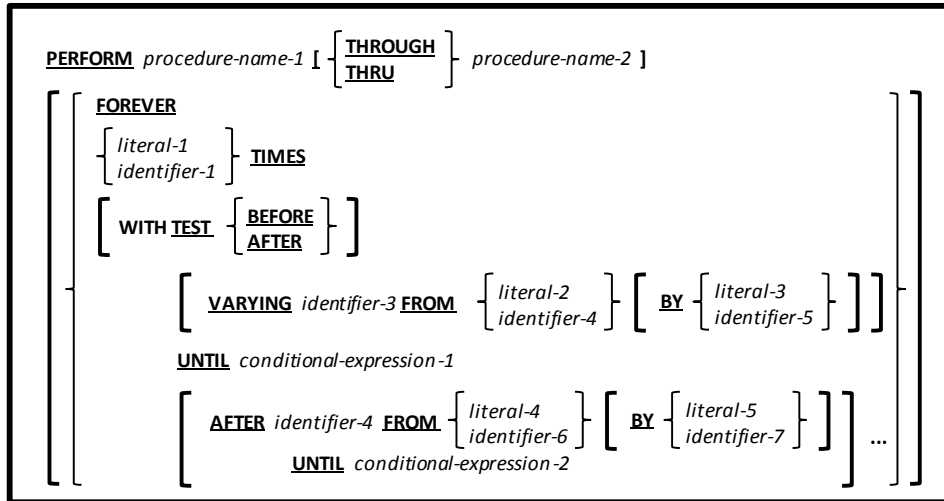
Devices that would be capable of supporting a WITH NO REWIND clause (tape drives) are pretty rare in the environments in which OpenCOBOL is intended to operate. No compiler or runtime message is issued if this option is used (it just won't do anything).

The WITH LOCK option is treated a little differently – it's officially "not implemented", and will generate a compilation warning if it is used.

## 6.33. PERFORM

### 6.33.1. PERFORM Format 1 – Procedural

Figure 6-76 - Procedural PERFORM Syntax



This format of the PERFORM statement is used to transfer control to one or more procedures and to return control when execution of the specified procedure(s) is complete. This invocation of the procedure(s) can be done a single time, multiple times, repeatedly until a condition becomes TRUE or forever (with – presumably – some way of breaking out of the control of the PERFORM within the procedure(s)).

1. The words THROUGH and THRU may be used interchangeably.
2. Both *procedure-name-1* and *procedure-name-2* must be PROCEDURE DIVISION sections or paragraphs defined in the same program unit as the PERFORM statement.
3. If *procedure-name-2* option is specified, it must follow *procedure-name-1* in the program's source code.
4. The *scope* of the PERFORM is defined as being the statements within *procedure-name-1*, the statements within *procedure-name-2* and all statements in all procedures defined between them.
5. Without the FOREVER, TIMES or UNTIL clauses, the code within the scope of the PERFORM will be executed (once) and control will return to the statement following the PERFORM.
6. The FOREVER option will repeatedly execute the code within the scope of the PERFORM with no conditions defined on the PERFORM statement itself for termination of the repetition. It will be up to the programmer to include code within the scope of the PERFORM that will either halt the program (STOP RUN) or break out of the PERFORM (EXIT PERFORM).
7. The TIMES option will repeat the execution of the instructions within the scope of the PERFORM a fixed number of times. Once that number of repetitions has concluded, control will fall into the next statement following the PERFORM.
8. The UNTIL clause will enable the statements within the scope of the PERFORM to be executed repeatedly until such time as the value of *conditional-expression-1* becomes TRUE.
9. The optional WITH TEST clause will control whether the UNTIL test is performed BEFORE the scope of the PERFORM is executed or AFTER. The default, if no WITH TEST clause is specified, is BEFORE.
10. The optional VARYING clause allows for the definition of a data item (*identifier-3*) that will have a unique numeric value for each iteration of the execution of the statements within the scope of the PERFORM. The first time, *identifier-3* will have the value specified by the FROM clause. At the conclusion of each iteration, the value defined by the BY clause will be added to *identifier-3* before *conditional-expression-1* is evaluated. The default BY value, if no BY clause is specified, is 1.
11. If a VARYING clause has been used, you may also use any number of additional AFTER clauses to create a secondary loop situation where each AFTER will create an additional series of iterations, will define an additional data item to be incremented during each iteration and will define an additional conditional expression to define the termination of that series of iterations. Functionally, this is basically a way of nesting a

PERFORM/VARYING/UNTIL within another PERFORM/VARYING/UNTIL without the need to code multiple statements. An example will probably help.

Observe the following code which defines a two-dimensional (3 row by 4 column) table and a pair of numeric data items to be used to subscript references to each element of the table:

```

01 PERFORM-DEMO.
   05 PD-ROW          OCCURS 3 TIMES.
      10 PD-COL      OCCURS 4 TIMES
         15 PD       PIC X(1).
01 PD-COL-No        PIC 9 COMP.
01 PD-Row-No        PIC 9 COMP.
    
```

PD (1, 1)	PD (1, 2)	PD (1, 3)	PD (1, 4)
PD (2, 1)	PD (2, 2)	PD (2, 3)	PD (2, 4)
PD (3, 1)	PD (3, 2)	PD (3, 3)	PD (3, 4)

Let's say we want to PERFORM a routine (100-Visit-Each-PD) which will – in turn – access each PD data item in the sequence shown to the right. Here's the PERFORM code:

```

PERFORM 100-Visit-Each-PD WITH TEST AFTER
  VARYING PD-Row-No FROM 1 BY 1 UNTIL PD-Row-No = 3
  AFTER PD-COL-No FROM 1 BY 1 UNTIL PD-COL-No = 4.
    
```

1	2	3	4
5	6	7	8
9	10	11	12

1	4	7	10
2	5	8	11
3	6	9	12

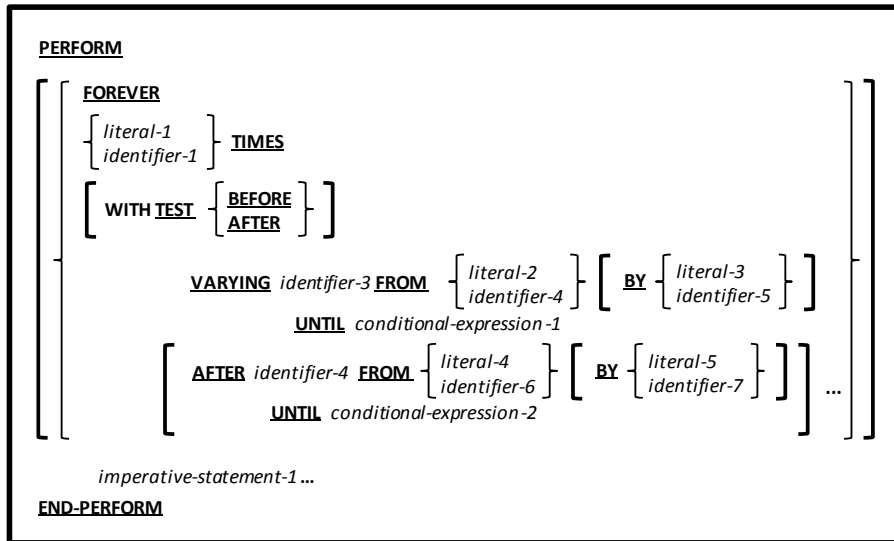
But, perhaps you needed to "visit" each PD in the sequence shown to the left. If so, then here's the PERFORM you need:

```

PERFORM 100-Visit-Each-PD WITH TEST AFTER
  VARYING PD-COL-No FROM 1 BY 1 UNTIL PD-COL-No = 4
  VARYING PD-Row-No FROM 1 BY 1 UNTIL PD-Row-No = 3.
    
```

### 6.33.2. PERFORM Format 2 – Inline

Figure 6-77 - Inline PERFORM Syntax



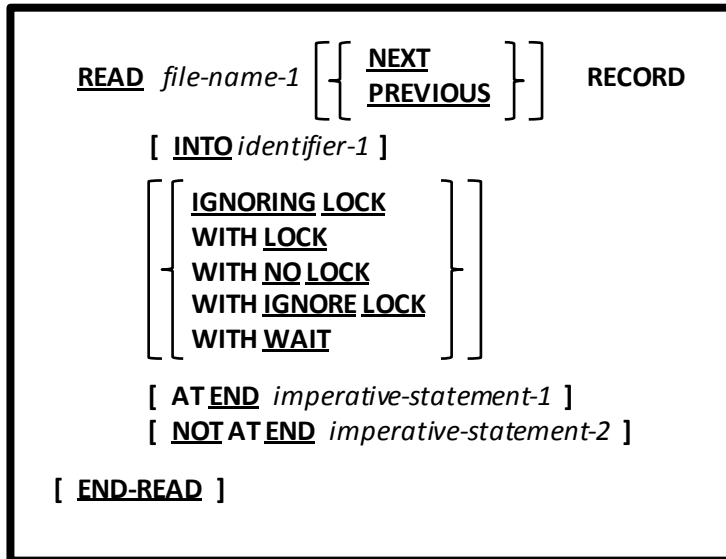
This format of the PERFORM statement is identical in operation to format 1, except for the fact that the statements that comprise the scope of the PERFORM are now specified in-line with the PERFORM code rather than in procedures located elsewhere within the program.

1. The FOREVER, TIMES, WITH TEST, VARYING, BY, AFTER and UNTIL clauses have the same use and effect as the same clauses on format 1 of the PERFORM statement.
2. The distinguishing characteristic of this format versus format 1 is that – with this version of the PERFORM statement – the code being executed is specified in-line (imperative-statement-1 ...) rather than in a procedure.

## 6.34. READ

### 6.34.1. READ Format 1 – Sequential READ

Figure 6-78 – READ (Sequential) Syntax



This form of the READ statement retrieves the next (or previous) record from a file.

1. *File-name-1* must currently be OPEN (section [6.32](#)) for INPUT or I-O.
2. If the ACCESS MODE of *file-name-1* is RANDOM, this format of the READ statement cannot be used.
3. If the ACCESS MODE is SEQUENTIAL, this is the only format of READ that is available. In such cases, the NEXT/PRIOR clauses are truly optional.
4. If the ACCESS MODE is DYNAMIC, this format of the READ statement may be used as well as format 2. The following minimalist READ statement...

```
READ file-name-1
```

...is perfectly legal according to both READ formats. For that reason, when ACCESS MODE DYNAMIC has been specified and you want to tell the OpenCOBOL compiler that a statement such as the one above should be treated as a sequential READ, you must add either NEXT or PRIOR to the statement (otherwise it will be treated as a random READ).

5. The next available record in *file-name-1* is retrieved and the contents of that record stored into the 01-level record structures subordinate to the file's FD (section [5.1](#)).
6. The keywords NEXT and PREVIOUS specify in what direction of travel the reading process will take through the file. If neither NEXT nor PREVIOUS clause is specified, NEXT is assumed.
7. The PREVIOUS option is available only for ORGANIZATION INDEXED files.
8. The optional INTO clause will cause a copy of the just-read record's contents to be MOVEd to *identifier-1*, assuming the READ succeeded.
9. See section [6.1.9.2](#) for a discussion of the record LOCK options.
10. The optional AT END clause will – if present – cause *imperative-statement-1* to be executed if the READ attempt fails due to a file status of 10 (end-of-file). The AT END clause **WILL NOT DETECT OTHER NON-ZERO FILE-STATUS VALUES**. Use a DECLARATIVES routine (section [6.3](#)) or an explicitly-declared file status field tested after the READ to detect error conditions other than end-of-file.
11. The optional NOT AT END clause will – if present – cause *imperative-statement-2* to be executed if the READ attempt is successful.

## 6.34.2. READ Format 2 – Random Read

Figure 6-79 - READ (Random) Syntax

```

READ file-name-1 RECORD

    [ INTO identifier-1 ]

    [ [ IGNORING LOCK
      [ WITH LOCK
      [ WITH NO LOCK
      [ WITH IGNORE LOCK
      [ WITH WAIT
      ]
      ]
      ]
      ]
    ]

    [ KEY IS identifier-2 ]

    [ INVALID KEY imperative-statement-3 ]
    [ NOT INVALID KEY imperative-statement-4 ]

    [ END-READ ]
  
```

This form of the READ statement retrieves an arbitrary record from a file.

1. *File-name-1* must currently be OPEN (section [6.32](#)) for INPUT or I-O.
2. If the ACCESS MODE of *file-name-1* is SEQUENTIAL, this format of the READ statement cannot be used.
3. If the ACCESS MODE is RANDOM, this is the only format of READ that is available.
4. If the ACCESS MODE is DYNAMIC, this format of the READ statement may be used as well as format 1. The following minimalist READ statement...

```
READ file-name-1
```

...is perfectly legal according to both READ formats. For that reason, when ACCESS MODE DYNAMIC has been specified for a file, a READ statement such as the above will be automatically treated as a random READ.

5. The optional KEY clause tells the compiler how a record is to be located in the file.

If the KEY clause is absent:

- If the file is an ORGANIZATION RELATIVE file, the contents of the field declared as the file's RELATIVE KEY will be used to identify a record.
- If the file is an ORGANIZATION INDEXED file, the contents of the field declared as the file's RELATIVE KEY will be used to identify a record.

If the KEY clause is specified:

- If the file is an ORGANIZATION RELATIVE file, the contents of *identifier-2* will be used as the relative record number of the record to be accessed. *Identifier-2* does not have to be the RELATIVE KEY field of the file (although it could be if you wish).
  - If the file is an ORGANIZATION INDEXED file, *identifier-2* must be the PRIMARY RECORD KEY or one of the file's ALTERNATE RECORD KEY fields (if any) – the current contents of that field will identify the record to be accessed. If an alternate record key is used, and that key allows duplicate values, the record accessed will be the 1<sup>st</sup> one having that key value.
6. The record identified by rule #5 will be retrieved from *file-name-1* and the contents of that record stored into the 01-level record structures subordinate to the file's FD (section [5.1](#)).
  7. The optional INTO clause will cause a copy of the just-read record's contents to be MOVED to *identifier-1*, assuming the READ succeeded.
  8. See section [6.1.9.2](#) for a discussion of the record LOCK options.

9. The optional INVALID KEY clause will – if present – cause *imperative-statement-1* to be executed if the READ attempt fails due to a file status of 23 (“Key Not Exists”). The INVALID KEY clause **WILL NOT DETECT OTHER NON-ZERO FILE-STATUS VALUES**. Use a DECLARATIVES routine (section [6.3](#)) or an explicitly-declared file status field tested after the READ to detect error conditions other than “Key Not Exists”.
10. The optional NOT INVALID KEY clause will – if present – cause *imperative-statement-2* to be executed if the READ attempt is successful.



## 6.35. RELEASE

Figure 6-80 - RELEASE Syntax

```
RELEASE record-name-1 [ FROM { literal-1  
                             identifier-1 } ]
```

The RELEASE statement adds a new record to a *sort file*.

1. The RELEASE statement is valid only within the INPUT PROCEDURE of a SORT statement. See section [6.41.1](#).
2. *Record-name-1* must be a record defined to a sort description (SD) entry. See section [5.2](#).

## 6.36. RETURN

Figure 6-81 - RETURN Syntax

```
RETURN file-name-1 RECORD  
  [ INTO identifier-1 ]  
  [ AT END imperative-statement-1 ]  
  [ NOT AT END imperative-statement-2 ]  
  [ END-RETURN ]
```

The RETURN statement reads a record from a sort- or merge file.

1. The RETURN statement is valid only within the OUTPUT PROCEDURE of a SORT (section [6.41.1](#)) or MERGE (section 6.28) statement.
2. *File-name-1* must be a sort- or merge file defined with a sort description (SD) entry. See section [5.2](#).
3. The INTO, AT END and NOT AT END clauses are used the same as with their READ statement (section [6.34](#)) equivalents.

## 6.37. REWRITE

Figure 6-82 - REWRITE Syntax

```

REWRITE record-name-1
  [ FROM { literal-1
            identifier-1 } ]
  [ [ WITH LOCK
      WITH NO LOCK ] ]
  [ INVALID KEY imperative-statement-3 ]
  [ NOT INVALID KEY imperative-statement-4 ]
  [ END-REWRITE ]

```

The REWRITE statement replaces a logical record on a disk file.

1. *Record-name-1* must be defined as an 01-level record subordinate to the File Description (FD – see section [5.1](#)) of a file that is currently OPEN (section [6.32](#)) for I-O.
2. The optional FROM clause will cause *literal-1* or *identifier-1* to be implicitly MOVEd into *record-name-1* prior to writing *record-name-1* to the file.
3. The REWRITE statement may not be used with ORGANIZATION IS LINE SEQUENTIAL files.
4. See section [6.1.9.2](#) for a discussion of the record LOCK options.
5. Rewriting a record does not cause the record contents of the file to be physically updated until the next block of the file is read, a COMMIT statement (section [6.11](#)) is issued or that file is closed.
6. If the file has ORGANIZATION RECORD BINARY SEQUENTIAL:
  - a. The record to be rewritten will be the one retrieved by the most-recently executed READ (section [6.34](#)) of the file.
  - b. The size of *record-name-1* cannot be altered (see the RECORD CONTAINS / RECORD IS VARYING clauses in section [5.1](#)).
7. If the file has ORGANIZATION RELATIVE or ORGANIZATION INDEXED:
  - a. If the file has ACCESS MODE SEQUENTIAL, the record to be rewritten will be the one retrieved by the most-recently executed READ (section [6.34](#)) of the file. If the file has ACCESS MODE RANDOM or ACCESS MODE DYNAMIC, no READ is required before a record may be rewritten – the RELATIVE KEY / RECORD KEY definition for the file will specify the record to be updated.
  - b. The size of *record-name-1* may be updated.
8. The ON INVALID KEY clause will be triggered (thus executing *imperative-statement-1*) if an error occurred during the REWRITE. Such errors might be actual I/O errors or “Key Not Exists” errors (file status 23), indicating no record exists that satisfies the RELATIVE KEY or RECORD KEY clause requirements.
9. The NOT ON INVALID KEY clause will be triggered, thus executing *imperative-statement-2*, if no error occurred during the REWRITE.

## 6.38. ROLLBACK

Figure 6-83 - ROLLBACK Syntax

**ROLLBACK**

The ROLLBACK verb reverts changes made to all files since the start of the program or since the last COMMIT.

1. OpenCOBOL does not (currently, at least) support file rollback. The OpenCOBOL ROLLBACK statement will have the same effect as the COMMIT verb (section [6.11](#)).

## 6.39. SEARCH

### 6.39.1. SEARCH Format 1 –Sequential Search

Figure 6-84 - Sequential SEARCH Syntax

```
SEARCH table-name
    [ VARYING index-name-1 ]
    [ AT END imperative-statement-1 ]
    { WHEN conditional-expression-1 imperative-statement-2 }...
[ END-SEARCH ]
```

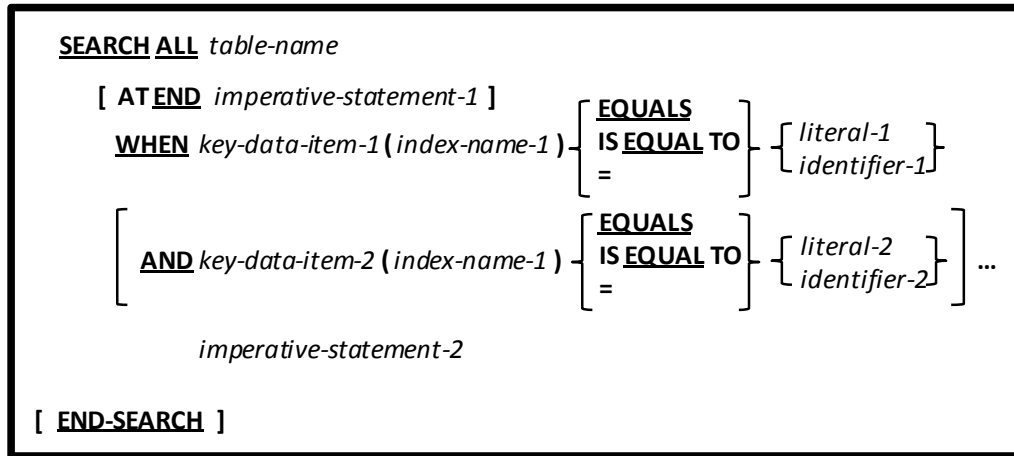
The SEARCH statement is used to sequentially search a table, stopping either once a specific value is located within the table or when the table has been completely searched.

1. The *index-name-1* identifier specified on the VARYING clause must be USAGE INDEX.
2. If no VARYING clause is specified, then the table being searched must have been created with an INDEXED BY clause (see section [5.3](#)).
3. At the time the SEARCH statement is executed, the current value of *index-name-1* (or the table's defined INDEXED BY index) will define the starting position in the table where the searching process will begin. Typically, one initializes that index to a value of 1 before starting the SEARCH, as follows:
 

```
SET index-name-1 TO 1
```
4. During the searching process, the *conditional-expression-1* will be evaluated and – if TRUE – will cause *imperative-statement-2* to be executed, after which control will fall into the next statement after the SEARCH.
5. If multiple WHEN clauses exist, each *conditional-expression-n* will be evaluated in-turn and the first one that evaluates to TRUE will cause the corresponding *imperative-statement-n* to be executed, after which control will fall into the next statement after the SEARCH.
6. If no *conditional-expression-n* evaluates to TRUE, the value of *index-name-1* will be incremented by 1. If the value of *index-name-1* is still within the OCCURS scope of *table-name*, the WHEN clause(s) will again be re-evaluated. This process will continue until a WHEN clause *conditional-expression-n* evaluates to TRUE or until the value of *index-name-1* is no longer within the OCCURS scope of *table-name*.
7. If no *conditional-expression-n* ever evaluates to TRUE and the value of *index-name-1* is no longer within the OCCURS scope of *table-name*, the *imperative-statement-1* which is part of the AT END clause will be executed. After this, control will fall into the next statement following the SEARCH. If there is no AT END clause, control simply falls into the next statement following the SEARCH.

## 6.39.2. SEARCH Format 2 – Binary, or Half-interval Search (SEARCH ALL)

Figure 6-85 - Binary SEARCH (ALL) Syntax



This format of the SEARCH statement performs a binary, or half-interval, search against a sorted table.

1. The definition of *table-name* must include the OCCURS, ASCENDING (and/or DESCENDING) KEY and INDEXED BY clauses.
2. In order for a table to be searchable via the SEARCH ALL statement, each of the following must be true:
  - a. The table meets the requirements of rule #1 above.
  - b. Just because the table has one or more KEY clauses doesn't mean the data is actually in that sequence in the table – the actual sequence of the data must agree with the KEY clause(s)!<sup>27</sup>
  - c. No two records in the table may have the same KEY field values. If the table has multiple KEY definitions, then no two records in the table may have the same combination of KEY field values.

If rule "a" is violated, the compiler will reject the SEARCH ALL. If rules "b" and/or "c" are violated, there will be no message issued by the compiler, but the run-time results of a SEARCH ALL against the table will probably be incorrect.
3. *Key-data-item-1* and *key-data-item-2* ... (if any) must be defined as keys of *table-name* via ASCENDING KEY or DESCENDING KEY clauses (see rule #1 above).
4. *Index-name-1* is the first INDEXED BY data item for *table-name*.
5. The WHEN clause is mandatory, unlike format 1 of the SEARCH statement.
6. There can only be one WHEN clause specified; there may be any number of AND clauses, but there cannot be more WHEN & AND clauses than there are KEY fields to the table. Each WHEN/AND clause should reference a different KEY field.
7. The function of the WHEN, along with any ANDs, is to compare the key field(s) of the table, as indexed by the first INDEXED BY item, against the specified literal and/or identifier values in order to locate the desired entry in the table. The table's index will be automatically varied by the SEARCH ALL statement in a manner designed to require the minimum number of tests.

<sup>27</sup> Of course, if the data sequence doesn't agree with the KEY clause, you can easily make it that way using a table SORT (see section SORT Format 2 – Table Sort)

8. The internal processing of the SEARCH ALL statement begins by setting internal “first” and “last” pointers to the 1<sup>st</sup> and last entry locations of the table. Processing then proceeds as follows<sup>28</sup>:
- a. The entry half-way between “first” and “last” is identified. We’ll call this the “current” entry, and will set its table entry location is saved into *index-name-1*.
  - b. The WHEN (along with any ANDs) is evaluated. This comparison of the keys against the target literal/identifier values will have one of three possible outcomes:
    - i. If the key(s) and value(s) match, *imperative-statement-2* is executed, after which control falls thru into the next statement following the SEARCH ALL.
    - ii. If the key(s) are LESS THAN the value(s), then the table entry being searched for can only occur in the “current” to “last” range of the table, so a new “first” pointer value is set (it will be set to the “current” pointer).
    - iii. If the key(s) are GREATER THAN the value(s), then the table entry being searched for can only occur in the “first” to “current” range of the table, so a new “last” pointer value is set (it will be set to the “current” pointer).
  - c. If the new “first” and “last” pointers are different than the old “first” and “last” pointers, there’s more left to be searched, so return to step “a” and continue.
  - d. If the new “first” and “last” pointers are the same as the old “first” and “last” pointers, the table has been exhausted and the entry being searched for cannot be found; *imperative-statement-1* is executed, after which control falls thru into the next statement following the SEARCH ALL.

The net effect of the above algorithm is that only a fraction of the number of elements in the table need ever be tested in order to decide whether or not a particular entry exists. This is because the SEARCH ALL discards half the remaining entries in the table each time it checks an entry.

Computer scientists will compare these two search techniques as follows:

- A sequential search (format 1) will need an average of  $n/2$  tests and a worst case of  $n$  tests in order to find an entry and  $n$  tests to identify that an entry doesn’t exist ( $n$  = the number of entries in the table).
- A binary search (format 2) will need worst case of  $\log_2 n$  tests in order to find an entry and  $\log_2 n$  tests to identify that an entry doesn’t exist ( $n$  = the number of entries in the table).

Here’s a more practical view of the difference. Let’s say that a table has 1,000 entries in it. With a sequential (format 1) search, on average, you’ll have to check 500 of them to find an entry and you’ll have to look at all 1,000 of them to find that an entry doesn’t exist. With a binary search, express the number of entries as a binary number ( $1,000_{10} = 1111101000_2$ ) and count the number of digits in the result (10) -THAT is the worst-case number of tests required to find an entry or to identify that it doesn’t exist. That’s quite an improvement.

---

<sup>28</sup> This is a simplified view of the algorithm intended purely as a pedagogical tool – an actual implementation of it requires a few additional picky little details to make it work (such as what to do when rule “a” identifies a “current” entry of 12.5!)

## 6.40. SET

### 6.40.1. SET Format 1 – SET ENVIRONMENT

Figure 6-86 - SET ENVIRONMENT Syntax

```
SET ENVIRONMENT { integer-1 } TO { integer-2 }
                 { identifier-1 }                { identifier-2 }
```

This format of the SET statement provides a straight-forward means of setting environment values from within a program.

1. Environment variables created or changed from within OpenCOBOL programs will be available to any sub-shell processes spawned by that program (i.e. CALL "SYSTEM") but will not be known to the shell or console window that started the OpenCOBOL program.
2. This is a much simpler and more readable means of setting environment variables than by using the DISPLAY statement (section [6.15.3](#)). For example, these two code sequences produce identical results:

```
DISPLAY "VARNAME" UPON ENVIRONMENT-NAME      SET ENVIRONMENT "VARNAME" TO "VALUE"
END-DISPLAY
DISPLAY "VALUE" UPON ENVIRONMENT-VALUE
```

### 6.40.2. SET Format 2 – SET Program-Pointer

Figure 6-87 - SET Program Pointer Syntax

```
SET program-pointer-1 TO ENTRY { literal-1 }
                                     { identifier-1 }
```

This form of SET allows you to retrieve the address of a PROCEDURE DIVISION code module – specifically a declared entry-point into the PROCEDURE DIVISION.

1. If you have used other versions of COBOL before (particularly mainframe implementations), you've possibly seen subroutine CALLs made passing a PROCEDURE DIVISION paragraph or SECTION name as an argument – that is not possible in OpenCOBOL; instead, you need to know how to use this form of the SET statement.
2. The USAGE of *program-pointer-1* must be PROGRAM-POINTER.
3. The *literal-1* or *identifier-1* value specified must name the PROGRAM-ID of the program or the entry-point named on an ENTRY statement.
4. Once the address of a PROCEDURE DIVISION code area has been acquired in this way, the address could be passed to a subroutine (usually written in C) for whatever use it needs it for. For examples of PROGRAM-POINTERS at work, see sections [7.3.1.21](#) and [7.3.1.22](#).

### 6.40.3. SET Format 3 – SET ADDRESS

Figure 6-88 - SET ADDRESS Syntax

```
SET [ ADDRESS OF ] { pointer-name-1 } ...
                       { identifier-1 }
TO [ ADDRESS OF ] { pointer-name-2 }
                       { identifier-2 }
```

This form of the SET statement can be used to work with the addresses of data items rather than their contents.



1. When the ADDRESS OF clause is used before the TO you will be using the SET to alter the address of a LINKAGE SECTION or BASED data item. Without that clause you will be assigning an address to one or more USAGE POINTER data items.
2. When the ADDRESS OF clause is used after the TO, SET will be identifying the address of *identifier-2* as the address to be assigned to *identifier-1* or stored in *pointer-name-1*. If the "ADDRESS OF" clause is absent after the TO, the contents of *pointer-name-2* will serve as the address to be assigned.

#### 6.40.4. SET Format 4 – SET Index

Figure 6-89 - SET Index Syntax

```
SET index-name-1 TO { literal-1
                     identifier-1 }
```

This SET statement assigns a value to a USAGE INDEX data item.

1. The USAGE of *index-name-1* should be INDEX, or *index-name-1* must be identified in a table INDEXED BY clause.

#### 6.40.5. SET Format 5 – SET UP/DOWN

Figure 6-90 - SET UP/DOWN Syntax

```
SET { index-name-1
      pointer-1 } { UP
                   DOWN }
    BY [ LENGTH OF ] { literal-1
                      identifier-2
                      function-reference 1 }
```

This format of SET is used to increment or decrement the value of an index or pointer by a specified amount.

1. The USAGE of *index-name-1* must be INDEX. The USAGE of *pointer-1* must be POINTER or PROGRAM-POINTER.
2. The typical usage when an *index-name-1* is specified is to set the value UP or DOWN by 1, since an *index-name-1* is usually being used to sequentially walk through the elements of a table.

#### 6.40.6. SET Format 6 – SET Condition Name

Figure 6-91 - SET Condition Name Syntax

```
SET { condition-name-1 } ... TO { TRUE
                                 FALSE }
```

This format provides one method of specifying the TRUE / FALSE value of a level-88 condition name.

1. By setting the specified condition name(s) to a TRUE or FALSE value, you will actually be assigning a value to the parent data item(s) to which the condition name data item(s) is subordinate to.
2. When specifying TRUE, the value assigned to each parent data item will be the first VALUE specified on the condition name's definition.
3. When specifying FALSE on the SET, the value assigned to each parent data item will be the value specified for the FALSE clause of the condition name's definition; if any *condition-name-1* occurrence lacks a FALSE clause, the SET statement will be rejected by the compiler.

### 6.40.7. SET Format 7 – SET Switch

Figure 6-92 - SET Switch Syntax

```
SET { mnemonic-name-1 } ... TO { ON  
                                          OFF }
```

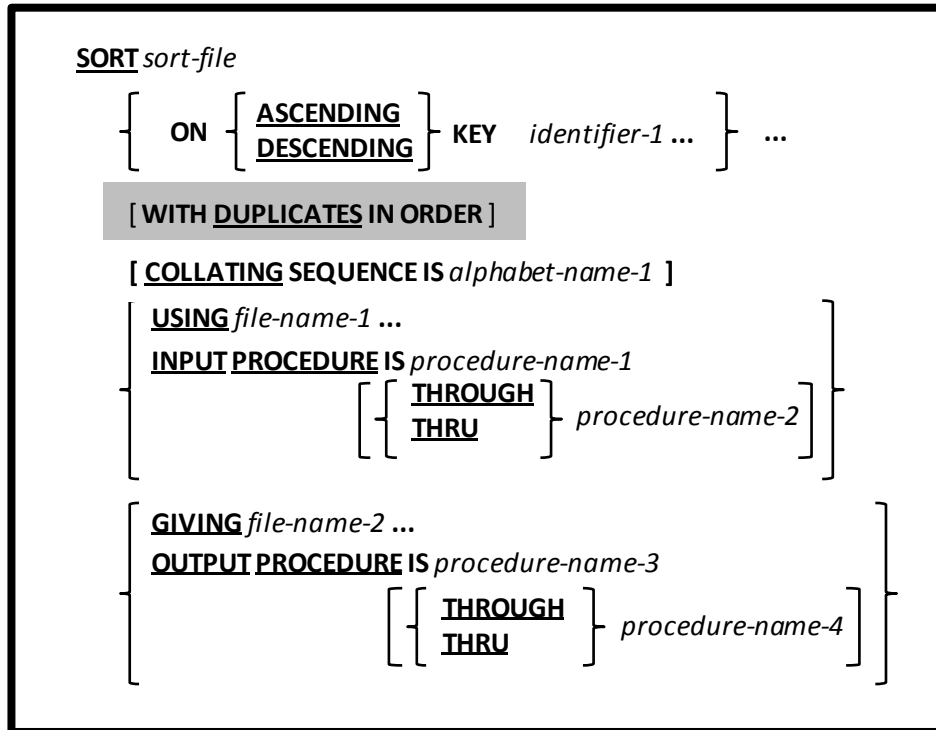
Use this SET statement type to turn a switch ON or OFF.

1. Switches are defined using the SPECIAL-NAMES paragraph. See section [4.1.4](#) for additional information.

## 6.41. SORT

### 6.41.1. SORT Format 1 – File-based Sort

Figure 6-93 - File-Based SORT Syntax



This format of the SORT statement is designed to sort large volumes of data according to one or more key fields.

1. The *sort-file* named on the SORT statement must be defined using a sort description (SD) in the FILE SECTION of the DATA DIVISION. See section [5.2](#). This file is referred to as the “sort file”.
2. If specified, *file-name-1* and *file-name-2* must reference ORGANIZATION LINE SEQUENTIAL or ORGANIZATION RECORD BINARY SEQUENTIAL files. These files must be defined using a file description (FD) in the FILE SECTION of the DATA DIVISION. See section [5.1](#). The same file may be used for *file-name-1* and *file-name-2*.
3. The *identifier-1 ...* field(s) must be defined as field(s) within a record of *sort-file*.
4. The WITH DUPLICATES IN ORDER clause is supported for compatibility purposes, but is non-functional.
5. A sort file (see #1) is never OPENed or CLOSEd.
6. The SORT statement works in three stages, as follows:

STAGE I (the input phase):

- a. The data to be sorted is loaded into the sort file. This is accomplished either by taking the entire contents of the file(s) named on the USING clause or by utilizing an INPUT PROCEDURE defined as *procedure-name 1* or *procedure-name-1 THRU procedure-name-2*.
- b. When USING is specified, *file-name-1 ...* must not be OPEN at the time the SORT is executed.
- c. When an INPUT PROCEDURE is used, records to be sorted are produced using whatever logic is necessary and are manually written to the sort file – one at a time – using the RELEASE statement (section [6.35](#)).
- d. A STOP RUN, EXIT PROGRAM or GOBACK executed within an INPUT PROCEDURE will terminate the currently executing program as well as the SORT.
- e. A GO TO statement that transfers control out of the INPUT PROCEDURE will terminate the SORT but allows the program to continue executing from the point where the GO TO transferred control to. Once an INPUT PROCEDURE has been aborted using a GO TO it cannot be resumed. You may, however, re-execute the SORT

statement itself. Any records previously RELEASEd to the sort file will be lost if a SORT is restarted in this manner. **USING A “GO TO” TO PREMATURELY TERMINATE A SORT, OR RE-STARTING A PREVIOUSLY-CANCELLED SORT IS NOT CONSIDERED GOOD PROGRAMMING STYLE AND SHOULD BE AVOIDED.**

- f. As data is loaded into the sort file, it is actually being buffered in dynamically-allocated memory. Only if the amount of data to be sorted exceeds the amount of available sort memory (128 MB)<sup>29</sup> will actual disk files be allocated and utilized. These “sort work files” will be discussed again shortly.
- g. An INPUT PROCEDURE is terminated either implicitly by a fall-thru of control past the last statement of *procedure-name-2* (or *procedure-name-1* if there is no *procedure-name-2*) or explicitly via an EXIT SECTION / EXIT PARAGRAPH executed in *procedure-name-2* (or *procedure-name-1* if there is no *procedure-name-2*). Once the INPUT PROCEDURE terminates, the input phase is complete.
- h. The scope of the INPUT PROCEDURE must not allow a file-based SORT, MERGE (section [6.28](#)) or RETURN (section [6.36](#)) statement to be executed.

STAGE 2 (the sort phase):

- a. The sort will take place by arranging the data records in the sequence defined by the ASCENDING KEY and/or DESCENDING KEY specification(s) on the SORT statement according to the COLLATING SEQUENCE specified on the SORT (if any) or – if none was defined – the PROGRAM COLLATING SEQUENCE specified or implied by the OBJECT-COMPUTER paragraph. Keys may be any supported data type and USAGE except for level-78 or level-88 data items.
- b. For example, let’s assume we’re sorting a series of financial transactions. The SORT statement might look like this:

```
SORT Sort-File
ASCENDING KEY Transaction-Date
ASCENDING KEY Account-Number
DESCENDING KEY Transaction-Amount
.
.
.
```

The effect of this statement will be to sort all transactions into ascending order of the date the transaction took place (oldest first, newest last). Unless the business running this program is going out of business, there are most-likely many transactions for any given date – therefore, within each grouping of transactions all with the same date, transactions will be sub-sorted into ascending sequence of the account number the transactions apply to. Since it’s quite possible there might be multiple transactions for an account on any given date, a third level sub-sort will arrange all transactions for the same account on the same date into descending sequence of the actual amount of the transaction (largest first, smallest last). If two or more transactions of \$100.00 were recorded for account #12345 on the 31<sup>st</sup> of August 2009, there will be no way of predicting exactly how those transactions are ordered relative to each other since there’s no additional “level” specified for sort keys.

- c. OpenCOBOL does not utilize a high-capacity, high-performance (and usually high expense) sorting package as would be the case on a mainframe computer system, but the SORT algorithms used<sup>30</sup> are more than adequate for the task.

Stage 3 (the output phase):

- a. Once the sort phase is complete, the sorted data will be written to *file-name-2* if the GIVING clause was specified, or by utilizing an OUTPUT PROCEDURE defined as *procedure-name 3* or *procedure-name-3 THRU procedure-name-4*.
- b. When GIVING is specified, *file-name-2* ... must not be OPEN at the time the SORT is executed.

<sup>29</sup> There is a runtime environment variable (COB\_SORT\_MEMORY) that you may use to allocate more or less memory to the sorting process. See section [7.2.4](#).

<sup>30</sup> The OpenCOBOL sort routines are entirely self-contained in the OpenCOBOL run-time library

- c. When an OUTPUT PROCEDURE is used, sorted records are manually read from the sort file – one at a time – using the RETURN statement (section [6.36](#)).
  - d. A STOP RUN, EXIT PROGRAM or GOBACK executed within an OUTPUT PROCEDURE will terminate the currently executing program as well as the SORT.
  - e. A GO TO statement that transfers control out of the OUTPUT PROCEDURE will terminate the SORT but allows the program to continue executing from the point where the GO TO transferred control to. Once an OUTPUT PROCEDURE has been aborted using a GO TO it cannot be resumed. You may, however, re-execute the SORT statement itself. Any records not yet RETURNed from the sort file will be lost if a SORT is restarted in this manner. **USING A “GO TO” TO PREMATURELY TERMINATE A SORT, OR RE-STARTING A PREVIOUSLY-CANCELLED SORT IS NOT CONSIDERED GOOD PROGRAMMING STYLE AND SHOULD BE AVOIDED.**
  - f. An OUTPUT PROCEDURE is terminated either implicitly by a fall-thru of control past the last statement of *procedure-name-4* (or *procedure-name-3* if there is no *procedure-name-4*) or explicitly via an EXIT SECTION / EXIT PARAGRAPH executed in *procedure-name-4* (or *procedure-name-3* if there is no *procedure-name-4*). Once the OUTPUT PROCEDURE terminates, the output phase – and the SORT statement itself - is complete.
  - g. The scope of the OUTPUT PROCEDURE must not allow a file-based SORT, MERGE (section [6.28](#)) or RELEASE (section [6.35](#)) statement to be executed.
7. Should disk work files be necessary due to the amount of data being sorted, they will be automatically allocated to disk in a folder defined by the TMPDIR, TMP or TEMP environment variables (see section [7.2.4](#)). These disk files WILL NOT be automatically purged upon program execution termination (normal or otherwise). Temporary sort work files will be named “**cobxxx.tmp**”, in case you want to delete them yourself or from within your program upon sort termination.

### 6.41.2. SORT Format 2 – Table Sort

Figure 6-94 - Table SORT Syntax

```

SORT table-name
  [ ON { ASCENDING
        | DESCENDING } KEY identifier-1 ... ] ...
  [ WITH DUPLICATES IN ORDER ]
  [ COLLATING SEQUENCE IS alphabet-name-1 ]

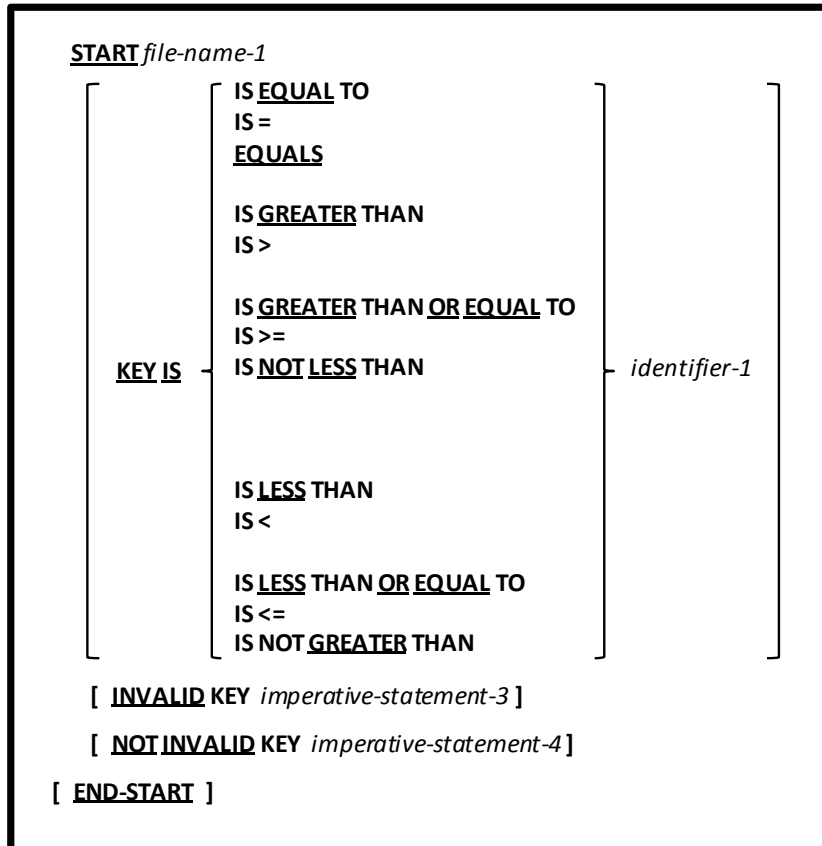
```

This format of the SORT statement sorts relatively small quantities of data – namely data contained in a DATA DIVISION table – according to one or more key fields.

1. The *table-name* data item must have an OCCURS clause.
2. The *identifier-1 ...* field(s), if any, must be defined as data items subordinate to *table-name*.
3. The WITH DUPLICATES IN ORDER clause is supported for compatibility purposes, but is non-functional.
4. The data within *table-name* will be sorted in-place (i.e. no sort file is required) according to the KEY specification(s) made on the SORT statement.
5. Currently, a table SORT with no KEY specification(s) made on the SORT statement is unsupported and will be rejected by the compiler.
6. The sort will take place by arranging the data records in the sequence defined by the ASCENDING KEY and/or DESCENDING KEY specification(s) on the SORT statement according to the COLLATING SEQUENCE specified on the SORT (if any) or – if none was defined – the PROGRAM COLLATING SEQUENCE specified or implied by the OBJECT-COMPUTER paragraph. Keys may be any supported data type and USAGE except for level-78 or level-88 data items.
7. The SORT will be performed in-place within *table-name* – no sort file is required.

## 6.42. START

Figure 6-95 - START Syntax



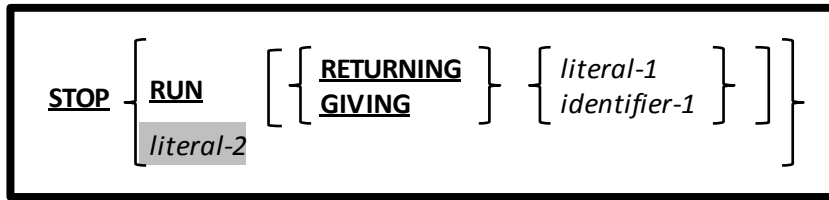
The START statement defines the logical starting point within a file for subsequent sequential read operations.

1. *File-name-1* must be an ORGANIZATION RELATIVE or ORGANIZATION INDEXED file.
2. *File-name-1* must have been SELECTed with an ACCESS MODE DYNAMIC or ACCESS MODE SEQUENTIAL.
3. *File-name-1* must be OPEN (section [6.32](#)) in either INPUT or I-O mode at the time the START is executed.
4. If no KEY clause is specified, “**KEY IS EQUAL TO** *identifier-1*” will be assumed.
5. If *file-name-1* is an ORGANIZATION RELATIVE file, *identifier-1* must be the RELATIVE KEY of the file. See section [4.2.1.2](#).
6. If *file-name-1* is an ORGANIZATION INDEXED file, *identifier-1* must be the RECORD KEY or one of the ALTERNATE RECORD KEY fields for the file. See section [4.2.1.3](#).
7. After successful execution of a START statement, the internal record pointer into the *file-name-1* data will be positioned such that the next sequential READ statement executed against *file-name-1* will read:
  - a. The FIRST record that satisfies the KEY clause specification if the relation check specified is EQUAL TO, GREATER THAN or GREATER THAN OR EQUAL TO (or any of their syntactical equivalents).
  - b. The LAST record that satisfies the KEY clause specification is the relation check specified is LESS THAN or LESS THAN OR EQUAL TO (or any of their syntactical equivalents).
8. The START statement only positions the file for a subsequent sequential READ – it does not actually populate *file-name-1*s 01-level records with new data. You must issue a sequential READ after a successful START to actually read the record that satisfies the KEY clause.
9. The ON INVALID KEY clause will be triggered (thus executing *imperative-statement-1*) if an error occurred during the START. Such errors might be actual I/O errors or “Key Not Exists” errors (file status 23), indicating no record exists that satisfies the KEY clause requirements.

10. The NOT ON INVALID KEY clause will be triggered, thus executing *imperative-statement-2*, if no error occurred during the START.
11. Once the START statement has located the desired record (or not) and executed any specified *imperative-statement-1* or *imperative-statement-2* (or not), control transfers to the next statement following the START.

## 6.43. STOP

Figure 6-96 - STOP Syntax



The STOP statement halts the program, returning control to the operating system.

1. The RETURNING and GIVING clauses may be used interchangeably.
2. The *literal-2* option is supported syntactically but will be rejected if used (with a WARNING), as it is obsolete.
3. The optional RETURNING/GIVING clause allows the program to return a numeric return code to the operating system. The return code value can be in the range -2147483648 to +2147483647.
4. The two code snippets below are equivalent. They show two different ways a return code may be passed back to the operating system:

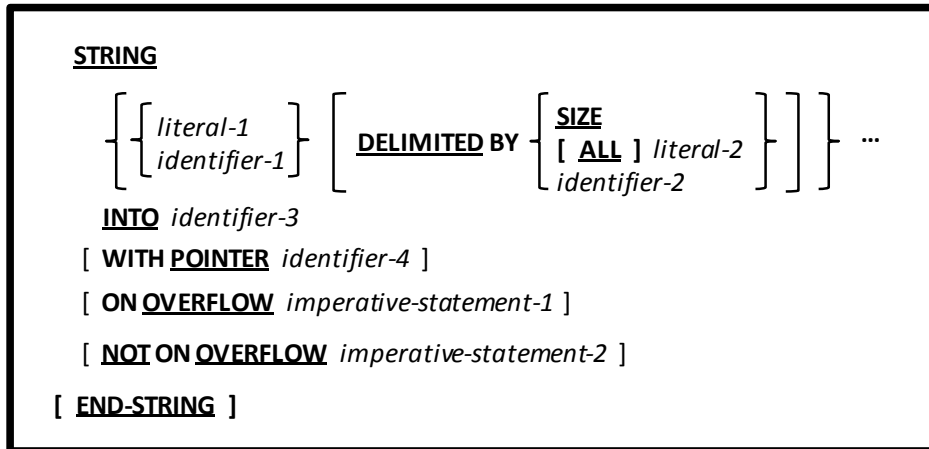
```
STOP RUN RETURNING 16
```

```
MOVE 16 TO RETURN-CODE
STOP RUN
```



## 6.44. STRING

Figure 6-97 - STRING Syntax



The STRING statement is used to concatenate all or a part of multiple strings together, forming a new string.

1. *Literal-1*, *literal-2*, *identifier-1*, *identifier-2* and *identifier-3* must be explicitly or implicitly defined as alphanumeric USAGE DISPLAY data. Any of those identifiers may be group items.
2. *Identifier-4* must be a non-edited elementary integer numeric data item with a value greater than zero.
3. Each *literal-1* / *identifier-1* will be known as the sending item while *identifier-3* will be known as the receiving item.
4. For each sending item, the contents of the sending item will be copied – character-by-character – into the receiving item; the first sending item will be copied into the receiving item beginning at the character position specified by the WITH POINTER clause (character positions are numbered upward from 1); if no WITH POINTER clause is specified, 1 will be assumed; the second sending item will be copied into the receiving item starting at the next character position after the last character transferred by the first item, and so forth.
5. Once the final character position of the receiving item has been filled, the STRING process will cease, regardless of whether or not there was more data to copy in the current sending item or even if there were more sending items to be processed.
6. If the DELIMITED BY SIZE option is specified for a sending item, the entire sending item will be copied. If no DELIMITED BY clause is specified, DELIMITED BY SIZE is assumed.
7. If a sending item has a DELIMITED BY clause without the SIZE option, the copying of the sending item will be terminated once the character sequence specified by *identifier-2* or **ALL** *literal-2* is found in the sending item.
8. The receiving item (*identifier-3*) is neither initialized (to SPACES or any other value) at the start of a STRING statement nor will it be SPACE filled should the total number of sending item characters copied into it be less than its size. You may explicitly INITIALIZE (section [6.25](#)) the receiving item yourself before executing the STRING if you wish.
9. If the value of *identifier-4* is less than 1 or if the receiving item runs out of space before all sending items have been fully processed, an overflow condition results. If an ON OVERFLOW clause is present in such a case, *imperative-statement-1* will be executed.
10. If there is no overflow condition and a NOT ON OVERFLOW clause is present, *imperative-statement-2* will be executed.
11. Once a STRING statement finishes and any imperative statements have been executed, control transfers to the next statement following the STRING.

## 6.45. SUBTRACT

### 6.45.1. SUBTRACT Format 1 – SUBTRACT FROM

Figure 6-98 - SUBTRACT FROM Syntax

```

SUBTRACT { [ LENGTH OF ] { literal-1 } } ...
           FROM { identifier-2 [ ROUNDED ] } ...
           [ ON SIZE ERROR imperative-statement-1 ]
           [ NOT ON SIZE ERROR imperative-statement-2 ]
           [ END-SUBTRACT ]

```

This format of the ADD statement generates the arithmetic sum of all arguments that appear before the FROM (*identifier-1* or *literal-1*) and then subtracts that sum from each of the identifiers listed after the TO (*identifier-2*).

1. *Identifier-1* and *identifier-2* must be numeric unedited data items.
2. *Literal-1* must be a numeric literal.
3. The **ROUNDED**, **ON SIZE ERROR** and **NOT ON SIZE ERROR** clauses are used in the same way as they are on the ADD statement (see section [6.5.1](#)).

### 6.45.2. SUBTRACT Format 2 – SUBTRACT GIVING

Figure 6-99 - SUBTRACT GIVING Syntax

```

SUBTRACT { [ LENGTH OF ] { literal-1 } } ...
           [ FROM identifier-2 ]
           GIVING { identifier-3 [ ROUNDED ] } ...
           [ ON SIZE ERROR imperative-statement-1 ]
           [ NOT ON SIZE ERROR imperative-statement-2 ]
           [ END-SUBTRACT ]

```

This format of the SUBTRACT statement generates the arithmetic sum of all arguments that appear before the FROM (*identifier-1* or *literal-1*), subtracts that sum from the contents of *identifier-2* and then replaces the contents of the identifiers listed after the GIVING (*identifier-3*) with that result.

1. *Identifier-1* and *identifier-2* must be numeric unedited data items.
2. *Identifier-3* must be a numeric (edited or unedited) data item.
3. *Literal-1* must be a numeric literal.
4. The **ROUNDED**, **ON SIZE ERROR** and **NOT ON SIZE ERROR** clauses are used in the same way as they are on the ADD statement (see section [6.5.1](#)).

### 6.45.3. SUBTRACT Format 3 – SUBTRACT CORRESPONDING

Figure 6-100 - SUBTRACT CORRESPONDING Syntax

```
SUBTRACT CORRESPONDING identifier-1 FROM identifier-2 [ ROUNDED ]  
    [ ON SIZE ERROR imperative-statement-1 ]  
    [ NOT ON SIZE ERROR imperative-statement-2 ]  
    [ END-SUBTRACT ]
```

This format of the SUBTRACT statement generates code equivalent to individual SUBTRACT FROM statements for corresponding matches of data items found subordinate to the two identifiers.

1. The rules for identifying corresponding matches are as discussed in section [6.29.2](#) – MOVE CORRESPONDING.
2. The ROUNDED, ON SIZE ERROR and NOT ON SIZE ERROR clauses are used in the same way as they are on the ADD statement (see section [6.5.1](#)).

## 6.46. SUPPRESS

Figure 6-101 - SUPPRESS Syntax



**SUPPRESS PRINTING**

Although syntactically recognized by the OpenCOBOL compiler, the SUPPRESS statement is non-functional because the RWCS (COBOL Report Writer) is not currently supported by OpenCOBOL.

## 6.47. TERMINATE

Figure 6-102 - TERMINATE Syntax

```
TERMINATE identifier-1 ...
```

Although syntactically recognized by the OpenCOBOL compiler, the TERMINATE statement is non-functional because the RWCS (COBOL Report Writer) is not currently supported by OpenCOBOL.

## 6.48. TRANSFORM

Figure 6-103 - TRANSFORM Syntax

```
TRANSFORM identifier-1 FROM { literal-1  
identifier-2 } TO { literal-2  
identifier-3 }
```

The TRANSFORM statement scans a data item performing a series of monoalphabetic substitutions, defined by the arguments before and after the “TO” clause.

1. The *literal-1* or *identifier-2* specified before the “TO” clause defines those characters in *identifier-1* that will be replaced. This will be referred to as the *target string*.
2. The *literal-2* or *identifier-3* specified after the “TO” clause defines those characters in *identifier-1* that will be replacing the characters specified by *literal-1* or *identifier-2*. This will be referred to as the *replacement string*.
3. The TRANSFORM verb was made obsolete in the 1985 standard of COBOL. Its function has been subsumed by the INSPECT statement – specifically the CONVERTING clause (section [6.27](#)).
4. The contents of *identifier-1* will be scanned – character by character. For each character, if that character appears in the target string, the corresponding character (by relative position) in the replacement string will then replace that character in *identifier-1*.
5. If the length of the replacement string exceeds that of the target string, the excess will be ignored.
6. If the length of the target string exceeds that of the replacement string, the replacement string will be assumed to be padded to the right with SPACES to make up the difference.

Figure 6-104 - The TRANSFORM Statement at Work

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DEMOTRANSFORM.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 Sample-Item PIC X(20) VALUE 'THIS IS A TEST'.
PROCEDURE DIVISION.
000-Main.
    TRANSFORM Sample-Item
        FROM 'ABCDEFGHIJKLMNQRSTUWXYZ'
        TO 'ZYXWVUTSRQPONMLKJIHGFE DCBA'
    DISPLAY
        Sample-Item
    END-DISPLAY
    STOP RUN
    .
```

Here's what the program DISPLAYs...

```
GSRH RH Z GVHG
```

## 6.49. UNLOCK

Figure 6-105 - UNLOCK Syntax

```
UNLOCK file-name-1 { RECORD  
                          RECORDS }
```

This statement syncs any as-yet unwritten file I/O buffers to the specified file (if any) and releases any record locks held for records belonging to the named file.

1. If *file-name-1* is a SORT file, no action will be taken.
2. Not all OpenCOBOL implementations support locking. Whether they do or not depends upon the operating system they were built for and the build options that were used when OpenCOBOL was generated.<sup>31</sup> When a program using one of those OpenCOBOL implementations issues an UNLOCK, it will be ignored. There will be no compiler message issued. Buffer syncing, if needed, will still occur.

---

<sup>31</sup> The author of this manual – for example – uses an OpenCOBOL build for Windows that utilizes the MinGW build/runtime environment and uses the Berkeley Database module for advanced file I/O. That OpenCOBOL build does NOT support LOCKing. Generally speaking, UNIX builds will support record locking.

## 6.50. UNSTRING

Figure 6-106 - UNSTRING Syntax

```

UNSTRING identifier-1
  DELIMITED BY { [ ALL ] literal-1 } [ OR { [ ALL ] literal-2 } ] ...
  INTO identifier-4 [ DELIMITER IN identifier-5 ] [ COUNT IN identifier-6 ]
    [ identifier-7 [ DELIMITER IN identifier-8 ] [ COUNT IN identifier-9 ] ] ...
  [ WITH POINTER identifier-10 ]
  [ TALLYING IN identifier-11 ]
  [ ON OVERFLOW imperative-statement-1 ]
  [ NOT ON OVERFLOW imperative-statement-2 ]
  [ END-UNSTRING ]

```

The UNSTRING statement parses a string, extracting any number of substrings from it.

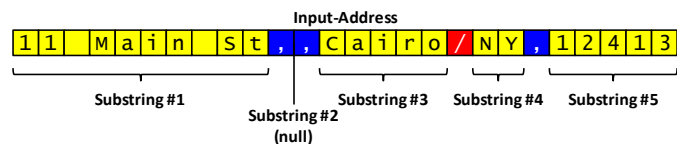
1. *Identifier-1* through *identifier-5*, *identifier-7* and *identifier-8* must be explicitly or implicitly defined as alphanumeric USAGE DISPLAY data. Any of those identifiers may be group items.
2. *Literal-1* and *literal-2* must be alphanumeric literals.
3. *Identifier-6* and *identifier-9* through *identifier-11* must be elementary non-edited integer numeric items.
4. *Identifier-10* must have a value greater than 0.
5. *Identifier-1* is known as the *source string*. *Identifier-4* and *identifier-7* are known as the *destination fields*.
6. The source string will be broken up into substrings starting from the character position indicated by *identifier-10* (or from position 1 if there is no WITH POINTER clause). If the initial value of *identifier-10* is less than 1 or greater than the size of the source string, an "overflow" condition results. Overflow is discussed in item #13.
7. Substrings are identified by using the various delimiter strings specified on the DELIMITED BY clause as inter-substring separators. Using the "ALL" option allows a delimiter sequence to be an arbitrarily long sequence of occurrences of the delimiter literal whereas its absence treats each occurrence as a separate delimiter.
8. Two consecutive delimiter sequences will identify a null substring.
9. Here is an example of how a source string will be parsed into substrings:

```

UNSTRING Input-Address
DELIMITED BY " " OR "/"
INTO
  Street-Address DELIMITER D1 COUNT C1
  Apt-Number     DELIMITER D2 COUNT C2
  City           DELIMITER D3 COUNT C3
  State         DELIMITER D4 COUNT C4
  Zip-Code      DELIMITER D5 COUNT C5
END-UNSTRING

```

Figure 6-107 - An UNSTRING Example



With the sample data shown, the UNSTRING statement will identify a total of five (5) substrings from the data. The result of this identification will be as if the following MOVE statements were executed:

```

MOVE "11 Main St" TO Street-Address
MOVE ""          TO Apt-Number32
MOVE "Cairo"    TO City
MOVE "NY"       TO State
MOVE "12413"   TO Zip-Code

```

If not enough substrings can get identified to populate all the destination fields, those for which no data can be found remain unchanged.

<sup>32</sup> A MOVE of the null string has the same effect as a MOVE of SPACES



If not enough destination fields are specified to receive all the substrings, the excess substrings are “thrown away”. An “overflow” condition will exist, however. Overflow is discussed in item #13.

10. Each destination field may have an optional DELIMITER clause. If a DELIMITER clause is specified, *identifier-5* (or *identifier-8*) will have the delimiter character string used to identify the substring for the destination field MOVED to it. Using the example shown earlier, the following implied MOVES will occur for the DELIMITER identifiers:

```
MOVE “,” TO D1
MOVE “,” TO D2
MOVE “/” TO D3
MOVE “,” TO D4
MOVE SPACES TO D533
```

11. Each destination field may have an optional COUNT clause. If a COUNT clause is specified, *identifier-6* (or *identifier-9*) will have the size of the substring for the destination field MOVED to it. Using the example shown earlier, the following implied MOVES will occur for the COUNT identifiers:

```
MOVE 10 TO C1
MOVE 0 TO C2
MOVE 5 TO C3
MOVE 2 TO C4
MOVE 5 TO C5
```

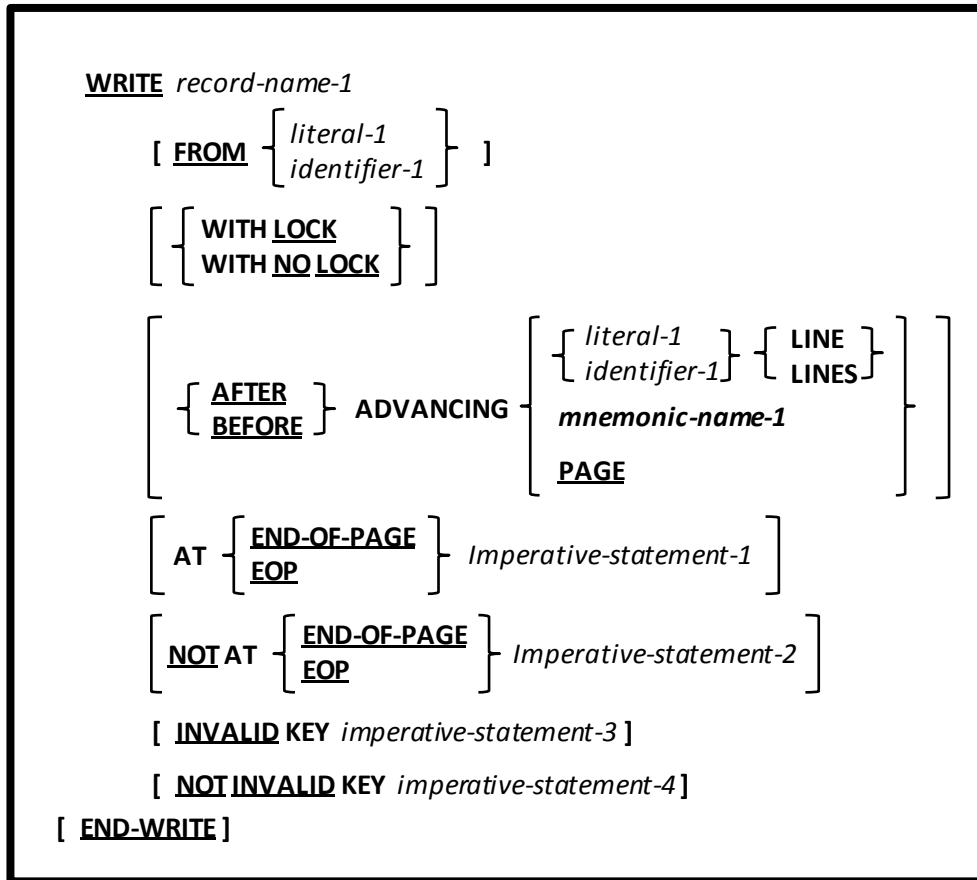
12. The TALLYING clause – if present – will be incremented by 1 each time a parsed substring is MOVED to a destination field. This field is NOT initialized to zero by UNSTRING, so you’ll want to do that yourself.
13. The optional ON OVERFLOW clause, if present, will trigger the execution of *imperative-statement-1* if an overflow condition occurs (see item #6 and #7). If the ON OVERFLOW clause triggers, the NOT ON OVERFLOW clause (if any) will be ignored.
14. The optional NOT ON OVERFLOW clause, if present, will trigger the execution of *imperative-statement-2* if no overflow condition occurred (see item #6 and #7). If the NOT ON OVERFLOW clause triggers, the ON OVERFLOW clause (if any) will be ignored.
15. Once the source string has been parsed, the appropriate destination fields have been updated (along with any DELIMITER/COUNT fields), *identifier-11* (TALLYING) has been incremented and any ON OVERFLOW or NOT ON OVERFLOW imperative statement has been executed, control will fall into the next statement following the UNSTRING.

---

<sup>33</sup> The last substring always has a delimiter of null which – when MOVED to the DELIMITER field - becomes SPACES.

## 6.51. WRITE

Figure 6-108 - WRITE Syntax



The WRITE statement writes a new record to an OPEN file.

1. *Record-name-1* must be defined as an 01-level record subordinate to the File Description (FD – see section [5.1](#)) of a file that is currently OPEN (section [6.32](#)) for OUTPUT, I-O or EXTEND.
2. *Literal-1* or *identifier-1* must be explicitly or implicitly defined as alphanumeric USAGE DISPLAY data. *Identifier-1* may be a group item.
3. The optional FROM clause will cause *literal-1* or *identifier-1* to be implicitly MOVEd into *record-name-1* prior to writing *record-name-1* to the file.
4. See section [6.1.9.2](#) for a discussion of the record LOCK options.
5. The ADVANCING clause is intended for use with ORGANIZATION LINE SEQUENTIAL files that will have reports written to them. Using this clause with any other ORGANIZATION will either be rejected outright by the compiler (ORGANIZATION IS RELATIVE or ORGANIZATION IS INDEXED) or may introduce unwanted characters into the file (ORGANIZATION IS RECORD BINARY SEQUENTIAL).
6. The ADVANCING n LINES clause will introduce the specified number of line-feed (X"10") characters into the file either before the written record (AFTER ADVANCING) or after the written record (BEFORE ADVANCING).
7. If no ADVANCING clause is specified on a WRITE to an ORGANIZATION LINE SEQUENTIAL file, AFTER ADVANCING 1 LINE will be assumed.
8. The ADVANCING PAGE clause will introduce a form-feed (X"0C") characters into the file either before the written record (AFTER ADVANCING) or after the written record (BEFORE ADVANCING).
9. If the file being written to contains a LINAGE clause (section [5.1](#)) in its FD, an internal line counter will be maintained by the runtime library and – when appropriate, the appropriate number of ASCII line-feed characters

will be automatically written to the file to accommodate the LINES AT TOP and/or LINES AT BOTTOM specification of the LINAGE definition.

10. The AT END-OF-PAGE and NOT AT END-OF-PAGE clauses are legal only for ORGANIZATION LINE SEQUENTIAL or ORGANIZATION RECORD BINARY SEQUENTIAL files whose file descriptions contain a LINAGE clause (section [5.1](#)).
11. The AT END-OF-PAGE clause will be triggered (thus executing *imperative-statement-1*) if an end-of-page condition occurred during the WRITE. End-of-page conditions occur when a WRITE statement introduces a data line or line-feed character into the file at a line position that occurs within the Page Footer area (see [Figure 5-3](#)).
12. The NOT AT END-OF-PAGE clause will be triggered (thus executing *imperative-statement-2*) if no end-of-page condition occurred during the WRITE.
13. The behavior of the combination of ADVANCING and AT END-OF-PAGE clauses needs to be understood in order to get the desired results. To that end, here is the sequence of events that will occur with a WRITE statement that involves these clauses:
  - a. If AFTER ADVANCING is specified:
 

If AFTER ADVANCING PAGE was specified, a form-feed character is written to the file and the internal end-of-page switch is set

OTHERWISE the appropriate number of line-feed characters (ADVANCING n LINES) will be written to the file; if the internal LINAGE counter shows that these line feeds have caused the maximum available usable lines on a logical page to be exhausted, the internal end-of-page switch is set.
  - b. The data record is written to the file. If the internal LINAGE counter shows that writing this record has caused the maximum available usable lines on a logical page to be exhausted, the internal end-of-page switch is set.
  - c. If BEFORE ADVANCING is specified:
 

If BEFORE ADVANCING PAGE was specified, a form-feed character is written to the file and the internal end-of-page switch is set

OTHERWISE the appropriate number of line-feed characters (ADVANCING n LINES) will be written to the file; if the internal LINAGE counter shows that these line feeds have caused the maximum available usable lines on a logical page to be exhausted, the internal end-of-page switch is set.
  - d. If the internal end-of-page switch is not set, *imperative-statement-2* (if any) is executed

OTHERWISE (the internal end-of-page switch is set), *imperative-statement-1* (if any) is executed
14. With the information from #13 above in mind, here's a cute trick for letting the AT END-OF-PAGE clause automatically generate page headings on your reports:

```

FD Report-File
  LINAGE IS 66 LINES
  .....WITH FOOTER AT 57
  .....LINES AT TOP 3
  .....LINES AT BOTTOM 3
  .
  .
  OPEN OUTPUT Report-File
  PERFORM Generate-Page-Header
  .
  .
  WRITE Report-Rec AFTER ADVANCING 1 LINE
  AT END-OF-PAGE PERFORM Generate-Page-Header
  END-WRITE
  .
  .
  CLOSE Report-File

```

15. The INVALID KEY and NOT INVALID KEY clauses are legal only on WRITE statements used against for ORGANIZATION RELATIVE or ORGANIZATION INDEXED files.

16. The ON INVALID KEY clause will be triggered (thus executing *imperative-statement-3*) if an error occurred during the WRITE. Such errors might be actual I/O errors or “Key Exists” errors (file status 22), indicating you tried to WRITE a record that already existed.
17. The NOT ON INVALID KEY clause will be triggered (thus executing *imperative-statement-4*) if no error occurred during the WRITE.



## 7. The OpenCOBOL System Interface

### 7.1. Using the OpenCOBOL Compiler (cobc)

#### 7.1.1. Introduction

Program source files should have extensions of “.cob” or “.cbl”.

Program filenames should match exactly the specification of PROGRAM-ID (including case). The reason for this was discussed in section [3](#).

Spaces cannot be included in PROGRAM-IDs and therefore should not be included in program filenames.

The OpenCOBOL compiler will translate your COBOL program into C source code, compile that C source code into executable binary form using the “C” compiler specified when OpenCOBOL was built and link that executable binary into either directly executable form, static-linkable form or dynamically-loadable executable form.

The OpenCOBOL compiler is named “cobc” (“cobc.exe” on a Windows system).

#### 7.1.2. Syntax and Options

The following describes the syntax and option switches of the cobc command. This information may be displayed by entering the command “cobc --help”.

Usage: cobc [options] file...

```

options:
--help                Display this message
--version, -v         Display compiler version
--info, -i            Display compiler build information
-v                   Display the commands invoked by the compiler
-x                   Build an executable program
-m                   Build a dynamically loadable module (default)
-std=<dialect>       warnings/features for a specific dialect :
                    cobol2002  Cobol 2002
                    cobol85   Cobol 85
                    ibm       IBM Compatible
                    mvs       MVS Compatible
                    bs2000    BS2000 Compatible
                    mf        Micro Focus Compatible
                    default    when not specified
                    See config/default.conf and config/*.conf
-free                Use free source format
-fixed               Use fixed source format (default)
-O, -O2, -Os         Enable optimization
-g                   Produce debugging information in the output
-debug              Enable all run-time error checking
-o <file>            Place the output into <file>
-b                  Combine all input files into a single
                    dynamically loadable module
-E                  Preprocess only; do not compile, assemble or link
-C                  Translation only; convert COBOL to C
-S                  Compile only; output assembly file
-c                  Compile and assemble, but do not link
-P                  Generate preprocessed program listing (.lst)
-xref               Generate cross reference through 'cobxref'
                    (V. Coen's 'cobxref' must be in path)
-I <directory>      Add <directory> to copy/include search path
-L <directory>      Add <directory> to library search path
-l <lib>             Link the library <lib>
-A <options>         Add <options> to the C compile phase
-Q <options>         Add <options> to the C link phase
-D <define>         Pass <define> to the C compiler
-conf=<file>        User defined dialect configuration - See -std=
--list-reserved      Display reserved words
--list-intrinsics    Display intrinsic functions
--list-mnemonics     Display mnemonic names
-save-temps(=<dir>) Save intermediate files (default current directory)
-MT <target>        Set target file used in dependency list
-MF <file>          Place dependency list into <file>
-ext <extension>    Add default file extension

-w                   Enable ALL warnings
-wall                Enable all warnings except as noted below
-wobsolete           warn if obsolete features are used

```

-warchaic	warn if archaic features are used
-wredefinition	warn incompatible redefinition of data items
-wconstant	warn inconsistent constant
-wparentheses	warn lack of parentheses around AND within OR
-wstrict-typing	warn type mismatch strictly
-wimplicit-define	warn implicitly defined data items
-wcall-params	warn non 01/77 items for CALL params (NOT set with -wall)
-wcolumn-overflow	warn text after column 72, FIXED format (NOT set with -wall)
-wterminator	warn lack of scope terminator END-XXX (NOT set with -wall)
-wtruncate	warn possible field truncation (NOT set with -wall)
-wlinkage	warn dangling LINKAGE items (NOT set with -wall)
-wunreachable	warn unreachable statements (NOT set with -wall)
-ftrace	Generate trace code (Executed SECTION/PARAGRAPH)
-ftraceall	Generate trace code (Executed SECTION/PARAGRAPH/STATEMENTS)
-fsyntax-only	Syntax error checking only; don't emit any output
-fdebugging-line	Enable debugging lines ('D' in indicator column)
-fsource-location	Generate source location code (Turned on by -debug or -g)
-fimplicit-init	Do automatic initialization of the Cobol runtime system
-fsign-ascii	Numeric display sign ASCII (Default on ASCII machines)
-fsign-ebcdic	Numeric display sign EBCDIC (Default on EBCDIC machines)
-fstack-check	PERFORM stack checking (Turned on by -debug or -g)
-ffold-copy-lower	Fold COPY subject to lower case (Default no transformation)
-ffold-copy-upper	Fold COPY subject to upper case (Default no transformation)
-fwrite-after	Use AFTER 1 for WRITE of LINE SEQUENTIAL (Default BEFORE 1)
-fnotrunc	Do not truncate binary fields according to PICTURE
-ffunctions-all	Allow use of intrinsic functions without FUNCTION keyword
-fmfcomment	'*' or '/' in column 1 treated as comment (FIXED only)
-fnull-param	Pass extra NULL terminating pointers on CALL statements

As discussed in section 2, program compilation units may consist of multiple programs defined sequentially in a single source file. By specifying multiple source files on the “cobc” command, it is possible for a single execution of the “cobc” command to process multiple compilation units.

### 7.1.3. Compiling Executable Programs

The simplest mode of compilation is to generate a single executable file from one or more OpenCOBOL source files:

```
cobc -x prog1.cbl prog2.cbl prog3.cbl
```

The main program must be the first program unit found in the “prog1.cbl” file. The remainder of “prog1.cbl” as well as all of “prog2.cbl” and “prog3.cbl” must be subprograms or nested subprograms.

This will generate a single executable file (UNIX) or exe file (Windows) which has all required COBOL programs included in the file. The dynamically-loadable runtime libraries for OpenCOBOL, GMP and BDB (or whatever other file I/O module was built-in to the OpenCOBOL package you are using) are still required to be available at run-time, however.

### 7.1.4. Dynamically-Loadable Subprograms

Subprograms that are to be dynamically loaded into memory at execution time must be compiled using the “-m” option on the cobc command, as follows:

```
cobc -m sprog1.cbl
```

- or -

```
cobc -m sprog1.cbl sprog2.cbl sprog3.cbl
```

The first command above generates a single dynamically-loadable module while the second example generates three.

The following rules apply to dynamically-loaded modules and the subroutines contained within them:

1. A Dynamically-loadable module generated from a source file named “xxxxxxx.cbl” or “xxxxxxx.cob” will be named “xxxxxxx.so” on a UNIX system or “xxxxxxx.dll” on a Windows system.
2. Dynamically-loadable modules containing only a single subprogram are created from OpenCOBOL source files having only a single program-unit. The PROGRAM-ID of that program unit must match exactly the filename (minus “.cbl” or “.cob”) of the source code as well the filename (minus the “.so” or “.dll”) of the dynamically-loadable module.

3. Dynamically-loadable modules containing multiple subprograms are created from a single OpenCOBOL source file containing multiple program-units. The PROGRAM-ID of one of those program units must match exactly the filename (minus “.cbl” or “.cob”) of the source code as well the filename (minus the “.so” or “.dll”) of the dynamically-loadable module. This PROGRAM-ID is known as the *primary entry-point* of the dynamically-loadable module.
4. When a program CALLs a subprogram that is contained within a dynamically-loadable module
  - a. The OpenCOBOL runtime library performs a search of all currently-loaded dynamically-loadable modules for the subprogram entry-point (the entry-point is the literal or identifier coded on the CALL statement (see section [6.8](#)). That entry-point will have been defined as either a PROGRAM-ID (section [3](#)) or ENTRY point (section [6.17](#)) within the source file that created the dynamically-loadable module.
  - b. If the entry-point was found, control is transferred to there and the subprogram begins execution.
  - c. If the entry-point was not found, the OpenCOBOL runtime library searches for a file named “xxxxxxx.so” (UNIX) or “xxxxxxx.dll” (Windows), where xxxxxxxx is the desired subroutine entry-point.
    - i. If a file was located, it is loaded and control is transferred to the entry-point within it so the subprogram may begin execution.
    - ii. If a file was not located, an error message (“**libcob: Cannot find module 'xxxxxxx'**”) is issued and program execution is aborted.
5. Rule #4 has a profound implication to subprogramming with dynamically-loadable modules containing multiple entry-points – You must successfully CALL the *primary entry-point* of the module (see rule #3) before you can CALL any other entry-points within the module.

It is also possible to generate main programs as dynamically-loadable libraries. Just use the “-m” option (as shown here) rather than the “-x” option. To execute these main programs, you’ll need to utilize the cobcrun command, as discussed in section [7.2.2](#).

### 7.1.5. Static Subroutines

You may also compile OpenCOBOL subroutines into assembler source code which can then be assembled and linked with a main program when that main program is compiled. To create such an assembler source file, compile the subprogram(s) as follows:

```
cobc -S sprog1.cbl (Note: “-S” is an uppercase-S)
```

This will create an assembler source file named “sprog1.s”. If you specify multiple input files, they’ll each create their own “.s” files.

To compile a main program, assemble an assembler source file and static-link it all together:

```
cobc -x mainprog.cbl sprog1.s
```

If multiple subprograms are needed, simply add their “.s” files to the command line. Any subprogram *entry-points* for which “.s” files were not specified will be CALLED at runtime as dynamically-loadable modules.

### 7.1.6. Combining COBOL and C Programs

Linkage between OpenCOBOL and C language programs is possible, but may require a little bit of special coding in one program or the other in order to meaningfully pass data between them. The issues involved deal predominantly with three topics, as follows. Each issue is discussed, with upcoming coding samples illustrating specifics as to how those issues are overcome in actual program code.

#### 7.1.6.1. OpenCOBOL Run-Time Library Requirements

Like most other implementations of the COBOL language, OpenCOBOL utilizes a run-time library. When the first program unit executed in a given execution sequence is an OpenCOBOL program, any run-time library initialization will



be performed by that COBOL code in a manner that is transparent to the C-language programmer. If, however, a C program unit is the first to execute, the burden of perform OpenCOBOL run-time library initialization falls upon the C program.

### 7.1.6.2. String Allocation Differences Between OpenCOBOL and C

Both languages store strings as a fixed-length continuous sequence of characters.

COBOL stores these character sequences up to a specific quantity limit imposed by the PICTURE clause of the data item. For example:

```
01 LastName PIC X(15).
```

There is never an issue of exactly what the length of a string contained in a USAGE DISPLAY data item is – there are always exactly how ever many characters as were allowed for by the PICTURE clause. In the example above, “LastName” will always contain exactly fifteen characters; of course, there may be anywhere from 0 to 15 trailing SPACES as part of the current LastName value.

C actually has no “string” datatype – rather, it stores strings as an array of “char” datatype items where each element of the array is a single character. Being an array, there is an upper limit to how many characters may be stored in a given “string”. For example:

```
char lastName[15]; /* 15 chars: lastName[0] thru lastName[14] */
```

C provides a robust set of string-manipulation functions to copy strings from one char array to another, search strings for certain characters, compare one char array to another, concatenate char arrays and so forth. To make these functions possible, it was necessary to be able to define the logical end of a string. C accomplishes this via the expectation that all strings (char arrays) will be terminated by a NULL character (x'00'). Of course, no one forces a programmer to do this, but if [s]he ever expects to use any of the C standard functions to manipulate that string they had better be doing it.

So, OpenCOBOL programmers expecting to pass strings to or receive strings from C programs had best be prepared to deal with the null-termination issue.

### 7.1.6.3. Matching C Data Types with OpenCOBOL USAGES

This is pretty simple, the OpenCOBOL and C programmer must just be aware of the following correspondence between C data types and COBOL USAGE specifications:

Figure 7-1 - C/OpenCOBOL Data Type Matches

This COBOL USAGE... (no PICTURE allowed)	Occupies this space...	Holds these numeric values...	And corresponds to this C data type...
BINARY-CHAR BINARY-CHAR UNSIGNED	1 byte	0 to 255	unsigned char
BINARY-CHAR SIGNED	1 byte	-128 to +127	signed char
BINARY-SHORT BINARY-SHORT UNSIGNED	2 bytes	0 to 65535	unsigned unsigned int unsigned short unsigned short int
BINARY-SHORT SIGNED	2 bytes	-32768 to +32767	int short short int signed int

This COBOL USAGE... (no PICTURE allowed)	Occupies this space...	Holds these numeric values...	And corresponds to this C data type...
			signed short signed short int
BINARY-LONG BINARY-LONG UNSIGNED	4 bytes	0 to 4294967295	unsigned long unsigned long int
BINARY-LONG SIGNED	4 bytes	-2147483648 to +2147483647	long long int signed long signed long int
BINARY-C-LONG SIGNED	4 bytes or 8 bytes	-2147483648 to +2147483647 Or -9223372036854775808 to +9223372036854775807	long (see the description of USAGE BINARY-C-LONG in <a href="#">Figure 5-10</a> )
BINARY-DOUBLE BINARY-DOUBLE UNSIGNED	8 bytes	0 to 18446744073709551615	unsigned long long unsigned long long int
BINARY-DOUBLE SIGNED	8 bytes	-9223372036854775808 to +9223372036854775807	long long int signed long long int
COMPUTATIONAL-1	4 bytes	$-3.4 \times 10^{38}$ to $+3.4 \times 10^{38}$ (six decimal digits of precision)	float
COMPUTATIONAL-2	8 bytes	$-1.7 \times 10^{308}$ to $+1.7 \times 10^{308}$ (15 decimal digits of precision)	double
N/A (no OpenCOBOL equivalent)	12 bytes	$-1.19 \times 10^{4932}$ to $+1.19 \times 10^{4932}$ (18 decimal digits of precision)	long double

There are other OpenCOBOL PICTURE/USAGE combinations that can define the same storage size and value range combinations, but (with the exception of COMP-1 and COMP-2), these are the ANSI2002 standard specifications for C-program data compatibility and OpenCOBOL programmers should get used to using them when data is being shared with C programs (they're good documentation too, highlighting the fact that the data will be "shared" with a C program).

The minimum values shown for the various SIGNED integer USAGEs are appropriate for a computer system that uses 2s-complement representation for negative signed binary values (such as those CPUs typically found in Windows PCs). A computer system using 1s-complement representation for negative signed binary values would have minimum values that are 1 greater (-127 instead of -128, for example).

### 7.1.6.4. OpenCOBOL Main Programs CALLing C Subprograms

Here are samples of an OpenCOBOL program that CALLs a C subprogram.

Figure 7-2 - OpenCOBOL CALLing C

(maincob.cbl)	(subc.c)
This OpenCOBOL MAIN PROGRAM...	...wants to CALL this C SUBPROGRAM
<pre> IDENTIFICATION DIVISION. PROGRAM-ID. maincob. DATA DIVISION. WORKING-STORAGE SECTION. 01 Arg1          PIC X(7). 01 Arg2          PIC X(7). 01 Arg3          USAGE BINARY-LONG. PROCEDURE DIVISION. 000-Main.   DISPLAY 'Starting cobmain'.   MOVE 123456789 TO Arg3.   STRING 'Arg1'         X'00'         DELIMITED SIZE         INTO Arg1   END-STRING.   STRING 'Arg2'         X'00'         DELIMITED SIZE         INTO Arg2   END-STRING.   CALL 'subc' USING BY CONTENT Arg1,                   BY REFERENCE Arg2,                   BY REFERENCE Arg3.    DISPLAY 'Back'.   DISPLAY 'Arg1=' Arg1.   DISPLAY 'Arg2=' Arg2.   DISPLAY 'Arg3=' Arg3.   DISPLAY 'Returned value='   RETURN-CODE. STOP RUN. </pre>	<pre> #include &lt;stdio.h&gt;  int subc(char *arg1,          char *arg2,          unsigned long *arg3) {   char nu1[7]="New1";   char nu2[7]="New2";   printf("Starting subc\n");   printf("Arg1=%s\n",arg1);   printf("Arg2=%s\n",arg2);   printf("Arg3=%d\n",*arg3);   arg1[0]='x';   arg2[0]='y';   *arg3=987654321;   return 2; } </pre>

The idea is to pass two string and one full-word unsigned arguments to the subprogram, have the subprogram print them out, change all three and pass a return code of 2 back to the caller. The caller will then re-display the three arguments (showing changes only to the two BY REFERENCE arguments), display the return code and halt. While simple, these two programs illustrate the techniques required quite nicely.

Note how the COBOL program ensures that a null end-of-string terminator is present on both string arguments.

Since the C program is planning on making changes to all three arguments, it declares all three as pointers in the function header and references the third argument as a pointer in the function body.<sup>34</sup>

<sup>34</sup> It actually had no choice for the two string (char array) arguments – they must be defined as pointers in the function even though the function code references them without the leading “\*” that normally signifies pointers.

These programs are compiled and executed as follows. The example assumes a UNIX system with an OpenCOBOL build that uses the native C compiler on that system; the technique works equally well regardless of which C compiler and which operating system you're using.

```
$ cc -c subc.c
$ cobc -x maincob.cbl subc.o
$ maincob
Starting cobmain
Starting subc
Arg1=Arg1
Arg2=Arg2
Arg3=123456789
Back
Arg1=Arg1
Arg2=Yrg2
Arg3=+0987654321
Returned value=+000000002
$
```

Remember that the null characters are actually in the OpenCOBOL "Arg1" and "Arg2" data items. They don't appear in the output, but they ARE there. When passing character strings to C programs, it's probably a good idea to make a null-terminated copy of the string items and pass those copies to the C program.

As was discussed in section [6.8](#), an OpenCOBOL CALL to a subprogram will need to specify the BY CONTENT clause to make an argument unchangeable by a subprogram if that subprogram is written in a language other than OpenCOBOL. When the CALLing and CALLED programs are both OpenCOBOL, the BY VALUE will be a faster alternative to BY CONTENT.

### 7.1.6.5. C Main Programs CALLing OpenCOBOL Subprograms

Now, the roles of the two languages in the previous section will be reversed, having a C main program execute an OpenCOBOL subprogram.

Figure 7-3 - C CALLing OpenCOBOL

(mainc.c) This C MAIN PROGRAM...	(subcob.cbl) ...wants to CALL this OpenCOBOL SUBPROGRAM
<pre>#include &lt;libcob.h&gt; #include &lt;stdio.h&gt;  int main (int argc, char **argv) {     int returnCode;     char arg1[7] = "Arg1";     char arg2[7] = "Arg2";     unsigned long arg3 = 123456789;     printf("Starting mainc...\n");     cob_init (argc, argv);     /* cob_init(0,NULL) if cmdline args not going     to COBOL */     returnCode = subcob(arg1,arg2,&amp;arg3);     printf("Back\n");     printf("Arg1=%s\n",arg1);     printf("Arg2=%s\n",arg2);     printf("Arg3=%d\n",arg3);     printf("Returned value=%d\n",returnCode);     return returnCode; }</pre>	<pre>IDENTIFICATION DIVISION. PROGRAM-ID. subcob. DATA DIVISION. LINKAGE SECTION. 01 Arg1          PIC X(7). 01 Arg2          PIC X(7). 01 Arg3          USAGE BINARY-LONG. PROCEDURE DIVISION USING     BY VALUE     Arg1,     BY REFERENCE Arg2,     BY REFERENCE Arg3. 000-Main.     DISPLAY 'Starting cobsub.cbl'.     DISPLAY 'Arg1=' Arg1.     DISPLAY 'Arg2=' Arg2.     DISPLAY 'Arg3=' Arg3.     MOVE 'X' TO Arg1 (1:1).     MOVE 'X' TO Arg2 (1:1).     MOVE 987654321 TO Arg3.     MOVE 2 TO RETURN-CODE.     GOBACK.</pre>

Since the C program is the one that will execute first, before the OpenCOBOL subroutine, the burden of initializing the OpenCOBOL run-time environment lies with that C program; it will have to invoke the "cob\_init" function, which is part of the "libcob" library. The two required C statements are shown, highlighted in boldface.

The arguments to the "cob\_init" routine are the argument count and value parameters passed to the main function when the program began execution. By passing them into the OpenCOBOL subprogram, it will be possible for that OpenCOBOL program to retrieve the command line or individual command-line arguments. If that won't be necessary, "cob\_init(0,NULL);" could be specified instead.

Since the C program wants to allow “arg3” to be changed by the subprogram, it prefixes it with a “&” to force a CALL BY REFERENCE for that argument. Since “arg1” and “arg2” are strings (char arrays), they are automatically passed by reference.

Here’s the output of the compilation process as well as the program’s execution. The example assumes a Windows system with an OpenCOBOL build that uses the GNU C compiler on that system; the technique works equally well regardless of which C compiler and which operating system you’re using.

```
C:\Users\Gary\Documents\Programs> cobc -S subcob.cb1
C:\Users\Gary\Documents\Programs> gcc mainc.c subcob.s -o mainc.exe -llibcob
C:\Users\Gary\Documents\Programs> mainc.exe
Starting mainc...
Starting cobsb.cb1
Arg1=Arg1
Arg2=Arg2
Arg3=+0123456789
Back
Arg1=Xrg1
Arg2=Xrg2
Arg3=987654321
Returned value=2
C:\Users\Gary\Documents\Programs>
```

Note that even though we told OpenCOBOL that the 1<sup>st</sup> argument was to be BY VALUE, it was treated as if it were BY REFERENCE anyway. String (char array) arguments passed from C callers to OpenCOBOL subprograms will be modifiable by the subprogram. It’s best to pass a copy of such data if you want to ensure that the subprogram doesn’t change it.

The third argument is different, however. Since it’s not an array you have the choice of passing it either BY REFERENCE<sup>35</sup> or BY VALUE<sup>36</sup>.

---

<sup>35</sup> Use “&” with the argument in the C calling program; specify the argument as BY REFERENCE in the COBOL subprogram

<sup>36</sup> Don’t use “&” with the argument in the C calling program; specify the argument as BY VALUE in the COBOL subprogram

### 7.1.7. Important Environment Variables

The following chart documents the various environment variables that can play a role in the compilation of OpenCOBOL programs.

Figure 7-4 - Compiler Environment Variables

Environment Variable	Use
COB_CC	Set to the name of the C compiler you wish OpenCOBOL to use.  <b>USE THIS FEATURE AT YOUR OWN RISK – YOU SHOULD ALWAYS USE THE C COMPILER YOUR OPENCOBOL BUILD WAS GENERATED FOR</b>
COB_CFLAGS <sup>37</sup>	Set to any switches that you'd like to pass on to the C compiler from the cobc compiler (in addition to any that cobc will specify). The default is " <b>-lprefix/include</b> ", where " <i>prefix</i> " is the path prefix specified when the OpenCOBOL binaries you are using were created.
COB_CONFIG_DIR	Set to the path to the folder where OpenCOBOL "config" files are kept. See section 7.1.8 for information on how those config files are used.
COB_COPY_DIR	If COPY modules your program needs are NOT stored in the same directory as your program, set this environment variable to the folder in which the COPY modules may be found (IBM mainframe programmers will recognize this as "SYSLIB").
COB_LDADD <sup>37</sup>	Set to any additional linker switches (ld) that can specify where standard libraries that must be linked with the program can be found. The default is "" (null).
COB_LDFLAGS <sup>37</sup>	Set to any linker/loader (ld) switches that you'd like to pass on to the C compiler from the cobc compiler (in addition to any that cobc will specify). The default is none.
COB_LIBS <sup>37</sup>	Set to any linker switches (ld) that specify where standard libraries that must be linked with the program can be found. The default is " <b>-Lprefix/lib -lcob</b> ", where " <i>prefix</i> " is the path prefix specified when the OpenCOBOL binaries you are using were created.
LD_LIBRARY_PATH	If you are planning on using static-linked subroutine libraries, set this variable to the path to the directory containing your libraries.

<sup>37</sup> These switches are intended for use only in very special circumstances by very advanced users; their usage is discouraged. A future release of OpenCOBOL will introduce a better way to pass switched to the C compiler and/or the loader from the cobc command.

Environment Variable	Use
TMPDIR  TMP (checked in this order)	Set to a directory/folder appropriate to create temporary files in. The intermediate working files created by cobc will be created here (and deleted once they're no longer needed).  On a Windows system, the TMP environment variable is normally set for you when you logon. If you wish to use a <u>different</u> temporary folder, you may set TMPDIR yourself and have no fear of disrupting other Windows software that relies on TMP.

### 7.1.8. Locating Copybooks at Compilation Time

The OpenCOBOL compiler will attempt to locate copybooks by searching for them in the following folders. The search will occur in this sequence:

- The folder in which the program being compiled resides.
- The folder named on the “-I” compiler switch (see section [7.1.2](#))
- The folder specified on the COB\_COPY\_DIR environment variable (see section [0](#)).

As each of the above folders is searched for a copybook - “COPYXXXXXXXX.”, for example – the OpenCOBOL compiler will attempt to locate the copybook file by any of the following names, in the sequence shown:

- XXXXXXXX.CPY
- XXXXXXXX.CBL
- XXXXXXXX.COB
- XXXXXXXX.cpy
- XXXXXXXX.cbl
- XXXXXXXX.cob
- XXXXXXXX

The COPY command is case-sensitive on UNIX systems; “COPY copybookname” and “COPY COPYBOOKNAME” will both fail to locate the “CopyBookName” copybook on a UNIX system. Windows implementations of OpenCOBOL may or may not be similarly case sensitive with regard to copybook names, depending upon the Windows version and OpenCOBOL build options – it is safest to simply treat the COPY command as case-sensitive in all environments.

### 7.1.9. Using Compiler Configuration Files

OpenCOBOL uses compiler configuration files to define various options that will control the compilation process. These configuration files are specified using the “-conf” compilation switch or are found in the folder defined by the COB\_CONFIG\_PATH environment variable.

The following is a verbatim listing of the “default” configuration file (the one used if you don't specify the “-conf” switch), just to show you the types of settings that may appear:

```
# COBOL compiler configuration                                -*- sh -*-
# Value: any string
name: "OpenCOBOL"

# Value: int
tab-width: 8
text-column: 72

# Value: 'cobo12002', 'mf', 'ibm'
#
assign-clause: mf

# If yes, file names are resolved at run time using environment variables.
# For example, given ASSIGN TO "DATAFILE", the actual file name will be
# 1. the value of environment variable 'DD_DATAFILE' or
# 2. the value of environment variable 'dd_DATAFILE' or
```

```

# 3. the value of environment variable 'DATAFILE' or
# 4. the literal "DATAFILE"
# If no, the value of the assign clause is the file name.
#
# Value: 'yes', 'no'
filename-mapping: yes

# Value: 'yes', 'no'
pretty-display: yes

# Value: 'yes', 'no'
auto-initialize: yes

# Value: 'yes', 'no'
complex-odo: no

# Value: 'yes', 'no'
indirect-redefines: no

# Binary byte size - defines the allocated bytes according to PIC
# Value:      signed  unsigned  bytes
#
# '2-4-8'      1 - 4      -----  -----
#              5 - 9      -----  -----
#              10 - 18     -----  -----
#
# '1-2-4-8'    1 - 2      -----  -----
#              3 - 4      -----  -----
#              5 - 9      -----  -----
#              10 - 18     -----  -----
#
# '1--8'       1 - 2      1 - 2      1
#              3 - 4      3 - 4      2
#              5 - 6      5 - 7      3
#              7 - 9      8 - 9      4
#              10 - 11     10 - 12     5
#              12 - 14     13 - 14     6
#              15 - 16     15 - 16     7
#              17 - 18     17 - 18     8
binary-size: 1-2-4-8

# Value: 'yes', 'no'
binary-truncate: yes

# Value: 'native', 'big-endian'
binary-byteorder: big-endian

# Value: 'yes', 'no'
larger-redefines-ok: no

# Value: 'yes', 'no'
relaxed-syntax-check: no

# Perform type OSVS - If yes, the exit point of any currently executing perform
# is recognized if reached.
# Value: 'yes', 'no'
perform-osvs: no

# If yes, linkage-section items remain allocated
# between invocations.
# Value: 'yes', 'no'
sticky-linkage: no

# If yes, allow non-matching level numbers
# Value: 'yes', 'no'
relax-level-hierarchy: no

# not-reserved:
# Value: word to be taken out of the reserved words list
# (case independent)

# Dialect features
# Value: 'ok', 'archaic', 'obsolete', 'skip', 'ignore', 'unconformable'
author-paragraph:      obsolete
memory-size-clause:    obsolete

```



multiple-file-tape-clause:	obsolete
label-records-clause:	obsolete
value-of-clause:	obsolete
data-records-clause:	obsolete
top-level-occurs-clause:	skip
synchronized-clause:	ok
goto-statement-without-name:	obsolete
stop-literal-statement:	obsolete
debugging-line:	obsolete
padding-character-clause:	obsolete
next-sentence-phrase:	archaic
eject-statement:	skip
entry-statement:	obsolete
move-noninteger-to-alphanumeric:	error
odo-without-to:	ok

## 7.2. Running OpenCOBOL Programs

### 7.2.1. Executing Programs Directly

OpenCOBOL programs compiled with the “-x” option will be generated as directly-executable programs. For example, on a Windows system, the “-x” option will be generated as an “.exe” file.

These native executables are appropriate for execution as non-graphical user interface programs.

On a UNIX system this means the programs may be executed from a command shell such as bash, csh, ksh and so forth. When an OpenCOBOL program runs on a Windows system, it runs within a console window (i.e. “cmd.exe”).

Interactions between the program and the user will take place using the standard input, standard output and standard error streams. Any SCREEN SECTION I/O performed by the program will take place within the command shell “window”.

Direct program execution syntax is as follows:

**[*path*]program [*arguments*]**

For example:

**/usr/local/printaccount ACCT=6625378**

Or...

**C:\Users\Me\Documents\Programs\printaccount.exe ACCT=6625378**

### 7.2.2. Using the “cobcrun” Utility

It is possible to generate executable modules for all OpenCOBOL programs, not just subroutines, by choosing to use the “-m” option to specify the compiler output format even for main programs (as discussed in section [7.1.4](#), this is the recommended output format option for subroutines).

Some may prefer to compile their OpenCOBOL main programs into these dynamically-loadable modules in the interests of using the same general compilation command for all programs without having to think “Is it a main program or a subroutine?”.

Main programs compiled in this manner should be executed as follows:

**[*path*]cobcrun program [*arguments*]**

Do not specify the “.so” or “.dll” extension on the program name. The “*program*” value must exactly match the PROGRAM-ID of the main program (including upper- and lower-case letters).

For an example of the use of cobcrun:

```
cd /usr/local
cobcrun printaccount ACCT=6625378
```

Or...

```
cd C:\Users\Me\Documents\Programs
cobcrun printaccount.exe ACCT=6625378
```

Note how the cobcrun command does not allow a path to be specified with the program name –the directory in which the programs dynamically loadable module exists must either be the current directory or must be defined in the current PATH.

### 7.2.3. Program Arguments

Regardless of the manner in which a program is executed, any arguments specified to the program may be retrieved via either of the following, documented in section [6.4.2](#):

- ACCEPT ... FROM COMMAND-LINE
- ACCEPT ... FROM ARGUMENT-VALUE

### 7.2.4. Important Environment Variables

The following chart documents the various environment variables that can play a role in the execution of OpenCOBOL programs.

Figure 7-5 - Run-Time Environment Variables

Environment Variable	Use
COB_LIBRARY_PATH	At runtime, OpenCOBOL will attempt to locate and load any application dynamically-loadable libraries from the PATH and the directory in which the program executable was found. If these library files could be somewhere else, specify the directory path using this variable.
COB_PRE_LOAD	If set to any non-null value, this variable will cause all dynamically-loadable libraries to be loaded when the program begins execution (rather than searching for and loading the module upon first use).
COB_SCREEN_ESC	If set to any non-blank value, this variable allows the ACCEPT verb to detect the Esc key. See <a href="#">Figure 4-8</a> for additional information.
COB_SCREEN_EXCEPTIONS	Setting this variable to any non-blank value will allow the ACCEPT verb to detect the Esc, PgUp and PgDn keys. See <a href="#">Figure 4-8</a> for additional information.
COB_SORT_MEMORY	The value of this variable (an integer) will be used to define how much memory will be allocated for use in sorting. If the value is 1048576 or greater, that value will be used “as is” as the amount of memory (in bytes) to allocate. If the value is less than 1048576. The default sort memory amount is 128 MB.
COB_SWITCH_n	(n=1 to 8); These environment variables correspond to SWITCH-1 through SWITCH-8. Setting them to “ON” will activate them; any other value turns them off. See section <a href="#">4.1.4</a> for more information.

Environment Variable	Use
COB_SYNC	If set to a value of upper- or lowercase “p”, this variable will force a file commit every time a file is written to (ensuring that data is <u>immediately</u> written to the file rather than retained in memory until a future commit occurs). This will slow-down update access to files, but will provide for better integrity in the event of a program failure.
DB_HOME	If your OpenCOBOL build uses the Berkeley Database (BDB) package, use this environment variable to specify the folder in which the lock management files to be associated with all non-SORT files opened by the program will be stored <sup>38</sup> . Having this variable defined will activate record locking features on the READ (section <a href="#">6.34</a> ), REWRITE (section <a href="#">6.37</a> ) and WRITE (section <a href="#">6.51</a> ) statements <sup>39</sup> .
PATH	The OpenCOBOL “bin” directory should be defined in the PATH.
TMPDIR TMP TEMP (checked in this order)	Set to a directory/folder appropriate to create temporary files in. This will be used by SORT and MERGE to create temporary work files. You may also use this folder for any temporary files your application may require.  Good form dictates that – if your application DOES create temporary working files – it should clean-up after itself. <sup>40</sup>

## 7.3. Built-In Subroutines

### 7.3.1. “Call by Name” Routines

There are a number of built-in subroutines included with OpenCOBOL. Generally, these routines are intended to match those available in Micro Focus COBOL (CBL\_...) or ACUCOBOL (C\$...).

These routines, all executed via their UPPERCASE NAMES, are capable of performing the following functions:

- Changing the current directory
- Copying files
- Creating a directory
- Creating, Opening, Closing, Reading and Writing byte-stream files
- Deleting directories (folders)
- Deleting files

<sup>38</sup> ORGANIZATION INDEXED files will also have their data file allocated in the DB\_HOME folder, if DB\_HOME exists.

<sup>39</sup> Even with DB\_HOME, locking will not work with ORGANIZATION SEQUENTIAL (either type) or ORGANIZAION RELATIVE files with OpenCOBOL builds created for Windows/MinGW. ORGANIZATION INDEXED locks will work with Windows/MinGW and all locks will work for all file organizations with UNIX OpenCOBOL builds.

<sup>40</sup> Take a look at the C\$DELETE and CBL\_DELETE\_FILE built-in subroutines.

- Determining how many arguments were passed to a subroutine
- Getting file information (size and last-modification date/time)
- Getting the length (in bytes) of an argument passed to a subroutine
- Justifying a field left-, right- or center-aligned
- Moving files (a destructive “copy”)
- Putting the program ‘to sleep’, specifying the sleep time in seconds
- Putting the program ‘to sleep’, specifying the sleep time in nanoseconds; CAVEAT: although you’ll express the time in nanoseconds, Windows systems will only be able to sleep at a millisecond granularity
- Submitting a command to the shell environment appropriate for the version of OpenCOBOL you are using for execution

The following table describes the various built-in subroutines. ALL SUBROUTINE ARGUMENTS ARE MANDATORY EXCEPT WHERE EXPLICITLY NOTED TO THE CONTRARY. Any subroutine returning a value to RETURN-CODE could utilize the RETURNING/GIVING clause on the CALL to return the result back to the full-word binary COMP-5 data item of your choice. See section [6.8](#).

### 7.3.1.1. CALL “C\$CHDIR” USING *directory-path*, *result*

This routine makes *directory-path* (an alphanumeric literal or identifier) the current directory.

The return code of the operation is returned both in the *result* argument (any non-edited numeric identifier) as well as in the RETURN-CODE special register. The return code of the operation will be either 0=Success or 128=failure.

The directory change remains in effect until the program terminates (in which the original current directory at the time the program was restarted will be automatically restored) or until another C\$CHDIR is executed.

Also see **CBL\_CHANGE\_DIR** – section [7.3.1.13](#).

### 7.3.1.2. CALL “C\$COPY” USING *src-file-path*, *dest-file-path*, 0

Use this subroutine to copy file *src-file-path* to *dest-file-path* as if it were done via the “CP” (Unix) or “COPY” (Windows) command.

Both file path arguments may be alphanumeric literals or identifiers.

The third argument is required, but is unused.

If the attempt to copy the file fails (for example, it or the destination directory doesn't exist), RETURN-CODE will be set to 128; on successful completion it will be set to 0.

Also see **CBL\_COPY\_FILE** – section [7.3.1.16](#).

### 7.3.1.3. CALL “C\$DELETE” USING *file-path*, 0

This routine deletes the file specified by the *file-path* argument (an alphanumeric literal or identifier) just as if that were done using the “RM” (Unix) or “ERASE” (Windows) command.

The second argument is required, but is unused.

If the attempt to delete the file fails (for example, it doesn't exist), RETURN-CODE will be set to 128; on successful completion it will be set to 0.

Also see **CBL\_DELETE\_FILE** – section [7.3.1.20](#).

### 7.3.1.4. CALL “C\$FILEINFO” USING *file-path*, *file-info*

With this routine you may retrieve the size of the file<sup>41</sup> specified as the *file-path* argument (an alphanumeric literal or identifier) and the date/time that file was last modified. The information is returned to the *file-info* argument, which is defined as the following 16-byte area:

```

01 File-Info.
   05 File-Size-In-Bytes PIC 9(18) COMP.
   05 Mod-YYYYMMDD      PIC 9(8)  COMP. *> Modification Date
   05 Mod-HHMMSS00     PIC 9(8)  COMP. *> Modification Time

```

The last two decimal digits in the modification time will always be 0.

If the subroutine is successful, a value of 0 will be returned in RETURN-CODE. Failure to retrieve the needed statistics on the file will cause a RETURN-CODE value of 35 to be passed back. Supplying less than two arguments will generate a 128 RETURN-CODE value.

Also see **CBL\_CHECK\_FILE\_EXIST** – section [7.3.1.14](#).

### 7.3.1.5. CALL “C\$JUSTIFY” USING *data-item*, “*justification-type*”

Use C\$JUSTIFY to left, right or center-justify an alphabetic, alphanumeric or numeric edited *data-item*. The *justification-type* argument indicates the type of the justification to be performed. The value of that argument will be interpreted as follows:

absent	Treated the same as if it were "R"
Cxxx...	If it begins with a capital "C", the value will be centered
Rxxx...	If it begins with a capital "R", the value will be right-justified, space-filled to the left
Lxxx...	If it begins with a capital "L", the value will be left-justified, space-filled to the right
anything else	Treated as if it were "R"

### 7.3.1.6. CALL “C\$MAKEDIR” USING *dir-path*

With this routine you may create a new directory – the name of which is supplied as the *dir-path* argument (an alphanumeric literal or identifier).

Only the lowest-level directory (last) in the specified path can be created – all others must already exist. This subroutine will NOT behave as a “mkdir -p” (Unix) or “mkdir /p” (Windows).

RETURN-CODE will be set to the return code of the operation; the value will be either 0=Success or 128=failure.

Also see **CBL\_CREATE\_DIR**- section [7.3.1.17](#).

### 7.3.1.7. CALL “C\$NARG” USING *arg-count-result*

C\$NARG returns the number of arguments passed to the subroutine that calls C\$NARG back to the numeric field *arg-count-result*.

When CALLED from a main program, the returned value will always be 0.

Also see the **NUMBER-OF-CALL-PARAMETERS**.register (section [6.1.8](#)).

<sup>41</sup> File size information may not be available in the particular OpenCOBOL build / Operating System combination you are using and may therefore always be returned as zero.

### 7.3.1.8. CALL “C\$PARAMSIZE” USING *argument-number*

This subroutine returns the size (in bytes) of the subroutine argument supplied using the *argument-number* parameter (a numeric literal or data item).

The size is returned in the RETURN-CODE special register.

If the specified argument does not exist, or an invalid argument number is specified, a value of 0 is returned.

### 7.3.1.9. CALL “C\$SLEEP” USING *seconds-to-sleep*

C\$SLEEP puts the program to sleep for the specified number of seconds. The *seconds-to-sleep* argument may be a numeric literal or data item.

Sleep times less than 1 will be interpreted as 0, which immediately returns without any sleep delay.

Also see the **CBL\_OC\_NANOSLEEP** subroutine – section [7.3.1.30](#).

### 7.3.1.10. CALL “C\$TOLOWER” USING *data-item*, BY VALUE *convert-length*

This routine will convert *convert-length* (a numeric literal or data item) leading characters of *data-item* (an alphanumeric identifier) to lower-case.

The *convert-length* argument must be specified **BY VALUE**. It specifies how many (leading) characters in *data-item* will be converted – any characters after that will remain unchanged.

If *convert-length* is negative or zero, no conversion will be performed.

Also see **CBL\_TOLOWER** – section [7.3.1.35](#).

### 7.3.1.11. CALL “C\$TOUPPER” USING *data-item*, BY VALUE *convert-length*

Use the C\$TOUPPER subroutine to change the *convert-length* (a numeric literal or data item) leading characters of *data-item* (an alphanumeric identifier) to upper-case.

The *convert-length* argument must be specified **BY VALUE**. It specifies how many (leading) characters in *data-item* will be converted – any characters after that will remain unchanged.

If *convert-length* is negative or zero, no conversion will be performed.

Also see **CBL\_TOUPPER** – section [7.3.1.36](#).

### 7.3.1.12. CALL “CBL\_AND” USING *item-1*, *item-2*, BY VALUE *byte-length*

This subroutine performs a bit-by-bit logical AND operation between the left-most  $8 \times \textit{byte-length}$  corresponding bits of *item-1* and *item-2*, storing the resulting bit string into *item-2*.

*Item-1* may be an alphanumeric literal or a data item. *Item-2* must be a data item. The length of both *item-1* and *item-2* must be at least  $8 \times \textit{byte-length}$ .

*Byte-length* may be a numeric literal or data item, and must be specified using **BY VALUE**.

The truth table shown to the right documents the “AND” process.

Any bits in *item-2* after the  $8 \times \textit{byte-length}$  point will be unaffected.

A result of zero will be passed back in the RETURN-CODE register.

Arg #1 bit	Arg #2 bit	New Arg #2 bit
0	0	0
0	1	0
1	0	0
1	1	1

### 7.3.1.13. CALL “CBL\_CHANGE\_DIR” USING *directory-path*

This routine makes *directory-path* (an alphanumeric literal or identifier) the current directory.

The directory change remains in effect until the program terminates (in which the original current directory at the time the program was restarted will be automatically restored) or until another CBL\_CHANGE\_DIR (or C\$CHDIR) is executed.

The return code of the operation is returned in the RETURN-CODE special register. The return code of the operation will be either 0=Success or 128=failure.

Also see C\$CHDIR – section [7.3.1.1](#).

### 7.3.1.14. CALL “CBL\_CHECK\_FILE\_EXIST” USING *file-path, file-info*

With this routine you may retrieve the size of the file<sup>42</sup> specified as the *file-path* argument (an alphanumeric literal or identifier) and the date/time that file was last modified. The information is returned to the *file-info* argument, which is defined as the following 16-byte area:

01	Argument-2.			
05	File-Size-In-Bytes	PIC 9(18)	COMP.	
05	Mod-DD	PIC 9(2)	COMP.	*> Modification Time
05	Mod-MO	PIC 9(2)	COMP.	
05	Mod-YYYY	PIC 9(4)	COMP.	*> Modification Date
05	Mod-HH	PIC 9(2)	COMP.	
05	Mod-MM	PIC 9(2)	COMP.	
05	Mod-SS	PIC 9(2)	COMP.	
05	FILLER	PIC 9(2)	COMP.	*> This will always be 00

If the subroutine is successful, a value of 0 will be returned in RETURN-CODE. Failure to retrieve the needed statistics on the file will cause a RETURN-CODE value of 35 to be passed back. Supplying less than two arguments will generate a 128 RETURN-CODE value.

Also see C\$FILEINFO - section [7.3.1.4](#).

### 7.3.1.15. CALL “CBL\_CLOSE\_FILE” USING *file-handle*

The CBL\_CLOSE\_FILE subroutine closes a bytestream file previously opened by either the CBL\_OPEN\_FILE or CBL\_CREATE\_FILE subroutines.

If the file defined by the *file-handle* argument (a PIC X(4) USAGE COMP-X data item) was opened for output, an implicit CBL\_FLUSH\_FILE will be performed before the file is closed.

If the subroutine is successful, a value of 0 will be returned in RETURN-CODE. Failure will cause a RETURN-CODE value of -1 to be passed back.

### 7.3.1.16. CALL “CBL\_COPY\_FILE” USING *src-file-path, dest-file-path*

Use this subroutine to copy file *src-file-path* to *dest-file-path* as if it were done via the “CP” (Unix) or “COPY” (Windows) command.

Both file path arguments may be alphanumeric literals or identifiers.

If the attempt to copy the file fails (for example, it or the destination directory doesn't exist), RETURN-CODE will be set to 128; on successful completion it will be set to 0.

Also see C\$COPY – section [7.3.1.2](#).

<sup>42</sup> File size information may not be available in the particular OpenCOBOL build / Operating System combination you are using and may therefore always be returned as zero.

### 7.3.1.17. CALL “CBL\_CREATE\_DIR” USING *dir-path*

With this routine you may create a new directory – the name of which is supplied as the *dir-path* argument (an alphanumeric literal or identifier).

Only the lowest-level directory (last) in the specified path can be created – all others must already exist. This subroutine will NOT behave as a “mkdir -p” (Unix) or “mkdir /p” (Windows).

RETURN-CODE will be set to the return code of the operation; the value will be either 0=Success or 128=failure.

Also see **C\$MAKEDIR** – section [7.3.1.6](#).

### 7.3.1.18. CALL “CBL\_CREATE\_FILE” USING *file-path*, 2, 0, 0, *file-handle*

The CBL\_CREATE\_FILE subroutine creates the new file specified using the *file-path* argument and opens it for output as a byte-stream file usable by **CBL\_WRITE\_FILE**.

Arguments 2, 3 and 4 should be coded as the constant values shown.<sup>43</sup>

A *file handle* (PIC X(4) USAGE COMP-X) will be returned, for any subsequent **CBL\_WRITE\_FILE** or **CBL\_CLOSE\_FILE** calls.

The success or failure of the subroutine will be reported back in the RETURN-CODE register, with a RETURN-CODE value of -1 indicating an invalid argument and a value of 0 indicating success.

Also see **CBL\_OPEN\_FILE** – section [7.3.1.31](#).

### 7.3.1.19. CALL “CBL\_DELETE\_DIR” USING *dir-path*

Delete an empty directory via CBL\_DELETE\_DIR.

The only argument – *dir-path* (an alphanumeric literal or identifier) – is the name of the directory to be deleted.

Only the lowest-level directory (last) in the specified path will be deleted, and that directory must be empty to be deleted.

RETURN-CODE will be set to the return code of the operation; the value will be either 0=Success or 128=failure.

### 7.3.1.20. CALL “CBL\_DELETE\_FILE” USING *file-path*

This routine deletes the file specified by the file-path argument (an alphanumeric literal or identifier) just as if that were done using the “RM” (Unix) or “ERASE” (Windows) command.

If the attempt to delete the file fails (for example, it doesn't exist), RETURN-CODE will be set to 128; on successful completion it will be set to 0.

Also see **C\$DELETE** – section [7.3.1.3](#).

### 7.3.1.21. CALL “CBL\_ERROR\_PROC” USING *function*, *program-pointer*

This routine registers a general error-handling routine.

The *function* argument must be a numeric literal or a 32-bit binary COMP-5 data item (USAGE BINARY-LONG, for example) with a value of 0 or 1. A value of 0 means that you will be registering (“installing”) an error procedure while a value of 1 indicates you’re deregistering (“uninstalling”) a previously-installed error procedure.

The *program-pointer* must be a USAGE PROGRAM-POINTER data item containing the address of your error procedure. See section [6.40.2](#) for instructions on how to populate such a data item.

---

<sup>43</sup> **CBL\_CREATE\_FILE** is actually a special-case of the **CBL\_OPEN\_FILE** routine - see that routine for a description of the meanings of arguments 2, 3 and 4.



A success (0) or failure (non-0) result will be passed back in the RETURN-CODE register.

A custom error-handler routine, if any, will trigger when a runtime error condition is encountered. The code within the handler will be executed and – once the handler issues an EXIT PROGRAM or a GOBACK - the system-standard error handling routine will be executed.

Only one user-defined error procedure may be in effect at any time.

An error procedure may be defined by a main program or a subprogram, but regardless of from where it was registered, it applies to the overall program compilation unit and will trigger when a runtime error occurs anywhere in the executable program. If the error procedure was defined by a subprogram, that program must be loaded at the time the error procedure is executed.

An error procedure should terminate using EXIT PROGRAM or GOBACK.

The following is a sample OpenCOBOL program that registers an error procedure. The output of that program is shown as well - as you can see, the error handler's messages appear followed by the standard OpenCOBOL message.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. demoerrproc.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
78 Err-Proc-Install          VALUE 0.
01 Current-Date             PIC X(8).
01 Current-Time             PIC X(8).
01 Err-Proc-Address         USAGE PROCEDURE-POINTER.
01 Formatted-Date           PIC X(4)/X(2)/X(2).
01 Formatted-Time           PIC X(2)/X(2)/X(2).
PROCEDURE DIVISION.
000-Register-Err-Proc.
    SET Err-Proc-Address TO ENTRY "999-Err"
    CALL "CBL_ERROR_PROC"
        USING Err-Proc-Install, Err-Proc-Address
    END-CALL
    IF RETURN-CODE NOT = 0
        DISPLAY 'Error: Could not' &
            'register Error Procedure'
    END-IF
.
099-Now-Test-Err-Proc.
    CALL "Tilt" END-CALL
    GOBACK
.
999-Err-Proc.
    ENTRY "999-Err"
    DISPLAY
        '** A Runtime Error Has Occurred **'
    END-DISPLAY
    ACCEPT
        Current-Date FROM DATE YYYYMMDD
    END-ACCEPT
    ACCEPT
        Current-Time FROM TIME
    END-ACCEPT
    MOVE Current-Date TO Formatted-Date
    MOVE Current-Time TO Formatted-Time
    INSPECT Formatted-Time REPLACING ALL '/' BY ':'
    DISPLAY
        '*** ' Formatted-Date ' ' Formatted-Time ' ***'
    END-DISPLAY
    GOBACK
.
```

Program output...

```
** A Runtime Error Has Occurred **
*** 2009/08/28 10:35:10 ***
libcob: cannot find module 'Tilt'
```

### 7.3.1.22. CALL “CBL\_EXIT\_PROC” USING *function, program-pointer*

This routine registers a general exit-handling routine.

The *function* argument must be a numeric literal or a 32-bit binary COMP-5 data item (USAGE BINARY-LONG, for example) with a value of 0 or 1. A value of 0 means that you will be registering (“installing”) an exit procedure while a value of 1 indicates you’re deregistering (“uninstalling”) a previously-installed exit procedure.

The *program-pointer* must be a USAGE PROGRAM-POINTER data item containing the address of your exit procedure. See section [6.40.2](#) for instructions on how to populate such a data item.

A success (0) or failure (non-0) result will be passed back in the RETURN-CODE register.

An exit procedure will trigger when a “STOP RUN” or its equivalent (i.e. “GOBACK” executed in a main program) is executed. The exit procedure code will be executed and – once it issues an EXIT PROGRAM or a GOBACK, the system-standard program termination routine will be executed.

Only one user-defined exit procedure may be in effect at any time.

An exit procedure may be defined by a main program or a subprogram, but regardless of from where it was registered, it applies to the overall program compilation unit and will trigger when a STOP RUN is executed anywhere in the executable program. If the exit procedure was defined by a subprogram, that program must be loaded at the time the exit procedure is executed.

An exit procedure should terminate using EXIT PROGRAM or a GOBACK.

The following is a sample OpenCOBOL program that registers an exit procedure. The output of that program is shown as well.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. demoexitproc.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
78 Exit-Proc-Install          VALUE 0.
01 Current-Date              PIC X(8).
01 Current-Time              PIC X(8).
01 Exit-Proc-Address         USAGE PROCEDURE-POINTER.
01 Formatted-Date            PIC XXXX/XX/XX.
01 Formatted-Time            PIC XX/XX/XX.
PROCEDURE DIVISION.
000-Register-Exit-Proc.
  SET Exit-Proc-Address TO ENTRY "999-Exit"
  CALL "CBL_EXIT_PROC"
    USING Exit-Proc-Install, Exit-Proc-Address
  END-CALL
  IF RETURN-CODE NOT = 0
    DISPLAY 'Error: Could not register Exit Procedure'
  END-IF
.
099-Now-Test-Exit-Proc.
  DISPLAY
    'Executing a STOP RUN...'
  END-DISPLAY
  GOBACK
.
999-Exit-Proc.
  ENTRY "999-Exit"
  DISPLAY
    '*** STOP RUN has been executed ***'
  END-DISPLAY
  ACCEPT
    Current-Date FROM DATE YYYYMMDD
  END-ACCEPT
  ACCEPT
    Current-Time FROM TIME
  END-ACCEPT
  MOVE Current-Date TO Formatted-Date
  MOVE Current-Time TO Formatted-Time
  INSPECT Formatted-Time REPLACING ALL '/' BY ':'
  DISPLAY
```

Program output...

```
Executing a STOP RUN...
*** STOP RUN has been executed ***
*** 2009/08/28 10:01:29 ***
```

```

      '***' ' Formatted-Date ' ' Formatted-Time ' '***'
END-DISPLAY
GOBACK
.
```

### 7.3.1.23. CALL “CBL\_EQ” USING *item-1*, *item-2*, BY VALUE *byte-length*

This subroutine performs a bit-by-bit test for equality between the left-most  $8 * \textit{byte-length}$  corresponding bits of *item-1* and *item-2*, storing the resulting bit string into *item-2*.

*Item-1* may be an alphanumeric literal or a data item. *Item-2* must be a data item. The length of both *item-1* and *item-2* must be at least  $8 * \textit{byte-length}$ .

*Byte-length* may be a numeric literal or data item, and must be specified using **BY VALUE**.

The truth table shown to the right documents the “EQ” process.

Any bits in *item-2* after the  $8 * \textit{byte-length}$  point will be unaffected.

A result of zero will be passed back in the RETURN-CODE register.

Arg #1 bit	Arg #2 bit	New Arg #2 bit
0	0	1
0	1	0
1	0	0
1	1	1

### 7.3.1.24. CALL “CBL\_FLUSH\_FILE” USING *file-handle*

In Micro Focus COBOL, CALLING this subroutine flushes any as-yet unwritten memory buffers for the (output) file whose *file-handle* is specified as the argument to disk.

This routine is non-functional in OpenCOBOL. It exists only to provide compatibility for applications that may have been developed for Micro Focus COBOL.

### 7.3.1.25. CALL “CBL\_GET\_CURRENT\_DIR” USING BY VALUE 0, BY VALUE *length*, BY REFERENCE *buffer*

This retrieves the fully-qualified pathname of the current directory, saving up to *length* characters of that name into the specified *buffer*.

The first argument is unused, but must be specified. It must be specified **BY VALUE**.

The *length* argument must be specified **BY VALUE**.

The *buffer* argument must be specified **BY REFERENCE**.

The value specified for the *length* argument (a numeric literal or data item) should not exceed the actual length of the *buffer* argument.

If the value specified for the *length* argument is LESS THAN the actual length of the *buffer* argument, the current directory path will be left-justified and space filled within the first *length* bytes of *buffer* – any bytes in *buffer* after that point will be unchanged.

If the routine is successful, a value of 0 will be returned to the RETURN-CODE register. If the routine failed because of a problem with an argument (such as a negative or 0 *length*), a RETURN-CODE value of 128 will result. Finally, if the 1<sup>st</sup> argument value is anything but zero, the routine will fail with a 129 RETURN-CODE.

### 7.3.1.26. CALL “CBL\_IMP” USING *item-1*, *item-2*, BY VALUE *byte-length*

This subroutine performs a bit-by-bit “implies” test between the left-most  $8 * \textit{byte-length}$  corresponding bits of *item-1* and *item-2*, storing the resulting bit string into *item-2*.

*Item-1* may be an alphanumeric literal or a data item. *Item-2* must be a data item. The length of both *item-1* and *item-2* must be at least  $8 * \text{byte-length}$ .

*Byte-length* may be a numeric literal or data item, and must be specified using **BY VALUE**.

The truth table shown to the right documents the “IMP” process.

Any bits in *item-2* after the  $8 * \text{byte-length}$  point will be unaffected.

A result of zero will be passed back in the RETURN-CODE register.

Arg #1 bit	Arg #2 bit	New Arg #2 bit
0	0	1
0	1	1
1	0	0
1	1	1

### 7.3.1.27. CALL “CBL\_NIMP” USING *item-1*, *item-2*, BY VALUE *byte-length*

This subroutine performs the negation of a bit-by-bit “implies” test between the left-most  $8 * \text{byte-length}$  corresponding bits of *item-1* and *item-2*, storing the resulting bit string into *item-2*.

*Item-1* may be an alphanumeric literal or a data item. *Item-2* must be a data item. The length of both *item-1* and *item-2* must be at least  $8 * \text{byte-length}$ .

*Byte-length* may be a numeric literal or data item, and must be specified using **BY VALUE**.

The truth table shown to the right documents the “NIMP” process.

Any bits in *item-2* after the  $8 * \text{byte-length}$  point will be unaffected.

A result of zero will be passed back in the RETURN-CODE register.

Arg #1 bit	Arg #2 bit	New Arg #2 bit
0	0	0
0	1	0
1	0	1
1	1	0

### 7.3.1.28. CALL “CBL\_NOR” USING *item-1*, *item-2*, BY VALUE *byte-length*

This subroutine performs the negation of a bit-by-bit “OR” test between the left-most  $8 * \text{byte-length}$  corresponding bits of *item-1* and *item-2*, storing the resulting bit string into *item-2*.

*Item-1* may be an alphanumeric literal or a data item. *Item-2* must be a data item. The length of both *item-1* and *item-2* must be at least  $8 * \text{byte-length}$ .

*Byte-length* may be a numeric literal or data item, and must be specified using **BY VALUE**.

The truth table shown to the right documents the “NOR” process.

Any bits in *item-2* after the  $8 * \text{byte-length}$  point will be unaffected.

A result of zero will be passed back in the RETURN-CODE register.

Arg #1 bit	Arg #2 bit	New Arg #2 bit
0	0	1
0	1	0
1	0	0
1	1	0

### 7.3.1.29. CALL “CBL\_NOT” USING *item-1*, BY VALUE *byte-length*

This subroutine “flips” the left-most  $8 \times \textit{byte-length}$  bits of *item-2*, storing the resulting bit string into *item-2*.

*Item-2* must be a data item. The length of *item-2* must be at least  $8 \times \textit{byte-length}$ .

*Byte-length* may be a numeric literal or data item, and must be specified using **BY VALUE**.

The truth table shown to the right documents the “NOT” process.

Any bits in *item-2* after the  $8 \times \textit{byte-length}$  point will be unaffected.

A result of zero will be passed back in the RETURN-CODE register.

Old Arg #2 bit	New Arg #2 bit
0	1
1	0

### 7.3.1.30. CALL “CBL\_OC\_NANOSLEEP” USING *nanoseconds-to-sleep*

CB\_OC\_NANOSLEEP puts the program to sleep for the specified number of nanoseconds.

The *nanoseconds-to-sleep* argument is a numeric literal or data item.

There are one BILLION nanoseconds in a second, so if you wanted to put the program to sleep for 1/4 second you'd use a *nanoseconds-to-sleep* value of 250000000.

Also see **C\$SLEEP** – section [7.3.1.9](#).

### 7.3.1.31. CALL “CBL\_OPEN\_FILE” *file-path*, *access-mode*, 0, 0, *handle*

.This routine opens an existing file for use as a byte-stream file usable by **CBL\_WRITE\_FILE** or **CBL\_READ\_FILE**.

The *file-path* argument is an alphanumeric literal or data-item.

The *access-mode* argument is a numeric literal or data item with a PIC X USAGE COMP-X (or USAGE BINARY-CHAR) definition; it specifies how you wish to use the file, as follows:

1 = input (read-only)

2 = output (write-only)

3 = input and/or output

The third and fourth arguments would specify a locking mode and device specification, respectively, but they're not implemented in OpenCOBOL (currently, at least) – just specify each as 0.

The final argument – *handle* - is a PIC X(4) USAGE COMP-X item that will receive the handle to the file. That handle is used on all other byte-stream functions to reference this specific file.

A RETURN-CODE value of -1 indicates an invalid argument, while a value of 0 indicates success. A value of 35 means the file does not exist.

Also see **CBL\_CREATE\_FILE** – section [7.3.1.18](#).

### 7.3.1.32. CALL “CBL\_OR” USING *item-1*, *item-2*, BY VALUE *byte-length*

This subroutine performs a bit-by-bit “OR” test between the left-most  $8 \times \textit{byte-length}$  corresponding bits of *item-1* and *item-2*, storing the resulting bit string into *item-2*.

*Item-1* may be an alphanumeric literal or a data item. *Item-2* must be a data item. The length of both *item-1* and *item-2* must be at least  $8 \times \textit{byte-length}$ .

*Byte-length* may be a numeric literal or data item, and must be specified using **BY VALUE**.

The truth table shown below documents the “OR” process.

Any bits in *item-2* after the  $8 * \text{byte-length}$  point will be unaffected.

A result of zero will be passed back in the RETURN-CODE register.

Arg #1 bit	Arg #2 bit	New Arg #2 bit
0	0	0
0	1	1
1	0	1
1	1	1

### 7.3.1.33. CALL “CBL\_READ\_FILE” USING *handle*, *offset*, *nbytes*, *flag*, *buffer*

This routine reads *nbytes* of data starting at byte number *offset* from the byte-stream file defined by *handle* into the specified *buffer*.

The *handle* argument (PIC X(4) USAGE COMP-X) must have been populated by a prior call to CBL\_OPEN\_FILE.

The *offset* argument (PIC X(8) USAGE COMP-X) defines the location in the file of the first byte to be read. The first byte of a file is byte offset 0.

The *nbytes* argument (PIC X(4) USAGE COMP-X) specifies how many bytes (maximum) will be read.

If the *flags* argument is specified as 128, the size of the file (in bytes) will be returned into the file offset argument (argument 2) upon completion.<sup>44</sup> The only other valid value for *flags* is 0. This argument may be specified either as a numeric literal or as a PIC X USAGE COMP-X data item.

Upon completion, RETURN-CODE will be set to 0 if the read was successful or to 10 if an “end-of-file” condition occurred. If RETURN-CODE has a value of -1, a problem was identified with the subroutine arguments.

### 7.3.1.34. CALL “CBL\_RENAME\_FILE” USING *old-file-path*, *new-file-path*

You may use this subroutine to rename a file.

The file specified by *old-file-path* will be “renamed” to the name specified as *new-file-path*. Each argument may be an alphanumeric literal or data item.

Despite what the name of this routine might make you believe, this routine is more than just a simple “rename” – it will actually move the file supplied as the 1<sup>st</sup> argument to the file specified as the 2<sup>nd</sup> argument. Think of it as a two-step sequence, first copying the *old-file-path* to the *new-file-path* and then a second step where the *old-file-path* is deleted.

If the attempt to move the file fails (for example, it doesn't exist), RETURN-CODE will be set to 128; on successful completion it will be set to 0.

### 7.3.1.35. CALL “CBL\_TOLOWER” USING *data-item*, BY VALUE *convert-length*

This routine will convert *convert-length* (a numeric literal or data item) leading characters of *data-item* (an alphanumeric identifier) to lower-case.

The *convert-length* argument must be specified **BY VALUE**. It specifies how many (leading) characters in *data-item* will be converted – any characters after that will remain unchanged.

If *convert-length* is negative or zero, no conversion will be performed.

Also see **C\$TOLOWER** – section [7.3.1.10](#).

<sup>44</sup> Not all operating system/OpenCOBOL environments may be able to retrieve file sizes – in such cases, a value of zero will be returned.

### 7.3.1.36. CALL “CBL\_TOUPPER” USING *data-item*, BY VALUE *convert-length*

Use C\$TOUPPER to change the *convert-length* (a numeric literal or data item) leading characters of *data-item* (an alphanumeric identifier) to upper-case.

The *convert-length* argument must be specified **BY VALUE**. It specifies how many (leading) characters in *data-item* will be converted – any characters after that will remain unchanged.

If *convert-length* is negative or zero, no conversion will be performed.

Also see C\$TOUPPER – section [7.3.1.11](#).

### 7.3.1.37. CALL “CBL\_WRITE\_FILE” USING *handle*, *offset*, *nbytes*, 0, *buffer*

This routine writes *nbytes* of data from the specified *buffer* to the byte-stream file defined by *handle* starting at byte number *offset*.

The *handle* argument (PIC X(4) USAGE COMP-X) must have been populated by a prior call to CBL\_OPEN\_FILE.

The *offset* argument (PIC X(8) USAGE COMP-X) defines the location in the file of the first byte to be written to. The first byte of a file is byte offset 0.

The *nbytes* argument (PIC X(4) USAGE COMP-X) specifies how many bytes (maximum) will be written.

The only allowable value or the *flags* argument is 0. This argument may be specified either as a numeric literal or as a PIC X USAGE COMP-X data item.

Upon completion, RETURN-CODE will be set to 0 if the write was successful or to 30 if an I/O error condition occurred. If RETURN-CODE has a value of -1, a problem was identified with the subroutine arguments.

### 7.3.1.38. CALL “CBL\_XOR” USING *item-1*, *item-2*, BY VALUE *byte-length*

This subroutine performs a bit-by-bit exclusive “OR” test between the left-most 8\**byte-length* corresponding bits of *item-1* and *item-2*, storing the resulting bit string into *item-2*.

*Item-1* may be an alphanumeric literal or a data item. *Item-2* must be a data item. The length of both *item-1* and *item-2* must be at least 8\**byte-length*.

*Byte-length* may be a numeric literal or data item, and must be specified using **BY VALUE**.

The truth table shown to the right documents the “XOR” process.

Any bits in *item-2* after the 8\**byte-length* point will be unaffected.

A result of zero will be passed back in the RETURN-CODE register.

Arg #1 bit	Arg #2 bit	New Arg #2 bit
0	0	0
0	1	1
1	0	1
1	1	0

### 7.3.1.39. CALL “SYSTEM” USING *command*

This subroutine submits the specified *command* (an alphanumeric literal or data item) to a command shell.

A shell will be opened subordinate to the OpenCOBOL program issuing the CALL to SYSTEM.

Output from the command (if any) will appear in the command window in which the OpenCOBOL program was executed.

On a Unix system, the shell environment will be established using the default shell program. This is also true when using an OpenCOBOL build created with and for the Cygwin Unix emulator.

With native Windows Windows/MinGW builds, the shell environment will be the Windows console window command processor (usually “cmd.exe”) appropriate for the version of Windows you’re using.

To trap output from the executed command and process it within the OpenCOBOL program, use a pipe (>) to send the command output to a temporary file which you then READ from within the program once control returns.

### 7.3.2. “Call by Number” Subroutines

Early versions of Micro Focus COBOL allowed programmers to access various runtime library routines by using a single two-digit hexadecimal number as the entry=point name. These were known as call-by-number routines. Over time, Micro Focus COBOL evolved, replacing most of the call-by-number routines with ones accessible using a more conventional call-by-name technique.

Most of the call-by-number routines have evolved into even more powerful call-by-name routines, many of which are supported by OpenCOBOL and were already presented in section 7.3

Three of the original call-by-number routines never evolved call-by-name equivalents; OpenCOBOL supports these routines.

#### 7.3.2.1. CALL X”91” USING *return-code*, *function-code*, *binary-variable-arg*

The original Micro Focus version of this routine is capable of providing a wide variety of functions – OpenCOBOL supports just three of those functions:

- Turning runtime switches (SWITCH-1, ... , SWITCH-8) on
- Turning runtime switches (SWITCH-1, ... , SWITCH-8) off
- Retrieving the number of arguments passed to a subroutine<sup>45</sup>

The *return-code* argument must be a binary numeric data item (USAGE BINARY-CHAR is recommended). It will receive a value of 0 if the operation was successful, 1 otherwise.

The *function code* argument must be either a numeric literal or a binary numeric data item (USAGE BINARY-CHAR is recommended).

The third argument – *variable-arg* – is defined differently depending upon the function-code value, as follows:

Value of <i>function-code</i>	Action To Be Performed	Definition and usage of <i>variable-arg</i>
11	Sets and/or clears <u>all eight</u> of the COBOL switches (SWITCH-1 through SWITCH-8) that are available for definition within SPECIAL-NAMES (see section 4.1.4) <sup>46</sup>	<i>Variable-arg</i> should be an OCCURS 8 TIMES array of USAGE BINARY-CHAR.  Each occurrence that is set to a value of zero prior to the CALL will cause the corresponding switch to be cleared. Each occurrence set to 1 prior to the CALL will cause the corresponding switch to be set.  Values other than 0 or 1 will be ignored.
12	Reads <u>all eight</u> of the COBOL switches (SWITCH-1 through SWITCH-8) that are available for definition within SPECIAL-NAMES (see section 4.1.4)	This argument should be an OCCURS 8 TIMES array of USAGE BINARY-CHAR.  Each of the 1 <sup>st</sup> eight occurrences of the array will be set to either 0 or 1 – 1 if the corresponding switch is set, 0 otherwise.
16	Retrieves the number of arguments passed to the program executing the CALL X”91”	This argument should be a binary numeric data item (USAGE BINARY-CHAR is recommended).  The number of arguments passed to the subroutine executing the CALL X”91” will be stored here.

<sup>45</sup> OpenCOBOL actually has two other ways to accomplish this task – the C\$NARG subroutine (section 7.3.1.7) and the NUMBER-OF-CALL-PARAMETERS special register (section 6.1.8); I recommend you use one of these methods instead of the X”91” routine when coding new programs

<sup>46</sup> If you only wish to set and/or clear some of the switches, it is recommended that you first use function 12 to read the current values of the switches and then change the *variable-arg* occurrences for the switch(es) you wish to change before using function 11 to actually make the changes.



### 7.3.2.2. CALL X"F4" USING *byte*, *table*

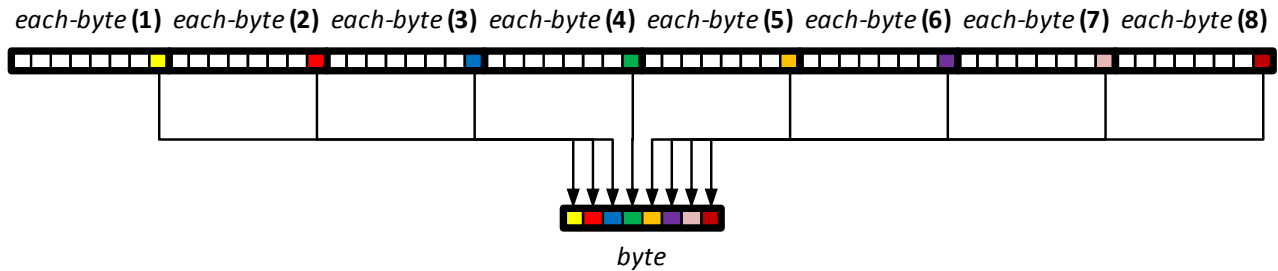
Routine X"F4" packs an 8-byte area containing 8 1-byte binary values of 0 or 1 into the corresponding bit positions of a 1-byte data item.

The *byte* data item need be only a single byte in size. If it is longer, the excess will be unaffected by this subroutine.

*Table* must be a data item at least 8 bytes long. If it is longer, the excess will be ignored by this subroutine. Typically, *table* is defined similarly to the following:

```
01 table.
05 each-byte OCCURS 8 TIMES USAGE BINARY-CHAR.
```

The following diagram illustrates how this subroutine works.



The colored squares represent the bits in the 1<sup>st</sup> 8 bytes of *array* that will be packed into *byte*. The white squares represent the bits in each *each-byte* that will be ignored.

### 7.3.2.3. CALL X"F5" USING *byte*, *table*

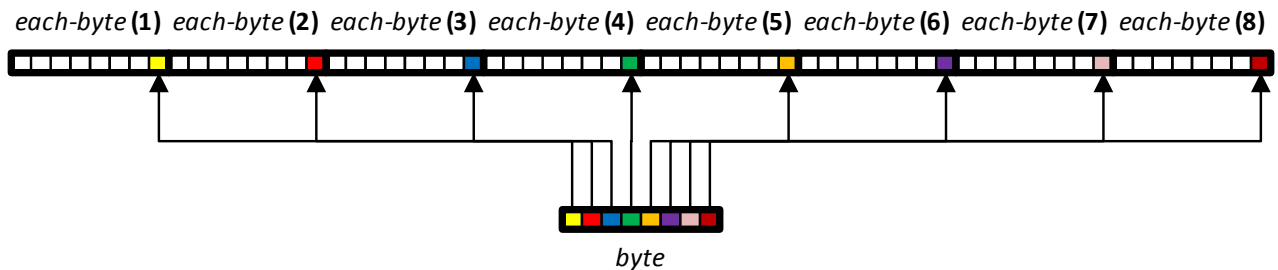
This routine unpacks each bit of a byte into an 8-byte area so they may be individually accessed and manipulated.

The *byte* data item need be only a single byte in size. If it is longer, the excess will be ignored by this subroutine.

*Table* must be a data item at least 8 bytes long. If it is longer, the excess will be unaffected by this subroutine. Typically, *table* is defined similarly to the following:

```
01 table.
05 each-byte OCCURS 8 TIMES USAGE BINARY-CHAR.
```

The following diagram illustrates how this subroutine works.



The colored squares represent each of the 8 bits in *byte*. The diagram shows how those bits will be "unpacked" into the rightmost bit of each of the 1<sup>st</sup> 8 consecutive bytes of *array*. The white squares represent the remaining bits in each of the 1<sup>st</sup> 8 *each-byte* occurrences – all of which will be set to 0.

### 7.3.2.4. Binary Truncation

By default, the OpenCOBOL compiler will truncate binary data items to the precision indicated by their PICTURE clause. For example, the following data item will have 2 bytes of storage allocated for it:

```
01 Comp-5-Item PIC 9(3) COMP-5.
```

Because of truncation, even though this field has enough bits allocated (16) to store values from 0 to 65535, it will be limited to values of 0 to 999 because of its PICTURE.

Or is it?

Take a look at the small demo program shown here. This program will perform three different types of operations against a binary field, displaying the results of each.

Here are the results when the program is compiled (with truncation in-effect by default) and executed:

```
Bin-Item-1=760 Disp-Item-1=032760
Bin-Item-1=765 Disp-Item-1=032765
Bin-Item-1=767 Disp-Item-1=032767
```

You can see that truncation affected the DISPLAY statements but appears to have had no impact whatsoever on the MOVE and ADD statements. This is the hidden secret about truncation in OpenCOBOL: it doesn't really truncate the internally-stored values – it just truncates the DISPLAY of them!

If that same program is recompiled without truncation (by adding the “-fnotrunc” switch to the ‘cobc’ command), the results are as follows:

```
Bin-Item-1=32760 Disp-Item-1=032760
Bin-Item-1=32765 Disp-Item-1=032765
Bin-Item-1=32767 Disp-Item-1=032767
```

against all types of numeric data items (even USAGE DISPLAY) are slowed down.

Before continuing, it's worth making the point that we're NOT talking about astronomical performance degradations here. Today's computers are FAST, and a user sitting at the keyboard, running an OpenCOBOL program is unlikely to notice. BUT ... if you have an OpenCOBOL program that has to process large amounts of data, performing some significant “number crunching” against that data as it goes, the impact of truncation could become noticeable.

Figure 7-6 - A Binary Truncation Demo Program

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DEMOTRUNC.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 Bin-Item-1 PIC 9(3)
COMP-5
VALUE 32760.

01 Disp-Item-1 PIC 9(6).

PROCEDURE DIVISION.
000-Main.
MOVE Bin-Item-1 TO Disp-Item-1
DISPLAY
'Bin-Item-1=' Bin-Item-1
'Disp-Item-1=' Disp-Item-1
END-DISPLAY
ADD 5 TO Bin-Item-1
MOVE Bin-Item-1 TO Disp-Item-1
DISPLAY
'Bin-Item-1=' Bin-Item-1
'Disp-Item-1=' Disp-Item-1
END-DISPLAY
MOVE 32767 TO Bin-Item-1
MOVE Bin-Item-1 TO Disp-Item-1
DISPLAY
'Bin-Item-1=' Bin-Item-1
'Disp-Item-1=' Disp-Item-1
END-DISPLAY
STOP RUN.
```

If this was all there was to the binary truncation issue it wouldn't be worth a section in this document. The fact is, however, that binary truncation has a significant effect on the performance of OpenCOBOL programs. When binary truncation is in effect, arithmetic operations performed

The demo program shown in [Figure 7-7](#) compares the performance of performing arithmetic operations (in a totally non-scientific, non-rigorous way) against USAGE DISPLAY, COMP, COMP-5 and BINARY-xxx<sup>47</sup> numeric data. It was actually my intent when I first wrote the program to merely demonstrate the relative performance differences between the first three types of numeric data storage, and it certainly met that objective.

Imagine my surprise, however, when I discovered that the use of “**-fnotrunc**” also made a significant difference!

Here’s what the program does:

- There are four numeric data items in the program – one USAGE DISPLAY, one USAGE COMP, one USAGE COMP-5 and one USAGE BINARY-LONG. Since the program was run on a computer with an Intel-architecture processor (actually it’s an AMD, but results are identical with Intel) I wanted to see just how much more efficient COMP-5 was over COMP.
- Each data item will have 7 added to it ten million times. You’ll see why shortly.
- The time (to one-one-hundredth of a second) will be retrieved before and after each test and the difference between the two will be DISPLAYed. This is why the computations were done so many times – it was to make sure the timing was “measurable” with only a 1/100 second “stopwatch”.

OpenCOBOL is retrieving wall-clock time, not actual CPU-used time, so other activities taking place on the computer had to be kept to a minimum while the tests were running. I also ran the tests multiple times, just to make sure I had consistent results (I did). Like I mentioned earlier – this is not a rigorous, scientific benchmark of numeric performance; it’s just a quick-and-dirty comparison.

[Figure 7-7](#) shows the program and the test results received when executing both with and without the “**-fnotrunc**” switch.

Here are the conclusions I drew from running these tests many times (30). The timings shown are average times from all tests:

With truncation ON:

- USAGE COMP has a significant performance advantage over USAGE DISPLAY
- USAGE COMP-5 has an even greater performance advantage over USAGE COMP, than COMP did over DISPLAY
- USAGE BINARY-LONG (and presumably the other BINARY-xxx USAGES as well) perform identically (within the measurement tolerances of the test) with COMP-5; this should be no surprise since COMP-5 and BINARY-xxx both allocate data the same way

With truncation OFF:

- There was a huge drop in both USAGE DISPLAY and USAGE COMP timings.
- The relative performance advantage of USAGE COMP over USAGE DISPLAY is even larger with truncation off than it was with it on.
- USAGE COMP-5 and USAGE BINARY-xxx appear to be virtually unaffected by the truncation on/off status, although there was a .01 second increase in average execution time of those tests without truncation over those with truncation. Given the number of times I ran the tests, it’s obvious that something makes COMP-5/BINARY-xxx run slower without truncation than with it; that difference, however, is so miniscule that I discount it as being statistically irrelevant<sup>48</sup>.

My final observation is that I see absolutely no reason whatsoever why the “**-fnotrunc**” option shouldn’t be used on all OpenCOBOL compilations.

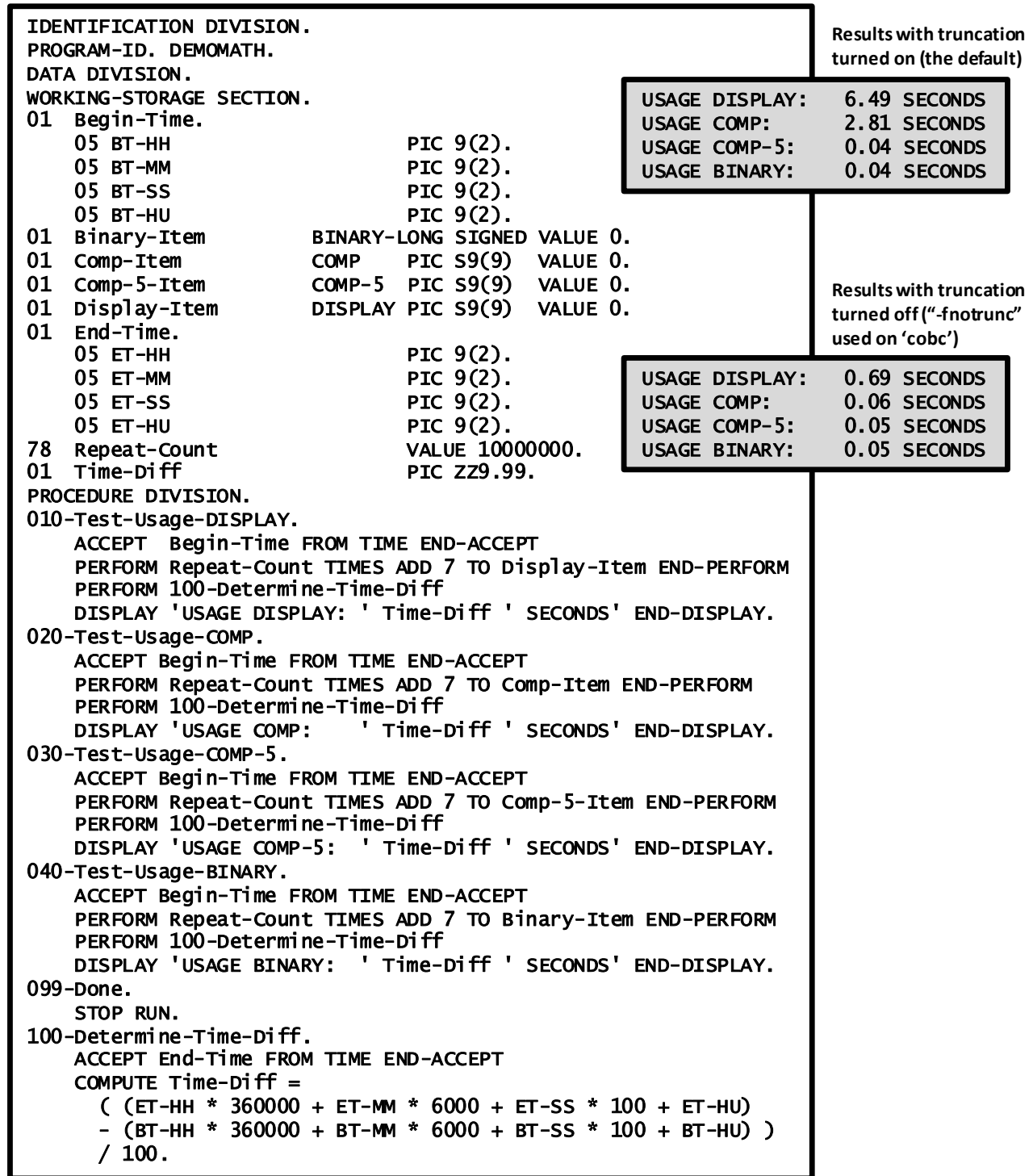
If you want to squeeze every last bit of performance out of your OpenCOBOL programs, don’t forget to investigate the various “**-O**” (optimization) switches. Actually run programs using various optimization switches (or not) and compare

<sup>47</sup> USAGE BINARY-xxx is supposed to store numeric data identically to USAGE COMP-5, but I felt it couldn’t hurt to check.

<sup>48</sup> Remember – that’s a .01 second difference over TEN MILLION iterations!

execution times, don't just compare the generated C code because sometimes the differences can't be "seen" at the C source-code level.

Figure 7-7 - A Non-Scientific Comparison of Numeric Data Item USAGE Performance





## 8. Sample Programs

### 8.1. FileStat-Msgs.cpy – File Status Values

This copybook contains an EVALUATE statement to translate the two-digit file status codes that may be generated by file I/O statements.

The copybook assumes that the file status data item name is “STATUS” and the error message data item is named “MSG”. By using the COPY statement’s REPLACING clause, however, you may use the data names you wish, as follows:

```
COPY FileStat-Msgs
  REPLACING STATUS BY Input-File-Status
           MSG     BY Error-Message.
```

Here’s the FileStat-Msgs.cpy copybook:

```
EVALUATE STATUS
  WHEN 00 MOVE 'SUCCESS' TO MSG
  WHEN 02 MOVE 'SUCCESS DUPLICATE' TO MSG
  WHEN 04 MOVE 'SUCCESS INCOMPLETE' TO MSG
  WHEN 05 MOVE 'SUCCESS OPTIONAL' TO MSG
  WHEN 07 MOVE 'SUCCESS NO UNIT' TO MSG
  WHEN 10 MOVE 'END OF FILE' TO MSG
  WHEN 14 MOVE 'OUT OF KEY RANGE' TO MSG
  WHEN 21 MOVE 'KEY INVALID' TO MSG
  WHEN 22 MOVE 'KEY EXISTS' TO MSG
  WHEN 23 MOVE 'KEY NOT EXISTS' TO MSG
  WHEN 30 MOVE 'PERMANENT ERROR' TO MSG
  WHEN 31 MOVE 'INCONSISTENT FILENAME' TO MSG
  WHEN 34 MOVE 'BOUNDARY VIOLATION' TO MSG
  WHEN 35 MOVE 'FILE NOT FOUND' TO MSG
  WHEN 37 MOVE 'PERMISSION DENIED' TO MSG
  WHEN 38 MOVE 'CLOSED WITH LOCK' TO MSG
  WHEN 39 MOVE 'CONFLICT ATTRIBUTE' TO MSG
  WHEN 41 MOVE 'ALREADY OPEN' TO MSG
  WHEN 42 MOVE 'NOT OPEN' TO MSG
  WHEN 43 MOVE 'READ NOT DONE' TO MSG
  WHEN 44 MOVE 'RECORD OVERFLOW' TO MSG
  WHEN 46 MOVE 'READ ERROR' TO MSG
  WHEN 47 MOVE 'INPUT DENIED' TO MSG
  WHEN 48 MOVE 'OUTPUT DENIED' TO MSG
  WHEN 49 MOVE 'I/O DENIED' TO MSG
  WHEN 51 MOVE 'RECORD LOCKED' TO MSG
  WHEN 52 MOVE 'END-OF-PAGE' TO MSG
  WHEN 57 MOVE 'I/O LINAGE' TO MSG
  WHEN 61 MOVE 'FILE SHARING FAILURE' TO MSG
  WHEN 91 MOVE 'FILE NOT AVAILABLE' TO MSG
END-EVALUATE.
```

### 8.2. cobdump – A Hex/Char Data Dump Subroutine

This next sample program is a useful little utility subroutine to produce a formatted hexadecimal and character dump of the data area passed to it.

If you follow the OpenCOBOL forums, you’ve undoubtedly heard about the CBL\_OC\_DUMP subroutine that was the winning entry in an OpenCOBOL programming contest. It’s a great tool for producing data dumps, and it’ll probably be in the official OpenCOBOL distribution one of these days.

For now though, I’ll keep using my good ol’ “cobdump” routine. It’s been my travelling companion from COBOL job to COBOL job since 1971. Here it is, all tuned up for OpenCOBOL, with new tires and a fresh coat of paint:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    cobdump.
*****
** This is a COBOL subroutine that will generated a formatted **
** Hex/Char dump of a storage area. To use this subroutine, **
** simply CALL it as follows: **
** ** **
** CALL "cobdump" USING <data-item> **
** FUNCTION LENGTH(data-item) **
** ** **
** >>> Note that the subroutine name MUST be specified in <<< **
** >>> lower-case ** <<< **
```

```

**
** The dump is generated to STDERR, so you may pipe it to a
** file when you execute your program using "2> file".
**
** AUTHOR:      GARY L. CUTLER
**              CutlerGL@gmail.com
**
** NOTE:       The author has a sentimental attachment to
**              this subroutine - it's been around since 1971
**              and it's been converted to and run on 10 dif-
**              ferent operating system/compiler environments
**
** DATE-WRITTEN: October 14, 1971
**
*****
**  DATE  CHANGE DESCRIPTION
** =====
** GC1071 Initial coding - Univac Dept. of Defense COBOL '68
** GC0577 Converted to Univac ASCII COBOL (ACOB) - COBOL '74
** GC1182 Converted to Univac UTS4000 COBOL - COBOL '74 w/
**         SCREEN SECTION enhancements
** GC0883 Converted to Honeywell/Bull COBOL - COBOL '74
** GC0983 Converted to IBM VS COBOL - COBOL '74
** GC0887 Converted to IBM VS COBOL II - COBOL '85
** GC1294 Converted to Micro Focus COBOL V3.0 - COBOL '85 w/
**         extensions
** GC0703 Converted to Unisys Universal Compiling System (UCS)
**         COBOL (UCOB) - COBOL '85
** GC1204 Converted to Unisys Object COBOL (OCOB) - COBOL 2002
** GC0609 Converted to OpenCOBOL 1.1 - COBOL '85 w/ some COBOL
**         2002 features
*****
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Addr-Pointer          USAGE POINTER.
01  Addr-Number          REDEFINES Addr-Pointer
                          USAGE BINARY-LONG.

01  Addr-Sub             USAGE BINARY-CHAR.

01  Addr-Value          USAGE BINARY-LONG.

01  Buffer-Sub           COMP-5 PIC 9(4).

01  Hex-Digits          VALUE '0123456789ABCDEF'.
05  Hex-Digit           OCCURS 16 TIMES PIC X(1).

01  Left-Nibble        COMP-5 PIC 9(1).
01  Nibble              REDEFINES Left-Nibble
                          BINARY-CHAR.

01  Output-Detail.
05  OD-Address.
    10 OD-Addr-Hex-Digit OCCURS 8 TIMES PIC X.
    05 FILLER             PIC X(1).
05  OD-Byte            PIC Z(3)9.
05  FILLER              PIC X(1).
05  OD-Hex-Byte        OCCURS 16 TIMES.
    10 OD-Hex-Byte-1     PIC X.
    10 OD-Hex-Byte-2     PIC X.
    10 FILLER            PIC X.
05  OD-ASCII-Byte      OCCURS 16 TIMES
                          PIC X.

01  Output-Detail-Sub  COMP-5 PIC 9(2).

01  Output-Header-1.
05  FILLER              PIC X(80) VALUE
    '<-Addr-> Byte ' &
    '<----- Hexadecimal -----> ' &
    '<---- Char ---->'.

01  Output-Header-2.
05  FILLER              PIC X(80) VALUE
    '===== ' &
    '===== ' &
    '===== '.

01  PIC-XX.
05  FILLER              PIC X VALUE LOW-VALUES.
05  PIC-X               PIC X.
01  PIC-Halfword        REDEFINES PIC-XX
                          PIC 9(4) COMP-X.

```

```

01 PIC-X10.
   05 FILLER                PIC X(2).
   05 PIC-X8                PIC X(8).

01 Right-Nibble            COMP-5 PIC 9(1).

LINKAGE SECTION.
01 Buffer                  PIC X ANY LENGTH.

01 Buffer-Length          USAGE BINARY-LONG.

```

PROCEDURE DIVISION USING Buffer, Buffer-Length.

```

000-Initialize.
  MOVE SPACES TO Output-Detail
  SET Addr-Pointer TO ADDRESS OF Buffer
  PERFORM 200-Generate-Address
  MOVE 0 TO Output-Detail-Sub
  DISPLAY
    Output-Header-1 UPON SYSERR
  END-DISPLAY
  DISPLAY
    Output-Header-2 UPON SYSERR
  END-DISPLAY
  .

010-Generate-Dump.
  PERFORM 100-Process-Byte
  VARYING Buffer-Sub FROM 1 BY 1
  UNTIL Buffer-Sub > Buffer-Length
  IF Output-Detail-Sub > 0
  DISPLAY
    Output-Detail UPON SYSERR
  END-DISPLAY
  .

099-Exit-COBDUMP.
  GOBACK.

100-Process-Byte.
  ADD 1
  TO Output-Detail-Sub
  END-ADD
  IF Output-Detail-Sub = 1
  MOVE Buffer-Sub TO OD-Byte
  END-IF
  MOVE Buffer (Buffer-Sub : 1) TO PIC-X
  IF (PIC-X < ' ')
  OR (PIC-X > '~')
  MOVE 'u' TO OD-ASCII-Byte (Output-Detail-Sub)
  *
  * Note: the character above was formed using the keystroke sequence ALT-151
  * (the 1, 5 and 1 entered using the numeric keypad) - this character
  * will appear as a small filled-in circle
  *
  ELSE
  MOVE PIC-X
  TO OD-ASCII-Byte (Output-Detail-Sub)
  END-IF
  DIVIDE PIC-Halfword BY 16
  GIVING Left-Nibble
  REMAINDER Right-Nibble
  END-DIVIDE
  ADD 1 TO Left-Nibble
  Right-Nibble
  END-ADD
  MOVE Hex-Digit (Left-Nibble)
  TO OD-Hex-Byte-1 (Output-Detail-Sub)
  MOVE Hex-Digit (Right-Nibble)
  TO OD-Hex-Byte-2 (Output-Detail-Sub)
  IF Output-Detail-Sub = 16
  DISPLAY
    Output-Detail UPON SYSERR
  END-DISPLAY
  MOVE SPACES TO Output-Detail
  MOVE 0 TO Output-Detail-Sub
  SET Addr-Pointer UP BY 16
  PERFORM 200-Generate-Address
  END-IF
  .

200-Generate-Address.

```



```

MOVE 8 TO Addr-Sub
MOVE Addr-Number TO Addr-Value
MOVE ALL '0' TO OD-Address
PERFORM WITH TEST BEFORE UNTIL Addr-Value = 0
    DIVIDE Addr-Value BY 16
        GIVING Addr-Value
        REMAINDER Nibble
    END-DIVIDE
    ADD 1 TO Nibble
    MOVE Hex-Digit (Nibble)
        TO OD-Addr-Hex-Digit (Addr-Sub)
    SUBTRACT 1 FROM Addr-Sub
END-PERFORM
.
```

### 8.3. OCic – an OpenCOBOL Full-Screen Compiler Front-End

This is more than a mere demonstration program – it's also a very practical utility! The "OCic" (OpenCOBOL Interactive Compiler) is a TUI program that may be used to connect the "cobc" compiler into a text editing framework so that – by simply pressing a keystroke sequence while editing an OpenCOBOL program – you may trigger a compilation of that program. The program is well documented (IMHO) and you should find it fairly easy to follow. The OCic.cbl was written to work with a native Windows or Windows/MinGW build of OpenCOBOL as well as a Windows/Cygwin or UNIX build – I suspect it'll work for MacOS, since that OS is actually a derivative of Linux, but such compatibility is untested.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. OCIC.
*****
** This program provides a Textual User Interface (TUI) to the **
** process of compiling and (optionally) executing an OpenCOBOL **
** program. **
**
** This programs execution syntax is as follows: **
**
** ocic <program-path-and-filename> [ <switch>... ] **
**
** Once executed, a display screen will be presented showing **
** the compilation options that will be used. The user will **
** have the opportunity to change options, specify new ones **
** and specify any program execution arguments to be used if **
** you select the "Execute" option. When you press the Enter **
** key the program will be compiled. **
**
** The SCREEN SECTION contains an image of the screen. **
**
** The "O10-Parse-Args" section in the PROCEDURE DIVISION has **
** documentation on switches and their function. **
*****
** AUTHOR: GARY L. CUTLER **
** CutlerGL@gmail.com **
**
** DATE-WRITTEN: June 14, 2009 **
**
*****
** DATE CHANGE DESCRIPTION **
**
** GC0609 Don't display compiler messages file if compilation **
** Is successful. Also don't display messages if the **
** output file is busy (just put a message on the **
** screen, leave the OC screen up & let the user fix **
** the problem & resubmit. **
** GC0709 when 'EXECUTE' is selected, a 'FILE BUSY' error will **
** still cause the (old) executable to be launched. **
** Also, the 'EXTRA SWITCHES' field is being ignored. **
** Changed the title bar to lowlighted reverse video & **
** the message area to highlighted reverse-video. **
** GC0809 Add a SPACE in from of command-line args when **
** executing users program. Add a SPACE after the **
** -ftraceall switch when building cobc command. **
** GC0909 Convert to work on Cygwin/Linux as well as MinGW **
*****
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
FUNCTION ALL INTRINSIC.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT Bat-File ASSIGN TO Bat-File-Name
    ORGANIZATION IS LINE SEQUENTIAL.

    SELECT Cobc-Output ASSIGN TO Cobc-Output-File
    ORGANIZATION IS LINE SEQUENTIAL.

    SELECT Source-Code ASSIGN TO File-Name
    ORGANIZATION IS LINE SEQUENTIAL
    FILE STATUS IS FSM-Status.
```

```

DATA DIVISION.
FILE SECTION.
FD Bat-File.
01 Bat-File-Rec                PIC X(2048).

FD Cobc-Output.
01 Cobc-Output-Rec            PIC X(256).

FD Source-Code.
01 Source-Code-Record        PIC X(80).

WORKING-STORAGE SECTION.
COPY screenio.

01 Bat-File-Name                PIC X(256).
GC0909 01 Cmd                    PIC X(512).
01 Cobc-Cmd                    PIC X(256).
01 Cobc-Output-File            PIC X(256).
01 Command-Line-Args          PIC X(256).
01 Config-File                 PIC X(12).
GC0909 01 Dir-Char                PIC X(1).
01 Dummy                       PIC X(1).
01 Env-TEMP                    PIC X(256).
01 File-Name.
05 FN-Char                    OCCURS 256 TIMES PIC X(1).
01 File-Status-Message.
05 FILLER                      PIC X(13) VALUE 'Status Code: '.
05 FSM-Status                  PIC 9(2).
05 FILLER                      PIC X(11) VALUE ', Meaning: '.
05 FSM-Msg                    PIC X(25).
01 Flags.
05 F-Compilation-Succeeded     PIC X(1).
GC0909 88 88-Compile-OK         VALUE 'Y'.
88 88-Compile-OK-Warn         VALUE 'W'.
88 88-Compile-Failed          VALUE 'N'.
GC0609 05 F-Complete            PIC X(1).
GC0609 88 88-Complete           VALUE 'Y'.
GC0609 88 88-Not-Complete       VALUE 'N'.
GC0809 05 F-IDENT-DIVISION      PIC X(1).
GC0809 88 88-1st-Prog-Complete  VALUE 'Y'.
GC0809 88 88-More-To-1st-Prog   VALUE 'N'.
05 F-LINKAGE-SECTION           PIC X(1).
88 88-Compile-As-Subpgm       VALUE 'Y'.
88 88-Compile-As-Mainpgm      VALUE 'N'.
05 F-No-Switch-Changes         PIC X(1).
88 88-No-Switch-Changes       VALUE 'Y'.
88 88-Switch-Changes           VALUE 'N'.
GC0709 05 F-Output-File-Busy     PIC X(1).
GC0709 88 88-Output-File-Busy   VALUE 'Y'.
GC0709 88 88-Output-File-Avail  VALUE 'N'.
GC0809 05 F-Source-Record-Type  PIC X(1).
GC0809 88 88-Source-Rec-Linkage VALUE 'L'.
GC0809 88 88-Source-Rec-Ident   VALUE 'I'.
GC0809 88 88-Source-Rec-Ignocob-COLOR-RED VALUE ' '.
05 F-Switch-Error              PIC X(1).
88 88-Switch-Is-Bad           VALUE 'Y'.
88 88-Switch-Is-Good          VALUE 'N'.

GC0909 01 Horizontal-Line        PIC X(80).
GC0909 01 I                      USAGE BINARY-LONG.
01 J                          USAGE BINARY-LONG.
GC0909 01 MS                     USAGE BINARY-LONG.
GC0909 01 ML                     USAGE BINARY-LONG.
01 OC-Compiled                 PIC XXXX/XX/XXBXX/XX.

GC0909 01 OS-Type                USAGE BINARY-LONG.
GC0909 88 OS-Unknown             VALUE 0.
GC0909 88 OS-Windows             VALUE 1.
GC0909 88 OS-Cygwin              VALUE 2.
GC0909 88 OS-UNIX                VALUE 3.

GC0909 01 OS-Type-Literal        PIC X(7).
01 Output-Message              PIC X(80).
01 Path-Delimiter              PIC X(1).
01 Prog-Folder                  PIC X(256).
01 Prog-Extension               PIC X(30).

```

```

01 Prog-File-Name          PIC X(40).
01 Prog-Name              PIC X(31).
78 Selection-Char        VALUE '>'.
01 Switch-Display.
05 SD-Switch-And-Value   PIC X(19).
05 FILLER                PIC X(1).
05 SD-Description        PIC X(60).
01 Switch-Keyword        PIC X(12).
88 Switch-Is-CONFIG      VALUE '/CONFIG', '/C'.
88 Switch-Is-DEBUG       VALUE '/DEBUG', '/D'.
88 Switch-Is-DLL         VALUE '/DLL'.
88 Switch-Is-EXECUTE     VALUE '/EXECUTE', '/E'.
88 Switch-Is-EXTRA       VALUE '/EXTRA', '/EX'.
88 Switch-Is-NOTRUNC     VALUE '/NOTRUNC', '/N'.
88 Switch-Is-TRACE       VALUE '/TRACE', '/T'.
01 Switch-Keyword-And-Value PIC X(256).
01 Switch-Value.
05 SV-1                  PIC X(1).
05 FILLER                PIC X(255).
01 Switch-Value-Alt      REDEFINES Switch-Value
                        PIC X(256).
88 Valid-Config-Filename
   VALUE 'BS2000', 'COBOL85', 'COBOL2002', 'DEFAULT',
   'IBM', 'MF', 'MVS'.
01 Switches.
05 S-ARGS                PIC X(75) VALUE SPACES.
05 S-Cfgs.
10 S-Cfg-BS2000          PIC X(1) VALUE ' '.
10 S-Cfg-COBOL85         PIC X(1) VALUE ' '.
10 S-Cfg-COBOL2002       PIC X(1) VALUE ' '.
10 S-Cfg-DEFAULT         PIC X(1) VALUE Selection-Char.
10 S-Cfg-IBM             PIC X(1) VALUE ' '.
10 S-Cfg-MF              PIC X(1) VALUE ' '.
10 S-Cfg-MVS             PIC X(1) VALUE ' '.
05 S-EXTRA               PIC X(75) VALUE SPACES.
05 S-Yes-No-Switches.
10 S-DEBUG               PIC X(1) VALUE 'N'.
10 S-DLL                 PIC X(1) VALUE 'N'.
10 S-EXECUTE             PIC X(1) VALUE 'N'.
10 S-NOTRUNC             PIC X(1) VALUE 'Y'.
10 S-SUBROUTINE          PIC X(1) VALUE 'A'.
10 S-TRACE               PIC X(1) VALUE 'N'.
10 S-TRACEALL            PIC X(1) VALUE 'N'.
01 Tally                 USAGE BINARY-LONG.
GC0809 01 Version-Number PIC X(5) VALUE '2.1'.

```

```

SCREEN SECTION.
>>SOURCE FREE

```

```

*
* Here is the layout of the OCic screen.
*
* Note that this program can utilize the traditional PC line-drawing characters,
* if they are available.
*
* If this program is run on windows, it must run with codepage 437 activated to
* display the line-drawing characters. With a native windows build or a
* windows/MinGW build, one could use the command "chcp 437" to set that codepage
* for display within a Windows console window (that should be the default, though).
* with a windows/Cygwin build, set the environment variable CYGWIN to a value of
* "codepage:oem" (this cannot be done from within the program though - you will
* have to use the "Computer/Advanced System Settings/Environment Variables" (Vista or
* Windows 7) function to define the variable. XP Users: use "My Computer/Properties/
* Advanced/Environment Variables".
*
* To use OCic without the line-drawing characters, comment-out the first set of
* 78 "LD" items and uncomment the second.
*
* The following sample screen layout shows how the screen looks with line-drawing
* characters disabled.
*
*-----
* openCobol Interactive Compilation Vxxxxx (YYYY/MM/DD HH:MM)                xxxxxxxx 01
*-----
* | Program: xxx...xxxxx                               Press an F-Key to 03
* | Folder:  xxx...xxxxxxxxxxxxxxxxx                   select the cor- 04
* | Filename: xxx...xxxxx.cbl                           responding option 05
* |-----
* | on/off switches:                                     Configuration (Select One): 07
* |-----
* | F1 Activate debugging lines                        S-F1 BS2000           Press ENTER 09
* | F2 Generate MAIN PROGRAMS as DLLs                  S-F2 COBOL85         to start   10
* | F3 This program is a SUBROUTINE                    S-F3 COBOL2002       compilation. 11
* | F4 Execute on successful compilation                S-F4 > Default      12
* | F5 > Do NOT truncate COMP/BINARY values            S-F5 IBM             Press Esc  13
* | F6 Trace procedures                                S-F6 Micro Focus    to cancel. 14
* | F7 Trace procedures + statements                    S-F7 MVS              15
* |-----
* | Additional "cobc" Switches (if any):                17

```



```

05 LINE 18 COL 02          VALUE LD-UL-Corner.
05                        PIC X(77) FROM Horizontal-Line.
05                        COL 80 VALUE LD-UR-Corner.

05 LINE 19 COL 02          VALUE LD-Vert-Line.
05                        COL 80 VALUE LD-Vert-Line.

05 LINE 20 COL 02          VALUE LD-LL-Corner.
05                        PIC X(77) FROM Horizontal-Line.
05                        COL 80 VALUE LD-LR-Corner.

05 LINE 22 COL 02          VALUE LD-UL-Corner.
05                        PIC X(77) FROM Horizontal-Line.
05                        COL 80 VALUE LD-UR-Corner.

05 LINE 23 COL 02          VALUE LD-Vert-Line.
05                        COL 80 VALUE LD-Vert-Line.

05 LINE 24 COL 02          VALUE LD-LL-Corner.
05                        PIC X(77) FROM Horizontal-Line.
05                        COL 80 VALUE LD-LR-Corner.
# # #
# # # TOP AND BOTTOM LINES
03 BACKGROUND-COLOR COB-COLOR-BLUE BLINK FOREGROUND-COLOR COB-COLOR-WHITE HIGHLIGHT.
05 LINE 01 COL 01 VALUE ' OpenCOBOL Interactive Compilation V'.
05                        PIC X(5) FROM Version-Number.
05                        VALUE ' ('.
05                        PIC X(16) FROM OC-Compiled.
05                        VALUE ')'.
05 LINE 01 COL 74 PIC X(7) FROM OS-Type-Literal.
05 LINE 25 COL 01 PIC X(81) FROM Output-Message.
# # #
# # # LABELS
03 BACKGROUND-COLOR COB-COLOR-BLACK FOREGROUND-COLOR COB-COLOR-CYAN HIGHLIGHT.
05 LINE 07 COL 04 VALUE 'On/Off Switches:'.
05                        COL 47 VALUE 'Configuration (Select One):'.
05 LINE 17 COL 04 VALUE 'Additional "cobc" Switches (if any):'.
05 LINE 21 COL 04 VALUE 'Program Execution Arguments (if any):'.
# # #
# # # TOP SECTION BACKGROUND
03 BACKGROUND-COLOR COB-COLOR-BLACK FOREGROUND-COLOR COB-COLOR-CYAN LOWLIGHT.
05 LINE 03 COL 04 VALUE 'Program: '.
05 LINE 04 COL 04 VALUE 'Folder: '.
05 LINE 05 COL 04 VALUE 'Filename: '.

05 LINE 03 COL 62 VALUE 'Press an F-Key to'.
05 LINE 04 COL 62 VALUE 'select the cor-'.
05 LINE 05 COL 62 VALUE 'responding option'.
# # #
# # # TOP SECTION PROGRAM INFO
03 BACKGROUND-COLOR COB-COLOR-BLACK FOREGROUND-COLOR COB-COLOR-WHITE HIGHLIGHT.
05 LINE 03 COL 14 PIC X(47) FROM Prog-Name.
05 LINE 04 COL 14 PIC X(47) FROM Prog-Folder.
05 LINE 05 COL 14 PIC X(47) FROM Prog-File-Name.
# # #
# # # MIDDLE LEFT SECTION F-KEYS
03 BACKGROUND-COLOR COB-COLOR-BLACK FOREGROUND-COLOR COB-COLOR-WHITE HIGHLIGHT.
05 LINE 09 COL 04 VALUE 'F1'.
05 LINE 10 COL 04 VALUE 'F2'.
05 LINE 11 COL 04 VALUE 'F3'.
05 LINE 12 COL 04 VALUE 'F4'.
05 LINE 13 COL 04 VALUE 'F5'.
05 LINE 14 COL 04 VALUE 'F6'.
05 LINE 15 COL 04 VALUE 'F7'.
# # #
# # # MIDDLE LEFT SECTION SWITCHES
03 BACKGROUND-COLOR COB-COLOR-BLACK FOREGROUND-COLOR COB-COLOR-RED HIGHLIGHT.
05 LINE 09 COL 07 PIC X(1) FROM S-DEBUG.
05 LINE 10 COL 07 PIC X(1) FROM S-DLL.
05 LINE 11 COL 07 PIC X(1) FROM S-SUBROUTINE.
05 LINE 12 COL 07 PIC X(1) FROM S-EXECUTE.
05 LINE 13 COL 07 PIC X(1) FROM S-NOTRUNC.
05 LINE 14 COL 07 PIC X(1) FROM S-TRACE.
05 LINE 15 COL 07 PIC X(1) FROM S-TRACEALL.
# # #
# # # MIDDLE LEFT SECTION BACKGROUND
03 BACKGROUND-COLOR COB-COLOR-BLACK FOREGROUND-COLOR COB-COLOR-CYAN LOWLIGHT.
05 LINE 09 COL 09 VALUE 'Activate debugging lines'.
05 LINE 10 COL 09 VALUE 'Generate MAIN PROGRAMS as DLLs'.
05 LINE 11 COL 09 VALUE 'This program is a SUBROUTINE'.
05 LINE 12 COL 09 VALUE 'Execute on successful compilation'.
05 LINE 13 COL 09 VALUE 'Do NOT truncate COMP/BINARY values'.
05 LINE 14 COL 09 VALUE 'Trace procedures'.
05 LINE 15 COL 09 VALUE 'Trace procedures + statements'.
# # #
# # # MIDDLE RIGHT SECTION F-KEYS
03 BACKGROUND-COLOR COB-COLOR-BLACK FOREGROUND-COLOR COB-COLOR-WHITE HIGHLIGHT.
05 LINE 09 COL 47 VALUE 'S-F1'.

```

```

05 LINE 10 COL 47 VALUE 'S-F2'.
05 LINE 11 COL 47 VALUE 'S-F3'.
05 LINE 12 COL 47 VALUE 'S-F4'.
05 LINE 13 COL 47 VALUE 'S-F5'.
05 LINE 14 COL 47 VALUE 'S-F6'.
05 LINE 15 COL 47 VALUE 'S-F7'.
#>
#> MIDDLE RIGHT SECTION SWITCHES
03 BACKGROUND-COLOR COB-COLOR-BLACK FOREGROUND-COLOR COB-COLOR-RED HIGHLIGHT.
05 LINE 09 COL 52 PIC X(1) FROM S-Cfg-BS2000.
05 LINE 10 COL 52 PIC X(1) FROM S-Cfg-COBOL85.
05 LINE 11 COL 52 PIC X(1) FROM S-Cfg-COBOL2002.
05 LINE 12 COL 52 PIC X(1) FROM S-Cfg-DEFAULT.
05 LINE 13 COL 52 PIC X(1) FROM S-Cfg-IBM.
05 LINE 14 COL 52 PIC X(1) FROM S-Cfg-MF.
05 LINE 15 COL 52 PIC X(1) FROM S-Cfg-MVS.
#>
#> MIDDLE RIGHT SECTION BACKGROUND
03 BACKGROUND-COLOR COB-COLOR-BLACK FOREGROUND-COLOR COB-COLOR-CYAN LOWLIGHT.
05 LINE 09 COL 54 VALUE 'BS2000      Press      ' .
05 LINE 10 COL 54 VALUE 'COBOL85      to start   ' .
05 LINE 11 COL 54 VALUE 'COBOL2002   compilation.' .
05 LINE 12 COL 54 VALUE 'Default     ' .
05 LINE 13 COL 54 VALUE 'IBM         Press Esc  ' .
05 LINE 14 COL 54 VALUE 'Micro Focus to cancel.' .
05 LINE 15 COL 54 VALUE 'MVS        ' .
#>
#> MIDDLE RIGHT SECTION FOREGROUND
03 BACKGROUND-COLOR COB-COLOR-BLACK FOREGROUND-COLOR COB-COLOR-WHITE HIGHLIGHT.
05 LINE 09 COL 73 VALUE 'ENTER'.
05 LINE 13 COL 73 VALUE 'Esc'.
#>
#> FREE-FORM OPTIONS FIELDS
03 BACKGROUND-COLOR COB-COLOR-BLACK FOREGROUND-COLOR COB-COLOR-WHITE HIGHLIGHT.
05 LINE 19 COL 04 PIC X(75) USING S-EXTRA.
05 LINE 23 COL 04 PIC X(75) USING S-ARGS.

>>SOURCE FIXED

PROCEDURE DIVISION.
*****
** Legend to procedure names:          **
**                                     **
** 00x-xxx  All MAIN driver procedures **
** 0xx-xxx  All GLOBAL UTILITY procedures **
** 1xx-xxx  All INITIALIZATION procedures **
** 2xx-xxx  All CORE PROCESSING procedures **
** 9xx-xxx  All TERMINATION procedures **
*****
DECLARATIVES.
000-File-Error SECTION.
  USE AFTER STANDARD ERROR PROCEDURE ON Source-Code.
000-Handle-Error.
  COPY FileStat-Msgs
    REPLACING STATUS BY FSM-Status
              MSG BY FSM-Msg.
  MOVE SPACES TO Output-Message
  IF FSM-Status = 35
    DISPLAY
      'File not found: "'
      TRIM(File-Name,TRAILING)
      '"'
    END-DISPLAY
  ELSE
    DISPLAY
      'Error accessing file: "'
      TRIM(File-Name,TRAILING)
      '"'
    END-DISPLAY
  END-IF
  GOBACK
  .
END DECLARATIVES.

000-Main SECTION.
  PERFORM 100-Initialization
  SET 88-Not-Complete TO TRUE
  GC0609 PERFORM UNTIL 88-Complete
  GC0609     PERFORM 200-Let-User-Set-Switches
  GC0609     PERFORM 210-Run-Compiler
  GC0709     IF (S-EXECUTE NOT = SPACES)
  GC0709     AND (88-Output-File-Avail)
  GC0609     PERFORM 230-Run-Program
  GC0609     END-IF
  GC0609 END-PERFORM
  .

009-Done.
  PERFORM 900-Terminate
  .
* -- Control will NOT return

```

```

010-Parse-Args SECTION.
*****
** Process a sequence of KEYWORD=VALUE items. These are items **
** specified on the command-line to provide the initial **
** options shown selected on the screen. When integrating **
** OCic into an editor or framework, include these switches on **
** the ocic.exe command the editor/framework executes. Any **
** underlined choice is the default value for that switch. **
**
** /CONFIG=BS2000|COBOL85|COBOL2002|DEFAULT|IBM|MF|MVS **
**          ===== **
** This switch specifies the default cobc compiler configura- **
** tion file to be used **
**
** /DEBUG=YES|NO **
**          == **
** This switch specifies whether (YES) or not (NO) debugging **
** lines (those with a "D" in column 7) will be compiled. **
**
** /DLL=YES|NO **
**          == **
** Use this switch to force ALL compiled programs to be built **
** as DLLs ("/DLL=YES"). When main programs are built as DLLs **
** they must be executed using the cobcrun utility. When **
** "/DLL=NO" is in effect, main programs are generated as **
** actual "exe" files and only subprograms will be generated **
** as DLLs. **
**
** /EXECUTE=YES|NO **
**          == **
** This switch specifies whether ("/EXECUTE=YES") or not **
** ("/EXECUTE=NO") the program will be executed after it is **
** successfully compiled. **
**
** /EXTRA=extra cobc argument(s) **
**
** This switch allows you to specify additional cobc arguments **
** that aren't managed by the other OC switches. If used, **
** this must be the last switch specified on the command line, **
** as everything that follows the "=" will be placed on the **
** cobc command generated by OC. **
**
** /NOTRUNC=YES|NO **
**          == **
** This switch specifies whether (YES) or not (NO) the sup- **
** pression of binary field truncation will occur. If a PIC **
** 99 COMP field (one byte of storage), for example, is given **
** the value 123, it may have its value truncated to 23 when **
** DISPLAYed. Regardless of the NOTRUNC setting, internally **
** the full precision of the field (allowing a maximum value **
** of 255) will be preserved. Even though truncation - if it **
** does occur - would appear to have a minimal disruption on **
** program operation, it has a significant effect on program **
** run-time speed. **
**
** /TRACE=YES|NO|ALL **
**          == **
** This switch controls whether or not code will be added to **
** the object program to produce execution-time logic traces. **
** A specification of "/TRACE=NO" means no such code will be **
** produced. By specifying "/TRACE=YES", code will be genera- **
** ted to display procedure names as they are entered. A **
** "/TRACE=ALL" specification will generate not only procedure **
** traces (as "/TRACE=YES" would) but also statement-level **
** traces too! All trace output is written to STDERR, so **
** adding a ">file" to the execution of the program will pipe **
** the trace output to a file. You may find it valuable to **
** add your own DISPLAY statements to the debugging output via **
** "DISPLAY xx UPON SYSERR" The SYSERR device corresponds to **
** the Windows or UNIX STDERR device and will therefore honor **
** any ">file" placed at the end of your program's execution. **
** Add a "D" in column 7 and you can control the generation or **
** ignoring of these DISPLAY statements via the "/DEBUG" **
** switch. **
*****
** On Entry: 'Command-Line-Args' contains the K=V items **
** On Exit: All 'S-xxxx' values corresponding to the switches **
** supplied in 'Command-Line-Args' will be set pro- **
** vided there weren't any problems with the values. **
** If there were problems, error messages will have **
** been generated. **
*****

```

```

011-Init.
  MOVE 1 TO I
  .

```

```

012-Extract-Kwd-And-Value.
  PERFORM UNTIL I NOT < LENGTH(Command-Line-Args)
    MOVE I TO J
    UNSTRING Command-Line-Args
      DELIMITED BY ALL SPACES
      INTO Switch-Keyword-And-Value
      WITH POINTER I
    END-UNSTRING
    IF Switch-Keyword-And-Value NOT = SPACES
      UNSTRING Switch-Keyword-And-Value

```

```

        DELIMITED BY '='
        INTO Switch-Keyword, Switch-Value
    END-UNSTRING
    PERFORM 030-Process-Keyword
END-IF
END-PERFORM
.

019-Done.
EXIT.

*****
** Since this program uses the SCREEN SECTION, it cannot do **
** conventional console DISPLAY operations. This routine **
** (which, I admit, is like using an H-bomb to hunt rabbits) **
** will submit an "ECHO" command to the system to simulate a **
** DISPLAY. **
*****
** On Entry: 'Output-Message' is expected to contain the mes- **
** sage to be displayed **
** On Exit: The message should now be on the console window **
*****
021-Build-And-Issue-Command.
    DISPLAY
        Output-Message
    END-DISPLAY
.

029-Done.
EXIT.

030-Process-Keyword SECTION.
*****
** Process a single KEYWORD=VALUE item. **
*****
** On Entry: The 'Switch-Keyword' and 'Switch-Value' items are **
** expected to have been populated (010-Parse-Args) **
** On Exit: Either the appropriate 'S-xxxx' item has been **
** changed or the appropriate error message has been **
** displayed **
*****

031-Init.
    MOVE UPPER-CASE(Switch-Keyword) TO Switch-Keyword
    SET 88-Switch-Is-Good TO TRUE
.

032-Process.
    EVALUATE TRUE
        WHEN Switch-Is-EXTRA
            UNSTRING Command-Line-Args DELIMITED BY '='
                INTO Dummy, S-EXTRA
            MOVE LENGTH(Command-Line-Args) TO I
            WHEN Switch-Is-CONFIG
                MOVE 'CONFIG' TO Switch-Keyword
                MOVE UPPER-CASE(Switch-Value)
                    TO Switch-Value
                EVALUATE Switch-Value
                    WHEN 'BS2000'
                        MOVE SPACES TO S-CfgS
                        MOVE Selection-Char TO S-Cfg-BS2000
                    WHEN 'COBOL85'
                        MOVE SPACES TO S-CfgS
                        MOVE Selection-Char TO S-Cfg-COBOL85
                    WHEN 'COBOL2002'
                        MOVE SPACES TO S-CfgS
                        MOVE Selection-Char TO S-Cfg-COBOL2002
                    WHEN 'DEFAULT'
                        MOVE SPACES TO S-CfgS
                        MOVE Selection-Char TO S-Cfg-DEFAULT
                    WHEN 'IBM'
                        MOVE SPACES TO S-CfgS
                        MOVE Selection-Char TO S-Cfg-IBM
                    WHEN 'MF'
                        MOVE SPACES TO S-CfgS
                        MOVE Selection-Char TO S-Cfg-MF
                    WHEN 'MVS'
                        MOVE SPACES TO S-CfgS
                        MOVE Selection-Char TO S-Cfg-MVS
                    WHEN OTHER
                        MOVE 'An invalid /CONFIG switch value ' &
                            'was specified on the command line ' &
                            '- ignored'
                            TO Output-Message
                END-EVALUATE
            WHEN Switch-Is-DEBUG
                MOVE 'DEBUG' TO Switch-Keyword
                MOVE UPPER-CASE(Switch-Value)
                    TO Switch-Value
                PERFORM 040-Process-Yes-No-Value
                IF 88-Switch-Is-Good
                    MOVE SV-1 TO S-DEBUG
                END-IF
            WHEN Switch-Is-EXECUTE
                MOVE 'EXECUTE' TO Switch-Keyword
                MOVE UPPER-CASE(Switch-Value)
                    TO Switch-Value
    END-EVALUATE

```



```

        PERFORM 040-Process-Yes-No-Value
        IF 88-Switch-Is-Good
            MOVE SV-1 TO S-EXECUTE
        END-IF
    WHEN Switch-Is-NOTRUNC
        MOVE 'NOTRUNC' TO Switch-Keyword
        MOVE UPPERCASE(Switch-Value)
            TO Switch-Value
        PERFORM 040-Process-Yes-No-Value
        IF 88-Switch-Is-Good
            MOVE SV-1 TO S-NOTRUNC
        END-IF
    WHEN Switch-Is-TRACE
        MOVE 'TRACE' TO Switch-Keyword
        MOVE UPPERCASE(Switch-Value)
            TO Switch-Value
        PERFORM 050-Process-Yes-No-All
        IF 88-Switch-Is-Good
            MOVE SV-1 TO S-TRACE
        END-IF
    WHEN OTHER
        MOVE SPACES TO Output-Message
        STRING " "
            TRIM(Switch-Keyword)
            " is not a valid switch ' &
            ' - ignored'
            DELIMITED SIZE
            INTO Output-Message
        END-STRING
        SET 88-Switch-Is-Bad TO TRUE
    END-EVALUATE
.
039-Done.
EXIT.

040-Process-Yes-No-Value SECTION.
*****
** Process a switch value of YES or NO **
*****
** On Entry: The 'Switch-Keyword' and 'Switch-Value' items are **
** expected to have been populated (010-Parse-Args) **
** On Exit: The 'Switch-Value' item will be updated with the **
** full word YES or NO (remember, the user could **
** have specified just a single letter) OR an error **
** message will have been generated **
*****

042-Process.
    EVALUATE SV-1
        WHEN 'Y'
            MOVE 'YES' TO Switch-Value
        WHEN 'N'
            MOVE 'NO' TO Switch-Value
        WHEN OTHER
            MOVE SPACES TO Output-Message
            STRING " *ERROR: " TRIM(Switch-Value)
                " is not a valid value for the "
                TRIM(Switch-Keyword) " switch"
                DELIMITED SPACES
                INTO Output-Message
            END-STRING
            SET 88-Switch-Is-Bad TO TRUE
    END-EVALUATE
.
049-Done.
EXIT.

050-Process-Yes-No-All SECTION.
*****
** Process a switch value of YES, NO or ALL **
*****
** On Entry: The 'Switch-Keyword' and 'Switch-Value' items are **
** expected to have been populated (010-Parse-Args) **
** On Exit: The 'Switch-Value' item will be updated with the **
** full word YES, NO or ALL (remember, the use could **
** have specified just a single letter) OR an error **
** message will have been generated **
*****

052-Process.
    IF SV-1 = 'A'
        MOVE 'ALL' TO Switch-Value
    ELSE
        PERFORM 040-Process-Yes-No-Value
    END-IF
.
059-Done.
EXIT.

060-Process-Yes-No-Auto SECTION.
*****
** Process a switch value of YES, NO or AUTO **
*****
** On Entry: The 'Switch-Keyword' and 'Switch-Value' items are **

```

```

** expected to have been populated (010-Parse-Args) **
** On Exit: The 'Switch-Value' item will be updated with the **
** full word YES or NO (remember, the use could have **
** specified just a single letter) OR an error **
** message will have been generated; in the event **
** that a value of AUTO was encountered, the source **
** code of the program will have been parsed, using **
** the presence or absence of an un-commented **
** "LINKAGE SECTION" (case-independent) to set a YES **
** or NO value, respectively **
*****

```

```

061-Init.
  IF SV-1 = 'A'
    PERFORM 070-Find-LINKAGE-SECTION
    IF 88-Compile-As-Subpgm
      MOVE 'Y' TO Switch-Value
    ELSE
      MOVE 'N' TO Switch-Value
    END-IF
  ELSE
    PERFORM 040-Process-Yes-No-Value
  END-IF
.

```

```

070-Find-LINKAGE-SECTION SECTION.
*****
** Determine if the program being compiled is a MAIN program **
*****
** On Entry: The 'File-Name' item is assumed to have been **
** populated with the complete path and filename of **
** the program being compiled (100-Initialization) **
** On Exit: The 'F-LINKAGE-SECTION' flag will be set to a 'Y' **
** or 'N', depending on whether or not a 'LINKAGE **
** SECTION' was found (uncommented) in the program. **
*****

```

```

071-Init.
  OPEN INPUT Source-Code
  SET 88-Compile-As-Mainpgm TO TRUE
  SET 88-More-To-1st-Prog TO TRUE
  PERFORM UNTIL 88-1st-Prog-Complete
    READ Source-Code AT END
    CLOSE Source-Code
    EXIT SECTION
  END-READ
  CALL 'checksource' USING Source-Code-Record
                        F-Source-Record-Type
  END-CALL
  IF 88-Source-Rec-Ident
    SET 88-1st-Prog-Complete TO TRUE
  END-IF
END-PERFORM
.

```

```

072-Process-Source.
  SET 88-Source-Rec-IgnocOB-COLOR-RED TO TRUE
  PERFORM UNTIL 88-Source-Rec-Linkage
    OR 88-Source-Rec-Ident
    READ Source-Code AT END
    CLOSE Source-Code
    EXIT SECTION
  END-READ
  CALL 'checksource' USING Source-Code-Record
                        F-Source-Record-Type
  END-CALL
END-PERFORM
CLOSE Source-Code
IF 88-Source-Rec-Linkage
  SET 88-Compile-As-Subpgm TO TRUE
END-IF
.

```

```

079-Done.
  EXIT.

```

```

100-Initialization SECTION.
*****
** Perform all program-wide initialization operations **
*****
** On Entry: There are no pre-requisites to this routine **
** On Exit: The 'OC-Compiled' string has been built & **
** all command line arguments have been processed. **
** If no program path/filename was found on the **
** command line a "syntax Error" message will have **
** displayed and the program will have halted. **
*****

```

```

GC0909 101-Determine-OS-Type.
GC0909   CALL 'getostype'
GC0909   END-CALL
GC0909   MOVE RETURN-CODE TO OS-Type
GC0909   EVALUATE TRUE
GC0909     WHEN OS-Unknown
GC0909       MOVE '\ ' TO Dir-Char
GC0909       MOVE 'Unknown' TO OS-Type-Literal

```

```

GC0909     WHEN OS-Windows
GC0909         MOVE '\\' TO Dir-Char
GC0909         MOVE 'windows' TO OS-Type-Literal
GC0909     WHEN OS-Cygwin
GC0909         MOVE '/' TO Dir-Char
GC0909         MOVE 'Cygwin' TO OS-Type-Literal
GC0909     WHEN OS-UNIX
GC0909         MOVE '/' TO Dir-Char
GC0909         MOVE 'UNIX' TO OS-Type-Literal
GC0909     END-EVALUATE
GC0909     .

102-Set-Environment-Vars.
    SET ENVIRONMENT 'COB_SCREEN_EXCEPTIONS' TO 'Y'
    SET ENVIRONMENT 'COB_SCREEN_ESC' TO 'Y'
    .

103-Generate-Cobc-Output-Fn.
    ACCEPT Env-TEMP
        FROM ENVIRONMENT "TEMP"
    END-ACCEPT
    MOVE SPACES TO Cobc-Output-File
    STRING TRIM(Env-TEMP,TRAILING)
GC0909         Dir-Char
GC0909         'OC-Messages.TXT'
        DELIMITED SIZE
        INTO Cobc-Output-File
    END-STRING
    .

104-Generate-Banner-Line-Info.
    MOVE WHEN-COMPILED (1:12) TO OC-Compiled
    INSPECT OC-Compiled
        REPLACING ALL '/' BY ':'
        AFTER INITIAL SPACE
    .

105-Establish-Switch-Settings.
    ACCEPT Command-Line-Args
        FROM COMMAND-LINE
    END-ACCEPT
    MOVE TRIM(Command-Line-Args, Leading)
        TO Command-Line-Args
    MOVE 0 TO Tally
    INSPECT Command-Line-Args TALLYING Tally FOR ALL '/'
    IF Tally = 0
        MOVE Command-Line-Args TO File-Name
        MOVE SPACES TO Command-Line-Args
    ELSE
        UNSTRING Command-Line-Args DELIMITED BY '/'
            INTO File-Name, Dummy
        END-UNSTRING
        INSPECT Command-Line-Args
            REPLACING FIRST '/' BY LOW-VALUES
        UNSTRING Command-Line-Args
            DELIMITED BY LOW-VALUES
            INTO Dummy, Cmd
        END-UNSTRING
        MOVE SPACES TO Command-Line-Args
        STRING '/' Cmd DELIMITED SIZE
            INTO Command-Line-Args
        END-STRING
    END-IF
    IF File-Name = SPACES
        DISPLAY
            'No program filename was specified'
        END-DISPLAY
        PERFORM 900-Terminate
    END-IF
    PERFORM 010-Parse-Args
    IF S-SUBROUTINE = 'A'
        MOVE 'S' TO Switch-Keyword
        MOVE 'A' TO Switch-Value
        PERFORM 070-Find-LINKAGE-SECTION
        IF 88-Compile-As-Subpgm
            MOVE 'Y' TO S-SUBROUTINE
        ELSE
            MOVE 'N' TO S-SUBROUTINE
        END-IF
    END-IF
    INSPECT S-Yes-No-Switches REPLACING ALL 'Y' BY Selection-Char
    INSPECT S-Yes-No-Switches REPLACING ALL 'N' BY ' '
    .

106-Determine-Folder-Path.
GC0909     Move 256 TO I
GC0909     IF OS-Cygwin AND File-Name (2:1) = ':'
GC0909         MOVE '\\' TO Dir-Char
GC0909     END-IF
    PERFORM UNTIL I = 0 OR FN-Char (I) = Dir-Char
        SUBTRACT 1 FROM I
    END-PERFORM
    IF I = 0
        MOVE SPACES TO Prog-Folder
        MOVE File-Name TO Prog-File-Name
    ELSE
        MOVE '*' TO FN-Char (I)

```

```

        UNSTRING File-Name DELIMITED BY '*'
            INTO Prog-Folder
                Prog-File-Name
        END-UNSTRING
        MOVE Dir-Char TO FN-Char (I)
    END-IF
    UNSTRING Prog-File-Name DELIMITED BY '.'
        INTO Prog-Name, Prog-Extension
    END-UNSTRING
    IF Prog-Folder = SPACES
        ACCEPT Prog-Folder
            FROM ENVIRONMENT 'CD'
        END-ACCEPT
    ELSE
        CALL "CBL_CHANGE_DIR"
            USING TRIM(Prog-Folder,TRAILING)
        END-CALL
    END-IF
    IF OS-Cygwin AND File-Name (2:1) = ':'
        MOVE '/' TO Dir-Char
    END-IF
    .

GC0909 107-Build-Box-Fields.
GC0909     MOVE ALL LD-Horiz-Line TO Horizontal-Line.
GC0909*    DELETE*    INSPECT Horizontal-Line REPLACING ALL SPACE BY LD-Horiz-Line
GC0909     .
GC0909 109-Done.
GC0909     EXIT.

200-Let-User-Set-Switches SECTION.
*****
** Show the user the current switch settings and allow them to **
** be changed. **
*****
** On Entry: The 'S-xxxx' items are set according to default **
** settings and any command line switches that were **
** provided. **
** On Exit: Any changes to the switches have been made and it **
** is time to 'rock-n-roll'. **
*****

201-Init.
    SET 88-Switch-Changes TO TRUE
    .

202-Show-And-Change-Switches.
    PERFORM UNTIL 88-No-Switch-Changes
        ACCEPT
            Switches-Screen
        END-ACCEPT
        IF COB-CRT-STATUS > 0
            EVALUATE COB-CRT-STATUS
                WHEN COB-SCR-F1
                    IF S-DEBUG = SPACE
                        MOVE Selection-Char TO S-DEBUG
                    ELSE
                        MOVE ' ' TO S-DEBUG
                    END-IF
                WHEN COB-SCR-F2
                    IF S-DLL = SPACE
                        MOVE Selection-Char TO S-DLL
                    ELSE
                        MOVE ' ' TO S-DLL
                    END-IF
                WHEN COB-SCR-F3
                    IF S-SUBROUTINE = SPACE
                        MOVE Selection-Char TO S-SUBROUTINE
                        MOVE ' ' TO S-EXECUTE
                    ELSE
                        MOVE ' ' TO S-SUBROUTINE
                    END-IF
                WHEN COB-SCR-F4
                    IF S-EXECUTE = SPACE
                        AND S-SUBROUTINE = SPACE
                            MOVE Selection-Char TO S-EXECUTE
                    ELSE
                        MOVE ' ' TO S-EXECUTE
                    END-IF
                WHEN COB-SCR-F5
                    IF S-NOTRUNC = SPACE
                        MOVE Selection-Char TO S-NOTRUNC
                    ELSE
                        MOVE ' ' TO S-NOTRUNC
                    END-IF
                WHEN COB-SCR-F6
                    IF S-TRACE = SPACE
                        MOVE Selection-Char TO S-TRACE
                        MOVE ' ' TO S-TRACEALL
                    ELSE
                        MOVE ' ' TO S-TRACE
                    END-IF
                WHEN COB-SCR-F7
                    IF S-TRACEALL = SPACE
                        MOVE Selection-Char TO S-TRACEALL
                        MOVE ' ' TO S-TRACE
                    END-IF
            END-EVALUATE
        END-IF
    END-UNTIL

```

```

ELSE
    MOVE ' ' TO S-TRACEALL
END-IF
WHEN COB-SCR-ESC
    PERFORM 900-Terminate
WHEN COB-SCR-F11
    MOVE SPACES TO S-CfgS
    MOVE Selection-Char TO S-Cfg-BS2000
WHEN COB-SCR-F12
    MOVE SPACES TO S-Cfgs
    MOVE Selection-Char TO S-Cfg-COBOL85
WHEN COB-SCR-F13
    MOVE SPACES TO S-CfgS
    MOVE Selection-Char TO S-Cfg-COBOL2002
WHEN COB-SCR-F14
    MOVE SPACES TO S-Cfgs
    MOVE Selection-Char TO S-Cfg-DEFAULT
WHEN COB-SCR-F15
    MOVE SPACES TO S-CfgS
    MOVE Selection-Char TO S-Cfg-IBM
WHEN COB-SCR-F16
    MOVE SPACES TO S-Cfgs
    MOVE Selection-Char TO S-Cfg-MF
WHEN COB-SCR-F17
    MOVE SPACES TO S-CfgS
    MOVE Selection-Char TO S-Cfg-MVS
WHEN OTHER
    MOVE 'An unsupported key was pressed'
    TO Output-Message
END-EVALUATE
ELSE
    SET 88-No-Switch-Changes TO TRUE
END-IF
END-PERFORM
.

209-Done.
EXIT.

210-Run-Compiler SECTION.
*****
** Run the compiler using the switch settings we've prepared. **
*****
** On Entry: Use the 'S-xxxx' items to build the desired **
** 'cobc' command and then submit it for processing. **
** On Exit: The program will have (hopefully) been compiled. **
** If compilation failed, a message to that effect **
** will be DISPLAYed and the program will halt. **
*****

211-Init.
MOVE SPACES TO Cmd
        Cobc-Cmd
        Output-Message

DISPLAY
    Switches-Screen
END-DISPLAY
MOVE 1 TO I
EVALUATE TRUE
    WHEN S-Cfg-BS2000 NOT = SPACES
        MOVE 'bs2000' TO Config-File
    WHEN S-Cfg-COBOL85 NOT = SPACES
        MOVE 'cobol85' TO Config-File
    WHEN S-Cfg-COBOL2002 NOT = SPACES
        MOVE 'cobol2002' TO Config-File
    WHEN S-Cfg-IBM NOT = SPACES
        MOVE 'ibm' TO Config-File
    WHEN S-Cfg-MF NOT = SPACES
        MOVE 'mf' TO Config-File
    WHEN S-Cfg-MVS NOT = SPACES
        MOVE 'mvs' TO Config-File
    WHEN OTHER
        MOVE 'default' TO Config-File
END-EVALUATE
.

212-Build-Compile-Command.
GC0909 MOVE SPACES TO CobC-Cmd
GC0909 STRING 'cobc -std='
GC0909 TRIM(Config-File,TRAILING)
GC0909 INTO Cobc-Cmd
GC0909 WITH POINTER I
GC0909 END-STRING
GC0909 IF S-SUBROUTINE NOT = ' '
GC0909     STRING '-m '
GC0909         DELIMITED SIZE INTO CobC-Cmd
GC0909         WITH POINTER I
GC0909     END-STRING
GC0909 ELSE
GC0909     STRING '-x '
GC0909         DELIMITED SIZE INTO CobC-Cmd
GC0909         WITH POINTER I
GC0909     END-STRING
GC0909 END-IF
GC0909 IF S-DEBUG NOT = ' '
GC0909     STRING '-fdebugging-line '

```

```

                DELIMITED SIZE INTO CobC-Cmd
                WITH POINTER I
            END-STRING
        END-IF
        IF S-NOTRUNC NOT = ' '
            STRING '-fnotrunc '
                DELIMITED SIZE INTO CobC-Cmd
                WITH POINTER I
            END-STRING
        END-IF
GC0809        IF S-TRACEALL NOT = ' '
            STRING '-ftraceall '
                DELIMITED SIZE INTO CobC-Cmd
                WITH POINTER I
            END-STRING
        END-IF
        IF S-TRACE NOT = ' '
            STRING '-ftrace '
                DELIMITED SIZE INTO CobC-Cmd
                WITH POINTER I
            END-STRING
        END-IF

GC0709        IF S-EXTRA > SPACES
GC0709            STRING ' '
GC0709                TRIM(S-Extra,TRAILING)
GC0709                ' '
GC0709                DELIMITED SIZE INTO Cobc-Cmd
GC0709                WITH POINTER I
GC0709            END-STRING
GC0709        END-IF
GC0909        STRING TRIM(Prog-File-Name,TRAILING)
GC0909            DELIMITED SIZE INTO Cobc-Cmd
GC0909            WITH POINTER I
GC0909        END-STRING
        .

213-Run-Compiler.
GC0609        SET 88-Output-File-Avail TO TRUE
                MOVE SPACES TO Cmd
                STRING TRIM(Cobc-Cmd,TRAILING)
                    '2>'
                    TRIM(Cobc-Output-File,TRAILING)
                    DELIMITED SIZE
                    INTO Cmd
                END-STRING
                CALL 'SYSTEM'
                    USING TRIM(Cmd,TRAILING)
                END-CALL
GC0909        IF RETURN-CODE = 0
GC0909            SET 88-Compile-OK TO TRUE
GC0909        ELSE
GC0909            SET 88-Compile-Failed TO TRUE
GC0909        END-IF
GC0909        IF 88-Compile-OK
GC0909            OPEN INPUT Cobc-Output
GC0909            READ Cobc-Output
GC0909                AT END
GC0909                CONTINUE
GC0909                NOT AT END
GC0909                SET 88-Compile-OK-Warn TO TRUE
GC0909            END-READ
GC0909            CLOSE Cobc-Output
GC0909        END-IF
GC0909        IF 88-Compile-OK
GC0909            MOVE 'Compilation was Successful' TO Output-Message
GC0909            DISPLAY
GC0909                Switches-Screen
GC0909            END-DISPLAY
GC0909            CALL 'CSLEEP'
GC0909                USING 2
GC0909            END-CALL
GC0909            MOVE SPACES TO Output-Message
GC0609            SET 88-Complete TO TRUE
        ELSE
GC0909            DISPLAY
GC0909                Blank-Screen
GC0909            END-DISPLAY
GC0909            IF 88-Compile-OK-Warn
GC0909                DISPLAY 'Compilation was successful, but ' &
GC0909                    'warnings were generated:'
GC0909                    AT LINE 24 COLUMN 1
GC0909                    WITH SCROLL UP 1 LINE
GC0909                END-DISPLAY
GC0909            ELSE
GC0909                DISPLAY 'Compilation Failed:'
GC0909                    AT LINE 24 COLUMN 1
GC0909                    WITH SCROLL UP 1 LINE
GC0909                END-DISPLAY
GC0909            END-IF
GC0609            SET 88-Compile-Failed TO TRUE
GC0609            SET 88-Complete TO TRUE
GC0909            DISPLAY
GC0909                AT LINE 24 COLUMN 1
GC0909                WITH SCROLL UP 1 LINE
GC0909            END-DISPLAY
GC0909            OPEN INPUT Cobc-Output

```

```

GC0909      PERFORM FOREVER
GC0909          READ Cobc-Output AT END
GC0909          EXIT PERFORM
GC0909      END-READ
GC0909          DISPLAY TRIM(Cobc-Output-Rec,TRAILING)
GC0909          AT LINE 24 COLUMN 1
GC0909          WITH SCROLL UP 1 LINE
GC0909      END-DISPLAY
GC0909      END-PERFORM
GC0909          CLOSE Cobc-Output
GC0909          DISPLAY ' '
GC0909          AT LINE 24 COLUMN 1
GC0909          WITH SCROLL UP 2 LINES
GC0909      END-DISPLAY
GC0909          DISPLAY 'Press ENTER to close:'
GC0909          AT LINE 24 COLUMN 1
GC0909          WITH SCROLL UP 1 LINE
GC0909      END-DISPLAY
GC0909          ACCEPT Dummy
GC0909          FROM CONSOLE
GC0909      END-ACCEPT
GC0909          DISPLAY
GC0909          Blank-Screen
GC0909      END-DISPLAY
END-IF
.

219-Done.
IF 88-Compile-Failed
    PERFORM 900-Terminate
END-IF
.

230-Run-Program SECTION.
*****
** Run the compiled program **
*****
** On Entry: Use the 'S-xxxx' items to build the desired **
** execution command and then submit it for proces- **
** sing. **
** On Exit: The program will have (hopefully) been executed. **
*****

232-Build-Command.
GC0909      MOVE SPACES TO Cmd
GC0909      MOVE 1 TO I
GC0909      IF S-SUBROUTINE NOT = ' '
GC0909      OR S-DLL NOT = ' '
GC0909          STRING 'cobcrun ' DELIMITED SIZE
GC0909          INTO Cmd
GC0909          WITH POINTER I
GC0909      END-STRING
GC0909      END-IF
GC0909      IF Prog-Folder NOT = SPACES
GC0909          IF OS-Cygwin AND Prog-Folder (2:1) = ':'
GC0909              STRING '/cygdrive/'
GC0909              INTO Cmd
GC0909              WITH POINTER I
GC0909          END-STRING
GC0909          STRING LOWER-CASE(Prog-Folder (1:1))
GC0909          INTO Cmd
GC0909          WITH POINTER I
GC0909          END-STRING
GC0909          PERFORM VARYING J FROM 3 BY 1
GC0909          UNTIL J > LENGTH(TRIM(Prog-Folder))
GC0909          IF Prog-Folder (J:1) = '\'
GC0909              STRING '/'
GC0909              INTO Cmd
GC0909              WITH POINTER I
GC0909          END-STRING
GC0909          ELSE
GC0909              STRING Prog-Folder (J:1)
GC0909              INTO Cmd
GC0909              WITH POINTER I
GC0909          END-STRING
GC0909          END-IF
GC0909      END-PERFORM
GC0909      ELSE
GC0909          STRING TRIM(Prog-Folder,TRAILING)
GC0909          INTO Cmd
GC0909          WITH POINTER I
GC0909      END-STRING
GC0909      END-IF
GC0909      STRING Dir-Char
GC0909      INTO Cmd
GC0909      WITH POINTER I
GC0909      END-STRING
GC0909      ELSE
GC0909          IF OS-Cygwin OR OS-UNIX
GC0909              STRING './'
GC0909              INTO Cmd
GC0909              WITH POINTER I
GC0909          END-STRING
GC0909          END-IF
GC0909      END-IF
GC0909      STRING TRIM(Prog-Name,TRAILING)
GC0909      INTO Cmd

```

```

GC0909     WITH POINTER I
GC0909     END-STRING
GC0909     IF S-SUBROUTINE = ' '
GC0909     AND S-DLL NOT = ' '
GC0909     STRING '.exe' DELIMITED SIZE
           INTO Cmd
           WITH POINTER I
           END-STRING
           END-IF
GC0809     IF S-ARGS NOT = SPACES
           STRING ' ' TRIM(S-ARGS,TRAILING)
           INTO Cmd
           WITH POINTER I
           END-STRING
           END-IF
           IF OS-Unknown OR OS-Windows
           STRING '&&pause'
           INTO Cmd
           WITH POINTER I
           END-STRING
           ELSE
           STRING ';echo "Press ENTER to cclose...";read'
           INTO Cmd
           WITH POINTER I
           END-STRING
           END-IF
           .

233-Run-Program.
GC0909     DISPLAY
GC0909     Blank-Screen
GC0909     END-DISPLAY

           CALL 'SYSTEM'
           USING TRIM(Cmd,TRAILING)
           END-CALL
           PERFORM 900-Terminate
           .

239-Done.
EXIT.

900-Terminate SECTION.
*****
** Display a message and halt the program          **
*****
** On Entry: There are no prerequisite conditions. **
** On Exit: There is no exit from this routine    **
*****

901-Display-Message.
GC0909     IF Output-Message > SPACES
GC0909     DISPLAY
GC0909     Switches-Screen
GC0909     END-DISPLAY
GC0909     CALL 'CSLEEP'
GC0909     USING 4
GC0909     END-CALL
GC0909     END-IF
GC0909     DISPLAY
GC0909     Blank-Screen
GC0909     END-DISPLAY
           .

909-Done.
GOBACK
           .

END PROGRAM OCic.

IDENTIFICATION DIVISION.
PROGRAM-ID.  getostype.
*****
** This subprogram determine the OS type the program is run- **
** ning under, passing that result back in RETURN-CODE as fol- **
** lows:                                                       **
** 0:  Cannot be determined                                   **
** 1:  Native windows or windows/MinGW                       **
** 2:  Cygwin                                                 **
** 3:  UNIX/Linux/MacOS                                     **
*****
** On Entry: There are no arguments to this subroutine      **
** On Exit: RETURN-CODE will be set as per above           **
*****
**  DATE   CHANGE DESCRIPTION                               **
** =====|===== **
** GC0909 Initial coding.                                  **
*****
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
FUNCTION ALL INTRINSIC.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Env-Path                                         PIC X(1024).

```



```

01 Tally                                     USAGE BINARY-LONG.
PROCEDURE DIVISION.
000-Main SECTION.
010-Get-TEMP-Var.
    MOVE SPACES TO Env-Path
    ACCEPT Env-Path
        FROM ENVIRONMENT "PATH"
        ON EXCEPTION
            MOVE 0 TO RETURN-CODE
        GOBACK
    END-ACCEPT
    IF Env-Path = SPACES
        MOVE 0 TO RETURN-CODE
    ELSE
        MOVE 0 TO Tally
        INSPECT Env-Path
            TALLYING Tally FOR ALL ";";
        IF Tally = 0 *> Must be some form of UNIX
            MOVE 0 TO Tally
            INSPECT Env-Path
                TALLYING TALLY FOR ALL "/cygdrive/"
            IF Tally = 0 *> UNIX/MacOS
                MOVE 3 TO RETURN-CODE
            ELSE *> Cygwin
                MOVE 2 TO RETURN-CODE
            END-IF
        ELSE *> Assume windows[/mingw]
            MOVE 1 TO RETURN-CODE
        END-IF
    END-IF
    GOBACK
END PROGRAM getostype.

IDENTIFICATION DIVISION.
PROGRAM-ID. checksource.
*****
** This subprogram will scan a line of source code it is given **
** looking for "LINKAGE SECTION" or "IDENTIFICATION DIVISION". **
** *****NOTE***** *****NOTE***** *****NOTE***** *****NOTE** **
** **
** These two strings must be found IN THEIR ENTIRETY within **
** the 1st 80 columns of program source records, and cannot **
** follow either a "*>" sequence OR a "g" in col 7. **
*****
** On Entry: Argument-1 contains the first 80 characters of **
** the program source **
** On Exit: Argument-2 will be set to one of the following: **
** "L" : LINKAGE SECTION found **
** "I" : IDENTIFICATION DIVISION found **
** " " : Neither of the above found **
*****
** DATE CHANGE DESCRIPTION **
** ===== **
** GC0809 Initial coding. **
*****
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
    FUNCTION ALL INTRINSIC.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 Compressed-Src.
    05 CS-Char                                     OCCURS 80 TIMES PIC X(1).

01 Flags.
    05 F-Found-SPACE                             PIC X(1).
    88 88-Skipping-SPACE                         VALUE 'Y'.
    88 88-Not-Skipping-SPACE                     VALUE 'N'.

01 I                                             USAGE BINARY-CHAR.

01 J                                             USAGE BINARY-CHAR.
LINKAGE SECTION.
01 Argument-1.
    02 A1-Char                                     OCCURS 80 TIMES PIC X(1).

01 Argument-2                                     PIC X(1).
    88 88-A2-LINKAGE-SECTION                       VALUE 'L'.
    88 88-A2-IDENTIFICATION-DIVISION              VALUE 'I'.
    88 88-A2-Nothing-Special                       VALUE ' '.
PROCEDURE DIVISION USING Argument-1, Argument-2.
000-Main SECTION.

010-Initialize.
    SET 88-A2-Nothing-Special TO TRUE
    IF A1-Char (7) = '*'
        GOBACK
    END-IF
    .

020-Compress-Multiple-SPACES.
    SET 88-Not-Skipping-SPACE TO TRUE
    MOVE 0 TO J
    MOVE SPACES TO Compressed-Src
    PERFORM VARYING I FROM 1 BY 1

```

```

        UNTIL I > 80
        IF A1-Char (I) = SPACE
            IF 88-Not-Skipping-SPACE
                ADD 1 TO J
                MOVE UPPER-CASE(A1-char (I)) TO CS-Char (J)
                SET 88-Skipping-SPACE TO TRUE
            END-IF
        ELSE
            SET 88-Not-Skipping-SPACE TO TRUE
            ADD 1 TO J
            MOVE A1-Char (I) TO CS-Char (J)
        END-IF
    END-PERFORM
.

030-Scan-Compressed-Src.
    PERFORM VARYING I FROM 1 BY 1
        UNTIL I > 66
        EVALUATE TRUE
            WHEN CS-Char (I) = '*'
                IF Compressed-Src (I : 2) = '*>'
                    GOBACK
                END-IF
            WHEN (CS-Char (I) = 'L') AND (I < 66)
                IF Compressed-Src (I : 15) = 'LINKAGE SECTION'
                    SET 88-A2-LINKAGE-SECTION TO TRUE
                    GOBACK
                END-IF
            WHEN (CS-Char (I) = 'I') AND (I < 58)
                IF Compressed-Src (I : 23) = 'IDENTIFICATION ' &
                    'DIVISION'
                    SET 88-A2-IDENTIFICATION-DIVISION TO TRUE
                    GOBACK
                END-IF
        END-EVALUATE
    END-PERFORM
.

099-Never-Found-Either-One.
    GOBACK
.

END PROGRAM checksource.

```

## 8.4. WINSYSTEM - Execute Windows Shell Commands (For Cygwin)

```

IDENTIFICATION DIVISION.
PROGRAM-ID.      WINSYSTEM.
*****
** This is an OpenCOBOL subroutine that will submit a windows **
** command to the windows "cmd.exe" command shell for proces- **
** sing. This is needed if your OpenCOBOL version was built **
** using Cygwin because the "SYSTEM" built-in subroutine will **
** submit commands to the Cygwin shell rather than the windows **
** shell. **
** **
** CALL "WINSYSTEM" USING <cmd> **
** **
** >>> Note that the subroutine name MUST be specified in <<< **
** >>> upper-case **
*****
** DATE CHANGE DESCRIPTION **
** ===== **
** GC0909 Initial coding **
*****

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
    FUNCTION ALL INTRINSIC.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 Cmd-Len          USAGE BINARY-LONG.
01 Shell-Cmd       PIC X(1024).
LINKAGE SECTION.
01 Cmd             PIC X(1) ANY LENGTH.
PROCEDURE DIVISION USING Cmd.
000-Main.
    CALL "C$PARAMSIZE" USING 1.
    MOVE RETURN-CODE TO Cmd-Len.

```

```
MOVE SPACES TO Shell-Cmd.  
STRING "cmd.exe /C "  
      Cmd(1:Cmd-Len)  
      INTO Shell-Cmd  
END-STRING  
CALL "SYSTEM"  
  USING TRIM(Shell-Cmd)  
END-CALL  
.  
099-Wave-Bye-Bye.  
GOBACK  
.
```

## 9. Glossary of Terms

There are many terms that are used throughout this document (as well as throughout ANY document dealing with the COBOL language) that are used to make discussions of syntax and semantics more concise. The following is a list of such terms and their definitions.

<b>Alphanumeric literal</b>	A string of characters enclosed within a pair of quotation marks (") or apostrophes ('). See section <a href="#">1.8</a> .
<b>Collating sequence</b>	The sequence in which the characters that are acceptable to a computer are ordered for purposes of all types of sorting, merging, comparing, and processing. OpenCOBOL programs may utilize standard character-set collating sequences (such as that defined by the ASCII or EBCDIC charactersets) or programmer-defined custom sequences as specified in the OBJECT-COMPUTER paragraph (section <a href="#">4.1.2</a> ) and defined in the SPECIAL-NAMES paragraph (section <a href="#">4.1.4</a> ).
<b>Compilation unit</b>	A single source file being compiled by the OpenCOBOL compiler. A compilation unit may contain one or more <a href="#">program units</a> .
<b>Division</b>	COBOL programs are broken into four major areas, called DIVISIONS. Divisions are used to collect program components oriented toward specific similar goals together in a single place. The COBOL divisions are: <ul style="list-style-type: none"> <li>• IDENTIFICATION DIVISION – names the program and, optionally, if it is a subprogram, defines it's high-level data initialization policy and/or global availability to other programs compiled in the same compilation unit.</li> <li>• ENVIRONMENT DIVISION – defines characteristics of the environment in which the program will be executed, such as files the program will be reading and/or writing, run-time switches that may be used to pass information into the program from the operating system environment and any special options that may be needed in order for the program to properly compile; typically, those special options are used to enable COBOL programs created using some other version of COBOL to be compiled and executed under a different version.</li> <li>• DATA DIVISION – provides detailed descriptions of the files, data and data structures the program will be working with.</li> <li>• PROCEDURE DIVISION – contains the actual executable program code.</li> </ul>
<b>Dynamically-loadable library</b>	The OpenCOBOL compiler can create dynamically-loadable library files when compiling <a href="#">subprograms</a> as their own separate <a href="#">compilation units</a> . On UNIX systems, these will be ".so" files while on Windows systems these will be DLLs. <a href="#">Main programs</a> can be created in this manner also. The "-m" compiler switch is used to create dynamically-loadable libraries.
<b>Dynamically-loadable module</b>	A synonym for <a href="#">Dynamically-loadable library</a> .
<b>Elementary Item</b>	A data item described as not being further logically subdivided.
<b>Entry-point</b>	A spot in the PROCEDURE DIVISION where a program may begin execution when it is executed from the operating system or CALLED by another program. Every program has at least one entry-point – known as the <i>primary entry-point</i> – which corresponds to the first executable <a href="#">statement</a> in the PROCEDURE DIVISION following the DECLARATIVES area, if any. Additional entry-points may be defined via the ENTRY statement (see section <a href="#">6.17</a> ).
<b>Entry-point name</b>	Every <i>entry-point</i> has a name. That name must be unique for all <a href="#">program units</a> that comprise an executable program. Entry-point names are defined using a <a href="#">subroutine's</a> PROGRAM-ID clause (see section <a href="#">3</a> ) or via ENTRY statements coded in the subroutine's PROCEDURE DIVISION (see section <a href="#">6.17</a> ).

<b>Executable file</b>	The OpenCOBOL compiler can create operating-system appropriate files that may be executed directly from the operating system environment. On Windows systems, these will be “.exe” files whereas on UNIX systems they will have no specific extensions. The “-x” compiler switch is used to create executable files. Only <a href="#">main programs</a> should be compiled in this manner.
<b>Figurative constants</b>	OpenCOBOL, like other COBOL implementations, supports a number of reserved words that may be used to represent a specific <a href="#">literal</a> value. These are known as figurative constants. See section <a href="#">1.9</a> .
<b>Group item</b>	A group item is an <a href="#">identifier</a> that is broken down into sub-items. For example, a MAILING-ADDRESS might be broken down into STREET-ADDRESS, APARTMENT-NUMBER, CITY, STATE and ZIP-CODE components.
<b>Identifiers</b>	These are data items a COBOL program will be working with. The vast majority of identifiers are defined by the user (programmer) while a few are pre-defined by the OpenCOBOL compiler. Identifiers pre-defined by the compiler are referred to as <a href="#">registers</a> . Other programming languages generally refer to identifiers as “variables”.
<b>Imperative statement</b>	A sequence of one or more non-conditional OpenCOBOL <a href="#">statements</a> or conditional OpenCOBOL <a href="#">statements</a> properly terminated with the correct “END-xxxx” trailer.
<b>Level number</b>	A user-defined word expressed as a 1- or 2-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 1 through 49 indicate the position of a data item in the hierarchical structure of a logical <a href="#">record</a> . Level numbers in the range 1 through 9 can be written either as a single digit or as a zero followed by the significant digit.  Level numbers 66, 77, 78 and 88 identify special properties of a data description entry.  See sections <a href="#">5.3</a> , <a href="#">5.4</a> , <a href="#">5.5</a> and <a href="#">0</a> .
<b>Literal</b>	A <a href="#">numeric literal</a> or an <a href="#">alphanumeric literal</a> .
<b>Main program</b>	An OpenCOBOL program that is to be executed directly from an operating system or shell event. Main programs are not executed from other programs unless such execution is accomplished via the CALL “SYSTEM” facility.
<b>Numeric literal</b>	A numeric constant. See section <a href="#">1.8</a> .
<b>Primary Entry-point</b>	See <a href="#">entry-point</a> .
<b>Procedure</b>	All executable code <a href="#">statements</a> within a single PROCEDURE DIVISION paragraph or SECTION.
<b>Procedure name</b>	A programmer-defined SECTION or paragraph name in the PROCEDURE DIVISION assigned to a <a href="#">procedure</a> . Procedure names serve as a means by which a <a href="#">statement</a> may refer to the <a href="#">statements</a> that follow the procedure name.
<b>Program unit</b>	An OpenCOBOL main program or subprogram. Subprogram program units may be nested inside of other program units and a main program unit may be followed by any number of subprogram program units in the same compilation unit.
<b>Qualification</b>	The process of establishing a unique reference to a data item whose name is duplicated in a program. This takes the form of using the duplicated data name and the name of any of its parent data items, connected by “OF” or “IN” such that the combination of those two data names is unique within the program.
<b>Record</b>	The most-inclusive, highest level, data item. The <a href="#">level number</a> for a record is 01. A record can be either an <a href="#">elementary item</a> or a <a href="#">group item</a> .
<b>Registers</b>	Special data items that are automatically defined for your use by the OpenCOBOL compiler. See section <a href="#">6.1.8</a> .

<b>Reserved word</b>	A COBOL word specified in the list of words that can be used in a COBOL source program, but that must not appear in the program as user-defined words or system names.
<b>Sentence</b>	Any number of COBOL <a href="#">statements</a> , followed by a period.
<b>Statement</b>	A single COBOL instruction. Every statement starts with a <a href="#">verb</a> which defines the overall action the statement will take. Any additional syntax following the <a href="#">verb</a> refines the actions that will be taken.
<b>Subprogram or subroutine</b>	These two interchangeable terms refer to OpenCOBOL programs that are executed by another program. Typically this is done via the CALL <a href="#">statement</a> from another OpenCOBOL program, although OpenCOBOL may execute subprograms written in other languages and other language programs may execute OpenCOBOL programs.
<b>User-defined names</b>	Either the name of an <a href="#">identifier</a> or a <a href="#">procedure</a> in the program. OpenCOBOL limits user-defined names to a maximum of 31 characters taken from the set of numeric digits, upper- and lower-case letters, hyphens and underscores. A user-defined name may neither begin nor end with a hyphen or underscore. User-defined names used as file names may additionally not begin with a digit although - unlike many other programming languages - user-defined names used as <a href="#">identifiers</a> or <a href="#">procedure</a> names may.
<b>Verb</b>	A single COBOL <a href="#">reserved-word</a> which defines an action a COBOL program will take at execution time. Every COBOL <a href="#">statement</a> begins with a verb. Some verbs perform relatively simple actions (MOVE, STOP, SET, etc.) while others can perform extremely complex actions (SEARCH, SORT, MERGE, STRING, UNSTRING, etc.).



## Index

- \*
  - \* In Column 7, 1-11
- >
  - >>D, 1-11
  - >>SOURCE FORMAT, 1-11
- A**
  - ACCEPT, 5-18, 6-24
    - Command-Line Arguments, 6-26
    - CONSOLE, 6-26
    - Date/Time, 6-29
    - Environment, 6-27
    - Screen Data, 6-28
    - Screen Size, 6-29
  - ACCESS MODE, 4-8, 4-9
    - DYNAMIC, 6-42, 6-73, 6-74, 6-78, 6-89
    - RANDOM, 6-42, 6-73, 6-74, 6-78
    - SEQUENTIAL, 6-42, 6-78, 6-89
  - ADD
    - CORRESPONDING, 6-32
    - GIVING, 6-32
    - TO, 6-31
  - ADDRESS OF
    - FREE, 6-54
    - SET, 6-84
  - AFTER, 6-72
    - INSPECT, 6-61, 6-62
    - PERFORM VARYING, 6-71
    - PERFORM WITH TEST, 6-71
    - WRITE ADVANCING, 6-101
  - ALL
    - INSPECT, 6-61, 6-62
    - VALUE, 5-11
  - ALL PROCEDURES, 6-25
  - ALLOCATE, 5-11, 6-33
  - ALPHABET, 4-4
  - ALPHABETIC, 6-6
  - ALPHABETIC-LOWER, 6-6
  - ALPHABETIC-UPPER, 6-6
  - Alphanumeric Literal, 1-13
  - ALTER, 6-34
  - ALTERNATE RECORD KEY, 4-9, 6-89
  - ALTERNATE RECORD KEY fields, 6-74
  - ANY, 6-50
  - ANY LENGTH, 5-11
  - ARGUMENT-NUMBER, 6-26
  - ARGUMENT-VALUE, 6-26, 6-27
  - Arithmetic Expressions, 6-2
  - ASCENDING KEY
    - SORT, 6-87, 6-88
    - Table, 5-11, 6-81
  - ASSIGN, 4-6
  - AT
    - ACCEPT, 6-28
    - DISPLAY, 6-44
    - END (READ), 6-73
    - END-OF-PAGE, 6-102
    - AUTO, 5-18
- B**
  - BACKGROUND-COLOR, 5-18, 5-19
  - BACK-TAB, 5-18
  - BASED, 5-10, 5-11, 6-84
  - BEEP, 5-17
  - BEFORE
    - INSPECT, 6-61, 6-62
    - PERFORM WITH TEST, 6-71
    - WRITE ADVANCING, 6-101
  - BELL, 5-17
  - Big-Endian, 5-13
  - BLANK, 5-18
  - BLANK WHEN ZERO, 5-11
  - BLINK, 5-19
  - BLOCK CONTAINS, 5-3
  - BY
    - CONTENT, 7-7
    - PERFORM VARYING, 6-71, 6-72
    - REFERENCE, 6-24, 6-36, 7-6, 7-8
    - VALUE, 6-24, 7-8
  - BYTE-LENGTH, 5-16
- C**
  - C\$CHDIR, 7-15
  - C\$COPY, 7-15
  - C\$DELETE, 7-15
  - C\$FILEINFO, 7-16
  - C\$JUSTIFY, 7-16
  - C\$MAKEDIR, 7-16
  - C\$NARG, 7-16, 7-27
  - C\$PARAMSIZE, 7-17
  - C\$SLEEP, 7-17
  - C\$TOLOWER, 7-17
  - C\$TOUPPER, 7-17
  - CALL, 6-6, 6-35, 7-3, 9-1
  - CANCEL, 5-1, 6-37
  - CBL\_AND, 7-17
  - CBL\_CHANGE\_DIR, 7-18
  - CBL\_CHECK\_FILE\_EXIST, 7-18
  - CBL\_CLOSE\_FILE, 7-18
  - CBL\_COPY\_FILE, 7-18
  - CBL\_CREATE\_DIR, 7-19
  - CBL\_CREATE\_FILE, 7-19
  - CBL\_DELETE\_DIR, 7-19
  - CBL\_DELETE\_FILE, 7-19
  - CBL\_EQ, 7-22
  - CBL\_ERROR\_PROC, 7-19
  - CBL\_EXIT\_PROC, 7-21
  - CBL\_FLUSH\_FILE, 7-22
  - CBL\_IMP, 7-22
  - CBL\_NIMP, 7-23



CBL\_NOR, 7-23  
 CBL\_NOT, 7-24  
 CBL\_OC\_NANOSLEEP, 7-24  
 CBL\_OPEN\_FILE, 7-19, 7-24  
 CBL\_OR, 7-24  
 CBL\_READ\_FILE, 7-24, 7-25  
 CBL\_RENAME\_FILE, 7-25  
 CBL\_TOUPPER, 7-25  
 CBL\_WRITE\_FILE, 7-19, 7-24, 7-26  
 CBL\_XOR, 7-26  
 CHAIN, 6-24  
 CHAINING, 6-24  
 CHARACTERS, 6-62  
 CLASS, 4-4  
 CLOSE, 6-23, 6-38, 6-70  
 cobcrun, 7-12  
 CODE-SET, 5-2  
 COL, 5-18  
 Collating Sequence, 9-1  
 COLLATING SEQUENCE, 4-2, 4-6  
 COLUMN, 5-18  
 Column 7  
   "\"", 1-11  
   "D", 1-11  
 COLUMNS, 6-29  
 Combined Conditions, 6-8  
 COMMAND-LINE, 6-26  
 comment, 1-11  
 COMMIT, 6-23, 6-39, 6-78  
 COMMON-STORAGE SECTION (Alternative To), 5-1  
 Compilation Unit, 2-1, 5-3, 5-10, 9-1  
 Compiler Switches  
   All Switches, 7-1  
   -conf, 7-10  
   -fdebugging-line, 1-11  
   -ffunctions-all, 4-2  
   -fixed, 1-11  
   -fnotrunc, 7-29  
   -free, 1-11  
   -g, 6-14, 6-15  
   -m, 7-2, 7-3, 7-12, 9-1  
   -m, 3-1  
   -S, 7-3  
   -Wobsolete, 3-1  
   -x, 7-3, 7-12, 9-2  
   -x, 3-1  
 COMPUTE, 6-40  
 Condition Names, 6-5  
 Conditional Expressions, 6-2, 6-5  
 Conditions  
   Combined, 6-8  
   Level-88 Condition Names, 6-5  
   Negated, 6-9  
   Relation, 6-8  
   Switch Status, 6-7  
 Configuration Files, 7-10  
 CONFIGURATION SECTION, 4-1  
 CONSOLE, 6-43  
 CONSOLE IS CRT, 4-3  
 CONSTANT, 5-16  
 Constant Descriptions, 5-16

CONTINUE, 6-41  
 CONVERTING, 6-61, 6-62, 6-97  
 CORRESPONDING, 6-32  
 COUNT, 6-100  
 CRT, 4-5, 6-43  
 CURRENCY SIGN, 4-5  
 CURSOR IS, 4-5

## D

D In Column 7, 1-11  
 DATA DIVISION, 1-7, 1-10  
 DATA RECORD, 5-2  
 DATE, 6-29  
 DATE YYYYMMDD, 6-29  
 DAY, 6-29  
 DAY YYYYDDD, 6-29  
 DAY-OF-WEEK, 6-29  
 DEBUGGING MODE, 4-1  
 DECIMAL POINT IS COMMA, 4-5  
 DECLARATIVES, 6-25, 6-42, 6-70, 9-1  
 DEFAULT, 6-59  
 DELETE, 6-42, 6-70  
 DELIMITED BY  
   STRING, 6-92  
   UNSTRING, 6-99  
 DELIMITED BY SIZE, 6-92  
 DELIMITER, 6-100  
 DESCENDING KEY  
   SORT, 6-87, 6-88  
   Table, 5-11, 6-81  
 DISK, 5-3  
 DISPLAY, 5-18  
   Command-Line Arguments, 6-43  
 CONSOLE, 6-43  
   Environment, 6-43  
   Screen Data, 6-44  
 DIVIDE  
   BY/GIVING, 6-47  
   BY/REMAINDER, 6-48  
   INTO, 6-46  
   INTO/GIVING, 6-46  
   INTO/REMAINDER, 6-47  
 DIVISION, 9-1  
 DYNAMIC, 4-8, 4-9  
 Dynamically-Loadable Library, 3-1, 9-1

## E

Elementary Item, 9-1  
 ELSE, 6-58  
 END PROGRAM, 2-1, 2-2  
 END-IF, 6-58  
 ENTRY, 6-49, 6-83, 7-3, 9-1  
 Entry Point, 9-1  
 ENVIRONMENT, 6-27  
 ENVIRONMENT DIVISION, 1-7, 4-1  
 Environment Variables  
   COB\_CC, 7-9  
   COB\_CFLAGS, 7-9  
   COB\_CONFIG\_DIR, 7-9

COB\_CONFIG\_PATH, 7-10  
 COB\_COPY\_DIR, 7-9, 7-10  
 COB\_LDADD, 7-9  
 COB\_LDFLAGS, 7-9  
 COB\_LIBRARY\_PATH, 7-13  
 COB\_LIBS, 7-9  
 COB\_PRE\_LOAD, 7-13  
 COB\_SCREEN\_ESC, 4-5, 7-13  
 COB\_SCREEN\_EXCEPTIONS, 4-5, 7-13  
 COB\_SORT\_MEMORY, 7-13  
 COB\_SWITCH\_n, 4-4  
 COB\_SWITCH\_n, 7-13  
 COB\_SYNC, 7-14  
 dd\_literal-1, 4-7  
 DD\_literal-1, 4-6  
 LD\_LIBRARY\_PATH, 7-9  
 literal-1, 4-7  
 PATH, 7-14  
 TEMP, 7-14  
 TMP, 7-10, 7-14  
 TMPDIR, 7-10, 7-14  
 ENVIRONMENT-NAME, 6-27  
 ENVIRONMENT-VALUE, 6-27  
 EOL, 5-18  
 EOS, 5-18  
 ERASE, 5-18  
 Error Procedure (user-defined), 6-70, 7-19  
 EVALUATE, 6-50  
 EVENT STATUS, 4-5  
 EXCEPTION  
   ACCEPT, 6-30  
   CALL, 6-35  
   DISPLAY, 6-45  
 Executable File, 3-1, 9-2  
 EXIT, 6-52  
   PARAGRAPH, 6-52, 6-65  
   PERFORM, 6-52  
   PERFORM CYCLE, 6-52  
   PROGRAM, 6-65, 6-86, 6-88  
   SECTION, 6-52, 6-65  
   Simple, 6-52  
 Exit Procedure (user-defined), 7-21  
 Expressions  
   Arithmetic, 6-2  
   Conditional, 6-2, 6-5  
 EXTEND, 6-70, 6-101  
 EXTERNAL  
   Data Item Description, 5-1, 5-10  
   FD, 5-3

**F**

FD, 6-101  
 -fdebugging-line, 4-1  
 Figurative Constant, 1-14, 9-2  
 File Description, 6-101  
 FILE SECTION, 1-7  
 FILE STATUS, 4-7  
 FILE-CONTROL, 1-7, 4-6  
 FILLER, 5-4  
 FIRST

INSPECT, 6-61, 6-62  
 -fixed, 1-11  
 fixed format, 1-11  
 FOREGROUND-COLOR, 5-18, 5-19  
 FOREVER, 6-71, 6-72  
 -free, 1-11  
 FREE, 6-54  
 free format, 1-11  
 FROM  
   PERFORM VARYING, 6-71  
   REWRITE, 6-78  
   Screen Item Description, 5-19  
   Screen Item Description, 5-19  
   WRITE, 6-101  
 FULL, 5-18  
 FUNCTION-ID, 2-2

**G**

GENERATE, 6-55  
 GIVING  
   CALL, 6-35  
   MERGE, 6-65  
   SORT, 6-87  
   STOP, 6-91  
 GLOBAL, 6-25  
   data item, 5-10  
   Data Item Description, 5-1  
   FD, 5-3  
 GO TO, 6-65, 6-71, 6-86, 6-88  
   DEPENDING ON, 6-57  
   Simple, 6-57  
 GOBACK, 6-53, 6-56, 6-65, 6-86, 6-88, 7-21  
 Group Item, 9-2

**H**

HIGHLIGHT, 5-19

**I**

IDENTIFICATION DIVISION, 3-1  
 Identifier, 9-2  
 IF, 6-58  
 IGNORING LOCK, 6-23  
 Imperative Statement, 9-2  
 INDEXED BY, 5-11, 6-81, 6-84  
 INITIAL, 6-35  
 INITIALIZE, 6-33  
   Verb, 6-59  
 INITIATE, 6-60  
 INPUT, 6-70  
 INPUT PROCEDURE, 6-76, 6-86  
 INPUT-OUTPUT SECTION, 1-7, 4-5  
 INSPECT, 6-61, 6-97  
 Intrinsic Functions (Supported)  
   ABS, 6-11  
   CHAR, 6-12  
   COMBINED-DATETIME, 6-13  
   CONCATENATE, 6-13  
   CURRENT-DATE, 6-13

DATE-OF-INTEGERS, 6-13  
 DATE-TO-YYYYMMDD, 6-13  
 DAY-OF-INTEGERS, 6-14  
 DAY-TO-YYYYDDD, 6-14  
 E, 6-14  
 EXCEPTION-FILE, 6-14  
 EXCEPTION-LOCATION, 6-14  
 EXCEPTION-STATEMENT, 6-14  
 EXCEPTION-STATUS, 6-15  
 EXP, 6-15  
 EXP10, 6-15  
 FACTORIAL, 6-15  
 FRACTIONAL-PART, 6-15  
 INTEGER, 6-15  
 INTEGER-OF-DATE, 6-15  
 INTEGER-OF-DAY, 6-15  
 INTEGER-PART, 6-16  
 LENGTH, 6-16  
 LOCALE-DATE, 6-16  
 LOCALE-TIME, 6-16  
 LOCALE-TIME-FROM-SECS, 6-16  
 LOG, 6-16  
 LOG10, 6-16  
 LOWER-CASE, 6-16, 6-21  
 MAX, 6-17, 6-20, 6-21  
 MIDRANGE, 6-17  
 MOD, 6-17  
 NUMVAL, 1-13, 6-17  
 NUMVAL-C, 1-13, 6-17  
 ORD, 6-18  
 ORD-MAX, 6-18  
 PI, 6-18  
 PRESENT-VALUE, 6-18  
 RANDOM, 6-18  
 RANGE, 6-19  
 REM, 6-19  
 REVERSE, 6-19  
 SECONDS-FROM-FORMATTED-TIME, 6-19  
 SECONDS-PAST-MIDNIGHT, 6-19  
 SIGN, 6-19  
 SIN, 6-19  
 SQRT, 6-19  
 STORED-CHAR-LENGTH, 6-20  
 SUBSTITUTE, 6-20  
 SUM, 6-20  
 TAN, 6-20  
 TEST-DATE-YYYYMMDD, 6-20  
 TEST-DAY-YYYYDDD, 6-20  
 TRIM, 6-20  
 WHEN-COMPILED, 6-21  
 YEAR-TO-YYYY, 6-21  
 Intrinsic Functions (Unsupported)  
 BOOLEAN-OF-INTEGERS, 6-11  
 CHAR-NATIONAL, 6-11  
 DISPLAY-OF, 6-11  
 EXCEPTION-FILE-N, 6-11  
 EXCEPTION-LOCATION-N, 6-11  
 HIGHEST-ALGEBRAIC, 6-11  
 INTEGER-OF-BOOLEAN, 6-11  
 LOCALE-COMPARE, 6-11  
 LOWEST-ALGEBRAIC, 6-11

NATIONAL-OF, 6-11  
 NUMVAL-F, 6-11  
 STANDARD-COMPARE, 6-11  
 TEST-NUMVAL, 6-11  
 TEST-NUMVAL-C, 6-11  
 TEST-NUMVAL-F, 6-11  
 INVALID KEY  
 DELETE, 6-42  
 REWRITE, 6-78  
 START, 6-89  
 WRITE, 6-102  
 I-O, 6-70, 6-101  
 I-O-CONTROL, 4-10

**J**

JUSTIFIED RIGHT, 5-11

**K**

KEY (START), 6-89

**L**

LABEL RECORD, 5-2  
 LEADING  
   INSPECT, 6-61, 6-62  
   SIGN, 5-6  
 LENGTH, 5-16  
 LENGTH OF, 6-31  
   Use With Alphanumeric Literals, 1-14  
 Level  
   01, 5-4  
   02-49, 5-4  
   66, 5-5, 5-17  
   77, 5-5  
   78, 5-16, 5-17  
   88, 5-17  
 Level Number, 9-2  
 LINAGE, 5-1, 5-3, 6-21, 6-101  
 LINE, 5-18  
 LINES  
   ACCEPT, 6-29  
   AT BOTTOM, 6-102  
   AT TOP, 6-102  
   WRITE ADVANCING, 6-101  
 LINKAGE SECTION, 6-24, 6-84  
 Literal, 9-2  
 Little-Endian, 5-13  
 LOCALE, 4-4  
 LOCAL-STORAGE SECTION, 5-1, 6-35  
 LOCK, 4-7  
 LOWLIGHT, 5-19

**M**

Main Program, 9-2  
 MEMORY SIZE, 4-2  
 MERGE, 6-64  
 MOVE, 5-11  
   CORRESPONDING, 6-66

Simple, 6-66  
 MULTIPLE FILE TAPE, 4-10  
 MULTIPLY  
 BY, 6-68  
 GIVING, 6-68

## N

Negated Conditions, 6-9  
 NEGATIVE, 6-6  
 Nested Source Programs, 2-2, 3-1  
 NEXT, 6-73  
 NEXT SENTENCE, 6-69  
 NO ADVANCING, 6-43  
 NO REWIND, 6-38  
 NOT AT END, 6-73  
 NOT AT END-OF-PAGE, 6-102  
 NOT EXCEPTION  
 ACCEPT, 6-30  
 DISPLAY, 6-45  
 NOT INVALID KEY  
 DELETE, 6-42  
 READ, 6-75  
 READ, 6-75  
 REWRITE, 6-78  
 START, 6-90  
 WRITE, 6-102  
 NOT ON OVERFLOW  
 STRING, 6-92  
 UNSTRING, 6-100  
 NOT ON SIZE ERROR  
 ADD, 6-31  
 COMPUTE, 6-40  
 DIVIDE, 6-46, 6-47, 6-48  
 MULTIPLY, 6-68  
 SUBTRACT, 6-93  
 NUMBER-OF-CALL-PARAMETERS, 7-16, 7-27  
 NUMERIC, 6-6  
 Numeric Literal, 1-12, 9-2

## O

OBJECT-COMPUTER, 4-1  
 OCCURS, 5-10, 6-81  
 OFF STATUS, 4-4  
 OMITTED, 6-6  
 ON OVERFLOW  
 STRING, 6-92  
 UNSTRING, 6-100  
 ON SIZE ERROR  
 ADD, 6-31  
 COMPUTE, 6-40  
 DIVIDE, 6-46, 6-47, 6-48  
 MULTIPLY, 6-68  
 SUBTRACT, 6-93  
 ON STATUS, 4-4  
 OPEN, 6-70, 6-73, 6-74, 6-89, 6-101  
 OPTIONAL, 4-6  
 ORGANIZATION  
 INDEXED, 1-7, 4-9, 6-42, 6-73, 6-78, 6-89, 6-101, 6-102, 7-14

LINE SEQUENTIAL, 1-5, 4-8, 5-3, 6-38, 6-64, 6-78, 6-86, 6-101  
 RECORD BINARY SEQUENTIAL, 1-6, 4-8, 5-3, 6-38, 6-64, 6-78, 6-86, 6-101  
 RELATIVE, 1-6, 4-8, 6-42, 6-78, 6-89, 6-101, 6-102  
 OUTPUT, 6-70, 6-101  
 OUTPUT PROCEDURE, 6-77  
 MERGE, 6-65  
 SORT, 6-87  
 OVERFLOW  
 CALL, 6-35  
 OVERLINE, 5-18

## P

PAGE  
 WRITE ADVANCING, 6-101  
 PERFORM, 6-25, 6-52  
 Inline, 6-72  
 Procedural, 6-71  
 POSITIVE, 6-6  
 PREVIOUS, 6-73  
 PRIMARY KEY, 4-9  
 PRIMARY RECORD KEY, 6-74  
 PRINTER, 4-8, 6-43  
 Procedure, 9-2  
 PROCEDURE DIVISION, 6-24  
 Procedure Name, 9-2  
 PROGRAM COLLATING SEQUENCE, 6-87, 6-88  
 Program Unit, 2-1, 9-2  
 PROGRAM-ID, 3-1, 7-3  
 PROGRAM-POINTER, 6-83  
 PROMPT, 5-18

## Q

Qualification, 6-1, 9-2

## R

RANDOM, 4-8, 4-9  
 READ, 6-42, 6-70, 6-73, 6-74, 6-78  
 Record, 9-2  
 RECORD CONTAINS, 5-3, 6-78  
 RECORD DELIMITER, 4-6  
 RECORD IS VARYING, 5-3, 6-78  
 RECORD KEY, 4-9, 6-42, 6-78, 6-89  
 RECORDING MODE, 5-2  
 REDEFINES, 5-10, 5-11  
 REEL, 6-38  
 Reference Modifier, 6-2  
 Registers, 9-2  
 Relation Conditions, 6-8  
 RELATIVE KEY, 4-8, 4-9, 6-42, 6-78, 6-89  
 RELEASE, 6-76, 6-86  
 REPLACING  
 INITIALIZE, 6-59  
 REPLACING (COPY), 1-12  
 REPLACING (INSPECT), 6-61, 6-62  
 REPORT IS, 5-3  
 REPORT SECTION, 5-1

REPOSITORY, 4-2, 6-11  
 REQUIRED, 5-18  
 RESERVE, 4-6  
 Reserved Word, 9-3  
 RETURN, 6-65, 6-77, 6-88  
 RETURN-CODE, 7-15, 7-16, 7-17, 7-18, 7-19, 7-20, 7-21, 7-22,  
 7-23, 7-24, 7-25, 7-26  
 RETURNING, 6-24, 6-33  
     CALL, 6-35  
     STOP, 6-91  
 REVERSE-VIDEO, 5-18  
 REWRITE, 6-78  
 ROLLBACK, 6-23, 6-79  
 ROUNDED, 6-31  
     ADD, 6-40  
     DIVIDE, 6-46, 6-47, 6-48  
     MULTIPLY, 6-68  
     SUBTRACT, 6-93

## S

SAME RECORD AREA, 4-10, 6-64  
 SAME SORT AREA, 4-10, 6-64  
 SAME SORT-MERGE AREA, 4-10, 6-64  
 SCREEN CONTROL, 4-5  
 SCREEN SECTION, 1-10, 5-1  
 SCROLL, 6-28  
 SEARCH  
     (Sequential), 6-80  
     ALL (Binary Search), 6-81  
 SECURE, 5-18  
 SEGMENT-LIMIT, 4-2  
 SELECT, 1-7  
 Sentence, 9-3  
 SEPARATE CHARACTER, 5-6  
 SEQUENTIAL, 4-8, 4-9  
 SET  
     Address, 6-83  
     Condition Name, 6-84  
     ENVIRONMENT, 6-83  
     Index, 6-84  
     Program-Pointer, 6-83  
     Switch, 6-85  
     UP/DOWN, 6-84  
 SHARING, 4-7, 6-70  
 SHARING WITH ALL OTHER, 4-6  
 Shift-TAB, 5-18  
 SIGN, 5-6  
 SIZE, 6-24  
 SIZE IS AUTO, 6-24  
 SORT  
     File, 6-86  
     Table, 6-88  
 SOURCE-COMPUTER, 4-1  
 Special Registers, 9-2  
 SPECIAL-NAMES, 4-3, 6-43, 7-27  
 Split Keys, 4-9  
 START, 4-8, 4-9, 6-70, 6-89  
 Statement, 9-3  
 STOP RUN, 6-56, 6-65, 6-71, 6-86, 6-88, 6-91, 7-21  
 STRING, 6-92

Subprogram, 9-3  
 Subroutine, 9-3  
 Subscripts, 6-1  
 SUBTRACT  
     CORRESPONDING, 6-94  
     FROM, 6-93  
     GIVING, 6-93  
 SUPPRESS, 6-95  
 Switch Status Conditions, 6-7  
 SWITCH-n, 4-4, 7-27  
 SYMBOLIC CHARACTERS, 4-4  
 SYNCHRONIZED, 5-14  
 SYSTEM, 7-26

## T

TAB, 5-18  
 TALLYING, 6-61  
     UNSTRING, 6-100  
 TERMINATE, 6-96  
 THROUGH, 6-50, 6-71  
 THRU, 6-50, 6-71  
 TIME, 6-29  
 TIMES, 6-52, 6-71, 6-72  
 TO  
     Screen Item Description, 5-19  
 TO VALUE, 6-59  
 TRAILING  
     INSPECT, 6-61, 6-62  
     SIGN, 5-6  
 TRANSFORM, 6-97

## U

UNDERLINE, 5-18  
 UNIT, 6-38  
 UNLOCK, 6-23, 6-39, 6-70, 6-98  
 UNSTRING, 6-99  
 UNTIL, 6-52, 6-71, 6-72  
 UPDATE, 6-28  
 UPON, 6-43  
 USAGE  
     BINARY-CHAR, 7-4, 7-27  
     BINARY-CHAR SIGNED, 7-4  
     BINARY-CHAR UNSIGNED, 7-4  
     BINARY-C-LONG SIGNED, 7-5  
     BINARY-DOUBLE, 7-5  
     BINARY-DOUBLE SIGNED, 7-5  
     BINARY-DOUBLE UNSIGNED, 7-5  
     BINARY-LONG, 7-5  
     BINARY-LONG SIGNED, 7-5  
     BINARY-LONG UNSIGNED, 7-5  
     BINARY-SHORT, 7-4  
     BINARY-SHORT SIGNED, 7-4  
     BINARY-SHORT UNSIGNED, 7-4  
     COMPUTATIONAL-1, 7-5  
     COMPUTATIONAL-2, 7-5  
     DISPLAY, 6-5  
     INDEX, 6-84  
     POINTER, 6-5, 6-33, 6-84  
     PROGRAM POINTER, 6-5

PROGRAM-POINTER, 6-84, 7-19, 7-21  
USE AFTER STANDARD ERROR PROCEDURE, 6-25  
USE BEFORE REPORTING, 6-25  
USE FOR DEBUGGING, 6-25  
user-defined name, 9-3  
User-Defined Name, 9-3  
USING, 6-24  
    Screen Item Description, 5-19  
USING (CALL), 6-35  
USING (SORT), 6-86

**V**

VALUE, 5-10, 5-19  
VALUE OF, 5-2  
VARYING, 6-52, 6-71, 6-72  
Verb, 9-3

**W**

WHEN, 6-50  
WITH  
    DUPLICATES, 4-9

DUPLICATES IN ORDER, 6-64, 6-86, 6-88  
IGNORE LOCK, 6-23  
LOCK, 6-23  
    CLOSE, 6-38, 6-70  
NO LOCK, 6-23  
NO REWIND  
    CLOSE, 6-70  
POINTER  
    STRING, 6-92  
    UNSTRING, 6-99  
TEST, 6-71, 6-72  
WITH FILLER, 6-59  
WITH WAIT, 6-23  
WORKING-STORAGE SECTION, 1-7, 5-1  
WRITE, 6-101

**X**

X"F4", 7-28  
X"F5", 7-28  
X"91", 7-27



# GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright (C) 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.

<<http://fsf.org/>>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

## 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially.

Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

## 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input



to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent.

An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

## 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

## 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition.

Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material.

If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

#### 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice.

These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

## 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number.

Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

## 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

## 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit.

When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form.

Otherwise they must appear on printed covers that bracket the whole aggregate.

## 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4.

Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

## 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

## 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number.

If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

## 11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.