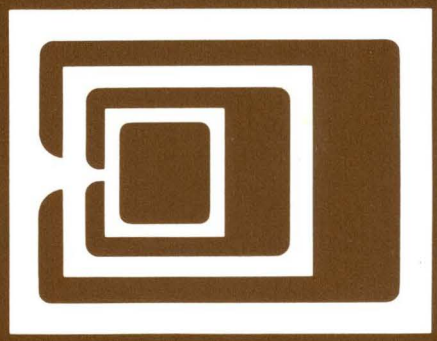
COBOL

COBOL-81
Language Reference Manual

Order No. AA–J434B–TC



digital software

COBOL-81
Language Reference Manual

Order No. AA–J434B–TC

May 1983

This document describes the COBOL–81 language.

OPERATING SYSTEM AND VERSION:	RSX–11M	V4
	RSX–11M–PLUS	V2
	RSTS/E	V8
SOFTWARE VERSION:	COBOL–81	V2

digital equipment corporation, maynard, massachusetts

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

The postage-paid READER'S COMMENTS form on the last page of this document requests your critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

digital™	DECwriter	RSTS
	DIBOL	RSX
DEC	MASSBUS	UNIBUS
DECmate	PDP	VAX
DECsystem-10	P/OS	VMS
DECSYSTEM-20	Professional	VT
DECUS	Rainbow	Work Processor

Chapter 1 General Program Concepts
COBOL Language Elements
COBOL-81 General Format Notation
Source Program Reference Formats
Program Structure
Sample Format Entry Page

Chapter 2 Identification Division
Format Entry Pages

Chapter 3 Environment Division
Format Entry Pages

Chapter 4 Data Division
Data Division Concepts

Format Entry Pages

Chapter 5 Procedure Division
Procedure Division Concepts

Format Entry Pages

Chapter 6 The COPY Statement
Format Entry Pages

Appendixes

COBOL-81/VAX–11 COBOL Reserved Words
Character Sets
File Status Key Values
Ensuring COBOL-81 Compatibility with VAX–11 COBOL

Glossary

Master Index

Book Map

Contents

Chapter 4 Data Division

Chapter 5 Procedure Division

Examples

Figures

Tables

To the Reader

Objectives

This manual describes the COBOL-81 language. It presents some general COBOL concepts and explains the use of each COBOL-81 language element.

Intended Audience

This manual is for the experienced COBOL programmer. It does not attempt to teach the COBOL language or operating system concepts and procedures. If you are a new COBOL user, you should read introductory COBOL textbooks and take DIGITAL COBOL courses – either self-paced or classroom.

Structure of This Document

The information in this manual is organized into six chapters. Supplementary information is also provided in the appendixes and glossary.

The document map, which follows the title page, lists the content areas of each chapter and appendix.

The master index at the end of this manual guides you to all the topics discussed in the COBOL-81 documentation set.

Associated Documents

Within the COBOL-81 documentation set:

- The *COBOL-81 RSTS/E User's Guide,* Order No. AA-J435C-TC, or the *COBOL-81 RSX-11M/M-PLUS User's Guide,* Order No. AA-M179B-TC, describes how to compile, debug, link, and run COBOL-81 programs. Your user's guide also discusses a variety of topics of interest to COBOL programmers.
- The *COBOL-81 Pocket guide,* Order No. AV-H630C-TC, summarizes key information from both this manual and the user's guide. The pocket guide lists all COBOL-81 language formats, commands, reserved words, and character sets.
- The *COBOL-81 RSTS/E Installation Guide/Release Notes,* Order No. AA-L028D-TC, or the *COBOL-81 RSX-11M/M-PLUS Installation Guide/Release Notes,* Order No. AA-M181C-TC, describes the installation and certification procedures for the COBOL-81 compiler. Your installation guide also contains release information that explains changes made to the compiler.
- The *PDP-11 COBOL to COBOL-81 Translator Utility,* Order No. AA-N339A-TC, contains information needed by users who have purchased the Translator Utility.

Outside the COBOL-81 documentation set:

The system directory lists and describes all manuals in your operating system's documentation set. One of the following directories can help you find the system information you need:

- *RSTS/E Documentation Directory*
- *RSX-11M/RSX-11S Information Directory and Index*
- *RSX-11M-PLUS Information Directory and Index*

Conventions Used in This Document

The following conventions apply to this manual:

Convention	Meaning
(RET)	A symbol with a one- to three-character abbreviation indicates that you must press a key on the terminal; for example, RET and TAB indicate that you press the RETURN key and the TAB key on your keyboard.
(CTRL/X)	The symbol (CTRL/X) indicates that you must press a key labeled CTRL while you simultaneously press another key; for example, (CTRL/C), (CTRL/O).
`$ COBOL (RET)`	Black ink indicates all output lines or prompting characters that the system prints or displays. Red ink indicates all user-entered commands.
`$File:  PAYROLL (RET)` `$`	
`PROCEDURE DIVISION.`	A vertical series of periods, or ellipses, means that not all the data a user would enter is shown.

`BEGIN-PROC.`
`.`
`.`
`.`
`END-PROC.`	

Summary of Technical Changes

This section lists, by chapter and appendix, the major technical changes documented in the *COBOL-81 Language Reference Manual*. These changes reflect additions and changes to the COBOL-81 programming language.

Chapter 1:

1. Special registers LINAGE-COUNTER, RMS-STS, and RMS-STV
2. System-name additions (mnemonics for devices)

Chapter 3:

3. Modification of SELECT clause to include relative files
4. Device-name and SWITCH clauses (SPECIAL-NAMES paragraph)
5. WINDOW option of the APPLY clause (I-O-CONTROL paragraph)
6. REEL/UNIT option of the RERUN clause (I-O-CONTROL paragraph)
7. Sort (or merge) files in SAME clause (I-O-CONTROL paragraph)

Chapter 4:

8. Sort/merge file description entry
9. New file description entry to include relative files
10. Minimum to maximum record size option of RECORD CONTAINS clause
11. LINAGE clause of file description entry
12. DEPENDING ON phrase of OCCURS clause
13. RENAMES clause of data description entry (level 66 items)
14. Condition-names in data description entry (level 88 items)

Chapter 5:

15. Qualification
16. Abbreviated combined relation conditions
17. Condition-name conditions
18. Extensions to the ACCEPT and DISPLAY statements to facilitate video forms design
19. CORRESPONDING phrase of the ADD, SUBTRACT, and MOVE statements
20. Separate formats and rules for EXIT and EXIT PROGRAM statements
21. Multiple receiving fields for arithmetic statements
22. VARYING phrase of the PERFORM statement
23. SORT, MERGE, RELEASE, and RETURN statements

Chapter 6:

24. REPLACING phrase of the COPY statement

Appendix A:

25. Modified reserved word list

Some formats that are not included in this list have minor rule modifications because of the new features for this release.

Incompatibilities with VAX-11 COBOL

COBOL-81 is a subset of VAX-11 COBOL, but the two products have some incompatibilities because of differences between the PDP-11 and the VAX-11 computer systems. Appendix D, Ensuring COBOL-81 Compatibility with VAX-11 COBOL, lists and describes all known incompatibilities.

Acknowledgment

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

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

The authors and copyright holders of the copyrighted material used herein are: FLOW-MATIC (trademark of Sperry Rand Corporation), Programming for the UNIVAC (R) I and II, Data Automation Systems, copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator Form No. F28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell.

They have specifically authorized the use of this material, in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

Chapter 1
General Program Concepts

This chapter contains general information about the language and structure of COBOL source programs. It describes COBOL language elements, source program reference formats, notation used in general formats, and program organization.

1.1 COBOL Language Elements

The character is the basic, indivisible unit of your COBOL program. To the COBOL-81 compiler, a COBOL program is a stream of contiguous characters that is syntactically correct according to the rules of the COBOL language. The compiler breaks down this continuous series of characters into character-strings and separators.

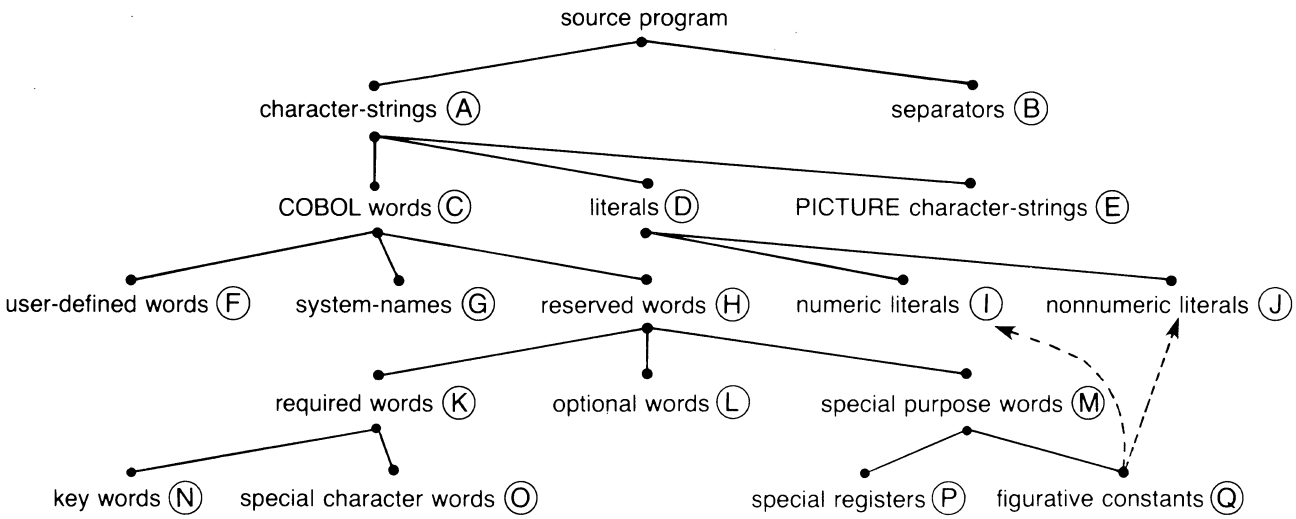
A character-string is any elementary unit of the COBOL language that provides information to the compiler. There are various types of character-strings. Each type is carefully defined in the COBOL language so that the compiler can only interpret it and use it in certain ways. For instance, PROGRAM-ID, "Enter employee number: ", and 9(10)V99 each represent a different type of character-string that appears in your source programs. Each of these character-strings provides the compiler with a different type of information.

Separators are the space characters and "punctuation" characters that mark the boundaries of character-strings. One of these separators, the period, also tells the compiler that it has reached the end of a header, sentence, or statement in a COBOL source program.

Some parts of your source program provide no information to the compiler. The compiler ignores comment lines and entries that make your program easier to understand and to maintain, but that do not add anything to program logic, data specifications, or device assignments. In short, the compiler does not consider "documentation only" sections as part of your source program.

Figure 1-1 illustrates how the compiler breaks down your source program into COBOL language elements. The sections following Figure 1-1 first discuss the COBOL character set, and then each COBOL language element in detail.

Figure 1-1: COBOL Language Elements



Examples:

```
          (N)          comment-entry
1.  AUTHOR.    LEN TRASTMART.(RET)
        (B)

          (F)     (F)        (N)   (E)     (N)   (L)   (I)
2.  01(TAB)TAX-RATE(TAB)(TAB)PIC 9V9(3) VALUE IS 1.253.(RET)
        (B)          (B)          (B)     (B)(B)     (B)

          (N)   (F)  (O)  (Q)     (N)          (J)
3.  (TAB)IF D-FLAG = ZEROS DISPLAY "Invalid data.",(RET)
          (B)     (B)(B)   (B)       (B)             (B)

          (N)         (P)          (O)(I)    (N)        (F)        (N)   (N)
4.  (TAB)IF LINAGE-COUNTER = 50, WRITE HEADER-REC AFTER PAGE.(RET)
          (B)                  (B)(B) (B)     (B)          (B)    (B)   (B)
```

1.1.1 The COBOL-81 Character Set

The COBOL-81 character set, shown in Table 1-1, is used to form character-strings and separators.

The only components of a COBOL-81 program that can contain characters outside this set are nonnumeric literals, comment-entries, and comment lines. Appendix B specifies the more inclusive computer character sets these components can use.

Table 1-1: The COBOL-81 Character Set

Character	Meaning
0, 1, ..., 9	digit
A, B, ..., Z	letter
a, b, ..., z	lowercase letter (equivalent to letter)
+	plus sign
–	minus sign (hyphen)
*	asterisk
/	slash (stroke, virgule)
=	equal sign
$	currency sign
>	greater than symbol
<	less than symbol
:	colon
_	underline (underscore)
	space
TAB	horizontal tab (equivalent to space)
(	left parenthesis
)	right parenthesis
,	comma (decimal point)
;	semicolon
.	period (decimal point, full stop)
"	quotation mark

Except in nonnumeric literals, the compiler treats lowercase letters as if they were uppercase. Therefore a program can contain COBOL words without regard to case. For example, the compiler recognizes the COBOL words in each of the following pairs as identical:

WORKING-STORAGE	Working-Storage
Input	input
file-a	FILE-A
INSPECT	InSpect

1.1.2 COBOL Words

A COBOL word is a character-string of not more than 30 characters that forms one of the following:

- A user-defined word
- A system-name
- A reserved word

A system-name or user-defined word cannot be a reserved word. However, a program can use the same COBOL word as both a user-defined word and a system-name. The compiler determines the word's class from its context.

1.1.2.1 User-Defined Words — A user-defined word is a COBOL word that you must supply to satisfy the format of a clause or statement. This word consists of characters selected from the set A through Z, 0 through 9, and hyphen (-). The hyphen can neither begin nor end a user-defined word.

COBOL-81 recognizes 13 types of user-defined words. Those that define program resources are grouped into sets. The letters preceding the word types show the set structure:

(A) alphabet-name

(B) condition-name

(B) data-name

(C) file-name

(D) index-name

level-number

(E) mnemonic-name

(F) paragraph-name

(G) program-name

(B) record-name

(H) section-name

segment-number

(I) text-name

All user-defined words in a program, except segment-numbers and level-numbers, can belong to one and only one of these sets. User-defined words in each set must be unique, or defined according to the rules for uniqueness of reference. (See Section 5.3, Uniqueness of Reference.) However, any segment-number or level-number can be the same as any other segment-number or level-number.

Except for section-names, paragraph-names, segment-numbers, and level-numbers, all user-defined words must contain at least one alphabetic character.

Table 1-2 describes the COBOL-81 user-defined words.

1.1.2.2 System-Names — A system-name is a COBOL word that has been defined by DIGITAL to refer to the program's operating environment. It is similar to a reserved word, except that its use is "reserved" only in particular places in the program. Anywhere else in the program, it can be used as a user-defined word. The compiler determines whether the word is a system-name or a user-defined word from its context in the program.

Table 1-3 lists the 16 COBOL-81 system-names, and specifies their locations in the source program.

1.1.2.3 Reserved Words — A reserved word can be used only as specified in the general formats. It cannot be a user-defined word. See Appendix A, COBOL-81/VAX–11 COBOL Reserved Words.)

There are three types of reserved words:

1. Required words
2. Optional words
3. Special-purpose words

Table 1-2: COBOL-81 User-Defined Words

User-Defined Word	Purpose
Alphabet-Name	Assigns a name to a character set and/or collating sequence. Alphabet-names must be defined in the SPECIAL-NAMES paragraph. (See Section 3.1.3, SPECIAL-NAMES Paragraph.)
Condition-Name	Assigns a name to a value, set of values, or range of values in the complete set of values that a data item can have. Data items with one or more associated condition-names are called conditional variables. Data Division entries define condition-names. Names assigned in the SPECIAL-NAMES paragraph to the "on" or "off" status of switches are also condition-names.
Data-Name	Names a data item described in a data description entry. When specified in a general format, *data-name* cannot be subscripted, indexed, or qualified unless specifically allowed by the rules for that format.
File-Name	Names a file connector. A file connector is the link between: • A file-name and a physical file • A file-name and its associated storage area File description and sort-merge file description entries describe file connectors.
Index-Name	Names an index associated with a specific table.
Level-Number	Is a one- or two-digit number that describes a data item's special properties or its position in the structure of a record. (See Section 4.1.1.1, Record Description, and Section 4.1.1.2, Level-Numbers.)
Mnemonic-Name	Associates a name with a system-name, such as CONSOLE, or SWITCH. (See Section 3.1.3, SPECIAL-NAMES Paragraph.)
Paragraph-Name	Names a Procedure Division paragraph. (See Section 1.4.3.) Paragraph-names are equivalent only if they are identical, that is, when they are composed of the same sequence and number of digits and/or characters. For example: START-UP — START-UP — Equivalent START-UP — STARTUP — Different Start-up — START-UP — Equivalent 001-START-UP — 01-START-UP — Different 017 — 017 — Equivalent 017 — 17 — Different
Program-Name	Identifies a COBOL source program. Only the first six characters of program-name are significant. (See Section 2.1, PROGRAM-ID Paragraph.)
Record-Name	Names a data item described with level-number 01 or 77.
Section-Name	Names a Procedure Division section. Section-names are equivalent only if they are identical: when they are composed of the same sequence and number of digits and/or characters. (See Section 1.4.3.)
Segment-Number	Is a one- or two-digit number that classifies a Procedure Division section for segmentation. In COBOL-81 programs, segment-numbers specify overlayable and nonoverlayable segments. (See Section 5.8, Segmentation.)
Text-Name	Identifies library text in a COBOL library. (See Chapter 6, COPY Statement.)

Table 1-3: COBOL-81 System-Names

System-Name	Location
CARD-READER	SPECIAL-NAMES paragraph
CONSOLE	SPECIAL-NAMES paragraph
CONTIGUOUS	APPLY clause of the I-O-CONTROL paragraph
DEFERRED-WRITE	APPLY clause of the I-O-CONTROL paragraph
EXTENSION	APPLY clause of the I-O-CONTROL paragraph
FILL-SIZE	APPLY clause of the I-O-CONTROL paragraph
ID	VALUE OF ID clause of the file description entry
LINE-PRINTER	SPECIAL-NAMES paragraph
MASS-INSERT	APPLY clause of the I-O-CONTROL paragraph
PAPER-TAPE-PUNCH	SPECIAL-NAMES paragraph
PAPER-TAPE-READER	SPECIAL-NAMES paragraph
PDP-11	SOURCE-COMPUTER and OBJECT-COMPUTER paragraphs
PREALLOCATION	APPLY clause of the I-O-CONTROL paragraph
PRINT-CONTROL	APPLY clause of the I-O-CONTROL paragraph
SWITCH	SPECIAL-NAMES paragraph
WINDOW	APPLY clause of the I-O-CONTROL paragraph

1.1.2.3.1 Required Words (Key Words and Special Characters) — A required word must be used whenever the statement or clause containing it is used in a program.

There are two types of required words: key words and special character words.

1. **Key Words**

   In general formats, key words are in uppercase and underlined.

   In Figure 1-2, the key words are COMPUTE, ROUNDED, SIZE, and ERROR.

Figure 1-2: Sample General Format

```
COMPUTE { rsult [ ROUNDED ] } ... = arithmetic-expression  [ ON SIZE ERROR stment ]
```

2. **Special Character Words**

   The arithmetic operators and relation characters are special character words. They are not underlined in general formats.

   In Figure 1-2, the equal sign (=) is a special character word.

1.1.2.3.2 Optional Words — In general formats, uppercase words that are not underlined are optional. They can make a program more readable, but have no semantic effect. In Figure 1-2, ON is an optional word.

1.1.2.3.3 Special-Purpose Words (Registers and Constants) — There are two types of special-purpose words: (1) special registers, which name and refer to special storage areas (special registers) that the compiler provides, and (2) figurative constants, which name and refer to specific constant values.

1. Special Registers

   The COBOL special registers appear only in Procedure Division statements. They store information related to or produced by specific COBOL features. The special registers are as follows:

   - For Linage Files

     LINAGE-COUNTER – The reserved word LINAGE-COUNTER names a line counter that the compiler provides when a file description entry contains a LINAGE clause. Its value is the number of the current record within the page body. (See Section 4.2.12, LINAGE Clause.) The implicit size of LINAGE-COUNTER is four decimal digits represented by PIC S9(4) COMP. You can qualify it with a file-name. Procedure Division statements can access the value of LINAGE-COUNTER but cannot change the value.

   - For PDP-11 Record Management Services (RMS-11)

     RMS-STS – The reserved word RMS-STS names a Record Management Services exception condition register. It contains the primary RMS status value of an I-O operation (RMS-STV is the secondary value). RMS-STS provides additional information on COBOL File Status values resulting from I-O operations. It is a four digit COMP item represented by PIC S9(4) USAGE IS COMP. You can qualify RMS-STS with a file-name. Before the program opens the file for the first time, the value of RMS-STS is undefined. After your program executes an OPEN or CLOSE statement, RMS-STS is set to the value of the STS field in the associated File Access Block. After execution of a READ, WRITE, REWRITE, DELETE, or START statement, RMS-STS is set to the value of the STS field in the associated Record Access Block. For an explanation and a listing of these values, refer to the *RMS-11 Macro Programmer's Guide*. Procedure Division statements can read the value in RMS-STS; however, only RMS-11 can change the value. For an example of its use, refer to the chapter on I-O exceptions conditions handling in Part IV of the COBOL-81 User's Guide for your system.

     RMS-STV – The reserved word RMS-STV names a Record Management Services exception condition register. It contains the secondary (RMS-STS is primary) RMS status value of an I-O operation. The interpretation of this value is dependent on the value in RMS-STS. It is a four digit COMP item represented by PIC S9(4) USAGE IS COMP. You can qualify RMS-STV with a file-name. The value in RMS-STV is undefined prior to the initial OPEN of the file. After your program executes an OPEN or CLOSE statement, RMS-STV is set to the value of the STV field in the associated File Access Block. After execution of a READ, WRITE, REWRITE, DELETE, or START statement, RMS-STV is set to the value of the STV field in the associated Record Access Block. For an explanation and a listing of these values, refer to the *RMS-11 Macro Programmer's Guide*. Procedure Division statements can read the value in RMS-STV; however, only RMS-11 can change the value. For an example of its use, refer to the chapter on I-O exceptions conditions handling in Part IV of the COBOL-81 User's Guide for your system.

2. Figurative Constants

Figurative constants name and refer to specific constant values generated by the compiler. The singular and plural forms of figurative constants are equivalent and interchangeable.

The figurative constants are:

ZERO, ZEROS, ZEROES

Represent the value zero, or one or more occurrences of the character 0 from the computer character set, depending on context. In the following example, the first use of the word ZERO represents a zero value; the second use represents six 0 characters:

```
03  ABC  PIC 9(5) VALUE ZERO.
03  DEF  PIC X(6) VALUE ZERO.
```

SPACE, SPACES

Represent one or more space characters from the computer character set.

HIGH-VALUE, HIGH-VALUES

Represent one or more occurrences of the character with the highest ordinal position in the program collating sequence. The value of HIGH-VALUE depends on the collating sequence specified by clauses in the OBJECT-COMPUTER and SPECIAL-NAMES paragraphs. (See Section 3.1.2, OBJECT-COMPUTER Paragraph and Section 3.1.3, SPECIAL-NAMES Paragraph.) For example, HIGH-VALUE for the NATIVE collating sequence is octal 377, but HIGH-VALUE for the STANDARD-1 collating sequence is octal 177.

LOW-VALUE, LOW-VALUES

Represent one or more occurrences of the character with the lowest ordinal position in the program collating sequence. The value of LOW-VALUE is octal 00, regardless of what collating sequence is specified. (See Section 3.1.2, OBJECT-COMPUTER Paragraph, and Section 3.1.3, SPECIAL-NAMES Paragraph.)

QUOTE, QUOTES

Represent one or more occurrences of the quotation-mark character ("). QUOTE or QUOTES cannot be used in place of a quotation mark to delimit a nonnumeric literal. The following examples are not equivalent:

```
QUOTE abcd QUOTE
"abcd"
```

ALL Literal

Represents one or more occurrences of the string of characters comprising the literal. The literal must be either nonnumeric, or a figurative constant other than ALL literal. When it precedes a figurative constant (for example, ALL ZEROES), the word ALL is redundant and serves only to enhance readability.

When a figurative constant represents a string of one or more characters, the string's length depends on its context:

1. The string's length can vary for a figurative constant in a VALUE IS clause, or for one associated with another data item (for example, when the figurative constant is moved to or compared with another data item). Proceeding from left to right, the compiler repeats the string of characters that represents the figurative constant. It repeats them, character by character, until the size of the resultant string equals that of the associated data item. This is done before and independent of the application of any JUSTIFIED clause specified for the data item.
2. When a figurative constant is not associated with another data item (for example, when it is in a DISPLAY, STRING, STOP, or UNSTRING statement), the length of the string is one occurrence of the ALL literal or one character in all other cases.

A figurative constant is valid wherever the word "literal" (or its abbreviation, "lit") appears in a General Format, or its associated rules. However, ZERO (ZEROS, ZEROES) is the only valid figurative constant for literals restricted to numeric characters.

The actual characters associated with HIGH-VALUE(S) depend on the program collating sequence. (See Section 3.1.2, OBJECT-COMPUTER Paragraph, and Section 3.1.3, SPECIAL-NAMES Paragraph.)

1.1.2.3.4 Literals (Numeric and Nonnumeric) — A literal is a character-string whose value is specified by: (1) the ordered set of characters it contains, or (2) a reserved word that is a figurative constant.

There are two types of literals: numeric and nonnumeric.

Numeric Literals

A numeric literal is a character string of 1 to 20 characters selected from the digits 0 through 9, the plus sign (+), the minus sign (–), and the decimal point (.).

The value of a numeric literal is the algebraic quantity represented by the characters in the literal. Its size equals the number of digits in the character-string.

The syntax rules for numeric literals are as follows:

1. A numeric literal must contain at least one digit and not more than 18 digits.
2. A numeric literal can contain only one sign character, which must be the leftmost character. If the literal is unsigned, its value is positive.
3. A numeric literal can contain only one decimal point. The decimal point is treated as an assumed decimal point. It can be used anywhere in the literal except as the rightmost character.

   If a numeric literal contains no decimal point, it is an integer.
4. The compiler treats a numeric literal enclosed in quotation marks as a nonnumeric literal.

Example 1-1: Size and Value of Numeric Literals

Literal	Value	Size in Digits
12	12	2
-123456789012345678	-123456789012345678	18
000000003	3	9
-34.455445555	-34.455445555	11
0	0	1
+0.000000000001	+0.000000000001	13
+0000000000001	+1	13

Nonnumeric Literals

A nonnumeric literal is a character-string of 0 to 256 characters. It is delimited on both ends by a quotation mark (").

The value of a nonnumeric literal is the value of the characters in the character-string. It does not include the quotation marks that delimit the character-string. All other punctuation characters in the nonnumeric literal are part of its value.

The compiler truncates nonnumeric literals to a maximum of 256 characters.

The syntax rules for nonnumeric literals are as follows:

1. A space or left parenthesis must immediately precede the opening quotation mark.
2. The closing quotation mark must be immediately followed by one of the following:
   - Space
   - Comma
   - Semicolon
   - Period
   - Right parenthesis
3. Because quotation marks are used as delimiters, two consecutive quotation marks must be used within the literal to represent the value of one quotation mark.

Example 1-2: Size and Value of Nonnumeric Literals

In the following examples, *s* represents a space character.

Literal	Value	Size in Characters
"ABC"	ABC	3
"01"	01	2
"s01"	s01	3
"D""E""F"	D"E"F	5
"a,b"	a,b	3
""""	"	1
"J""""K"	J""K	4
"O""P""Q"	O"P"Q	5
"R""""S""""T"	R""S""T	7

1.1.3 PICTURE Character-Strings

A PICTURE character-string defines the size and category of an elementary data item. It can consist of the currency symbol and certain combinations of characters in the COBOL character set. (See Section 4.2.14, PICTURE Clause.)

A punctuation character that is part of a PICTURE character-string is not considered to be a punctuation character. Instead, the compiler treats it as a symbol within the PICTURE character-string.

1.1.4 Separators

A separator delimits character-strings. It can be one character or two contiguous characters formed according to the following rules:

Space

The space can be a separator or part of a separator.

1. Where a space is used as a separator or part of a separator, more than one space can be used.
2. A space can immediately precede any separator except:
   a. As specified by the rules for reference formats (See Section 1.3)
   b. The closing quotation mark of a nonnumeric literal; the space is then considered part of the nonnumeric literal rather than a separator
3. A space can immediately follow any separator except the opening quotation mark of a nonnumeric literal. After an opening quotation mark, the space is considered part of the nonnumeric literal rather than a separator.

Comma and Semicolon

The comma and semicolon are separators when they immediately precede a space. In this case, the comma and semicolon are interchangeable with each other and with the separator space. They can be used anywhere in a source program that a separator space can be used.

Period

The period is a separator when it immediately precedes a space or a return character. It can be used only where allowed by:

1. Statement and sentence structure definitions (Section 5.1, Verbs, Statements, and Sentences)
2. Reference format rules (Section 1.3, Source Program Reference Formats)

Parentheses

Parentheses can be used only in balanced pairs of left and right parentheses to delimit:

- Subscripts
- Indexes
- Arithmetic expressions
- Conditions

Quotation Marks

An opening quotation mark (") must be immediately preceded by a separator space or a left parenthesis. A closing quotation mark (") must be immediately followed by one of the separators: space, comma, semicolon, period, or right parenthesis.

Horizontal Tab

The horizontal tab aligns statements or clauses on successive columns of the source program listing. It is interchangeable with the separator space. When the compiler detects a tab character (other than in a nonnumeric literal), it generates one or more space characters consistent with the tab character position in the source line. (See Section 1.3, Source Program Reference Format.)

1.2 COBOL-81 General Format Notation (Meta-Language)

Throughout this manual, general formats are shown for all COBOL-81 clauses and statements. General formats show the specific arrangement of the parts of an entry, paragraph, clause, or statement. When you can use more than one arrangement of its parts, the general format is separated into separate formats (Format 1, Format 2, and so forth). Unless the general format's rules state otherwise, you must write clauses in the sequence shown.

The notation used in the general formats is called the COBOL meta-language. Because it illustrates the rules to follow when writing a source program, COBOL meta-language helps you to write your own statements or clauses. However, some of the elements of COBOL meta-language would not actually appear in any source program.

The following meta-language elements are combined into general formats. Those elements that do not actually appear in a source program are followed by an asterisk (*):

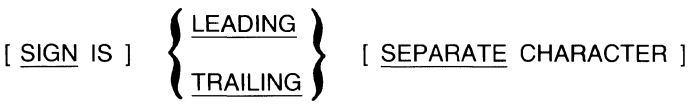
- Uppercase and special-character words
- Lowercase words
- Brackets *
- Braces *
- Choice indicators *
- Ellipsis *
- Separator period

1.2.1 Uppercase and Special-Character Words

All uppercase and special-character words are COBOL reserved words; that is, they cannot appear in your program as words you define or as system-names.

Underlined uppercase words are key words. A key word is required and must be spelled correctly when it is included in the source program. The following special-character words are not underlined in general formats but are required where they appear: +, –, <, >, comma (,), and =.

In the general format for the SIGN clause, the key words are SIGN, LEADING, TRAILING, and SEPARATE:



Uppercase words not underlined are optional. They serve only to improve the source program's readability. In the preceding general format, the optional words are IS and CHARACTER.

1.2.2 Lowercase Words

Lowercase words are generic terms. They indicate entries the programmer must provide. Lowercase words can represent COBOL words, literals, PICTURE character-strings, comment-entries, or complete syntactical entries.

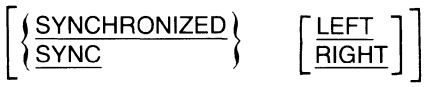
1.2.3 Brackets and Braces

Brackets ([ ]) enclose an optional part of a general format. When they enclose vertically stacked entries, brackets indicate that you can select one (but no more than one) of the enclosed entries.

Braces ({ }) indicate that you *must* select one (but no more than one) of the enclosed entries. If one of the entries contains only reserved words that are not key words, that entry is the default option when no other entry is selected.

In the general format for the SYNCHRONIZED clause:

- The entire clause is optional
- If the clause is used, it must contain either SYNCHRONIZED or SYNC
- The clause can contain either LEFT or RIGHT (or neither)



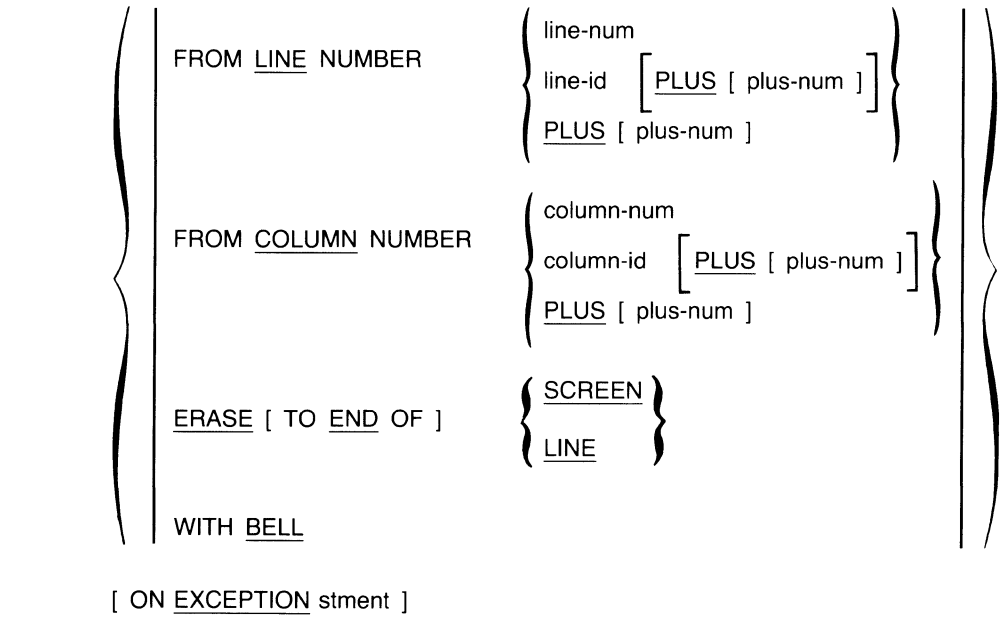
The following SYNCHRONIZED clause entries are valid:

```
SYNCHRONIZED
SYNCHRONIZED LEFT
SYNCHRONIZED RIGHT
SYNC
SYNC LEFT
SYNC RIGHT
```

1.2.4 Choice Indicators

If choice indicators, {| |}, enclose part of a general format, you *must* select *one or more* of the enclosed entries (in any order). However, no entry can be used more than once.

In the general format for the ACCEPT statement (format 4), one or more of the vertically stacked entries must be selected:



Some valid ACCEPT statement entries are:

```
ACCEPT KEY IN A-KEY LINE 10 COLUMN 20.
ACCEPT CONTROL KEY IN A-KEY ERASE LINE WITH BELL.
ACCEPT CONTROL KEY A-KEY COLUMN 15.
```

1.2.5 The Ellipsis

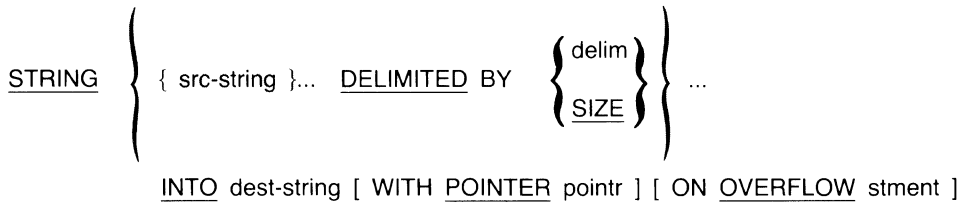
In general formats, the ellipsis (...) allows repetition of a part of the format.

To determine which part of the format can be repeated:

1. Find the ellipsis.
2. Scanning to the left, find the first right delimiter, either] or }.
3. Continuing to the left, find its logically matching left delimiter, either [or {.

The ellipsis applies to the part of the format between the matched pair of delimiters.

In the general format for the STRING statement, the ellipsis allows repetition of the shaded part:



Some valid STRING statement entries are:

```
STRING A B DELIMITED BY SIZE INTO C.
STRING A DELIMITED BY B, C DELIMITED BY D INTO E.
STRING A B DELIMITED BY SIZE, C DELIMITED BY D INTO E.
```

1.2.6 The Separator Period

The separator period (.) is the period used to "punctuate" the source program. Separator periods are required where shown in a general format. For example, there are eight separator periods in the general format for the Identification Division:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. program-name.
[ AUTHOR. [ comment-entry ] ... ]
[ INSTALLATION. [ comment-entry ] ... ]
[ DATE-WRITTEN. [ comment-entry ] ... ]
[ DATE-COMPILED. [ comment-entry ] ... ]
[ SECURITY. [ comment-entry ] ... ]
```

The separator periods following the words, DIVISION, PROGRAM-ID, and *program-name* must appear in every source program. The separator periods following the words AUTHOR, INSTALLATION, DATE-WRITTEN, DATE-COMPILED, and SECURITY must appear in a source program that includes these optional paragraphs.

1.3 Source Program Reference Formats

The COBOL-81 compiler recognizes two source program formats: terminal and ANSI.

- Terminal format is a compact DIGITAL-specified format.
- ANSI format conforms to the American National Standard COBOL reference format.

Terminal format is the default program reference format (unless this was changed by your system manager at installation time). In other words, the compiler expects terminal format source lines if the compiler command line either: (1) includes the qualifier /NOANSI_FORMAT, or (2) has no source format qualifier. The compiler expects ANSI format only when the command line includes the /ANSI_FORMAT qualifier.

The program reference format spacing rules take precedence over all other spacing rules.

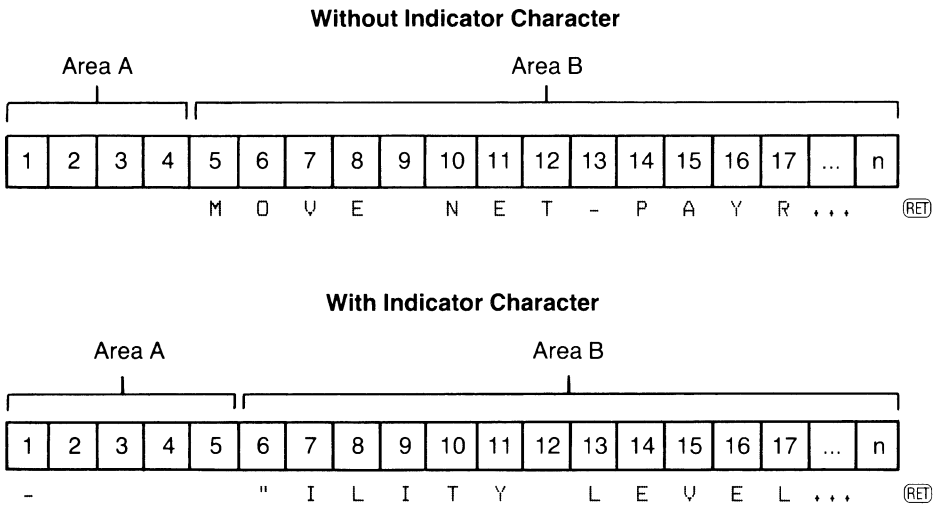
1.3.1 Terminal Format

COBOL-81 terminal format shortens program preparation time and reduces storage space for source programs. If you have used only ANSI format in the past, note that terminal format:

- Combines the indicator area with Area A
- Eliminates the sequence number and identification areas
- Permits up to 200 characters in a source program line

Figure 1-3 illustrates how Areas A and B are mapped to character positions in the source code line. The value n can be less than or equal to 200.

Figure 1-3: Terminal Program Reference Format



The following five sections discuss the definitions and rules that apply to terminal format.

1.3.1.1 Source Line Structure

Indicator

An indicator occupies the first character position. In terminal format, the compiler recognizes the following valid indicator characters in the first character position:

Character	Source Line Interpretation
hyphen (-)	Continuation line. The compiler processes the line as a continuation of the previous source line.
asterisk (*)	Comment line. The compiler ignores the contents of the line. However, the source line appears on the program listing.
slash (/)	New listing page. The compiler treats the line as a comment line. However, it advances the program listing to the top of the next page before printing the line.

Area A — When no indicator is present, Area A occupies character positions 1 through 4. When an indicator is present, Area A occupies character positions 1 through 5.

Area A contains division headers, section headers, paragraph headers, paragraph-names, level indicators, and certain level-numbers.

Area B — Area B begins with the character position immediately following Area A. It ends when the compiler detects a carriage return.

Area B contains all other COBOL text.

1.3.1.2 Line Continuation — Sentences, entries, phrases, and clauses that continue in Area B of subsequent lines are called continuation lines. The line being continued is called the continued line.

A hyphen in a line's indicator area causes its first nonblank character in Area B to be the immediate successor of the last nonblank character of the preceding line. This continuation excludes intervening comment lines and blank lines.

However, if the continued line contains a nonnumeric literal without a closing quotation mark, the first nonblank character in Area B of the continuation line must be a quotation mark. The continuation starts with the character immediately after the quotation mark. Area A of the continuation line must be blank.

If the indicator area is blank:

1. The compiler treats the first nonblank character on the line as if it followed a space.
2. The compiler treats the last nonblank character on the preceding line as if it preceded a space.

Example 1-3 illustrates the use of line continuation in terminal format. The example shows continuation of a numeric literal, a nonnumeric literal, and a sentence (in that order).

Example 1-3: Line Continuation of Numeric and Nonnumeric Literals (Terminal Format)

```
01  NUMERIC-CONTINUATION.
    03  NUMERIC-LITERAL                    PIC  9(16) VALUE IS 123
-    4567890123456.
01  NONNUMERIC-CONTINUATION.
    03  NONNUMERIC-LITERAL                 PIC  X(40) VALUE IS "AB
-    "CDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmn".
PROCEDURE DIVISION.
SENTENCE-CONTINUATION.
    IF NUMERIC-LITERAL NOT = SPACES
        DISPLAY "NUMERIC-LITERAL NOT = SPACES"
    ELSE
        DISPLAY NUMERIC-LITERAL.
```

1.3.1.3 Blank Lines — A blank line contains no characters. The compiler recognizes a blank line by the presence of the carriage return.

1.3.1.4 Comment Lines — A comment line is any source line with an asterisk (*) or slash (/) in its indicator area. Area A and Area B can contain any character(s) from the computer character set. Comment lines can be anywhere in a source program or library text.

1.3.1.5 Short Lines and Tab Characters — Because terminal format does not have a fixed 80-character line length, you can press RETURN to delimit lines that are shorter than that. The RETURN key inserts a return character into the source program file, and the compiler recognizes the return character as the end of the line.

The TAB key inserts a tab character into the source program file. Tab characters, other than those in nonnumeric literals, cause the compiler to generate enough space characters to position the next character you enter at the next tab stop.

In terminal format, the compiler's tab stops are: (1) on the first character position of Area B and (2) every eight character positions to the right, until the end of the line.

Using TAB makes it easy for you to move directly into Area B without counting spaces. Using TAB also makes it easier to obtain consistent vertical alignment within Area B to improve program readability.

Note

Although the lines in a source program can be as long as 200 characters, you must remember that the maximum length of the source line includes all spaces represented by a tab character.

Also, only 125 characters of the source program line appear on the program listing. The compiler processes the complete source line but displays only the first 125 characters on the listing.

Example 1-4 shows how the compiler interprets carriage return and tab characters:

Example 1-4: Compiler Interpretation of Shortened Source Lines (Terminal Format)

Source Lines Entered from Terminal

```
*The following record description shows the source line format(RET)
01(TAB)RECORD-A.(RET)
(TAB)03  GROUP-A.(RET)
(TAB)(TAB)05  ITEM-A(TAB)PIC X(10).(RET)
*(TAB)The tab character in the nonnumeric literal(RET)
*(TAB)on the next line is stored as one character(RET)
(TAB)(TAB)05  ITEM-B(TAB)PIC X VALUE IS "(TAB)".(RET)
(TAB)03  ITEM-C(TAB)(TAB)PIC X(10).(RET)
```

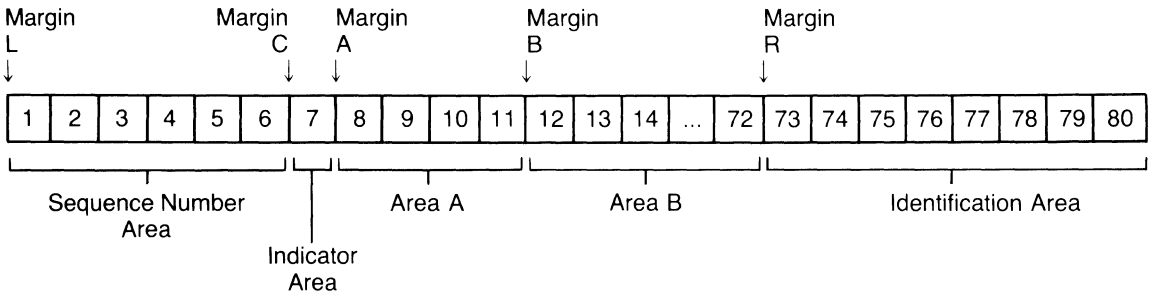
Compiler Interpretation

```
*The following record description shows the source line format
01  RECORD-A.
    03  GROUP-A.
            05  ITEM-A      PIC X(10).
*    The tab character in the nonnumeric literal
*    on the next line is stored as one character
            05  ITEM-B      PIC X VALUE IS "(TAB)".
    03  ITEM-C              PIC X(10).
```

1.3.2 ANSI Format

ANSI program reference format describes COBOL programs so that they can be stored on punched card media. For compatibility with card format, a source program line must be limited to 80 characters. Also, each area of the input line is defined as a set sequence of character positions. "Margins" between the areas are fixed to define the columns of the punched card. Figure 1-4 illustrates how ANSI format areas are mapped to character positions in the input source line.

Figure 1-4: ANSI Program Reference Format



The following five sections discuss the definitions and rules that apply to ANSI format.

1.3.2.1 Source Line Structure

Margin L	Immediately to the left of the leftmost character position.
Margin C	Between character positions 6 and 7.
Margin A	Between character positions 7 and 8.
Margin B	Between character positions 11 and 12.
Margin R	Between character positions 72 and 73.
Sequence Number Area	The six character positions between Margin L and Margin C. The contents can be any character(s) from the computer character set. The compiler does not check the contents of this area for either uniqueness, or ascending sequence.
Indicator Area	The seventh character position. The character in this position directs the compiler to interpret the source line in one of the following ways:

Character	Source Line Interpretation
space ( )	Default. The compiler processes the line as normal COBOL text.
hyphen (–)	Continuation line. The compiler processes the line as a continuation of the previous source line.
asterisk (*)	Comment line. The compiler ignores the contents of the line.
slash (/)	New listing page. The compiler treats the line as a comment line. However, it advances the program listing to the top of the next page before printing the line.

(continued on next page)

Area A	The four character positions between Margin A and Margin B. Area A contains division headers, section headers, paragraph headers, paragraph-names, level indicators, and certain level-numbers.
Area B	The 61 character positions between Margin B and Margin R. Area B contains all other COBOL text.
Identification Area	The eight character positions immediately following Margin R. The compiler ignores the contents of the identification area.

1.3.2.2 Line Continuation — Sentences, entries, phrases, and clauses that continue in Area B of subsequent lines are called continuation lines. The line being continued is called the continued line.

A hyphen in a line's indicator area causes its first nonblank character in Area B to be the immediate successor of the last nonblank character of the preceding line. This continuation excludes intervening comment lines and blank lines.

However, if the continued line contains a nonnumeric literal without a closing quotation mark, the first nonblank character in Area B of the continuation line must be a quotation mark. The continuation starts with the character immediately after the quotation mark. The compiler considers all 61 character positions in Area B of the continued line as part of the literal. Pressing RETURN will not suppress spaces following text entry on the continued line. Area A of the continuation line must be blank.

If the indicator area is blank:

1. The compiler treats the first nonblank character on the line as if it followed a space.
2. The compiler treats the last nonblank character on the preceding line as if it preceded a space.

Example 1-5 illustrates use of line continuation in ANSI format.

Example 1-5: Line Continuation of Numeric and Nonnumeric Literals (ANSI Format)

```
001010 01  NUMERIC-CONTINUATION.
001020     03  NUMERIC-LITERAL                    PIC  9(16) VALUE IS 123
001030-    4567890123456.
001040 01  NONNUMERIC-CONTINUATION.
001050     03  NONNUMERIC-LITERAL                 PIC  X(40) VALUE IS "AB
001060-    "CDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmn".
001070 PROCEDURE DIVISION.
001080 SENTENCE-CONTINUATION.
001090     IF NUMERIC-LITERAL NOT = SPACES
001100         DISPLAY "NUMERIC-LITERAL NOT = SPACES"
001110     ELSE
001120         DISPLAY NUMERIC-LITERAL.
```

Lines 001020 and 001030 show continuation of a numeric literal. Lines 001050 and 001060 continue a nonnumeric literal. A sentence that spans four lines begins on line 001090.

1.3.2.3 Blank Lines — A blank line contains no characters other than spaces between Margin C and Margin R. Blank lines can be anywhere in a source program or in library text that you intend to include in a source program with the COPY statement.

1.3.2.4 Comment Lines — A comment line is any source line with an asterisk or slash in its indicator area. Area A and Area B can contain any character(s) from the computer character set. Comment lines can be anywhere in a source program or library text.

1.3.2.5 Short Lines and Tab Characters — If the source program input medium is not punched cards, using the TAB and RETURN keys can shorten source program lines. A tab character is inserted into the source program by the TAB key; a return character is inserted by the RETURN key.

The compiler recognizes the end of the input line as Margin R. Tab characters, other than those in nonnumeric literals, cause the compiler to generate enough space characters to position the next character at the next tab stop. The compiler's tab stops are at character positions 8 (first character postion in Area A), 12 (first character position in Area B), 20, 28, 36, 44, 52, 60, 68, and 76.

Example 1-6 shows how the compiler interprets carriage return and horizontal tab characters in ANSI format.

Example 1-6: Compiler Interpretation of Shortened Source Lines (ANSI Format)

Shortened ANSI format source line

```
000100*The following record description shows the source line format↵
000110 01⇥RECORD-A.↵
000120⇥⇥03  GROUP-A.↵
000130⇥⇥⇥05  ITEM-A⇥PIC X(10).↵
000140*⇥The tab character in the nonnumeric literal↵
000150*⇥on the next line is stored as one character↵
000160⇥⇥⇥05  ITEM-B⇥PIC X VALUE IS "⇥".↵
000170⇥⇥03  ITEM-C⇥⇥PIC X(10).↵
```

Source line as interpreted by compiler

```
000100*The following record description shows the source line format
000110 01  RECORD-A.
000120     03  GROUP-A.
000130             05  ITEM-A      PIC X(10).
000140*    The tab character in the nonnumeric literal
000150*    on the next line is stored as one character
000160             05  ITEM-B      PIC X VALUE IS "⇥".
000170     03  ITEM-C              PIC X(10).
```

Do not use the TAB key more than necessary. You will get compiler error diagnostics if you insert tab characters beyond the permissible character position(s) for a COBOL statement or entry. Example 1-7 shows how the compiler treats a source program line with tab characters inserted incorrectly. The problem is in line 000004: it contains one too many tab characters. This places the paragraph-name P0 out of Area A.

Example 1-7: Incorrect Use of TAB

Shortened ANSI Format Source Line

```
000001(TAB)IDENTIFICATION DIVISION.
000002(TAB)PROGRAM-ID. ANSI-TEST.
000003(TAB)PROCEDURE DIVISION.
000004(TAB)(TAB)P0.
000005(TAB)(TAB)STOP RUN.
```

Source Line as Interpreted by Compiler

```
     1        000001 IDENTIFICATION DIVISION.
     2        000002 PROGRAM-ID. ANSI-TEST.
     3        000003 PROCEDURE DIVISION.
     4        000004              P0.
                                  ^^^
***  F  450  Undefined reference.
***  F  590  A section header or paragraph-name is required.
***  F  519  Invalid statement syntax.
     5        000005              STOP RUN.
                                  ^
***  I  501  *Compilation resumed at this point.
```

1.4 Program Structure

Figure 1-5 shows the basic program structure of a COBOL program. It illustrates the organization of a program into divisions, sections, paragraphs, sentences, and entries.

Figure 1-5: Structure of a COBOL Program

IDENTIFICATION DIVISION.
PROGRAM-ID. *main-program.*
AUTHOR.
INSTALLATION.
DATE-WRITTEN.
DATE-COMPILED.
SECURITY.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.
OBJECT-COMPUTER.
SPECIAL-NAMES.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
I-O-CONTROL.

DATA DIVISION.
FILE SECTION.
file and record description entries
sort-merge file and record description entries
WORKING-STORAGE SECTION.
record description entries
LINKAGE SECTION.
record description entries

(continued on next page)

PROCEDURE DIVISION
DECLARATIVES.
sections
paragraphs
sentences
END DECLARATIVES.
sections
paragraphs
sentences

1.4.1 Division Header

A division header identifies and marks the beginning of a division. It is a specific combination of reserved words followed by a separator period. Division headers start in Area A.

Except for the COPY statement (See Chapter 6, The COPY Statement.), the statements, entries, paragraphs and sections of a COBOL source program are grouped into four divisions in this order:

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

The end of a COBOL source program is indicated by the end of that program's Procedure Division.

Note

The Procedure Division header can contain a USING phrase. (See Section 5.9, Procedure Division General Format and Rules.)

Only these items can immediately follow a division header:

- Another division header
- A section header
- A paragraph header or paragraph-name
- A comment line
- A blank line
- DECLARATIVES (after the Procedure Division header only)
- PROGRAM-ID (after the Identification Division header only)

1.4.2 Section Header

A section header identifies and marks the beginning of a section in the Environment, Data, and Procedure Divisions. In the Environment and Data Divisions, a section header is a specific combination of reserved words followed by a separator period. In the Procedure Division, a section header is a user-defined word followed by the word SECTION (and an optional segment-number). A separator period always follows a section header. Section headers start in Area A.

The valid section headers follow for each division.

In the Environment Division:

CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.

In the Data Division:

FILE SECTION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.

In the Procedure Division:

user-name SECTION [segment-number].

Only these items can immediately follow a section header:

- A division header
- Another section header
- A paragraph header or paragraph-name
- A comment line
- A USE statement (in the Declaratives part of the Procedure Division only)
- A blank line
- A Data Division entry (in the Data Division)

1.4.3 Paragraph, Paragraph Header, Paragraph-Name

A paragraph consists of a paragraph header or paragraph-name (depending on the division) followed by zero, one, or more entries (or sentences).

A paragraph header is a reserved word followed by a separator period. Paragraph headers identify paragraphs in the Identification and Environment Divisions.

The paragraph headers are:

Identification Division	Environment Division
PROGRAM-ID.	SOURCE-COMPUTER.
AUTHOR.	OBJECT-COMPUTER.
INSTALLATION.	SPECIAL-NAMES.
DATE-WRITTEN.	FILE-CONTROL.
DATE-COMPILED.	I-O-CONTROL.
SECURITY.	

A paragraph-name is a user-defined word followed by a separator period. Paragraph-names identify Procedure Division paragraphs.

Paragraph headers and paragraph-names start in Area A of any line after the first line of a division or section.

The first entry or sentence of a paragraph begins in either:

- On the same line as the paragraph header or paragraph-name
- In Area B of the next nonblank line that is not a comment line

Successive sentences or entries begin in Area B of either:

- The same line as the preceding entry or sentence
- The next nonblank line that is not a comment line

1.4.4 Data Division Entries

A Data Division entry begins with a level indicator or level-number and is followed, in order, by:

1. A space
2. The name of a data item or file connector
3. A sequence of independent descriptive clauses
4. A separator period

The level indicators are:

- FD (for file description entries)
- SD (for sort-merge file description entries)

Level indicators start in Area A.

Entries that begin with level-numbers are called data description entries. The level-number values are 01 through 49, 66, 77, and 88. Level-numbers 01 through 09 can be one- or two-digit numbers.

Level 01 and 77 data description entries begin in Area A. All other data description entries can begin on the first character position of Area B. Further indentation has no effect on level-number magnitude; it merely enhances readability.

1.4.5 Declaratives

Declaratives specify procedures to be executed only when certain conditions occur. You must write declarative procedures at the beginning of the Procedure Division in consecutive sections. The key word DECLARATIVES begins the declaratives part of the Procedure Division; the pair of key words END DECLARATIVES ends it. Each of these reserved word phrases must: (1) be on a line by itself, starting in Area A; and (2) be followed by a separator period.

For example:

```
PROCEDURE DIVISION.
DECLARATIVES.
IOERROR SECTION.
    USE AFTER STANDARD ERROR PROCEDURE ... .
PAR-1.
    .
    .
    .
END DECLARATIVES.
FIRST-ONE SECTION.
PARAG-1.
    .
    .
    .
```

When you use declarative procedures, you must divide the remainder of the Procedure Division into sections.

1.5 Sample Format Entry Page

Most entries in this manual adhere to the format on the following sample page. Each COBOL division or major topic begins a new chapter and each entry begins on a new page.

Entry-Name

Entry-Name

Function

The function paragraph describes the function or the effect of the entry.

General Format

A general format shows the specific arrangement of elements in the entry. If there is more than one arrangement, the formats are numbered. All clauses (mandatory and optional) must be used in the sequence shown in the format. However, the syntax rules sometimes allow exceptions.

generic-term

Following the general format are definitions of its generic terms. These terms are supplied by the programmer and appear in the rules in italics. Restrictions applied to generic terms are equivalent to syntax rules.

Syntax Rules

Syntax rules define or clarify the arrangement of words or elements. They can also impose or relax restrictions implied by the general format. Syntax rule violations are detected at compile time.

General Rules

General rules define or clarify the meaning (or relationship of meanings) of an element or set of elements. They also define the semantics of an entry, describing its effects on program compilation or execution. General rule violations are detected at run time.

Technical Notes

Technical notes describe an entry's effects in system-specific terms. They define relationships between the COBOL program and the operating system, RMS-11, the hardware, and other components in the PDP-11 system.

Additional References

Additional references point to other relevant information in this manual, the COBOL-81 User's Guide for your system, and manuals in the operating system documentation set.

Examples

Examples show the use of a statement, clause, or other entry. The COBOL-81 User's Guide for your system contains examples in application contexts.

Chapter 2
Identification Division

Function

The Identification Division marks the beginning of a COBOL program. It also identifies a program and its source listing.

General Format

```
    IDENTIFICATION DIVISION.
    PROGRAM-ID. program-name.
  [ AUTHOR. [ comment-entry ] ... ]
* [ INSTALLATION. [ comment-entry ] ... ]
* [ DATE-WRITTEN. [ comment-entry ] ... ]
* [ DATE-COMPILED. [ comment-entry ] ... ]
* [ SECURITY. [ comment-entry ] ... ]
```

* These paragraphs are not described in individual entries; they follow the same format as the AUTHOR paragraph and are for documentation only.

Syntax Rules

1. The Identification Division must be the first entry in a COBOL program.
2. The Identification Division must begin with the Identification Division header. The header consists of the reserved words IDENTIFICATION DIVISION followed by a separator period.
3. The PROGRAM-ID paragraph must immediately follow the Identification Division header.

2.1 PROGRAM-ID Paragraph

Function

The PROGRAM-ID paragraph identifies a program.

General Format

PROGRAM–ID. program-name.

Syntax Rules

1. The PROGRAM-ID paragraph must be present in every program.
2. *Program-name* must contain 1 to 30 characters. Only the first six characters of *program-name* are significant to to the compiler.

General Rules

1. *Program-name* is a user-defined word that identifies a COBOL program and its source listing. The first six characters of *program-name* appears as the first word in the first line of every page in the compiler source listing.
2. *Program-name* represents the object program entry point.
3. If an executable image includes more than one separately compiled program, the first six characters of the *program-name* for each separately compiled program must be unique.

Additional References

Chapter 6 COPY Statement

Examples

```
PROGRAM-ID. PROGA.

PROGRAM-ID. JOB6a.

PROGRAM-ID.
    WRITEMASTERREPORT.
```

2.2 AUTHOR Paragraph

Function

The AUTHOR paragraph is for documentation only.

General Format

AUTHOR. [comment-entry] ...

Syntax Rules

1. *Comment-entry* can consist of any combination of characters from the computer character set.
2. *Comment-entries* can span several lines in Area B. However, they cannot be continued by using a hyphen in the indicator area.
3. The end of *comment-entry* is the line before the next entry in Area A.

Examples

```
AUTHOR. STAN GUSSO.

AUTHOR. This program was written by Phil Goodrich
                                    122 Thompson Ln.
                                    Grover Corners, MN

AUTHOR.
```

Chapter 3
Environment Division

Function

The Environment Division describes the program's physical environment. It also specifies input-output control and describes special control techniques and hardware characteristics.

General Format

```
[ ENVIRONMENT DIVISION.

 [ CONFIGURATION SECTION.
 [ SOURCE-COMPUTER.  [ source-computer-entry. ] ]
 [ OBJECT-COMPUTER.  [ object-computer-entry. ] ]
 [ SPECIAL-NAMES.  [ special-names-entry. ] ] ]

 [ INPUT-OUTPUT SECTION.
  FILE-CONTROL.  { file-control-entry. } ...
 [ I-O-CONTROL.  [ input-output-control-entry. ] ] ] ]
```

Syntax Rules

1. The Environment Division follows the Identification Division.
2. The general format defines the order of appearance of Environment Division entries.

3.1 Configuration Section

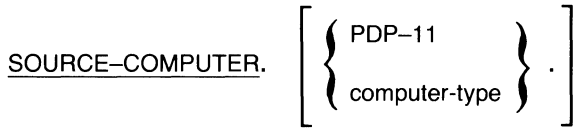
The Configuration Section can contain three paragraphs: SOURCE-COMPUTER, OBJECT-COMPUTER, and SPECIAL-NAMES.

3.1.1 SOURCE-COMPUTER Paragraph

Function

The SOURCE-COMPUTER paragraph specifies the computer on which the source program is to be compiled.

General Format



computer-type
is a user-defined word that names the computer.

Syntax Rule

The word PDP-11 is a system-name. It is not a reserved word.

General Rule

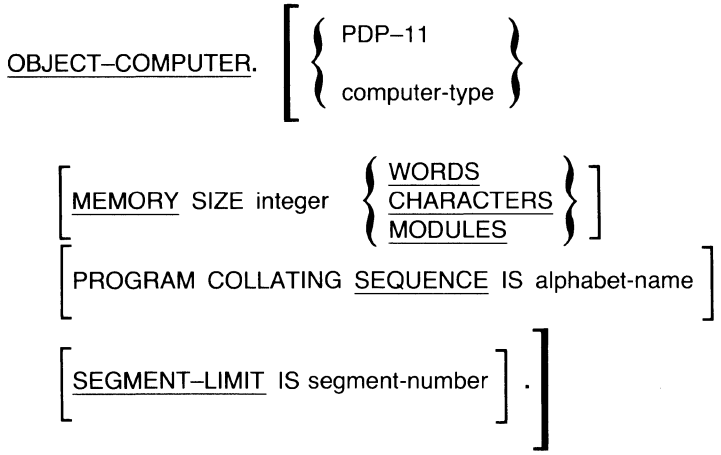
This paragraph is for documentation only.

3.1.2 OBJECT-COMPUTER Paragraph

Function

The OBJECT-COMPUTER paragraph describes the computer on which the program is to execute.

General Format



computer-type
: is a user-defined word that names the computer.

alphabet-name
: the name of a collating sequence defined in the ALPHABET clause of the SPECIAL-NAMES paragraph.

segment-number
: is an integer from 1 through 49.

integer
: is an integer from 1 through 65,535.

Syntax Rules

1. The word *PDP-11* is a system-name. It is not a reserved word and is for documentation only.
2. either *PDP-11* or *computer-type* must be specified if any other OBJECT-COMPUTER clauses appear.

General Rules

1. The MEMORY SIZE clause is for documentation only.
2. The PROGRAM COLLATING SEQUENCE clause causes the program to use the collating sequence of *alphabet-name* to determine the truth value of nonnumeric comparisons in:
   - Relation conditions
   - Condition-name conditions

OBJECT-COMPUTER
Continued

3. The PROGRAM COLLATING SEQUENCE clause also applies to nonnumeric merge and sort keys. However, the COLLATING SEQUENCE phrase in a MERGE or SORT statement takes precedence over the PROGRAM COLLATING SEQUENCE clause.
4. If there is no PROGRAM COLLATING SEQUENCE clause, the program uses the NATIVE collating sequence.
5. The SEGMENT-LIMIT clause determines how a program is overlaid in memory. When the SEGMENT-LIMIT clause is specified, segments with numbers from *segment-number* through 49 are overlaid; that is, they are swapped in and out of a given memory area as needed. Those segments with numbers less than *segment-number* comprise the program "root."

   When the executable image contains only one program, segments in the root always remain in memory.

   When the executable image contains two or more programs (and if at least two programs contain the SEGMENT-LIMIT clause), there is more than one root. In this case, one root is always in memory, but the various roots are overlaid.

Additional References

Section 3.1.3	SPECIAL-NAMES Paragraph
Section 4.2.2	Sort-Merge File Description
Section 5.5.1	Relation Condition
Section 5.5.3	Condition-Name Condition
Section 5.8	Segmentation
Part IV of the COBOL-81 User's Guide for your system	Refer to the chapter on Sorting Records and Merging Files

Examples

1. Computer name only:

```
OBJECT-COMPUTER. PDP-11.
```

2. No computer name (if the computer is not specified, then no other clause can appear):

```
OBJECT-COMPUTER.
```

3. With PROGRAM COLLATING SEQUENCE clause:

   The SPECIAL-NAMES paragraph must define ALPH-A.

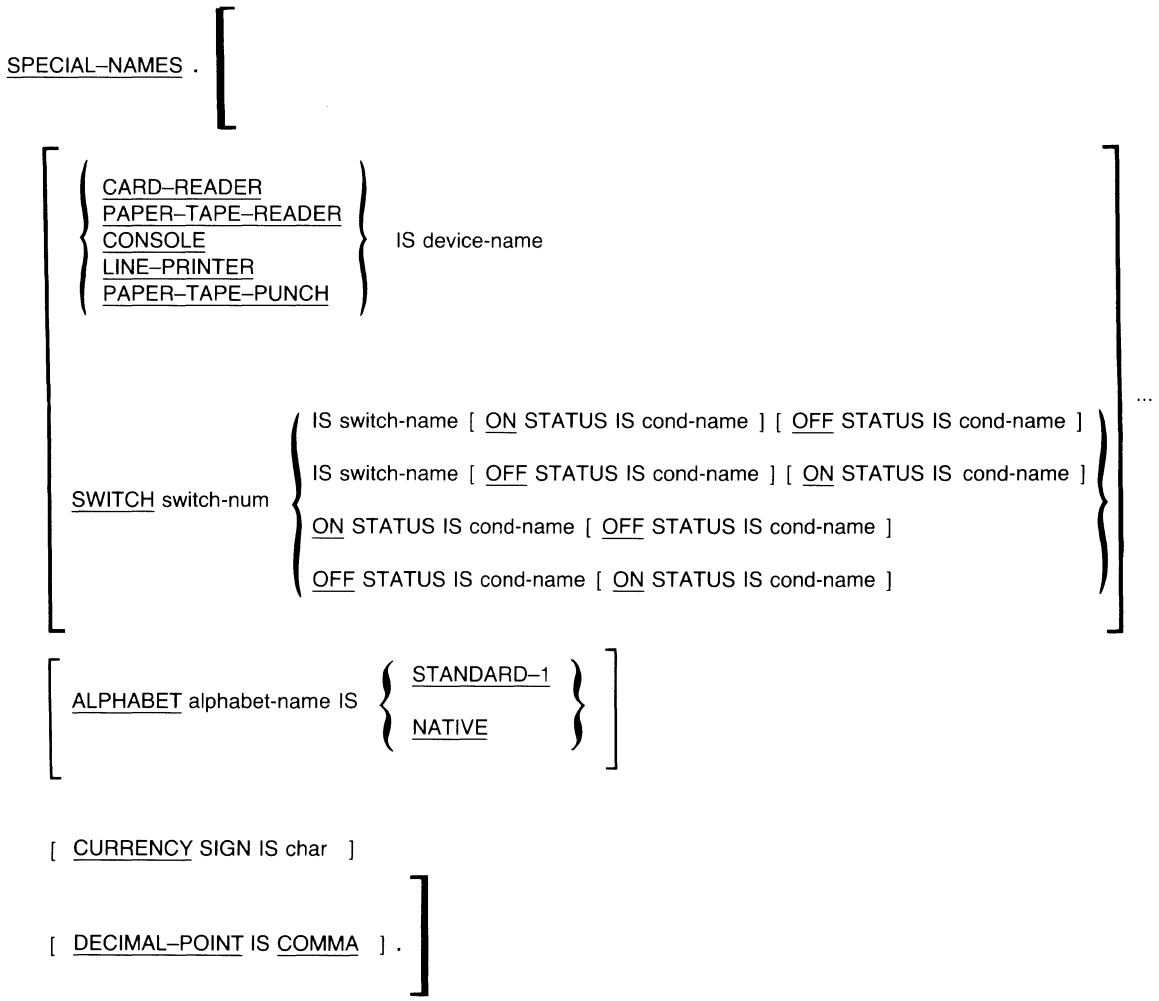
```
OBJECT-COMPUTER. PDP-11
    PROGRAM COLLATING SEQUENCE IS ALPH-A.
```

3.1.3 SPECIAL-NAMES Paragraph

Function

The SPECIAL-NAMES paragraph: (1) associates operating system device names with user-defined mnemonic-names, (2) specifies the currency sign, (3) selects the decimal point, and (4) relates alphabet-names to character sets or collating sequences.

General Format



device-name
: is a mnemonic-name for a device. Only the ACCEPT and DISPLAY statements can refer to it.

switch-num
: is the number of a program switch. Its value can range from 1 through 16.

SPECIAL-NAMES
Continued

switch-name
is a mnemonic-name for the program switch.

cond-name
is a condition-name for the "on" or "off" status of the switch. Its truth value is "true" when the STATUS phrase matches the status of the switch, "false" when it does not.

alphabet-name
is the user-defined word for a character set and/or collating sequence.

char
is a one-character nonnumeric literal that specifies the currency symbol.

General Rules

device-name Clause

1. The *device-name* clause associates an operating system device name with a user-defined mnemonic-name (*device-name*). The COBOL-81 system-names PAPER-TAPE-READER, CARD-READER, CONSOLE, LINE-PRINTER, and PAPER-TAPE-PUNCH act as "connectors" between the *device-names* specified in the program and system devices. Therefore, an ACCEPT or DISPLAY statement that refers to a program specific *device-name* can transfer data from (or to) the device associated with the COBOL-81 system-name.

   The system-names and their default device equivalents are:

System-Name	Device
CARD-READER	CR:
PAPER-TAPE-READER	PR:
CONSOLE	TI:
LINE-PRINTER	LP:
PAPER-TAPE-PUNCH	PP:

Note

By default, CONSOLE is associated with the interactive terminal (TI:) that begins program execution, rather than with the computer console.

SWITCH Clause

2. The ON STATUS (or OFF STATUS) phrase of the SWITCH clause associates the status of *switch-name* with a corresponding *cond-name*. The program uses a switch-status condition in the Procedure Division to test the switch.

ALPHABET Clause

3. The ALPHABET clause relates a name to a character code set, collating sequence, or both.

The ALPHABET clause specifies:

- A character code set, when *alphabet-name* is in a CODE-SET clause in the file description entry.
- A collating sequence, when *alphabet-name* is in: (1) the PROGRAM COLLATING SEQUENCE clause in the OBJECT-COMPUTER paragraph or (2) the COLLATING SEQUENCE phrase of a SORT or MERGE statement

4. STANDARD-1 refers to the ASCII character set. The ASCII character set is defined in *American National Standard X3.4-1968*, "Code for Information Interchange."
5. NATIVE refers to the native character set. It consists of 256 characters. The lowest-valued 128 characters are the ASCII character set. The highest-valued 128 characters are reserved for later standardization and definition by DIGITAL.
6. The character with the highest ordinal position in the program collating sequence equals the figurative constant HIGH-VALUE.
7. The character with the lowest ordinal position in the program collating sequence equals the figurative constant LOW-VALUE.

CURRENCY SIGN Clause

8. In the CURRENCY SIGN clause, *char* specifies the PICTURE clause currency symbol. It can be any printable character from the computer character set except:
   - 0 through 9
   - A, B, C, D, P, R, S, V, X, Z, or the space
   - Asterisk (*), plus sign (+), minus sign (–), comma (,), period (.), semicolon (;), comma (,), quotation mark ("), equal sign (=), or slash (/)
9. The CURRENCY SIGN clause can contain lowercase counterparts of the valid uppercase alphabetic characters. However, lowercase and uppercase alphabetic characters are equivalent in PICTURE character-strings. Therefore, lowercase letters in the CURRENCY SIGN clause cannot match any PICTURE character-string entry.
10. If there is no CURRENCY SIGN clause, the PICTURE clause uses the currency sign ($) as the default.

DECIMAL-POINT IS COMMA Clause

11. The DECIMAL-POINT IS COMMA clause exchanges the functions of the comma and period in: (1) the PICTURE clause character-string and (2) numeric literals.

Additional References

Section 3.1.2	OBJECT-COMPUTER Paragraph
Section 4.2.6	CODE-SET Clause
Section 5.5.4	Switch-Status Condition
Section 5.9.1	ACCEPT Statement
Section 5.9.7	DISPLAY Statement
Appendix B	Computer Character Set

SPECIAL-NAMES
Continued

Examples

1. Device-name clause:

   This example allows ACCEPT and DISPLAY statements to use THE-CARDS to refer to the device CR: and LOCAL-USER to refer to the device TI:.

```
CARD-READER IS THE-CARDS
CONSOLE IS LOCAL-USER
```

2. SWITCH clause:

   (Procedure Division statements can use the condition-names defined in the SWITCH clause. At run time, COBOL-81 prompts you to enter the numbers for the switch(es) you want on during program execution.)

```
SWITCH 1 IS FIRST-SWITCH ON IS ONE-ON OFF IS ONE-OFF
SWITCH 4 ON FOUR-ON
```

   The following results assume that switch 1 is on and switch 4 is off.

Condition	Truth Value
IF FOUR-ON	false
IF ONE-ON	true
IF NOT ONE-OFF	true
IF ONE-ON AND NOT FOUR-ON	true

3. ALPHABET clause:

```
ALPHABET MY-SET IS STANDARD-1.
```

   This clause defines the alphabet named MY-SET to be the ASCII character set.

   In the results of the following examples, the character *s* represents a space. The examples assume these data description entries:

```
01  ITEMA  PIC X(5).
01  ITEMB  PIC X(5).
01  ITEMC  PIC GG,GG9.99.
01  ITEMD  PIC ZZZ,ZZ9,99.
01  ITEME  PIC ZZZ,.
```

4. CURRENCY SIGN clause:

```
CURRENCY SIGN "G"
```

The following MOVE statements show the effect of the CURRENCY SIGN clause:

Statement	ITEMC Value
MOVE 12.34 TO ITEMC	sssG12.34
MOVE 100 TO ITEMC	ssG100.00
MOVE 1000 TO ITEMC	G1,000.00

5. DECIMAL-POINT IS COMMA clause:

Statement	Result
MOVE 1 TO ITEMD	ITEMD = ssssss1,00
MOVE 1000 TO ITEMD	ITEMD = ss1.000,00
MOVE 1,1 TO ITEMD	ITEMD = ssssss1,10
MOVE 12 TO ITEME	ITEME = s12,

3.2 INPUT-OUTPUT SECTION

The INPUT-OUTPUT Section can contain two paragraphs: FILE-CONTROL and I-O-CONTROL.

3.2.1 FILE-CONTROL Paragraph

Function

The FILE-CONTROL paragraph contains file-related specifications.

General Format

FILE–CONTROL.

Format 1 – Sequential File

```
SELECT [ OPTIONAL ] file-name
    ASSIGN TO file-spec
[   RESERVE reserve-num [ AREA  ] ]
                        [ AREAS ]
[ [ ORGANIZATION IS ] SEQUENTIAL ]
[ ACCESS MODE IS SEQUENTIAL ]
[ FILE STATUS IS file-stat ] .
```

Format 2 – Relative File

```
SELECT file-name
    ASSIGN TO file-spec
[   RESERVE reserve-num [ AREA  ] ]
                        [ AREAS ]
[ ORGANIZATION IS ] RELATIVE
[   ACCESS MODE IS { SEQUENTIAL [ RELATIVE KEY IS rel-key ] }
                   { { RANDOM  }                            }
                   { { DYNAMIC } RELATIVE KEY IS rel-key    } ]
[ FILE STATUS IS file-stat ] .
```

(continued on next page)

Format 3 – Indexed File

```
SELECT file-name

    ASSIGN TO file-spec
[   RESERVE reserve-num [ AREA  ] ]
                        [ AREAS ]

[ ORGANIZATION IS ] INDEXED

[ ACCESS MODE IS { SEQUENTIAL } ]
                 { RANDOM     }
                 { DYNAMIC    }

[ RECORD KEY IS rec-key ]
[ ALTERNATE RECORD KEY IS alt-key [ WITH DUPLICATES ] ] ...
[ FILE STATUS IS file-stat ] .
```

Format 4 – Sort or Merge File

```
SELECT file-name ASSIGN TO file-spec .
```

file-name
: names a file within your COBOL program. Each *file-name* must have a file description (or sort-merge file description) entry in the Data Division. The same *file-name* cannot appear more than once in the FILE-CONTROL paragraph.

Syntax Rules

All Formats

1. The FILE-CONTROL paragraph must have at least one SELECT clause.
2. SELECT must be the first clause in the FILE-CONTROL paragraph. The other clauses can follow it in any order.
3. Each file described in the Data Division must be specified only once in the FILE-CONTROL paragraph.

Format 1

4. You can specify the OPTIONAL phrase only for input files.

General Rules

Format 1

1. You must specify an OPTIONAL phrase for input files that need not be present when the program runs.

Formats 2 and 3

2. The rules for the OPEN statement describe the effects of the OPTIONAL phrase.

Additional Reference

Section 5.9.17 OPEN Statement

Examples

The following examples assume that the VALUE OF ID clause is not in any associated file description entry.

1. Sequential file:

   (This SELECT clause refers to two files with sequential organization.)

```
SELECT FILE-A
  ASSIGN TO "REPORT".
SELECT FILE-B
  ASSIGN TO "UPDATE".
```

2. Indexed file:

```
SELECT FILE-A
  ASSIGN TO "DK1:ALUMNI.DAT"
  ORGANIZATION INDEXED
  ACCESS IS DYNAMIC
  RECORD KEY IS STUDENT-NUM.
```

3. Sort or merge file:

```
SELECT INPUT-FILE
  ASSIGN TO "DK1:MAILST.DAT".
```

3.2.1.1 ACCESS MODE Clause

Function

The ACCESS MODE clause specifies the order of access for a file's records.

General Format

Format 1 – Sequential File

```
[ACCESS MODE IS] SEQUENTIAL
```

Format 2 – Relative File

```
                                  ⎧ SEQUENTIAL [ RELATIVE KEY IS rel-key ] ⎫
[ACCESS MODE IS] ⎨ ⎧ RANDOM  ⎫    ⎩                                        ⎭
                 ⎩ ⎩ DYNAMIC ⎭    RELATIVE KEY IS rel-key
```

Format 3 – Indexed File

```
[ACCESS MODE IS] ⎧ SEQUENTIAL ⎫
                 ⎨ RANDOM     ⎬
                 ⎩ DYNAMIC    ⎭
```

rel-key
is the file's relative key data item.

Syntax Rules

1. *Rel-key* must be the data-name of an unsigned integer data item whose description does not contain a PICTURE symbol "P". *Rel-key* can be qualified.
2. If the USING or GIVING phrase of a SORT or MERGE statement contains the name of the file, the ACCESS MODE RANDOM clause cannot be used for the file.
3. If a START statement references a relative file, the program must specify the RELATIVE KEY phrase for that file.

ACCESS MODE
Continued

General Rules

All Formats

1. If there is no ACCESS MODE clause, the access mode is sequential.
2. For sequential access, record access sequence depends on file organization:
   - Sequential files – The sequence is the same as that established by the execution of WRITE statements that created or extended the file.
   - Relative files – The sequence is the order of ascending relative record numbers of the file's existing records.
   - Indexed files – The sequence is the order of ascending record key values within a given key of reference according to the collating sequence of the file.

Formats 2 and 3

3. For random access, the value of *rel-key* (for relative files) or a record key data item (for indexed files) indicates the record to be accessed.
4. For dynamic access, the program can access records sequentially and randomly.

Format 2

5. Relative record numbers uniquely identify records in relative files. A record's relative record number identifies its ordinal position in the file. The first record in the file has a relative record number of 1. Subsequent records have progressively higher relative record numbers. However, if the file is created with random access, the numbers need not be consecutive (for example, "1,2,4,7,8,9,11").
6. The relative key data item associated with the execution of an input/output statement is *rel-key* in the SELECT clause of the file associated with the statement.

3.2.1.2 ALTERNATE RECORD KEY Clause

Function

The ALTERNATE RECORD KEY clause specifies an alternate access path to indexed file records.

General Format

```
ALTERNATE RECORD KEY IS alt-key [ WITH DUPLICATES ]
```

alt-key
is the alternate record key for the file. It is the data-name of a data item in the file's record description entry. The data item must be described as: (1) alphanumeric or alphabetic or (2) a group item.

Syntax Rules

1. *Alt-key* can be qualified; however, it cannot be subscripted or indexed.
2. *Alt-key* cannot be a group item that contains a variable-occurrence data item.
3. *Alt-key* cannot have the same leftmost character position as that of the prime record key data item or any other *alt-key* for the same file.

General Rules

1. When a program creates an indexed file with one or more ALTERNATE RECORD KEY clauses, each subsequent program referencing this indexed file must:
   - Use the same data description for *alt-key*
   - Define the same relative location in the record as *alt-key*
   - Specify the same number of ALTERNATE RECORD KEY clauses
   - Maintain the same order of ALTERNATE RECORD KEY clauses
2. The DUPLICATES phrase specifies that two or more records in the file can have duplicate values in the same *alt-key* data item. If there is no DUPLICATES phrase, two records cannot have the same value in corresponding alternate record keys.
3. If the file has more than one record description entry, you need to describe *alt-key* in only one of those entries. The character positions referenced by *alt-key* in any one record description entry are implicitly referenced as an alternate key for all other record description entries of that file.
4. A file can have up to 254 alternate record keys.

3.2.1.3 ASSIGN Clause

Function

The ASSIGN clause associates a file with a partial or a complete file specification.

General Format

ASSIGN TO file-spec

file-spec
is a nonnumeric literal that provides a partial or a complete file specification.

General Rules

1. If there is no VALUE OF ID clause in the file description entry, or that clause contains no file specification, *file-spec* is the file specification.
2. If there is a full or a partial file specification in an associated VALUE OF ID clause, those file specification components will override *file-spec*.
3. *File-spec* can contain a logical name.

Technical Note

When an OPEN statement executes, PDP-11 Record Management Services (RMS-11):

- Removes leading and trailing spaces and tab characters from the file specification
- Translates lowercase letters in the file specification to uppercase
- Performs logical name translation

Additional Reference

Section 4.2.22 VALUE OF ID Clause

3.2.1.4 FILE STATUS Clause

Function

The FILE STATUS clause names a data item that contains the status of an input-output operation.

General Format

```
FILE STATUS IS file-stat
```

file-stat
Working-Storage Section or Linkage Section. *File-stat* is the file's FILE STATUS data item.

Syntax Rule

File-stat can be qualified.

General Rule

After execution of every I-O statement that refers to the file, a value is moved to *file-stat*. That value indicates the statement's execution status.

Additional References

Section 5.7 I-O Status
Appendix C File Status Values

ORGANIZATION

3.2.1.5 ORGANIZATION Clause

Function

The ORGANIZATION clause specifies a file's logical structure.

General Format

```
[ ORGANIZATION IS ] { SEQUENTIAL }
                    { RELATIVE   }
                    { INDEXED    }
```

General Rules

1. File organization is fixed when the file is created. It cannot be subsequently changed.
2. If there is no ORGANIZATION clause, the default is sequential.

3.2.1.6 RECORD KEY Clause

Function

The RECORD KEY clause specifies the primary access path to indexed file records.

General Format

RECORD KEY IS rec-key

rec-key
is the data-name of a data item in a record description entry for the file. The data item must be described as: (1) alphanumeric or alphabetic or (2) a group item.

Syntax Rules

1. *Rec-key* can be qualified.
2. *Rec-key* cannot be a group item that contains a variable-occurrence data item.

General Rules

1. The RECORD KEY clause specifies the prime record key for a file.
2. The values of the prime record key cannot be duplicated in the file's records.
3. The data description of *rec-key*, and its relative location in the record, must be the same as those used when the file was created.
4. If the file has more than one record description entry, you need to describe *rec-key* in only one of those entries. The character positions referenced by *rec-key* in any one record description entry are implicitly referenced as the prime record key for all other record description entries of that file.

Additional Reference

Section 3.2.1.2 ALTERNATE RECORD KEY Clause

RESERVE

3.2.1.7 RESERVE Clause

Function

The RESERVE clause specifies the number of input-output buffers for a file.

General Format

```
RESERVE reserve-num [ AREA  ]
                    [ AREAS ]
```

reserve-num
is an integer literal from 1 through 127. It specifies the number of input-output areas for the file.

General Rule

If there is no RESERVE clause, the number of input-output areas defaults to:

- One, for sequential files
- One, for relative files
- Two, for indexed files

Additional References

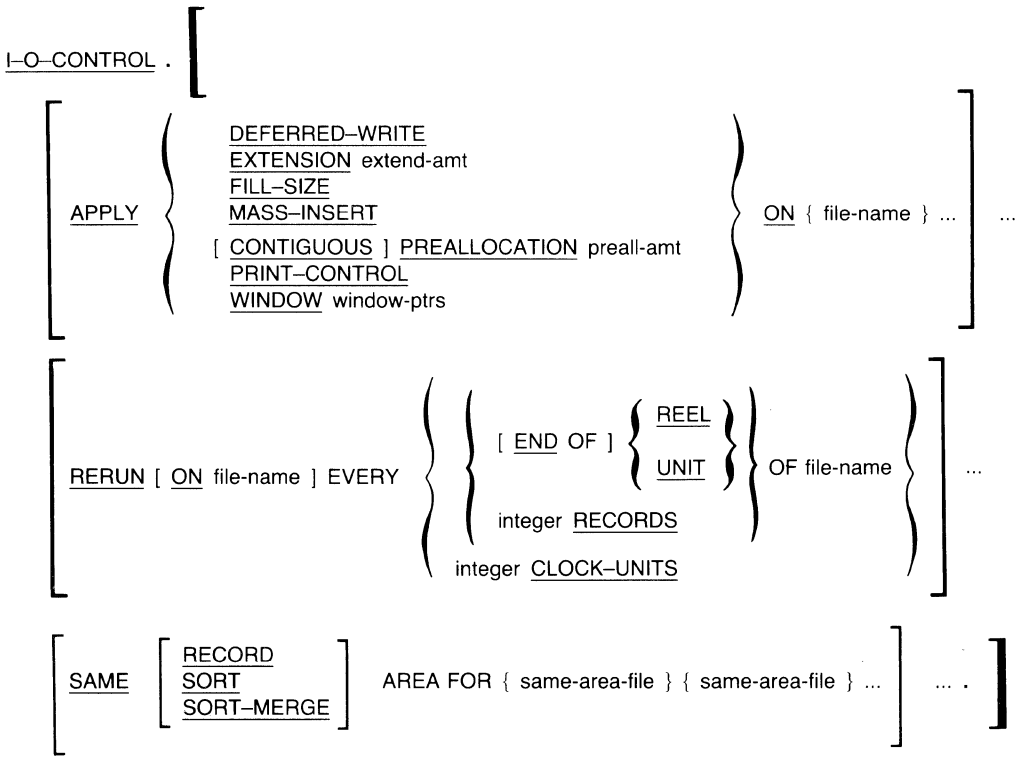
Section 3.2.2 APPLY Clause

3.2.2 I-O-CONTROL Paragraph

Function

The I-O-CONTROL paragraph specifies the input-output techniques to be used for a file.

General Format



extend-amt
: is an integer from 0 through 65535. It specifies the number of blocks in each extension of a disk file.

preall-amt
: is an integer from 0 through 2,147,483,647. It specifies the number of blocks to allocate when the program creates a disk file.

window-ptrs
: is an integer whose permissible values are dependent on your operating system. See Technical Note 7.

file-name
: names a file described in a Data Division file description entry.

same-area-file
: names a file described in a Data Division file description entry to share storage areas with every other *same-area-file*.

I-O-CONTROL
Continued

Syntax Rules

1. The I-O-CONTROL clauses can appear in any order.
2. Each phrase of the APPLY clause can refer only to some file types:

Phrase	File Type
DEFERRED-WRITE	Relative or indexed organization
EXTENSION	Disk file
FILL-SIZE	Indexed organization
MASS-INSERT	Indexed organization
PREALLOCATION	Disk file
PRINT-CONTROL	Sequential organization
WINDOW	Disk file

3. More than one APPLY clause can refer to the same *file-name*.
4. The phrases of the APPLY clause can appear in any order. However, each phrase can be used only once for each *file-name*.
5. In the SAME AREA clause, SORT and SORT-MERGE are equivalent.
6. If *same-area-file* refers to a sort or merge file, you must use the SORT, SORT-MERGE, or RECORD phrase.
7. A program can contain more than one SAME clause. However, the following conditions apply:
   - A *same-area-file* cannot be in more than one SAME AREA clause.
   - A *same-area-file* cannot be in more than one SAME RECORD AREA clause.
   - A *same-area-file* that refers to a sort or merge file cannot be in more than one SAME SORT AREA or SAME SORT-MERGE AREA clause.
   - If any file in a SAME AREA clause appears in a SAME RECORD AREA clause, *all* files in the SAME AREA clause must appear in the SAME RECORD AREA clause. In addition, other files that are not in that SAME AREA clause can appear in the SAME RECORD AREA clause.

     The rule that only one file in a SAME AREA clause can be open at a time takes precedence over the rule that more than one file in a SAME RECORD AREA clause can be open at once.
   - If a file that is not a sort or merge file appears in a SAME AREA clause and also in one or more SAME SORT AREA or SAME SORT-MERGE AREA clauses, *all* files in the SAME AREA clause must appear in the SAME SORT AREA or SAME SORT-MERGE AREA clauses.

General Rules

APPLY Clause

1. The DEFERRED-WRITE phrase causes a physical write operation to occur only when the input-output buffer for *file-name* is full. If there is no DEFERRED-WRITE phrase, a physical write occurs each time an output statement executes for *file-name*. The DEFERRED-WRITE phrase applies only to relative and indexed files.

2. The EXTENSION phrase specifies the number of disk blocks to be added each time a file is extended. RMS-11 extends a file when it needs more file space to add a record.

   If *extend-amt* equals zero, RMS-11 extends the file by its default value.

3. The FILL-SIZE phrase causes RMS-11 to use the fill size specified when the file was created to fill the file's buckets. If there is no FILL-SIZE phrase, RMS-11 fills buckets completely. The FILL-SIZE phrase applies only to indexed files.

4. The MASS-INSERT phrase optimizes the addition of records to an indexed file. However, optimization occurs only if the records are in ascending order by prime record key.

5. The PREALLOCATION phrase causes RMS-11 to allocate *preall-amt* disk blocks when it creates the file.

   The CONTIGUOUS phrase specifies that the preallocated disk blocks must be contiguous.

   If RMS-11 cannot find *preall-amt* (or contiguous *preall-amt*) disk blocks, the open fails.

6. The PRINT-CONTROL phrase specifies that the file has print file format and it applies only to sequentially organized files.

   The PRINT-CONTROL phrase is redundant if: (1) the file description entry contains a LINAGE clause, or (2) the program contains a WRITE statement with the ADVANCING phrase for the file. However, the PRINT CONTROL phrase is required for print files on magnetic tape.

7. The WINDOW phrase specifies the technique RMS-11 uses to map your file. See Technical Note 7 for a discussion of its effect and the permissible values it can contain.

SAME AREA Clause

8. The SAME AREA clause causes two or more files named by *same-area-file* to use the same input-output buffer.

9. If you specify the SAME AREA clause, only one *same-area-file* can be open at one time.

SAME RECORD AREA Clause

10. The SAME RECORD AREA clause causes two or more files named by *same-area-file* to share the same memory area for the current logical records.

11. If you specify the SAME RECORD AREA clause, more than one *same-area-file* (or all of them) can be open at the same time.

12. Any record in the shared area becomes the current logical record of:

    - Each *same-area-file* of the SAME RECORD AREA clause open in OUTPUT mode
    - The most recently read *same-area-file* of the SAME RECORD AREA clause open in INPUT mode

    The logical records start with the same leftmost character position. Thus, the SAME RECORD AREA clause is equivalent to an implicit redefinition of the shared area.

SAME SORT (SORT-MERGE) AREA Clause

In the following rules, the terms SORT, sort, and sort file also imply SORT-MERGE, merge, and merge file.

13. At least one *same-area-file* in the SAME SORT AREA clause must be a sort file.
14. The SAME SORT AREA clause causes two or more sort files named by *same-area-file* to use the same memory area.
15. Files other than sort files do not share the same storage area unless their names are in a SAME AREA or SAME RECORD AREA clause.
16. No other *same-area-file* can be open during the execution of a SORT statement that refers to any *same-area-file*.

RERUN Clause

17. The RERUN clause is for documentation only. It has no effect on program execution.

Technical Notes

The following notes describe the effects of APPLY clause phrases on parameters in the File Access Block (FAB) and Record Access Block (RAB) associated with *file-name*. Descriptions of FAB and RAB fields are in the *RMS-11 Macro Programmer's Guide*.

1. The DEFERRED-WRITE phrase sets the DFW bit in the FOP field of the FAB.
2. The EXTENSION phrase stores *extend-amt* in the DEQ field of the FAB.
3. The FILL-SIZE phrase sets the LOA bit in the ROP field of the RAB.
4. The MASS-INSERT phrase sets the MAS bit in the ROP field of the RAB.
5. The PREALLOCATION phrase stores *preall-amt* in the ALQ field of the FAB.

   The CONTIGUOUS phrase sets the CTG bit in the FOP field of the FAB.
6. The PRINT-CONTROL phrase has no effect on FAB parameters. (For print control files, COBOL-81 uses variable length records with embedded print control characters, rather than a specific record format.)
7. The WINDOW phrase stores *window-ptrs* in the RTV field of the FAB. The effect of the WINDOW phrase varies according to your operating system.

   On an RSX-11M/M-PLUS system, *window-ptrs* overrides the default window size. The value of *window-ptrs* must fall in the range 0 to 127 inclusive, or be equal to 255.

   On a RSTS/E system, *window-ptrs* overrides the default clustersize. Its value can be 0, a power of two, or 255.

Additional References

Section 3.2.1.7	RESERVE Clause
Section 5.9.17	OPEN Statement
Section 5.9.19	READ Statement
Section 5.9.22	REWRITE Statement
Section 5.9.26	START Statement
Section 5.9.32	WRITE Statement
Part IV of the COBOL-81 User's Guide for your system	Refer to the chapter on file optimization techniques

Chapter 4
Data Division

This chapter first discusses the logical and physical concepts that apply to the Data Division. It then presents the general formats for all Data Division entries and clauses, describes their basic elements, and lists applicable rules of use.

4.1 Data Division Concepts

The Data Division defines the data processed by your COBOL program in both physical and logical terms. It also specifies whether the data is contained in files or is developed only for local use in your program.

The File Section of your program defines data contained in files. A *file description* or a *sort-merge file description entry* creates a logical reference to a file. It also can contain clauses that define physical file characteristics. A file description or sort-merge file description entry must be associated with at least one *record description entry*, which logically defines a set of related data within the file. A record description entry is a set of one or more *data description entries*, organized in a hierarchical structure. The data description entries themselves specify all the data used in your program. You logically define the record hierarchy by the level numbers you use for the data description entries (or entry). Your logical link to a record or to a field in a record is the data-name you assign in a corresponding data description entry. The clauses in a data description entry also specify physical data attributes, such as storage format and initial values.

The Working-Storage and Linkage Sections also contain data description entries, which describe characteristics of data developed for use in your program.

The following sections explain in more detail how a COBOL program specifies physical and logical characteristics. It shows how record descriptions impose logical structures on data, and how the physical attributes of data affect the way it is stored and manipulated.

4.1.1 Logical Concepts

Because a record description is a logical, rather than a physical structure, a program can define more than one record description for the same file. However, this redefinition does not mean that the physical data changes in any way. Multiple record descriptions for a file all apply to one physical data unit on the file medium.

When you refer to a data-name in a COBOL source statement, you are referring to a *logical* unit, either a logical record or a logical subset of that record. When your COBOL source statements execute, the logical units to which they refer are mapped to physical units on media. The logical units are then manipulated according to their physical attributes.

The correspondence between a logical record and a physical record is not necessarily a one-to-one correspondence. The term *physical record* applies to a data unit that is media dependent and defined by PDP-11 Record Management Services (RMS-11). A logical record can correspond to one physical record, either alone or grouped with other logical records. Or, at least on disk, a logical record could need more than one physical record to contain it.

Several COBOL clauses (in the Environment and Data Divisions) describe the relationships between logical records and physical records. Programs can then access data as logical entities with little regard to the physical data definitions that RMS-11 requires.

4.1.1.1 Record Description — Logical records do not have to be subdivided; however, they often are. Subdivision can continue for each of the record's parts, allowing progressively more detailed data definition.

The basic subdivision of a record is the *elementary data item* (or *elementary item*), which you define by specifying a PICTURE clause. As the term implies, elementary items are never subdivided. A logical record consists of one or more sets of elementary items, or is itself an elementary item.

A *group data item* (or *group item*) is a data set within a record that contains other subordinate data items. The lowest-level group item is always a named sequence of one or more elementary items. Group items can combine to form more inclusive group items. Therefore, an elementary item can be subordinate to more than one group item in the record.

Figure 4-1 represents a personnel record that illustrates how elementary and group items can be related in a record hierarchy. The record contains three group items directly subordinate to the top level: Identification Data, History, and Payroll Data. The first group item, Identification Data, directly contains two elementary items, Name and Job Title, and two other group items, Employee Number and Address. The group item, Employee Number, contains two elementary items: Department Code and Badge Number. The group item, Address, contains four elementary items: Street, City, State, and ZIP Code. The elementary item, City, belongs to three group items. It is subordinate to Address, Identification Data, and Personnel Record. The second group item, History, directly contains three elementary items: Hire Date, Last Promotion, and Termination Date. The third group item, Payroll Data, also directly contains two elementary items: Current Salary and Previous Salary.

4.1.1.2 Level-Numbers — Record description entries use a system of level-numbers to specify the hierarchical organization of elementary and group items. Level-numbers that specify hierarchical structure can range from 01 through 49.

The record is the most inclusive data item; that is, there is no hierarchical relationship between one record description entry and any other. However, there is a hierarchical relationship between a group item and its subordinate group or elementary items. The level-number for records is 01. Less inclusive data items have greater (although not necessarily consecutive) level-numbers.

All items subordinate to a group item must have level-numbers greater than the group's level-number. In a record description, a group item is delimited by the next subsequent level number that is less than or equal to that group's level number.

Figure 4-1: Hierarchical Record Structure

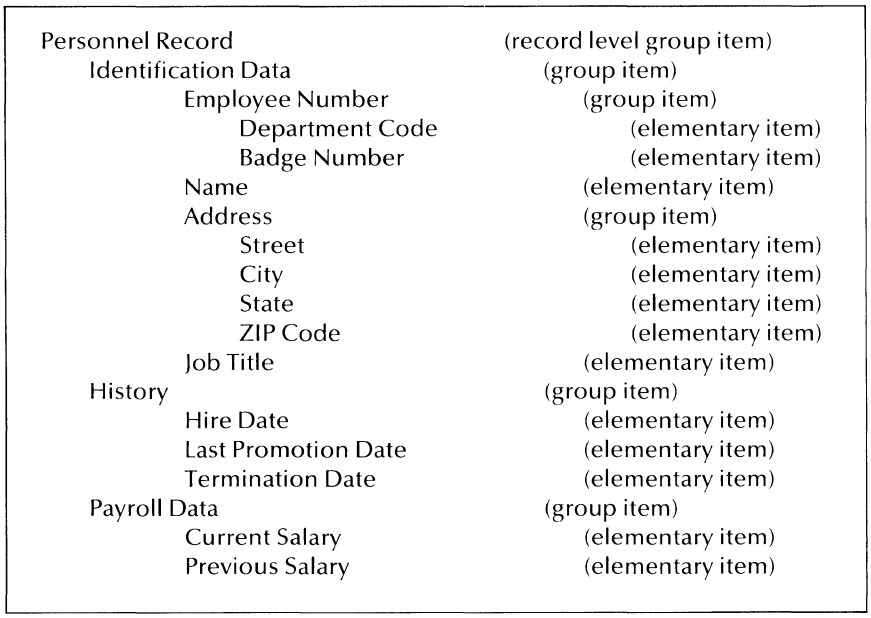


Figure 4-2 shows how level-numbers specify hierarchical structure and how the presence of the PICTURE clause defines an elementary item. Although line indentation can make record descriptions easier to read, it does not affect record structure; only the level-number values specify the hierarchy. The ellipsis (...) indicates that parts of the program line have been omitted.

Figure 4-2: Level-Number Record Structure

```
01  PERSONNEL-RECORD.                                  (record)
    03  IDENTIFICATION-DATA.                           (group item)
        05  EMPLOYEE-NUMBER.                           (group item)
            07  DEPARTMENT-CODE        PIC ...         (elementary item)
            07  BADGE-NUMBER           PIC ...         (elementary item)
        05  NAME                       PIC ...         (elementary item)
        05  ADDRESS.                                   (group item)
            07  STREET                 PIC ...         (elementary item)
            07  CITY                   PIC ...         (elementary item)
            07  STATE                  PIC ...         (elementary item)
            07  ZIP-CODE               PIC ...         (elementary item)
        05  JOB-TITLE                  PIC ...         (elementary item)
    03  HISTORY.                                       (group item)
        04  HIRE-DATE                  PIC ...         (elementary item)
        04  LAST-PROMOTION-DATE        PIC ...         (elementary item)
        04  TERMINATION-DATE           PIC ...         (elementary item)
    03  PAYROLL-DATA.                                  (group item)
        05  CURRENT-SALARY             PIC ...         (elementary item)
        05  PREVIOUS-SALARY            PIC ...         (elementary item)
```

Three special level-numbers – 66, 77, and 88 – neither specify hierarchical structure nor actually indicate level. Rather, they define special types of data entries:

- Level-number 66 identifies RENAMES items, which regroup other data items. (See Section 4.2.17.)

- Level-number 77 specifies noncontiguous (elementary) items in the Working-Storage and Linkage Sections. These data items are not subdivisions of other items and cannot themselves be subdivided. For all other purposes, they are identical to level 01 elementary entries.
- Level-number 88 associates condition-names with values of a corresponding data item (the conditional variable). (See Section 1.1.2.1, Condition-Name.)

4.1.1.3 Multiple Record Definitions — Example 4-1 shows a sample file description entry (FD) that contains three record description entries. It defines three logical templates the program can impose on a record to access data from it.

Example 4-1: Multiple Record Definition Structure

```
FD  MASTER-FILE.
01  T1.
    02  T1-ACCOUNT-NO              PIC 9(6).
    02  T1-TRAN-CODE               PIC 99.
    02  T1-NAME                    PIC X(13).
    02  T1-BALANCE                 PIC 9(5)V99.
    02  REC-TYPE                   PIC XX.
01  T2.
    02  T2-ACCOUNT-NO              PIC 9(6).
    02  T2-ADDRESS.
        03  T2-STREET              PIC X(15).
        03  T2-CITY                PIC X(7).
    02  REC-TYPE                   PIC XX.
01  RECORD-TYPE.
    02                             PIC X(28).
    02  REC-TYPE                   PIC XX.
```

T1, T2, and RECORD-TYPE each define a record of 30 characters. Once the program reads a record, it can use the last two characters (REC-TYPE) to determine which record description to use. (Example 4-1 defines a fixed length record. However, if the example had defined a variable length record, REC-TYPE would have been the first field in the record – not the last.)

4.1.2 Physical Concepts

COBOL programs describe files and data in physical terms for storage on input-output media. The physical description of data includes:

- The mapping and grouping of logical records within the structure of the file medium
- The unit used to transfer records to and from your program
- The size and storage format of an elementary data item

The size and recording mode of a physical record depend on the hardware device involved in an input or output operation. For example, tape and disk media store physical records differently. On tape, a physical record is written between interrecord gaps. On disk, a physical record is written in multiples of 512-byte units.

On DIGITAL systems, the term used for a physical record differs according to file organization. A physical record in a sequential file is called a *block*. A physical record in a relative or indexed file is called a *bucket*. A block (or bucket) corresponds to the unit used by RMS-11 (Record Management Services) to transfer records from a file to your program (and vice versa). The number of records (in logical terms) actually transferred by an input-output operation depends upon:

- The block size specified by the BLOCK CONTAINS clause
- The number of logical records contained in a physical record

When your COBOL program executes a READ or WRITE statement, it can affect more than one record. Whether it does or not depends upon file media defaults and how your program uses the BLOCK CONTAINS clause. For this reason, it is important to keep in mind the correspondence between logical and physical records when balancing input-output optimization against file sharing needs. Refer to Part IV of the COBOL-81 User's Guide for your system for more information on file and record processing.

The size and storage format of an elementary data item depends upon what class and category of data it represents and how that data can be used. A data item's PICTURE clause determines its class and category. The item's PICTURE clause and USAGE clause, in combination, specify its size and storage format. (See Section 4.2.14 and Section 4.2.20.)

When an arithmetic or data-movement statement transfers data into an elementary item, the category of the item affects the way the data is positioned in storage. The COBOL Standard Alignment Rules specify the relationship between category and positioning.

The following sections discuss categories and classes of data, the Standard Alignment Rules, and record storage format in greater detail.

4.1.2.1 Categories and Classes of Data — Depending on the symbols that its PICTURE clause contains, a data item belongs to one of the following five categories:

- Alphabetic
- Alphanumeric
- Alphanumeric edited
- Numeric
- Numeric edited

These categories are grouped into three classes:

- Alphabetic
- Alphanumeric
- Numeric

Every elementary item, except an index data item or an index-name, belongs to one of these classes and categories. (Index data items and index-names are the only elementary items for which you do not specify a PICTURE clause. They therefore cannot belong to a category).

The class of a group item is treated as alphanumeric regardless of the class of elementary items subordinate to it. Therefore, the statements in your program should not specify a group item when a numeric item is expected or required.

Table 4-1 shows the relationship of classes and categories of data items.

Table 4-1: Classes and Categories of Data Items

Level	Class	Category
Elementary	Alphabetic	Alphabetic
	Numeric	Numeric
	Alphanumeric	Numeric Edited Alphanumeric Edited Alphanumeric
Group	Alphanumeric	Alphabetic Numeric Numeric Edited Alphanumeric Edited Alphanumeric

4.1.2.2 Standard Alignment Rules — The Standard Alignment Rules specify how characters are positioned in an elementary item. Positioning depends on the item's category:

1. For a numeric receiving data item:
   - The data is aligned by decimal point. It is moved to the receiving character positions with zero fill or truncation, if necessary.
   - When an assumed decimal point is not explicitly specified, the data item is treated as if it had an assumed decimal point immediately after its rightmost character. It is aligned as in the preceding paragraph.
2. For a numeric edited receiving data item, the data is aligned by decimal point with zero fill or truncation, if necessary. Editing requirements can replace leading zeros with some other symbol.
3. For receiving data items that are alphabetic, alphanumeric edited, or alphanumeric (without editing), the data is aligned at the leftmost character position in the data item, with space fill or truncation to the right, if necessary.

If the JUSTIFIED clause applies to the data item, the rules for the JUSTIFIED clause override rule 3. (See Section 4.2.9.)

As stated earlier, the Standard Alignment Rules specify data positioning only within elementary items. DIGITAL defines other alignment rules that affect the positioning of: (1) records on the file media, (2) group items within the record, and (3) elementary items within a group item. The following section discusses these additional alignment rules.

4.1.2.3 Record Allocation — The byte is the smallest addressable storage unit in PDP-11 memory, and each byte can have an even or an odd address. When referring to data alignment, the even address is often called the even byte, word, or 2-byte boundary. The following discussion will use the term "2-byte boundary" to refer to even addresses.

As the COBOL-81 compiler allocates storage for data, it tries to locate each data item at the next unassigned byte (either odd or even address) for greatest storage efficiency. This method of storage allocation is sometimes called the left-to-right technique. However, the compiler must also apply alignment rules that require certain data items to align on 2-byte boundaries or, in a few cases, to align on larger byte offsets from a record boundary. The compiler may then have to "skip" one or more bytes before assigning a location to one of these data items. The skipped bytes, called fill bytes, are spaces that precede data items in a file. Fill bytes can precede data items described in your program as:

1. A record description entry for a file at level 01 (Relative locations of records in the Working-Storage and Linkage Sections are neither defined nor predictable.)
2. A binary data item (An elementary item whose description contains a COMP or INDEX usage clause.)
3. A group item containing any elementary items that have special alignment requirements
4. A data item that does not require special alignment itself, but that has been redefined by another item with special alignment requirements

Unless you write the SYNCHRONIZED clause for COMP data items, the compiler inserts no more than one fill byte between data items. However, more than one fill byte might precede COMP SYNC items in your program.

Note

The SYNCHRONIZED clause ensures that COMP items in data files can be read by a VAX–11 COBOL program. For more information, refer to Appendix D, Ensuring Compatibility between COBOL-81 and VAX–11 COBOL.

Figure 4-3 shows alignment boundaries for a record. The boundary is the leftmost location of the one-, two-, four-, or eight-byte area. All boundaries for items within a record are relative to the record's beginning. Therefore, alignment of the record as a unit has no effect on data access.

Figure 4-3: Record Alignment Boundaries

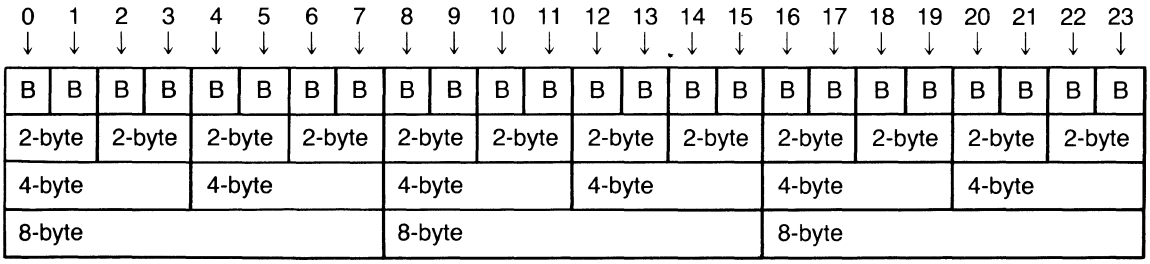


Table 4-2 shows the required boundaries for data items needing special alignment. The asterisk (*) indicates items you use to automatically ensure compatibility with VAX–11 COBOL.

Table 4-2: Data Items Requiring Alignment

Data Type	Required Boundary
INDEX	2-byte
PIC 9 to 9(4) COMP	2-byte
* PIC 9 to 9(4) COMP SYNC	2-byte
PIC 9(5) to 9(9) COMP	2-byte
* PIC 9(5) to 9(9) COMP SYNC	4-byte
PIC 9(10) to 9(18) COMP	2-byte
* PIC 9(10) to 9(18) COMP SYNC	8-byte

Because fill bytes *precede* a data item in storage, they have no effect on the size of elementary data items being aligned. However, they can increase the size of group items containing the specially aligned data items.

The presence of fill bytes makes a group item's storage structure different from what you might expect, based on a simple count of characters. If the group item contains many subordinate items requiring alignment, its size can increase significantly. If you are unaware of fill bytes and try to move a group item containing them into an elementary data item, right-end truncation will occur. You do not have this problem, however, if you move the group item into another identically described group item. This is because the compiler applies additional alignment rules to ensure that identically described group items have the same storage structure. Therefore, group moves always produce predictable results.

The compiler allocates storage for group items using the Major-Minor Equivalence Technique. This technique is based on two alignment rules:

1. Location Equivalence - the leftmost location of a group item must be the same as the leftmost location of its first subordinate item.
2. Boundary Equivalence - a group item must align on the same type of boundary as the most restrictive boundary required by its subordinate items or items that redefine the group.

4.1.2.4 Location Equivalence — Location equivalence forces a group (major) item to the same storage location as its first subordinate (minor) item. Consider this example:

```
01   ITEM-A.
     03   ITEM-B.
          05   ITEM-C PIC 9(4) COMP.
     03   ITEM-D PIC X.
     03   ITEM-E.
          05   ITEM-F PIC 9(4) COMP.
     03   ITEM-G PIC X.
```

Figure 4-4 compares data item alignment using location equivalence with alignment using a left-to-right technique. The left-to-right technique assigns locations to allocation a group item *before* assigning locations to the group's subsidiary items. However, location equivalence first

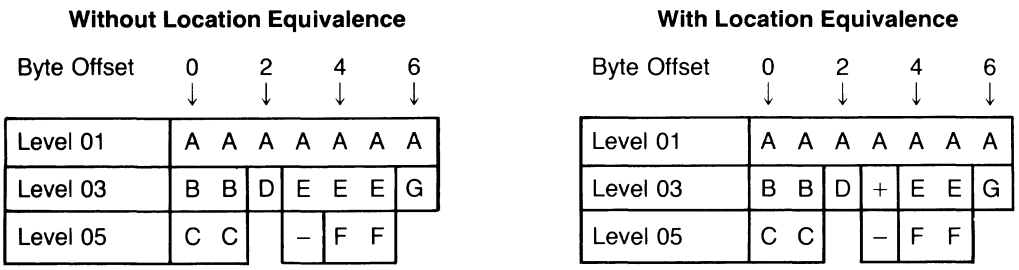
assigns locations to the subsidiary items. Note that the location equivalence rule adds a fill byte before ITEM-E. This forces ITEM-E to align on the same boundary as ITEM-F, which is implicitly aligned because it is a COMP data item.

Figure 4-4: Data Alignment Requirements Without and With Location Equivalence

Data Item	Without Location Equivalence	With Location Equivalence
ITEM-A	00	00
ITEM-B	00	00
ITEM-C	00	00
ITEM-D	02	02
ITEM-E	03	04
ITEM-F	04	04
ITEM-G	06	06

Figure 4-5 shows the storage allocation for the record ITEM-A using both techniques. The symbol "–" indicates the fill byte caused by the alignment requirement for COMP items. The symbol "+" represents the fill byte resulting from location equivalence. Any even offset number indicates a 2-byte boundary. Level 01 data items provide reference points for boundary calculations and are therefore aligned on "0".

Figure 4-5: Record Allocation Without and With Location Equivalence



Regardless of the record allocation technique, an elementary move such as:

```
MOVE ITEM-C TO ITEM-F
```

always produces the expected result. However, a group move could cause different results. Using the previous example, consider the statement:

```
MOVE ITEM-B TO ITEM-E.
```

Without location equivalence, the structures of ITEM-B and ITEM-E are not the same because the alignment of their subordinate items is different. The leftmost location of ITEM-E (03) is one byte to the left of its subordinate item, ITEM-F. Therefore, the contents of ITEM-C do not align properly with ITEM-F. The first byte of ITEM-C moves to the fill byte before ITEM-F, and the contents of ITEM-C and ITEM-F are not the same.

With location equivalence, the group move produces the expected result because the structures of ITEM-B and ITEM-E are identical.

When referring to Figure 4-5, note that the fill byte inserted by alignment requirements only increases the size of the group item, ITEM-A (01). The fill byte does not increase the size of the group item, ITEM-E (03), because ITEM-F (05) is its first subordinate item.

4.1.2.5 Boundary Equivalence — Boundary equivalence forces a group item to align on the same type of boundary as the most restrictive boundary required by any item in its scope. Its scope includes any items that:

- Are subordinate to the group
- Redefine the group
- Are subordinate to a data item that redefines the group

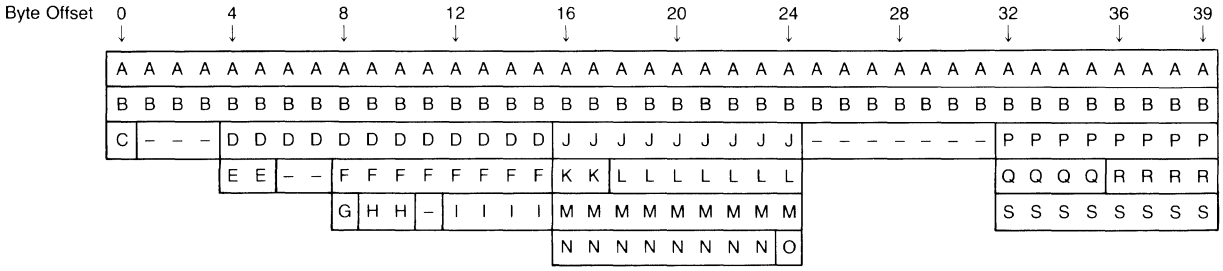
Figure 4-6 shows how the compiler determines the boundary where each item begins.

Figure 4-6: Effect of Boundary and Location Equivalence Rules on Sample Record

	Boundary	Reason
01 ITEM-A.	–	Level 01 is reference point
03 ITEM-B.	8-byte	Contains ITEM-J
05 ITEM-C PIC X.	byte	Default alignment
05 ITEM-D.	4-byte	Contains ITEM-F
07 ITEM-E PIC 9(4) COMP SYNC.	2-byte	Explicit SYNC clause
07 ITEM-F.	4-byte	Contains ITEM-I
09 ITEM-G PIC X.	byte	Default alignment
09 ITEM-H PIC 9(4).	byte	Default alignment
09 ITEM-I PIC 9(8) COMP SYNC.	4-byte	Explicit SYNC clause
05 ITEM-J.	8-byte	Redefined by ITEM-M which contains ITEM-N
07 ITEM-K PIC 9(1) COMP SYNC.	2-byte	Explicit SYNC clause
07 ITEM-L PIC X(7).	byte	Default alignment
05 ITEM-M REDEFINES ITEM-J.	8-byte	Contains ITEM-N
07 ITEM-N PIC 9(15) COMP SYNC.	8-byte	Explicit SYNC clause
07 ITEM-O PIC X.	byte	Default alignment
05 ITEM-P.	8-byte	Redefined by ITEM-S
07 ITEM-Q PIC 9(5) COMP.	2-byte	Binary data item
07 ITEM-R PIC 9(7) COMP.	2-byte	Binary data item
05 ITEM-S REDEFINES ITEM-P PIC 9(15) COMP SYNC.	8-byte	Explicit SYNC clause

Figure 4-7 graphically represents the preceding example. It shows the result of location and boundary equivalence applied to the description of record ITEM-A. The symbol "–" indicates fill bytes.

Figure 4-7: Storage Allocation for Sample Record



Note the location of ITEM-D. Location equivalence requires only that ITEM-D have the same location as ITEM-E, its first subordinate item. ITEM-E requires only two-byte boundary alignment. However, another of ITEM-D's subordinate items, ITEM-F, contains ITEM-I, which must be aligned on a four-byte boundary. Therefore, boundary equivalence forces ITEM-D to a four-byte boundary as well, causing two fill bytes between ITEM-E and ITEM-F.

This example shows how boundary equivalence helps make group moves predictable:

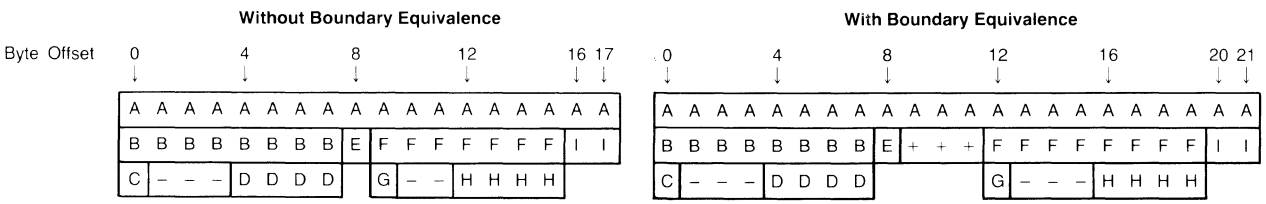
```
01     ITEM-A.
    03     ITEM-B.
        05     ITEM-C     PIC X.
        05     ITEM-D     PIC 9(8) COMP SYNC.
    03     ITEM-E         PIC X.
    03     ITEM-F.
        05     ITEM-G     PIC X.
        05     ITEM-H     PIC 9(8) COMP SYNC.
    03     ITEM-I         PIC XX.
```

The descriptions of ITEM-B and ITEM-F are equivalent. Therefore, you would not expect the following sentence to change the values of ITEM-C and ITEM-D:

```
MOVE ITEM-B TO ITEM-F
MOVE ITEM-F TO ITEM-B.
```

Figure 4-8 shows how storage for the record would be allocated without and with boundary equivalence. The symbol "–" indicates fill bytes caused by the SYNCHRONIZED clause. The symbol "+" represents fill bytes resulting from boundary equivalence.

Figure 4-8: Record Allocation Without and With Boundary Equivalence



Without boundary equivalence, ITEM-B occupies eight bytes, and ITEM-F occupies seven bytes. Moving the contents of ITEM-B to ITEM-F truncates the last byte of ITEM-D. Moving the contents of ITEM-F to ITEM-B pads the last byte of ITEM-D with a space character.

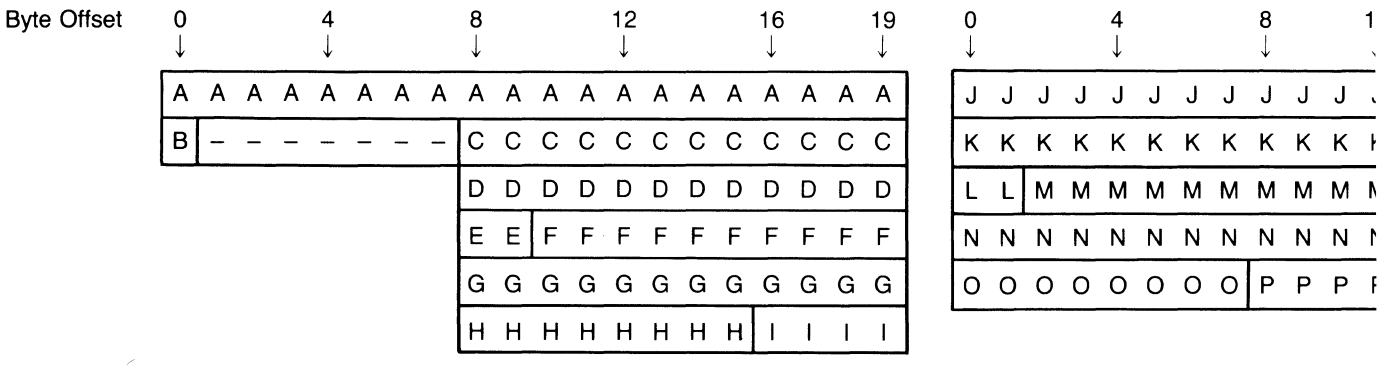
In contrast, boundary equivalence eliminates this unforeseen result. The elementary items occupy the same relative positions in each group. Therefore, the structures of ITEM-B and ITEM-F are the same, and the results of both group and elementary moves are predictable.

Examples

This series of examples shows major-minor storage allocation. The notes after each example indicate its significant features. The symbol "–" represents fill bytes.

Example 1

```
WORKING-STORAGE SECTION.
01    ITEM-A.
    03    ITEM-B    PIC X.
    03    ITEM-C.
        05    ITEM-D.
            07    ITEM-E    PIC 999 COMP SYNC.
            07    ITEM-F    PIC X(10).
        05    ITEM-G REDEFINES ITEM-D.
            07    ITEM-H    PIC 9(14) COMP SYNC.
            07    ITEM-I    PIC XXXX.
01    ITEM-J.
    03    ITEM-K.
        05    ITEM-L    PIC 999 COMP SYNC.
        05    ITEM-M    PIC X(10).
    03    ITEM-N REDEFINES ITEM-K.
        05    ITEM-O    PIC 9(14) COMP SYNC.
        05    ITEM-P    PIC XXXX.
```

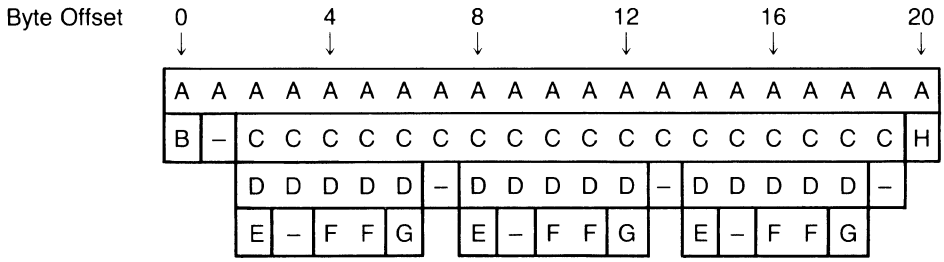


Note:

The structures of ITEM-J (a record) and ITEM-C (a group item within a record) are identical because of location and boundary equivalence rules.

Example 2

```
WORKING-STORAGE SECTION.
01    ITEM-A.
    03    ITEM-B              PIC X.
    03    ITEM-C.
        05    ITEM-D OCCURS 3 TIMES.
            07    ITEM-E      PIC X.
            07    ITEM-F      PIC 9999 COMP SYNC.
            07    ITEM-G      PIC X.
    03    ITEM-H              PIC X.
```

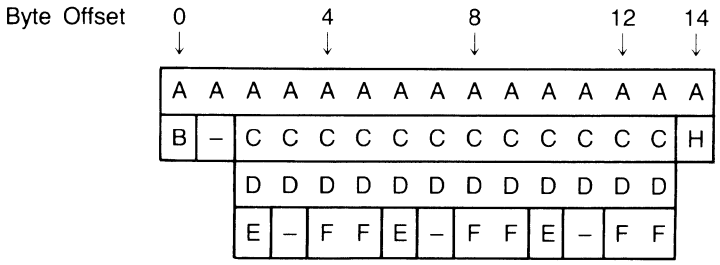


Notes:

1. ITEM-D naturally falls on an odd byte boundary. Therefore, a fill byte precedes all but the first occurrence of ITEM-D to maintain 2-byte boundary alignment of the item.
2. ITEM-D is five bytes long. The fill byte preceding ITEM-D is not included in its length.
3. ITEM-C is 18 bytes long. Its length includes the three fill bytes associated with ITEM-D, as well as the three fill bytes associated with ITEM-F.
4. The record ITEM-A is 21 bytes long. Its length includes the fill bytes associated with ITEM-C, ITEM-D, and ITEM-F.

Example 3

```
WORKING-STORAGE SECTION.
01    ITEM-A.
    03    ITEM-B               PIC X.
    03    ITEM-C.
        05    ITEM-D OCCURS 3 TIMES.
            07    ITEM-E      PIC X.
            07    ITEM-F      PIC 9999 COMP SYNC.
    03    ITEM-H              PIC X.
```



Notes:

1. This example is the same as Example 2 except that it omits ITEM-G.
2. ITEM-D is four bytes long. No fill bytes precede it, since its next occurrence is already aligned on a two-byte boundary.
3. ITEM-C is 12 bytes long. It includes only the three fill bytes associated with ITEM-F.
4. The record ITEM-A is 15 bytes long. It includes four fill bytes – the one fill byte associated with ITEM-C and the three fill bytes associated with ITEM-F.

Data Division
Format Entry Pages

4.2 DATA DIVISION General Format and Rules

Function

The DATA DIVISION describes data the program creates, receives as input, manipulates, and produces as output.

General Format

```
[ DATA DIVISION.

[ FILE SECTION.
      [ file-description-entry { record-description-entry } ... ] ...
      [ sort-merge-file-description-entry { record-description-entry } ... ] ... ]

[ WORKING-STORAGE SECTION.
      [ record-description-entry ] ... ]

[ LINKAGE SECTION.
      [ record-description-entry ] ... ] ]
```

Syntax Rules

1. The Data Division follows the Environment Division.
2. The reserved words DATA DIVISION, followed by a separator period, identify and begin the Data Division.

General Rules

1. The Data Division has three sections. These sections must be in this order:

   FILE SECTION.
   WORKING-STORAGE SECTION.
   LINKAGE SECTION.

File Section

2. The File Section defines the structure of data files. It begins with the File Section header: the reserved words FILE SECTION, followed by a separator period.
3. File description entries and sort-merge file description entries follow the File Section header. They can be in any order.
4. The file description entry consists of a level indicator (FD), a file-name, and a series of independent clauses.

DATA DIVISION General Format and Rules
Continued

5. FD clauses specify: (1) how the file records data; (2) the sizes of logical and physical records, and (3) the names of data records.
6. An FD entry, followed by one or more record description entries, defines a sequential, relative, or indexed file. Record description entries must immediately follow the associated FD entry.
7. The sort-merge file description entry consists of a level indicator (SD), a file-name, and a series of independent clauses.
8. SD clauses specify: (1) how the file records data, (2) the sizes of logical and physical records, and (3) the names of data records.
9. An SD entry specifies the sizes and names of data records for a sort-merge file referred to by SORT and MERGE statements.
10. An SD entry, followed by one or more record description entries, defines a file. Record description entries must immediately follow the associated SD entry.

Working-Storage Section

11. The Working-Storage Section describes records and subordinate data items. These records are not parts of files; rather, the program develops and processes them internally.
12. The Working-Storage Section also describes data items assigned values by the source program.
13. The Working-Storage Section consists of a section header, followed by record description entries.
14. The section header consists of the reserved words WORKING-STORAGE SECTION, followed by a separator period.
15. A record description entry groups data items that bear a hierarchical relationship to each other. Unrelated data items in the Working-Storage Section can be described as records that are individual elementary items.
16. Record description clauses can be used in the File Section, the Working-Storage Section, or the Linkage Section.
17. The VALUE IS clause can specify the initial value of any item in the Working-Storage Section except index data items (described by the USAGE IS INDEX clause) and index-names (described by the INDEXED BY phrase of the OCCURS clause).
18. If the VALUE IS clause does not specify an initial value, the default initial value for an item is undefined. However, index-names are preset to occurrence number one.

Linkage Section

19. The Linkage Section is only in a called program.
20. The Linkage Section describes data available through the calling program; both the calling and called programs can access this data.
21. To access calling program data items through the Linkage Section, the called program must have a Procedure Division header USING phrase.
22. The structure of the Linkage Section is the same as that of the Working-Storage Section. It consists of a section header followed by record description entries. The section header consists of the reserved words LINKAGE SECTION followed by a separator period.
23. The VALUE IS clause cannot appear in the Linkage Section except in condition-name entries (level 88). The initial value of Linkage Section items is defined by the calling program in each call.

Additional References

Section 1.1.2.1	User-defined Words
Section 4.2.21	VALUE IS Clause
Section 5.3.3	CALL Statement

FD (File Description Entry for Sequential, Relative, and Indexed Files)

4.2.1 FD (File Description) Complete Entry Skeleton

Function

A file description entry describes the physical structure, identification, and record names for sequential, relative, and indexed files.

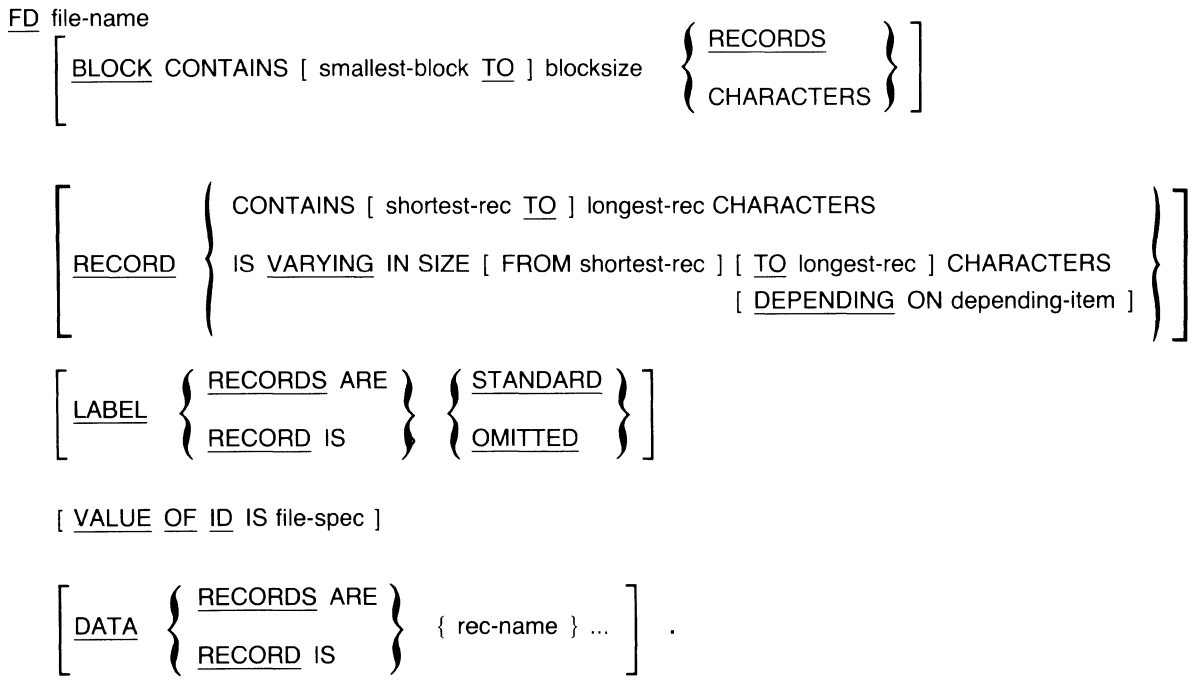
General Formats

Format 1 – Sequential File

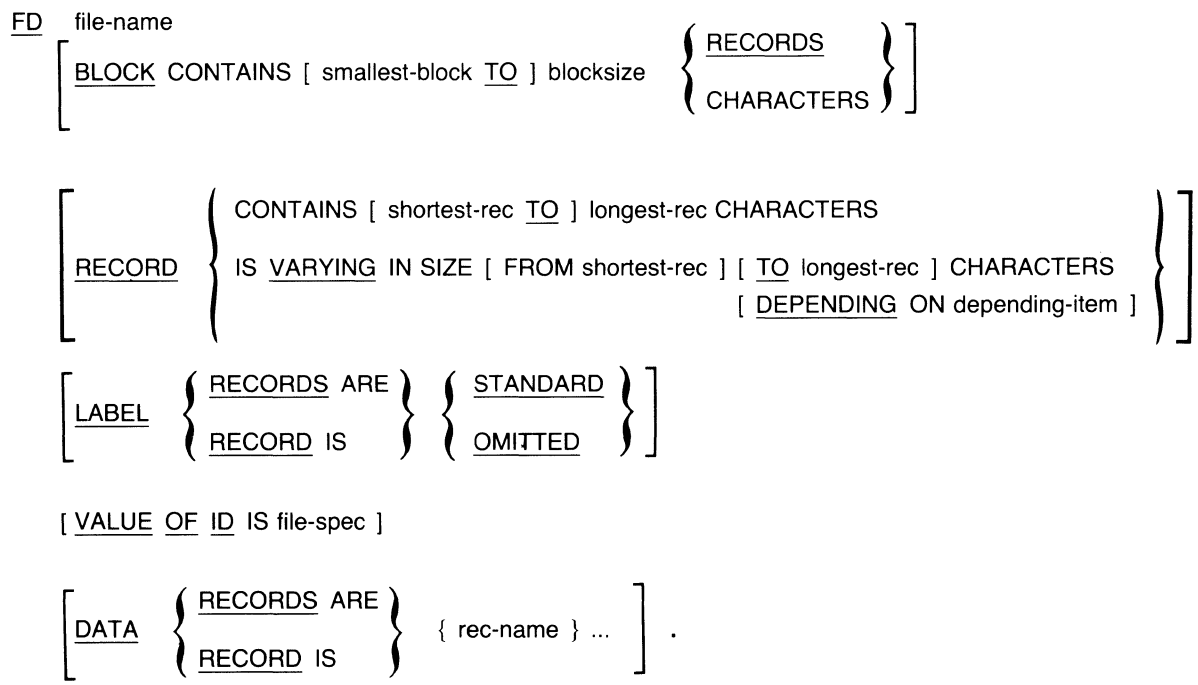
```
FD file-name

[ BLOCK CONTAINS [ smallest-block TO ] blocksize { RECORDS      } ]
                                                 { CHARACTERS }

[ RECORD { CONTAINS [ shortest-rec TO ] longest-rec CHARACTERS                                  } ]
         { IS VARYING IN SIZE [ FROM shortest-rec ] [ TO longest-rec ] CHARACTERS               }
         {                                          [ DEPENDING ON depending-item ]             }

[ LABEL { RECORDS ARE } { STANDARD } ]
        { RECORD IS   } { OMITTED  }

[ VALUE OF ID IS file-spec ]

[ DATA { RECORDS ARE } { rec-name } ... ]
       { RECORD IS   }

[ LINAGE IS { page-size } LINES [ WITH FOOTING AT footing-line ]
                                [ LINES AT TOP top-lines ] [ LINES AT BOTTOM bottom-lines ] ]

[ CODE-SET IS alphabet-name ] .
```

(continued on next page)

Format 2 – Relative File



Format 3 – Indexed File



FD
Continued

Syntax Rules

Formats 1, 2, and 3

1. The level indicator FD identifies the start of a file description entry. It must precede *file-name*.
2. The clauses following *file-name* can appear in any order.
3. A separator period must terminate a file description entry.

Format 1

4. *File-name* can refer only to a sequential file.
5. One or more record description entries must follow the file description entry.

Format 2

6. *File-name* can refer only to a relative file.
7. If a START statement refers to *file-name*, the file description must include the RELATIVE KEY phrase within the ACCESS MODE clause.
8. One or more record description entries must follow the file description entry.

Format 3

9. *File-name* can refer only to an indexed file.
10. One or more record description entries must follow the file description entry.

General Rule

A file description entry associates *file-name* with a file connector.

Examples

Part IV of the COBOL-81 User's Guide for your system contains examples of each file description entry format.

4.2.2 SD (Sort-Merge File Description) Complete Entry Skeleton

Function

A sort-merge file description entry describes a sort or merge file's physical structure, identification, and record names.

General Format

```
SD  file-name

[ RECORD { CONTAINS [ shortest-rec TO ] longest-rec CHARACTERS                          } ]
         { IS VARYING IN SIZE [ FROM shortest-rec ] [ TO longest-rec ] CHARACTERS       }
                                                    [ DEPENDING ON depending-item ]

[ DATA { RECORDS ARE } { rec-name } ... ] .
       { RECORD IS   }
```

Syntax Rules

1. The level indicator SD identifies the start of a sort-merge file description. It must precede *file-name*.
2. The clauses following *file-name* can appear in any order.
3. A separator period must terminate a sort-merge file description entry.
4. One or more record description entries must follow the sort-merge file description entry.

General Rule

No input-output statements can refer to a *file-name* in a sort-merge file description.

Examples

The COBOL-81 User's Guide for your system (Part IV, Processing Files and Records) contains examples of the sort-merge file description entry.

4.2.3 Data Description – Complete Entry Skeleton

Function

A data description entry specifies the characteristics of a data item.

General Formats

Format 1

```
level-number   [ data-name ]
               [ FILLER   ]

  [ REDEFINES other-data-item ]

  [ { PICTURE } IS character-string ]
  [ { PIC    }                      ]

  [ [ USAGE IS ] { COMPUTATIONAL   } ]
                 { COMP            }
                 { COMPUTATIONAL-3 }
                 { COMP-3          }
                 { DISPLAY         }
                 { INDEX           }

  [ [ SIGN IS ] { LEADING  } [ SEPARATE CHARACTER ] ]
                { TRAILING }

  [ OCCURS table-size TIMES

      [ { ASCENDING  } KEY IS { key-name } ... ] ...
        { DESCENDING }

      [ INDEXED BY { ind-name } ... ]

    OCCURS min-times TO max-times TIMES DEPENDING ON depending-item

      [ { ASCENDING  } KEY IS { key-name } ... ] ...
        { DESCENDING }

      [ INDEXED BY { ind-name } ... ]           ]
```

(continued on next page)

```
[ { SYNCHRONIZED } [ LEFT  ] ]
  { SYNC         }   RIGHT ]

[ { JUSTIFIED } RIGHT ]
  { JUST      }

[ BLANK WHEN ZERO ]

[ VALUE IS lit ] .
```

Format 2

```
66  new-name RENAMES rename-start [ { THRU      } rename-end ] .
                                    { THROUGH }
```

Format 3

```
88  condition-name { VALUE IS  } { low-val [ { THRU    } high-val ] } ... .
                   { VALUES ARE }           { THROUGH }
```

Syntax Rules

1. *Level-number* in Format 1 can be any number from 01 through 49, or 77.
2. Data description clauses can appear in any order, with two exceptions:
   - The optional *data-name* or FILLER clause must immediately follow *level-number*.
   - The optional REDEFINES clause must immediately follow the optional *data-name* or FILLER clause.
3. There must be a PICTURE clause for all elementary items except: (1) an index data item and (2) the subject of a RENAMES clause. (In these cases, there must be no PICTURE clause.)
4. The words THRU and THROUGH are equivalent.
5. The SYNCHRONIZED, PICTURE, JUSTIFIED, and BLANK WHEN ZERO clauses can appear only in data description entries for elementary items.

DATA DESCRIPTION
Continued

General Rules

1. Each *condition-name* requires a separate Format 3 entry. The level 88 entry associates one or more values, or ranges of values, with *condition-name*.

   All *condition-name* entries for an associated data item (the conditional variable) must immediately follow that item's data description entry.

   A *condition-name* can be associated with a data item at any level except:

   - Another *condition-name*
   - A level 66 item
   - A group that contains items with JUSTIFIED, SYNCHRONIZED, or USAGE (other than USAGE IS DISPLAY) clauses
   - An index data item

2. Multiple level 01 data description entries subordinate to an FD or SD entry implicitly redefine the same area.

4.2.4 BLANK WHEN ZERO Clause

Function

The BLANK WHEN ZERO clause replaces zeros with spaces when a data item's value is zero.

General Format

```
BLANK WHEN ZERO
```

Syntax Rules

1. The BLANK WHEN ZERO clause can be used only for a numeric or numeric edited elementary item.
2. A BLANK WHEN ZERO data item must be implicitly or explicitly described with DISPLAY usage.
3. The BLANK WHEN ZERO clause cannot be used for a data item that has an asterisk (*) in its PICTURE string.

General Rules

1. The BLANK WHEN ZERO clause causes an item to contain spaces when its value is zero.
2. When the data item has a numeric PICTURE string, the BLANK WHEN ZERO clause makes the item's category numeric edited.

BLOCK CONTAINS

4.2.5 BLOCK CONTAINS Clause

Function

The BLOCK CONTAINS clause specifies the size of a physical record.

General Format

```
BLOCK CONTAINS [ smallest-block TO ] blocksize { RECORDS    }
                                               { CHARACTERS }
```

smallest-block
: is an integer literal. It specifies the minimum physical record size and must be less than *blocksize*.

blocksize
: is an integer literal. It specifies the exact or maximum physical record size.

General Rules

1. The BLOCK CONTAINS clause specifies the physical record size.
2. At run time, COBOL-81 ignores *smallest-block*.
3. The RECORDS phrase specifies the physical record size in terms of logical records.
   - For a magnetic tape file with fixed length records, each physical record except the last contains *blocksize* records.
   - For a magnetic tape file with variable length records, the compiler computes the physical record size. It equals the size of the largest logical record, plus any overhead bytes, multiplied by *blocksize*.
   - For a disk file with sequential organization, there are no unused bytes in any physical record. Logical records can span physical record boundaries.
   - For a disk file with relative or indexed organization, the compiler uses *blocksize* to compute the size of the physical record. Because of the overhead bytes required by Record Management Services (RMS-11), the size can differ from record size multiplied by *blocksize*.
4. The CHARACTERS phrase specifies physical record size in terms of characters.
   - For files assigned to magnetic tape, the physical record size is the larger of: (1) *blocksize* bytes, or (2) the size of the largest logical record; plus any overhead bytes for variable length records. *Blocksize* must be a multiple of four.
   - For sequential disk files, there are no unused bytes in any physical record. Logical records can span physical record boundaries.
   - For relative and indexed files, the physical record size is *blocksize* bytes. *Blocksize* must be at least as large as the largest logical record, plus any overhead bytes. It should be a multiple of 512.

5. If there is no BLOCK CONTAINS clause, the physical record size assumes a default value.
   - For a magnetic tape file, the physical record size is the size of the largest logical record plus any overhead bytes.
   - For a sequential disk file, there are no unused bytes in any physical record. Logical records can span physical record boundaries.
   - For a relative or indexed file, the physical record size is the smallest number of 512-byte units that can contain at least one logical record (including any overhead bytes).
6. For files assigned to magnetic tape, the size of physical records (in characters) must be a multiple of four. Otherwise, RMS-11 rounds up the physical record size to the next multiple of four.

4.2.6 CODE-SET Clause

Function

The CODE-SET clause specifies the representation of data on external media.

General Format

```
CODE-SET IS alphabet-name
```

alphabet-name
: is the name of a character set defined in the ALPHABET clause of the SPECIAL-NAMES paragraph.

Syntax Rule

The CODE-SET clause applies only to files with sequential organization.

General Rule

COBOL-81 ignores the CODE-SET clause during input-output operations. Because both NATIVE and STANDARD-1 (in the ALPHABET clause) describe the ASCII character set, no character conversion can occur.

Additional Reference

Section 3.1.3 SPECIAL-NAMES Paragraph

4.2.7 Data-Name Clause

Function

Data-name specifies a data item that your program can explicitly reference. FILLER specifies an item that cannot be explicitly referenced.

General Format

```
[ data-name ]
[ FILLER    ]
```

Syntax Rule

In the File, Working-Storage, and Linkage Sections, *data-name* or the key word FILLER (if present) must be the first word after the level-number in each data description entry.

General Rules

1. If there is no data-name or FILLER clause, the compiler treats the data item as a FILLER item.
2. The key word FILLER can name a data item. However, a program cannot explicitly refer to FILLER items.
3. The key word FILLER can name a conditional variable. A program cannot refer to the conditional variable. However, it can refer to the value of the conditional variable by referring to its associated condition-names.

Examples

1. Elementary FILLER items:

   In this example, the program can refer only to the group item, ITEMA.

```
01  ITEMA.
    03  FILLER PIC X(10) VALUE SPACES.
    03  PIC X(2) VALUE "AB".
    03  PIC 9 VALUE 6.
```

2. Group FILLER items:

   In this example, the program can refer to any elementary item. However, it cannot refer to the record or to the group item that contains ITEMC and ITEMD.

```
01  FILLER.
    03  ITEMA      PIC X(4).
    03  ITEMB      PIC 9(7).
    03  FILLER.
        05  ITEMC PIC X.
        05  ITEMD PIC 9(8)V99.
    03  ITEME      PIC X.
```

4.2.8 DATA RECORDS Clause

Function

The DATA RECORDS clause documents the names of a file's record description entries.

General Format

rec-name
: is the name of a data record. It must be defined by a level 01 data description entry subordinate to the file description entry.

Syntax Rule

The order of appearance of multiple *rec-name* entries is not significant.

General Rule

The DATA RECORDS clause is for documentation only.

4.2.9 JUSTIFIED Clause

Function

The JUSTIFIED clause specifies nonstandard data positioning in an alphanumeric receiving item.

General Format

Syntax Rules

1. The JUSTIFIED clause can be used only for elementary items.
2. The JUSTIFIED clause cannot be used for: (1) index data items; (2) numeric data items; or (3) edited data items. It can be used only for alphanumeric data items.
3. JUST is an abbreviation for JUSTIFIED.

General Rules

1. If a COBOL statement transfers data to a receiving item whose data description contains the JUSTIFIED clause, the COBOL-81 Object Time System:
   - Truncates the excess leftmost characters if the sending item is larger than the receiving item
   - Aligns the data at the rightmost character position of the receiving item if the sending item is smaller than the receiving item (Spaces fill the excess leftmost character positions.)
2. If there is no JUSTIFIED clause, data movement follows the rules for aligning data in elementary items (Standard Alignment Rules).

Additional References

Section 4.1.2.2 Standard Alignment Rules
Section 5.9.15 MOVE Statement

Examples

The Procedure Division entry for the MOVE statement contains examples using this clause.

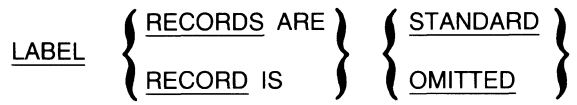
LABEL RECORDS

4.2.10 LABEL RECORDS Clause

Function

The LABEL RECORDS clause specifies the presence or absence of labels.

General Format



General Rule

The LABEL RECORDS clause is for documentation only.

4.2.11 Level-Number

Function

The level-number shows the position of a data item within the hierarchical structure of a logical record. It also identifies entries for condition-names and the RENAMES clause.

General Format

level-number

Syntax Rules

1. The level-number must be the first element in a data description entry.
2. Data description entries that are subordinate to a file description (FD) entry have level-numbers 01 through 49, 66, or 88.
3. Data description entries in the Working-Storage and Linkage Sections have level-numbers 01 through 49, 66, 77, or 88.

General Rules

1. The level-number 01 identifies the first entry in a record description.
2. Multiple level 01 entries subordinate to a file description entry represent implicit redefinitions of the same area.
3. Level-number 66 identifies a RENAMES entry. It can be used only in a Format 2 data description entry.
4. Level-number 77 identifies a noncontiguous data item entry in the Working-Storage and Linkage Sections. The level 77 entry can have no subordinate data description entries except level 88 items.
5. Level-number 88 defines a condition-name associated with a conditional variable. It can be used only in a Format 3 data description entry.
6. Level-numbers 66, 77, and 88 do not imply hierarchical position.

Additional References

Section 1.1.2.1	User-Defined Words – Condition-Name
Section 4.1.1.1	Record Description
Section 4.2.3	Data Description
Section 4.2.17	RENAMES Clause

4.2.12 LINAGE Clause

Function

The LINAGE clause specifies the number of lines on a logical page. It can also specify the size of the logical page's top and bottom margins and the line where the footing area begins in the page body.

General Format

```
LINAGE IS { page-lines } LINES [ WITH FOOTING AT footing-line ]
    [ LINES AT TOP top-lines ] [ LINES AT BOTTOM bottom-lines ]
```

page-lines
: is a positive integer or the data-name of an elementary unsigned integer numeric data item. Its value must be greater than zero. It specifies the number of lines that can be written or spaced on the logical page. If *page-lines* is a data-name, it can be qualified.

footing-line
: is a positive integer or the data-name of an elementary unsigned integer numeric data item. Its value must be greater than zero, but cannot be greater than *page-lines*. *Footing-line* specifies the line number where the footing area begins in the page body. If *footing-line* is a data-name, it can be qualified.

top-lines
: is an integer or the data-name of an elementary unsigned integer numeric data item. Its value can be zero. *Top-lines* specifies the number of lines in the top margin of the logical page. If *top-lines* is a data-name, it can be qualified.

bottom-lines
: is an integer or the data-name of an elementary unsigned integer numeric data item. Its value can be zero. *Bottom-lines* specifies the number of lines in the bottom margin of the logical page. If *bottom-lines* is a data-name, it can be qualified.

General Rules

1. The LINAGE clause specifies the number of lines on a logical page.
2. Logical page size is the sum of the values specified in all phrases except FOOTING. If there is no LINES AT TOP or LINES AT BOTTOM phrase, the default value of *top-lines* or *bottom-lines* is zero. If there is no FOOTING phrase, the default value of *footing-line* equals the value of *page-lines*.
3. Logical and physical page sizes are not necessarily the same.
4. The page body is the logical page area in which the program can write or space lines. Its size equals the value of *page-lines*.
5. The footing area comprises the area of the logical page between *footing-line* and *page-lines*, inclusive.

6. When the program opens the file by executing an OPEN statement with the OUTPUT phrase, it uses the values of *page-lines*, *top-lines*, and *bottom-lines* to define the logical page sections. When these values are integers, they apply to all logical pages the program writes to the file during its execution.
7. When *page-lines*, *top-lines*, and *bottom-lines* are data-names, their values affect OPEN and WRITE statement execution as follows:
   - When the program executes an OPEN statement with the OUTPUT phrase for the file, the values specify the number of lines in each of the associated sections of the first logical page.
   - When the program executes a WRITE statement with the ADVANCING PAGE phrase, or when a page overflow condition occurs, the values specify the number of lines in each of the associated sections of the next logical page.
8. The value of *footing-line* defines the footing area for the first logical page when the program executes an OPEN statement with the OUTPUT phrase for the file. The value defines the footing area for the next logical page when: (a) the program executes a WRITE statement with the ADVANCING PAGE phrase or, (b) a page overflow condition occurs.
9. For each file with a LINAGE clause, the program has a corresponding special register called LINAGE-COUNTER. At any time, the value in LINAGE-COUNTER is the line number in the current page body at which the device is positioned.
10. LINAGE-COUNTER is a special register whose implicit size is four decimal digits (represented by PIC S9(4) COMP). Procedure Division statements can refer to LINAGE-COUNTER but cannot change its value.
11. If the program has more than one LINAGE-COUNTER, all Procedure Division references to it must be qualified by *file-name*.
12. Execution of a WRITE statement for a file with the LINAGE clause changes the value of the associated LINAGE-COUNTER:
    - If the WRITE statement has the ADVANCING PAGE phrase, its execution resets LINAGE-COUNTER to one. (The resetting operation implicitly increments the value of LINAGE-COUNTER to exceed the value of *page-lines*).
    - If the WRITE statement has the ADVANCING LINES phrase, its execution increments LINAGE-COUNTER by the value in the ADVANCING phrase.
    - If the WRITE statement does not have the ADVANCING phrase, it increments LINAGE-COUNTER by one.
13. Execution of an OPEN statement for the file sets its LINAGE-COUNTER to one.
14. Each logical page follows the preceding logical page with no spacing between them.
15. Execution of a CLOSE statement for the file causes the remainder of the current page to be written to the file.

LINAGE
Continued

Technical Note

The LINAGE clause causes a file to be in print-file format. When a WRITE statement positions the file to the top of the next logical page, device positioning occurs by line spacing rather than by page ejection or form feed.

The default operating system PRINT command causes the insertion of a form-feed character when a form nears the end of a page. Therefore, when the default PRINT command refers to a LINAGE file, unexpected page spacing can result.

To print linage-files correctly on a RSTS/E system, use the /NOFEED file qualifier of the PRINT command to suppress the insertion of form-feed characters. For example:

```
$ PRINT OUTPUT-REPORT/NOFEED
```

To print linage-files correctly on an RSX-11M/M-PLUS system, use the /LENGTH=0 qualifier of the PRINT command to suppress the insertion of form-feed characters. For example:

```
> PRINT/LENGTH=0 OUTPUT-REPORT
```

Additional References

Section 5.9.32 WRITE Statement

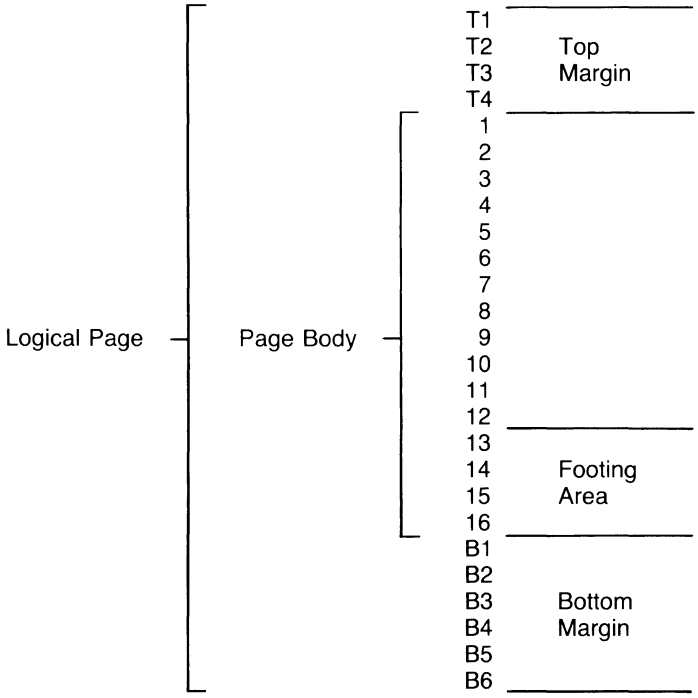
Example

This example specifies a logical page whose size is 26 lines. The first line to which the page can be positioned is the fifth line. The end-of-page condition occurs when a WRITE statement causes the LINAGE-COUNTER value to be in the range 13 through 16. The page overflow condition occurs when a WRITE statement causes the LINAGE-COUNTER value to exceed 16.

```
FD  PRINT-FILE
    VALUE OF ID IS "REPORT1.LIS"
    LINAGE IS 16 LINES WITH FOOTING AT 13
      LINES AT TOP 4 LINES AT BOTTOM 6.
```

Figure 4-9 shows the logical page areas resulting from the example.

Figure 4-9: Logical Page Areas Resulting from a LINAGE Clause



4.2.13 OCCURS Clause

Function

The OCCURS clause defines tables and provides the basis for subscripting and indexing. It eliminates the need for separate entries for repeated data items.

General Format

Format 1

```
OCCURS table-size TIMES

[ { ASCENDING  } KEY IS { key-name } ... ] ...
  { DESCENDING }

[ INDEXED BY { ind-name } ... ]
```

Format 2

```
OCCURS min-times TO max-times TIMES DEPENDING ON depending-item

[ { ASCENDING  } KEY IS { key-name } ... ] ...
  { DESCENDING }

[ INDEXED BY { ind-name } ... ]
```

table-size
: is an integer that specifies the exact number of occurrences of a table element.

min-times
: is an integer that specifies the minimum number of occurrences of a table element. Its value must be greater than or equal to one.

max-times
: is an integer that specifies the maximum number of occurrences of a table element. Its value must be greater than *min-times*.

key-name
: is the data-name of an entry that contains the OCCURS clause or an entry subordinate to it. *Key-name* can be qualified. Each *key-name* after the first must name an entry subordinate to the entry that contains the OCCURS clause. The values in each *key-name* are the basis of the ascending or descending arrangement of the table's repeated data.

ind-name
: is an index-name. It associates an index with the table and allows indexing in table element references.

depending-item
: is the data-name of an elementary unsigned integer data item. Its value specifies the current number of occurrences. *Depending-item* can be qualified.

Syntax Rules

1. The subject of the entry is the data-name that contains the OCCURS clause.
2. A *key-name* cannot contain an OCCURS clause. However, this rule does not apply to the first *key-name* if it is the subject of the entry.
3. There can be no OCCURS clauses between the data description entries for *key-names* and the subject of the entry.
4. In the OCCURS clause of the data description entry, *key-name* cannot be subscripted or indexed.
5. There must be an INDEXED BY phrase if any Procedure Division statements contain indexed references to the subject of the entry or to any of its subordinates.
6. The INDEXED BY phrase implicitly defines *ind-name*. The program cannot define *ind-name* elsewhere.
7. A Format 2 OCCURS clause can be used in a record description entry for a file with variable length records. If it is, *depending-item* must be in the same record.
8. The subject of a Format 2 OCCURS clause can be followed, in the same record description, only by data description entries subordinate to it.
9. The OCCURS clause cannot be used in a data description entry that has:
   - A level-number of 01, 66, 77, or 88
   - A subordinate variable occurrence data item (Format 2 OCCURS clause)
10. The data item defined by *depending-item* cannot occupy any character position in the range delimited by the following:
    - The character position defined by the subject of the OCCURS clause
    - The last character position defined by the record description entry containing the OCCURS clause
11. Each *ind-name* must be a unique word in the program.

General Rules

1. The OCCURS clause defines tables and provides the basis for subscripting and indexing.
2. Except for the OCCURS clause itself, all data description clauses associated with the subject of the OCCURS clause apply to each occurrence of the item.

OCCURS
Continued

3. Format 1 specifies that the subject of the entry has a fixed number of occurrences.
4. Format 2 specifies that the subject of the entry has a variable number of occurrences. *Min-times* and *max-times* specify the minimum and maximum number of occurrences. Only the number of the subject's occurrences is variable; its size is fixed.

   The value of *depending-item* must fall in the range *min-times* through *max-times*.

   The contents of data items with occurrence numbers exceeding the current value of *depending-item* are unpredictable.
5. If a group item with a subordinate entry that has a Format 2 OCCURS clause is a sending item, the operation uses only the part of the table area specified by *depending-item* at the start of the operation.

   If the group is a receiving item, the operation uses the maximum length of the group. The operation ignores *depending-item* and does not change its value unless it is in the group.
6. The KEY IS phrase indicates that the repeated data is arranged in ascending or descending order according to the values in the data items named by *key-name*. The rules for operand comparison determine the ascending or descending order. The position of each *key-name* in the list determines its significance. The first is the most significant, and the last is least significant.
7. If a Format 2 OCCURS clause is in a record description entry *and* the associated file description entry has the VARYING phrase of the RECORD clause, the records are variable length.

   If the RECORD clause does not have the DEPENDING ON phrase, the program must set the OCCURS clause *depending-item* to the number of occurrences before executing a RELEASE, REWRITE, or WRITE statement. The *depending-item* value determines the length of the record to be written.

Technical Note

A table can sometimes contain fill bytes that increase its size. Fill bytes are caused by alignment requirements that apply either to the subject of the OCCURS clause or to its subordinates. Unless the table contains a COMP SYNC data item, the compiler will insert no more than one fill byte between items when assigning storage. However, if the table contains COMP SYNC items larger than 4 digits, the size of the table could increase significantly.

The compiler issues an informational diagnostic when an item will contain fill bytes. If you compile your program using the /SHOW:MAP qualifier, the listing also indicates where the fill bytes appear in the table. Refer to Part I of the COBOL-81 User's Guide for your system for instructions on using the COBOL command with this qualifier. The chapter on table handling in Part III of the User's Guide contains additional information on this subject.

Additional References

Section 4.1.2.3	Record Allocation
Section 5.5.1.1	Comparison of Numeric Operands
Section 5.5.1.2	Comparison of Nonnumeric Operands
Section 5.9.23	SEARCH Statement

Examples

Additional examples of the OCCURS clause are in Section 4.1.2.3 and Section 5.9.23, SEARCH Statement.

1. One-dimensional table:

   This record description entry describes a 20-character record. The record contains ten occurrences of ITEMB, a 2-character data item.

```
01   ITEMA.
     03   ITEMB OCCURS 10 TIMES PIC XX.
```

2. Two-dimensional table:

   This record description entry describes a 320-character record. The record contains eight occurrences of ITEMB, a 40-character data item. ITEMB contains ten occurrences of ITEMC, a 4-character data item. Each ITEMC contains two data items: ITEMD and ITEME.

```
01   ITEMA.
     03   ITEMB OCCURS 8 TIMES.
          05   ITEMC OCCURS 10 TIMES.
               07   ITEMD PIC X.
               07   ITEME PIC XXX.
```

   ITEMB (1) refers to a 40-character data item, the first ten occurrences of ITEMC. Similarly, ITEMB (5) refers to the fifth group of ten occurrences of ITEMC.

   ITEME (3,4) refers to ITEME in the fourth occurrence of ITEMC in the third occurrence of ITEMB:

```
ITEMB (1)    DEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEE
ITEMB (2)    DEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEE
ITEMB (3)    DEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEE
ITEMB (4)    DEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEE
ITEMB (5)    DEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEE
ITEMB (6)    DEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEE
ITEMB (7)    DEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEE
ITEMB (8)    DEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEE
```

3. Variable occurrence data item:

   When ITEMA is a receiving item, its size is 2128 characters. When it is a sending item, its size can vary from 70 to 2128 characters, depending on the value in ITEMC.

   Each ITEME is 42 characters long. Its size cannot change. The only effect of the value of ITEMC is to determine the number of ITEME occurrences.

OCCURS
Continued

There are ten occurrences of ITEMH and ITEMI in each occurrence of ITEME.

```
01  ITEMA.
    03  ITEMB  PIC X(6).
    03  ITEMC  PIC 99.
    03  ITEMD  PIC X(20).
    03  ITEME OCCURS 1 TO 50 TIMES DEPENDING ON ITEMC.
        05  ITEMF PIC XX.
        05  ITEMG OCCURS 10 TIMES.
            07  ITEMH  PIC X.
            07  ITEMI  PIC XXX.
```

4.2.14 PICTURE Clause

Function

The PICTURE clause specifies the general characteristics and editing requirements of an elementary item.

General Format

Syntax Rules

1. You can use the PICTURE clause only for an elementary item.
2. *Character-string* contains allowable combinations of characters in the COBOL character set. These characters are called the "symbols" of the PICTURE character-string.
3. *Character-string* can contain no more than 30 symbols.

   When the symbols define nonnumeric data, they can represent an item that is more than 30 characters in length.

   When they define numeric data, the symbols can represent an item with no more than 18 digit positions.
4. The PICTURE clause is required for every elementary item except: (a) an item specified by the USAGE IS INDEX clause and (b) the subject of a RENAMES clause. Data description entries for these items cannot contain a PICTURE clause.
5. PIC is an abbreviation for PICTURE.
6. The asterisk (*), when used as a zero suppression symbol, and the BLANK WHEN ZERO clause cannot be used in the same entry.

General Rules

1. The PICTURE clause defines a data item to belong to a particular category, and determines what the item can contain. Table 4-3 shows the valid contents of both *character-string* and the data item itself for each category. General rules 2 through 5 following Table 4-3 supplement the information it contains.

Table 4-3: Summary of PICTURE Clause Rules

Category of Receiving Item	PICTURE of Receiving Item	Valid Contents of Sending Item	Examples
Alphabetic	Must contain one or more As.	One or more alphabetic characters.	AA A(9)
Numeric	Must contain at least one 9. Can contain one S and one V. Can contain Ps. Must describe 1 to 18 digit positions, which can be represented by 9s or Ps.	One or more numeric characters.	S9(4)V99 9PPP SPP9
Alphanumeric	Can contain combinations of As, Xs and 9s. Can be all Xs. Cannot be all As or all 9s.	One or more characters in computer character set.	XX99XX AAXA(4) X(32)
Alphanumeric Edited	Must contain at least one A or X. Must also contain at least one B, 0, or /. Can contain one or more 9s.	One or more characters in computer character set.	XXBXXB9(4) XX/99/00 9(6)/X
Numeric Edited	Must contain at least one 0, B, /, Z, *, +, (comma), ., –, CR, DB, or cs. Can contain Ps, 9s, and one V. Must describe 1 to 18 digit positions, which can be represented by 9s, zero suppression symbols (Z, *), and floating insertion symbols (+, –, cs).	One or more numeric characters.	*,***.** ZZ,ZZZ.9(4) $$,$$$DB $9,999CR ZZZCR **.**

2. In an alphanumeric item definition, each character position is treated as if it were represented by an X, even though A or 9 may be specified.
3. Some PICTURE symbols represent character positions and some do not. A data item's size is determined by adding up all the symbols that represent a character position. For example, a numeric data item with a PICTURE of 999V99 has a size of five characters. V does not count toward the item's size.
4. *Character-string* can contain a repeat count to represent consecutive occurrences of the following symbols: A, the comma (,), X, 9, P, Z, *, B, /, 0, +, –, and the currency symbol (represented by cs). The repeat count must be an unsigned, nonzero integer enclosed in parentheses. For example, S9(6)V9(4) is equivalent to S999999V9999. However, *character-string* can contain no more than one of the following symbols: S, V, the period (.), CR, and DB.
5. The PICTURE clause symbols and their functions are as follows:

   A — Represents a character position that can contain only an alphabetic character. (An alphabetic character belongs to the set of characters: A through Z, a through z, and the space.)

Can occur more than once.

Counts toward data item size.

B — Represents a character position into which a space is inserted.

Can occur more than once.

Counts toward data item size.

P — Specifies an assumed decimal scaling position, defining the location of the decimal point when one is not specified in *character-string*.

Can occur more than once, but only as a contiguous string of Ps at either the leftmost or rightmost end (not both) of *character-string*. The assumed decimal point character (V) is redundant when specified. However, when it is specified, V can appear to the left of the leftmost P or to the right of the rightmost P.

Does not count toward data item size. However, each P counts toward the maximum number of digit positions (18) in a numeric or numeric edited item.

Cannot be used if an explicit decimal point (.) appears in *character-string*.

In certain operations that refer to a data item with P characters in *character-string*, the compiler treats each P position as if it contained the value zero. For example, a data item with PICTURE 99PPP can have 100 unique values that range from 0 to 99,000 (0, 1000, 2000, ..., 99,000). A data item with PICTURE PP9 can have ten unique values (0, .001, .002, ... .009). These operations are any of the following:

- Any operation requiring a numeric sending operand
- A MOVE statement where the sending operand is a numeric or numeric edited data item and *character-string* contains the symbol P
- A comparison operation where both operands are numeric

In all other operations, the compiler ignores the digit positions specified with the symbol P and does not count them toward the size of the operand.

S — Indicates the presence of an operational sign, but does not specify the sign representation or position.

Can occur only once, as the leftmost character in *character-string*.

Does not count toward data item size unless the data description entry contains a SIGN IS SEPARATE clause. If the SIGN clause does not appear in the item's data description, S is equivalent to SIGN IS TRAILING.

V — Specifies the location of the assumed decimal point.

Can occur only once.

Does not count toward data item size.

Cannot be used if an explicit decimal point (.) appears in the PICTURE.

PICTURE
Continued

X — Represents a character position that can contain any character from the computer character set.

Can occur more than once.

Counts toward data item size.

Z — Represents a leading digit position that is replaced by a space when its value and the value of the digits to its left are zero.

Can occur more than once.

Counts toward data item size.

Use of Z excludes the use of the asterisk (*) for zero suppression and replacement.

9 — Represents a digit position that can contain only the digits 0 through 9.

Can occur more than once.

Counts toward data item size.

0 — Represents a character position into which 0 is inserted.

Can occur more than once.

Counts toward data item size.

/ — Represents a character position into which a slash (/) is inserted.

Can occur more than once.

Counts toward data item size.

. — Represents a character position into which a comma (,) is inserted.

Can occur more than once.

Counts toward data item size.

, — Represents a character position into which a decimal point (.) is inserted. It also represents the decimal point for alignment purposes.

Can occur only once.

Counts toward data item size.

Cannot be used if V or P appear in *character-string*.

Note

When a program contains the DECIMAL POINT IS COMMA clause, the functions and rules for the period (.) and comma (,) are exchanged. In other words, the rules that apply to the period apply to the comma, and vice versa.

+ –
: Represent the editing sign control symbols, plus (+) and minus (–).
Each can occur more than once.
Each counts as one character toward data item size.
Character-string can contain occurrences of + or –, but not both. Also, the use of either character excludes the use of both CR and DB.

CR DB
: Represent the editing sign control symbols, credit (CR) and debit (DB).
Each can occur only once, as the two rightmost character positions.
Each counts as two characters toward data item size.
Character-string can contain either CR or DB, but not both. Also, the use of either excludes the use of both + and – as fixed insertion characters.

*
: Represents a leading digit position that is replaced by * when its value and the values of all digit positions to its left are zero.
Can occur more than once.
Counts toward data item size.
Use of * excludes the use of Z for zero suppression and replacement.

CS
: Represents a character position into which the currency symbol is inserted. This symbol is either the currency sign ($) or the character specified in the CURRENCY SIGN clause of the SPECIAL-NAMES paragraph.
Can occur more than once.
Counts as one character toward data item size.

Editing Rules

1. There are two PICTURE clause editing methods: (a) insertion editing and (b) suppression and replacement editing. Each method has the following variations:

Insertion Editing
Simple insertion
Special insertion
Fixed insertion
Floating insertion

Suppression and Replacement Editing
Zero suppression and replacement with spaces
Zero suppression and replacement with asterisks

2. The types of editing a program can perform on a data item depends on the item's category:

Category	Types of Editing	Valid Editing Characters
Alphabetic	None	None
Numeric	None	None
Alphanumeric	None	None
Alphanumeric Edited	Simple insertion	0, B, and /
Numeric Edited	All	All, subject to Editing Rule 3

3. Floating insertion editing and editing by zero suppression and replacement are mutually exclusive. That is, a PICTURE clause can use one type of editing or the other, but not both.

   Furthermore, a PICTURE clause can use only one type of replacement symbol for zero suppression. The Z (space) and * (asterisk) symbols cannot appear in the same PICTURE clause.

4. **Simple Insertion Editing**

   The , (comma), B (space), 0 (zero), and / (slash) are the symbols used in simple insertion editing. They indicate a data item position to contain the character they represent. These symbols count toward data item size.

   If the comma is the last symbol in *character-string*, the PICTURE clause must be the last clause of the data description entry. In this case, ",." are the last two characters of the data description entry. However, if the DECIMAL-POINT IS COMMA clause is in the SPECIAL-NAMES paragraph, the data description entry ends with two consecutive periods.

5. **Special Insertion Editing**

   The period (.) is the only symbol used in special insertion editing. It represents the data item position to contain the actual decimal point. However, it also represents the decimal point for alignment purposes. Therefore, the assumed decimal point (V) and the actual decimal point (.) cannot be used in the same *character-string*. The period counts toward data item size.

   If the period is the last symbol in *character-string*, the PICTURE clause must be the last clause of the data description entry. In this case, the data description entry ends with two periods. However, if the DECIMAL-POINT IS COMMA clause is in the SPECIAL-NAMES paragraph, ",." are the last two characters of the data description entry.

6. **Fixed Insertion Editing**

   The currency symbol and the editing sign control symbols +, –, CR, and DB are the symbols used in fixed insertion editing. *Character-string* can contain only one currency symbol and only one of the editing sign control symbols as fixed insertion characters.

   CR and DB each represent two character positions, which must be the two rightmost positions.

   The symbols + and – must be either the leftmost or rightmost character position that counts toward the size of the item.

   The currency symbol must be the leftmost character position that counts toward the size of the item. However, a + or – symbol can precede it.

   Fixed insertion editing causes the insertion symbol to occupy the same position in the edited data item as in *character-string*. Table 4-4 shows that the results of using editing sign control symbols depend on the data item's value.

Table 4-4: Using Sign Control Symbols in Fixed Insertion Editing

Editing Symbol in PICTURE Character-String	Result	
	Data Item Positive or Zero	Data Item Negative
+	+	–
–	space	–
CR	2 spaces	CR
DB	2 spaces	DB

7. **Floating Insertion Editing**

The currency symbol and editing sign control symbols + and – are the symbols used in floating insertion editing. They are mutually exclusive in *character-string*. That is, if any floating insertion symbol appears in *character-string*, no other floating insertion symbol can appear.

To indicate floating insertion editing, you must use a string of at least two floating insertion symbols. You can include simple insertion symbols either within the floating string or immediately to the right of the floating string. These simple insertion symbols are treated as part of the floating string. (That is, they only appear in results when the value of the data item is large enough to include the position occupied by the simple insertion symbol.) You can append the fixed insertion symbols CR or DB immediately to the right of a floating string.

The leftmost symbol of the floating insertion string represents the leftmost position in which a floating insertion character can appear. This character position cannot be filled by a digit.

The second floating symbol from the left represents the leftmost limit of the numeric data the data item can store. Nonzero numeric characters can replace all symbols at or to the right of this limit.

You can use the floating insertion symbol in only two ways. It can represent:

a. Any or all leading numeric character positions to the left of the decimal point

   In this case, run-time results show a single insertion character in the position immediately preceding either: (a) the first nonzero digit in the data item or (b) the decimal point, whichever appears leftmost in the data. For example, a data item whose PICTURE is $$$.99 and whose value is zero would appear as $.00.

b. All numeric character positions in the PICTURE character-string

   In this case, you must specify at least one insertion symbol to the left of the decimal point. When the data item has a nonzero value, run-time results are the same as when *all* the insertion symbols are to the left of the decimal point. However, when the data item has a zero value, run-time results show neither a floating insertion character nor the decimal point. For example, a data item whose PICTURE is $$$.$$ and whose value is zero would appear as spaces.

If the floating insertion symbol is + or –, the actual character inserted depends on the value of the data item. Table 4-5 shows the possible results of using editing sign control symbols in floating insertion editing.

Table 4-5: Using Sign Control Symbols in Floating Insertion Editing

Editing Symbol in PICTURE Character-String	Result	
	Data Item Positive or Zero	Data Item Negative
+	+	–
–	space	–

To avoid truncation, the minimum size of *character-string* must be the sum of:

- The number of characters in the sending item
- The number of simple, special, or fixed insertion characters edited into the receiving item
- One, for the floating insertion character

8. **Zero Suppression and Replacement Editing**

One or more occurrences of Z or * define a floating suppression string, which can suppress leading zeros in numeric character positions. Z causes spaces to replace the zeros; * causes asterisks to replace them.

The suppression symbols are mutually exclusive. That is, *character-string* can contain either Z or *, but not both.

Each suppression symbol counts toward data item size.

You can include simple insertion symbols either within the floating string or immediately to its right. These simple insertion symbols are treated as part of the floating string. (That is, they only appear in results when the value of the data item is large enough to include a position occupied by a simple insertion symbol.)

You can use zero suppression symbols to represent either:

- Any or all leading numeric character positions to the left of the decimal point
- All numeric character positions on both sides of the decimal point

  For example, both ZZZ9.99 and ZZ.ZZ are valid *character-strings*, but ZZZ.Z9 is not.

The following actions occur if the suppression symbols represent any or all leading numeric character positions to the left of the decimal point:

- The replacement character replaces any leading zero in the data that corresponds to a suppression symbol in the string.
- Suppression ends at either: (1) the first nonzero digit in the data represented by the suppression string or (2) the decimal point, whichever appears first in the data.

The following events occur if the suppression symbols represent all numeric positions in *character-string*:

- If the value of the data is not zero, the result is the same as if all suppression symbols were to the left of the decimal point. That is, zeros to the right of the decimal point are not suppressed.

- If the value is zero and the suppression symbol is Z, all character positions in the edited data item (including any editing characters) contain spaces.
- If the value is zero and the suppression symbol is *, all character positions in the edited data item (including any insertion editing characters other than the decimal point) contain asterisks. The decimal point appears in the data item.

9. The symbols +, –, *, Z, and the currency symbol are mutually exclusive when they are used as floating replacement characters. That is, if any one of these symbols appears as a floating replacement character, none of the other symbols can appear as a floating replacement character in the same PICTURE clause.

PICTURE Symbol Precedence Rules

1. *Character-string* must contain either:
   - At least one of the symbols A, X, Z, 9, or *
   - At least two of the symbols +, –, or cs
2. Table 4-6 summarizes the rules for combining symbols to form *character-strings* more complex than the basic possibilities listed in rule 1. The table shows that the use of one symbol in a *character-string* excludes the use of certain others before or after it.

   The table uses the following conventions:

   a. A *Y* at an intersection means the symbol(s) at the top of the column (*First Symbol*) can precede the symbol(s) at the left of the row (*Second Symbol*).

   b. Braces { } enclose symbols that are mutually exclusive.

   c. The currency symbol appears as *cs*.

   d. Symbols appear twice in a column or row when their rules of use depend upon their location in a *character-string*. These double entry symbols are:
      - Fixed insertion symbols + and –
      - Floating symbols Z, *, +, –, and cs
      - P

      The uppermost entry in a column (or the leftmost entry in a row) represents symbol use left of the actual or implied decimal point position. The second entry represents symbol use to the right of the decimal point.

PICTURE
Continued

Table 4-6: PICTURE Symbol Precedence Rules

Second Symbol \ First Symbol		Simple, Special, and Fixed Insertion Symbols								Floating Insertion and Suppression Symbols						Other Symbols					
		B	0	/	,	.	{+ −}	{+ −}	{CR DB}	cs	{Z *}	{Z *}	{+ −}	{+ −}	cs	cs	9	A X	S	V	P	P
	B	Y	Y	Y	Y	Y	Y		Y	Y	Y	Y	Y	Y	Y	Y	Y		Y		Y
Simple,	0	Y	Y	Y	Y	Y	Y		Y	Y	Y	Y	Y	Y	Y	Y	Y		Y		Y
Special,	/	Y	Y	Y	Y	Y	Y		Y	Y	Y	Y	Y	Y	Y	Y	Y		Y		Y
and	,	Y	Y	Y	Y	Y	Y		Y	Y	Y	Y	Y	Y	Y	Y			Y		Y
Fixed	.	Y	Y	Y	Y		Y		Y	Y		Y		Y		Y					
Insertion	{+ −}																				
Symbols	{+ −}	Y	Y	Y	Y	Y			Y	Y	Y			Y	Y	Y			Y	Y	Y
	{CR DB}	Y	Y	Y	Y	Y			Y	Y	Y			Y	Y	Y			Y	Y	Y
	cs						Y														
Floating	{Z *}	Y	Y	Y	Y		Y		Y	Y											
Insertion	{Z *}	Y	Y	Y	Y	Y	Y		Y	Y	Y								Y		Y
and	{+ −}	Y	Y	Y	Y				Y			Y									
Suppression	{+ −}	Y	Y	Y	Y	Y			Y			Y	Y						Y		Y
Symbols	cs	Y	Y	Y	Y		Y							Y							
	cs	Y	Y	Y	Y	Y	Y							Y	Y				Y		Y
	9	Y	Y	Y	Y	Y	Y		Y	Y		Y		Y		Y	Y	Y	Y		Y
	A X	Y	Y	Y												Y	Y				
Other	S																				
Symbols	V	Y	Y	Y	Y		Y		Y	Y		Y		Y		Y		Y		Y	
	P	Y	Y	Y	Y		Y		Y	Y		Y		Y		Y		Y		Y	
	P						Y		Y									Y	Y		Y

Additional References

Section 4.2.18 SIGN Clause
Section 5.9.15 MOVE Statement

Examples

The Procedure Division entry for the MOVE statement contains examples that illustrate this clause.

4.2.15 RECORD Clause

Function

The RECORD clause specifies: (1) the number of character positions in a fixed length record, (2) variable length record format, or (3) the minimum and maximum number of character positions in a variable length record.

General Format

Format 1

```
RECORD CONTAINS [ shortest-rec TO ] longest-rec CHARACTERS
```

Format 2

```
RECORD IS VARYING IN SIZE    [ FROM shortest-rec ] [ TO longest-rec ] CHARACTERS
                                                   [ DEPENDING ON depending-item ]
```

shortest-rec
: is an integer that specifies the minimum number of character positions in a variable length record.

longest-rec
: is an integer greater than *shortest-rec*. It specifies the maximum number of character positions in a variable length record or the size of a fixed length record.

depending-item
: is the data-name of an elementary unsigned integer data item in the Working-Storage or Linkage Section. It specifies the number of character positions for an output operation, and it contains the number of character positions after a successful input operation.

Syntax Rules

1. No record description entry for a file can specify:
   - Fewer character positions than *shortest-rec*
   - More character positions than *longest-rec*
2. In a sort-merge file description entry, the first *shortest-rec* character positions of the record must be large enough to include all keys specified in any SORT or MERGE statement for the sort or merge file.
3. For an indexed file, the first *shortest-rec* character positions of the record must be large enough to include all record keys.

General Rules

Both Formats

1. The absence of a RECORD clause is the same as a Format 1 RECORD clause with: (1) no *shortest-rec* phrase, and (2) *longest-rec* equal to the greatest number of character positions described for any of the file's records.
2. The number of characters described by a record description entry is the sum of both:
   - The number of character positions in all elementary items excluding redefinitions and renamings
   - The number of fill bytes added because of alignment requirements

   If the record description entry contains a table definition, the sum includes the number of character positions in the maximum number of table elements.

Format 1

3. If there is no *shortest-rec* phrase, Format 1 specifies fixed length records. *Longest-rec* then specifies the number of character positions in each record of the file.
4. If there is a *shortest-rec* phrase, Format 1 specifies variable length records, the same as Format 2 without the DEPENDING phrase.
5. For variable length records:
   - The maximum record size for a READ or RETURN operation is the number of character positions described in the largest record description entry for the file.
   - During execution of a RELEASE, REWRITE, or WRITE statement, the number of character positions in a record equals the number of character positions in the record description entry referred to by the statement.
   - If all record description entries for the file describe records of the same size, RELEASE, REWRITE, and WRITE statements for the file transfer fixed length records in variable length format.

Format 2

6. Format 2 specifies variable length records.
7. If the clause does not contain *shortest-rec*, the minimum number of character positions in any of the file's records is the least number of character positions described by a record description entry for the file.
8. If the clause does not contain *longest-rec*, the maximum number of character positions in any of the file's records is the greatest number of character positions described by a record description entry for the file.
9. If there is a DEPENDING phrase, the program must set *depending-item* to the number of character positions in the record before executing a RELEASE, REWRITE, or WRITE statement for the file.

10. After successful execution of a READ or RETURN statement for the file, the value of *depending-item* indicates the number of character positions in the accessed record.
11. The *depending-item* value is not changed by executions of:
    - DELETE and START statements
    - Unsuccessful READ and RETURN statements
12. For RELEASE, REWRITE, and WRITE statement execution, determining the number of character positions in the record depends partly upon whether or not the record contains a variable occurrence data item (an item described by the OCCURS clause or that is subordinate to another item so described). During execution of these statements, three rules determine the number of character positions in the record:
    - If there is a *depending-item*, its value specifies the number of character positions.
    - If there is no *depending-item* and the record does not contain a variable occurrence data item, the number of character positions described by the record description entry specifies the number of character positions.
    - If there is no *depending-item* and the record contains a variable occurrence data item, the number of character positions is the sum of the character positions in: (1) the fixed part of the record, and (2) the table elements specified by the OCCURS clause *depending-item* when the output statement executes.

Additional References

Section 4.2.19	SYNCHRONIZED Clause
Section 4.2.20	USAGE Clause
Part I of the COBOL-81 User's Guide for your system	Refer to the chapter on program compilation (/CHECK qualifier)

4.2.16 REDEFINES Clause

Function

The REDEFINES clause allows different data description entries to describe the same storage area.

General Format

```
level-number  [ data-name ]  REDEFINES other-data-item
              [ FILLER    ]
```

other-data-item
is a data-name. It identifies the data description entry that first defines the storage area.

Note

Level-number, data-name, and FILLER are not actually part of the REDEFINES clause. They are included in the general format only to clarify the relative position of the clauses.

Syntax Rules

1. The subject of the REDEFINES clause is the data-name or FILLER in a Format 1 data description entry.
2. The REDEFINES clause must immediately follow its subject.
3. The level-numbers of the subject of the REDEFINES clause and *other-data-item* must be the same. However, they cannot be either 66 or 88.
4. The REDEFINES clause cannot be used in a level 01 entry in the File Section.
5. The data description entry for *other-data-item* cannot contain an OCCURS clause. However, *other-data-item* can be subordinate to an item whose data description entry contains an OCCURS clause. In that case, the reference to *other-data-item* in the REDEFINES clause cannot be subscripted or indexed.
6. Neither the original definition nor the redefinition can contain a variable occurrence data item.
7. If *other-data-item* is other than a level 01 entry, the number of character positions it contains must be greater than or equal to the number in the subject of the REDEFINES clause. If *other-data-item* is a level 01 entry, its description need not follow this rule; that is, *other-data-item* can contain fewer character positions than the subject of the REDEFINES clause.
8. *Other-data-item* cannot be qualified even if it is not unique. The reference to *other-data-item* is unique without qualification because of the placement of the REDEFINES clause.

9. A program can have multiple redefinitions of the same character positions. However, they must all refer to *other-data-item*, the data-name that originally defined the area.
10. The redefining entries cannot contain VALUE clauses except in condition-name entries.
11. No entry with a level-number lower than that of *other-data-item* can occur between the data description entry for *other-data-item* and the redefinition.
12. The entries redefining the storage area must immediately follow those that originally defined it. There can be no intervening entries that define additional storage areas.

General Rules

1. Storage allocation starts at the location of *other-data-item*. Storage allocation continues until it defines the number of character positions in the data item referred to by the subject of the REDEFINES clause.
2. If more than one data description entry defines the same character position, the program can refer to the character postion using the data-name associated with any of those data description entries.

Additional References

Section 4.1.2.3 Record Allocation
Section 4.2.3 Data Description – Format 1

REDEFINES
Continued

Example

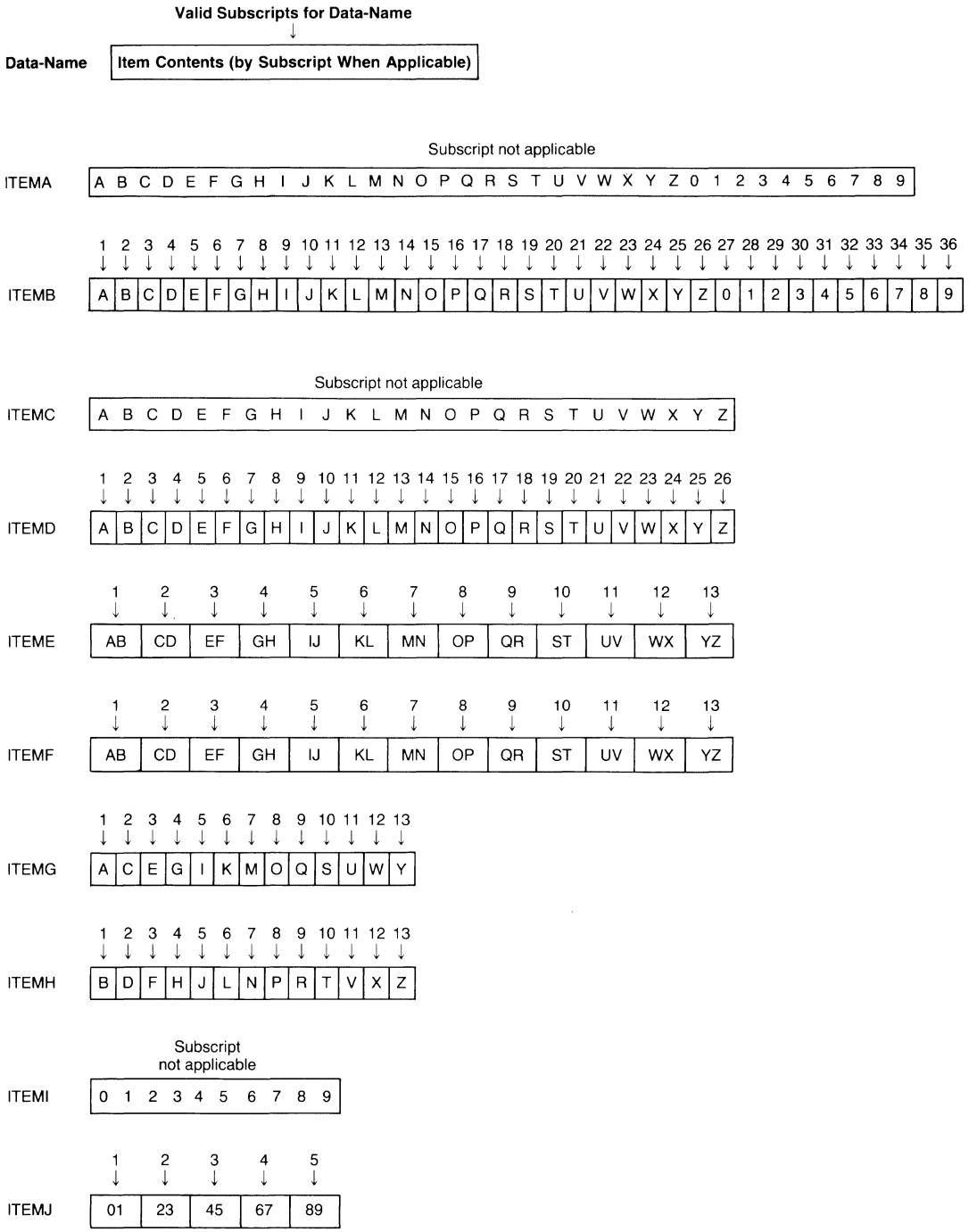
This example shows:

- A sample program containing multiple redefinitions of the same area
- The results of the sample program statements
- The allowable subscripts and the contents for each data item in the program

```
 1 IDENTIFICATION DIVISION.
 2 PROGRAM-ID. REDEFINES-TEST.
 3 DATA DIVISION.
 4 WORKING-STORAGE SECTION.
 5 01  ITEMA.
 6     03  FILLER PIC X(26) VALUE "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
 7     03  FILLER PIC X(10) VALUE "0123456789".
 8 01  REDEFINES ITEMA.
 9     03  ITEMB  OCCURS 36 TIMES PIC X.
10 01  REDEFINES ITEMA.
11     03  ITEMC.
12         05 ITEMD OCCURS 26 TIMES PIC X.
13     03  REDEFINES ITEMC.
14         05  ITEME OCCURS 13 TIMES.
15             07  ITEMF PIC XX.
16             07  REDEFINES ITEMF.
17                 09  ITEMG PIC X.
18                 09  ITEMH PIC X.
19     03  ITEMI.
20         05  ITEMJ OCCURS 5 TIMES PIC XX.
21 PROCEDURE DIVISION.
22 XXX.
23     DISPLAY ITEMA.
24     DISPLAY ITEMB(1).
25     DISPLAY ITEMB(26).
26     DISPLAY ITEMB(27).
27     DISPLAY ITEMB(36).
28     DISPLAY ITEMC.
29     DISPLAY ITEMD(1).
30     DISPLAY ITEMD(26).
31     DISPLAY ITEME(1).
32     DISPLAY ITEME(13).
33     DISPLAY ITEMF(1).
34     DISPLAY ITEMF(13).
35     DISPLAY ITEMG(1).
36     DISPLAY ITEMG(13).
37     DISPLAY ITEMH(1).
38     DISPLAY ITEMH(13).
39     DISPLAY ITEMI.
40     DISPLAY ITEMJ(1).
41     DISPLAY ITEMJ(5).
42     STOP RUN.
```

Results

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
A
Z
0
9
ABCDEFGHIJKLMNOPQRSTUVWXYZ
A
Z
AB
YZ
AB
YZ
A
Y
B
Z
0123456789
01
89
```

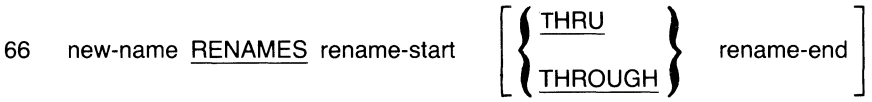

Valid Subscripts for Data-Name
Data-Name
Item Contents (by Subscript When Applicable)
Subscript not applicable
ITEMA
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
ITEMB
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9
Subscript not applicable
ITEMC
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
ITEMD
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
1 2 3 4 5 6 7 8 9 10 11 12 13
ITEME
AB CD EF GH IJ KL MN OP QR ST UV WX YZ
ITEMF
AB CD EF GH IJ KL MN OP QR ST UV WX YZ
1 2 3 4 5 6 7 8 9 10 11 12 13
ITEMG
A C E G I K M O Q S U W Y
ITEMH
B D F H J L N P R T V X Z
Subscript
not applicable
ITEMI
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5
ITEMJ
01 23 45 67 89

4.2.17 RENAMES Clause

Function

The RENAMES clause groups elementary items in alternative or overlapping ways.

General Format



new-name
: is the data-name of the item being described. It identifies an alternate grouping of one or more items in a record.

rename-start
: is the data-name of the leftmost data item in the area. It can be qualified.

rename-end
: is the data-name of the rightmost data item in the area. It can be qualified.

Note

Level-number 66 and *new-name* are not actually part of the RENAMES clause. They are in the general format only to clarify the relationships between the clauses.

Syntax Rules

1. A logical record can have any number of RENAMES entries.
2. All RENAMES entries referring to data items in a logical record must immediately follow the last data description entry of the record description entry.
3. The program cannot qualify data-names with *new-name*.
4. The program can qualify *new-name* only by the names of the associated level 01, FD, or SD entries.
5. The data description entries for *rename-start* and *rename-end*:
   - Cannot have an OCCURS clause
   - Cannot be subordinate to an item whose data description entry has an OCCURS clause
6. *Rename-start* and *rename-end* must be the names of elementary items or groups of elementary items in the same logical record. They cannot be the same data-name.
7. A level 66 entry cannot rename another level 66 entry. Nor can it rename a level 88, level 01, or level 77 entry.

8. None of the items in the range, including *rename-start* and *rename-end*, can be variable occurrence data items.
9. The words THRU and THROUGH are equivalent.
10. *Rename-end* cannot be subordinate to *rename-start*. The beginning of *rename-end* cannot be to the left of the beginning of *rename-start*. The end of *rename-end* must be to the right of the end of *rename-start*.

General Rules

1. If *rename-end* is used, *new-name* includes all elementary items:
   - Starting with: (a) *rename-start*, if *rename-start* is an elementary item, or (b) the first elementary item in *rename-start*, if *rename-start* is a group item
   - Ending with: (a) *rename-end*, if *rename-end* is an elementary item, or (b) the last elementary item in *rename-end*, if *rename-end* is a group item
2. If *rename-end* is not used, all data attributes for *rename-start* become data attributes for *new-name*. In this case, you are renaming a single data item. If that item is a group item, *new-name* is also treated as a group item. If that item is an elementary item, *new-name* is also treated as an elementary item.

Additional Reference

Section 4.2.3 Data Description

Example

```
WORKING-STORAGE SECTION.
01  AA.
    02  BB   PIX XX    VALUE "$$".
    02  F    PIC X     VALUE "=".
    66  B-CODE RENAMES BB.

01  A.
    02  B    PIC XX    VALUE "3-".
    02  C    PIC XX    VALUE "2-".
    02  D    PIC XX    VALUE "1-".
    02  E    PIC X(9)  VALUE "Blast Off".
    66  F    RENAMES B THROUGH E.

PROCEDURE DIVISION.
000-BEGIN.
    DISPLAY BB.
    DISPLAY B-CODE.
    DISPLAY B.
    DISPLAY C.
    DISPLAY D.
    DISPLAY E.
    DISPLAY F OF A.
    STOP RUN.
```

Results

```
$$
$$
3-
2-
1-
Blast Off
3-2-1-Blast Off
```

4.2.18 SIGN Clause

Function

The SIGN clause specifies the operational sign's position and type of representation.

General Format

```
[ SIGN IS ] { LEADING  } [ SEPARATE CHARACTER ]
            { TRAILING }
```

Syntax Rules

1. The SIGN clause can be used only in a numeric data description entry whose PICTURE contains the S symbol. It can also be used for a group item containing such entries.
2. The data items to which the SIGN clause applies must have display usage.
3. If a file description entry has a CODE-SET clause, all signed numeric data description entries associated with the file must contain the SIGN IS SEPARATE clause.

General Rules

1. The SIGN clause specifies the operational sign's position and type of representation. It applies to a numeric data description entry or to each numeric data description entry subordinate to a group.
2. The SIGN clause applies only to numeric data description entries whose PICTURE clause contains the S symbol. S indicates the presence of an operational sign. However, S does not specify the sign's representation or, necessarily, its position.
3. If you specify the SIGN clause for both a group item and a group item subordinate to it, the SIGN clause for the subordinate group overrides the group item SIGN clause.
4. If you specify the SIGN clause for both a group item and an elementary numeric item subordinate to it, the SIGN clause for the elementary item overrides the group item SIGN clause.
5. A numeric data description entry to which no optional SIGN clause applies, but whose PICTURE contains an S symbol, has an operational sign. The entry is equivalent to an entry that contains the SIGN IS TRAILING clause without the SEPARATE CHARACTER phrase.
6. If you specify the SEPARATE CHARACTER phrase:
   - The operational sign is the leading (or trailing) character position of the elementary numeric data item. The sign does not share this position with a digit.
   - The S symbol in the PICTURE counts toward data item size. That is, it represents a character position.
   - The operational signs for positive and negative are the characters + and –.

7. If you do not specify the SEPARATE CHARACTER phrase:
   - The operational sign is associated with the leading (or trailing) digit position of the elementary numeric item. The sign shares this character position with a digit.
   - The S symbol in the PICTURE does not count toward data item size. That is, it does not represent a character position.
   - The character in the operational sign position represents both a numeric digit and the item's algebraic sign. Table 4-7 shows the characters representing positive and negative signs for all numeric digits. Where more than one character appears, the first is the character generated as the result of machine operations.

Table 4-7: Positive and Negative Signs for All Numeric Digits

Digit Values	Sign	
	Positive	Negative
0	{, [, ?, or 0	},], :, or !
1	A or 1	J
2	B or 2	K
3	C or 3	L
4	D or 4	M
5	E or 5	N
6	F or 6	O
7	G or 7	P
8	H or 8	Q
9	I or 9	R

8. Every numeric data item whose PICTURE contains the S symbol is a signed numeric data item. If you specify the SIGN clause for such an item, necessary conversions for computations or comparisons occur automatically.

4.2.19 SYNCHRONIZED Clause

Function

The SYNCHRONIZED clause specifies elementary item alignment on word boundary offsets relative to a record's beginning. These offsets are related to the size and usage of the item being stored.

General Format

Syntax Rules

1. SYNC is an abbreviation for SYNCHRONIZED.
2. The SYNCHRONIZED clause can be used only for an elementary item.

General Rules

1. The SYNCHRONIZED clause aligns a data item in a record so that no other data item occupies any character positions between the required boundaries to the left and right of the data item.
2. The SYNCHRONIZED clause does not change the size or operational sign position of the data item it specifies.
3. If the number of character positions needed to store the data item is less than the number of positions between the required boundaries, no other data items occupy the unused positions.

   However, the unused character positions are included in the size of those group item(s):

   - To which the elementary item belongs
   - In which the elementary item is not the first subordinate item

   In other words, the SYNCHRONIZED clause can increase the size of a group item only when the specified data item is both subordinate to the group item and is other than the first elementary data item in the group. (The first elementary item in a group item always aligns on the same boundary as the group item. In this case, any unused character positions do not affect the size of that group item.)
4. The LEFT and RIGHT phrases have the same effect; they are equivalent to: (a) each other and (b) the SYNCHRONIZED clause with neither the LEFT nor RIGHT phrases.
5. Each occurrence of the data item is SYNCHRONIZED if the clause applies to: (a) a data item whose data description entry also has an OCCURS clause, or (b) a data item subordinate to another data item whose data description entry has an OCCURS clause.

Technical Notes

1. The SYNCHRONIZED clause affects alignment only of COMP data items.
2. Specify the SYNCHRONIZED clause for COMP data items to ensure COBOL-81 compatibility with VAX–11 COBOL.
3. All records are aligned on 2-byte (word) boundaries. A COMP SYNC data item must align on the same boundary as the record containing it, or align on a boundary that is in 2-, 4-, or 8-byte increments from the record boundary.
4. Table 4-8 illustrates the effect of the SYNCHRONIZED clause on data item alignment.

Table 4-8: COMP and COMP SYNC Alignment Differences

Data Type	Required Boundary
PIC 9 to 9(4) COMP	2-byte
PIC 9 to 9(4) COMP SYNC	2-byte
PIC 9(5) to 9(9) COMP	2-byte
PIC 9(5) to 9(9) COMP SYNC	4-byte
PIC 9(10) to 9(18) COMP	2-byte
PIC 9(10) to 9(18) COMP SYNC	8-byte

Additional References

Section 4.1.2.3 Record Allocation

Appendix D Ensuring COBOL-81 Compatibility with VAX–11 COBOL

4.2.20 USAGE Clause

Function

The USAGE clause specifies the internal format of a data item.

General Format

```
[ USAGE IS ]  { COMPUTATIONAL   }
              { COMP            }
              { COMPUTATIONAL-3 }
              { COMP-3          }
              { DISPLAY         }
              { INDEX           }
```

Syntax Rules

1. COMP is an abbreviation for COMPUTATIONAL.
2. COMP-3 is an abbreviation for COMPUTATIONAL-3.
3. You can use the USAGE clause in any data description entry with a level-number other than 66 or 88.
4. If the USAGE clause is in the data description for a group item, it can also be in data description entries for subordinate elementary and group items. However, the usage of a subordinate item must be the same as that in the group item data description entry.
5. The PICTURE character-string of a COMP or COMP-3 item can contain only the symbols 9, S, V, and P.
6. An index data item reference can appear in only:
   - A SEARCH or SET statement
   - A relation condition
   - The USING phrase of the Procedure Division header
   - The USING phrase of the CALL statement
7. The data description entry for a USAGE IS INDEX data item cannot contain any of the following clauses:
   - BLANK WHEN ZERO
   - JUSTIFIED
   - PICTURE
   - VALUE IS
8. An elementary item with the USAGE IS INDEX clause cannot be a conditional variable; that is, the elementary item's value cannot be specified by level 88 items.

General Rules

1. You can specify the USAGE clause in the data description entry for a group item. In this case, it applies to each elementary item in the group. However, you cannot reference the group item in any operations that do not permit alphanumeric operands. (See rules 4 and 8.)
2. The USAGE clause specifies the representation of an elementary data item in storage. It does not affect the way that the program uses the item. However, the rules for some Procedure Division statements restrict the USAGE clause of statement operands.
3. A COMP, or COMP-3 item can represent a value used in computations. The PICTURE clauses for COMP and COMP-3 items must be numeric.
4. If the data description entry for a group item specifies COMP or COMP-3 usage, the usage applies to elementary items in the group. It does not apply to the group itself; and the program cannot use the group item in computations.
5. The USAGE IS DISPLAY clause specifies that the data item is in Standard Data Format.
6. If no USAGE clause applies to an elementary item, its usage is DISPLAY.
7. If the USAGE IS INDEX clause applies to an elementary item, the elementary item is called an index data item. It contains a value that must correspond to an occurrence number of a table element.
8. If the data description entry for a group item specifies USAGE IS INDEX, all elementary items in the group are index data items. However, the group itself is not an index data item.
9. When a MOVE or input-output statement refers to a group that contains an index data item, the index data item is not converted to another format during the operation.

Technical Notes

1. COMP is the standard binary format. A COMP item is a binary value with an assumed decimal scaling position. Depending on its size, it occupies two, four, or eight bytes in storage. (Refer to Tables 4-9 and 4-10 to determine storage allocation, as related to data item size.)

Note

COMP and COMP SYNC data items have the same storage requirements. The SYNCHRONIZED (SYNC) clause affects the alignment of the computational data item, but not its size. However, a record containing COMP SYNC data items can, under certain circumstances, occupy more bytes in storage than a record containing COMP data items. Refer to Section 4.1.2.3 and Section 4.2.19 for more information.

2. COMP-3 specifies the packed-decimal format. COMP-3 items are stored two decimal digits per byte with an assumed decimal scaling position. The sign occupies the rightmost (least significant) four bits of the rightmost byte.

   If the PICTURE for a COMP-3 item specifies an even number of decimal digits, the value zero occupies the leftmost (most significant) four bits of the leftmost byte.

   Signs resulting from operations in which the receiving item usage is COMP-3 are:

   Positive sign: binary 1100, hexadecimal C
   Negative sign: binary 1101, hexadecimal D
   Unsigned: binary 1111, hexadecimal F

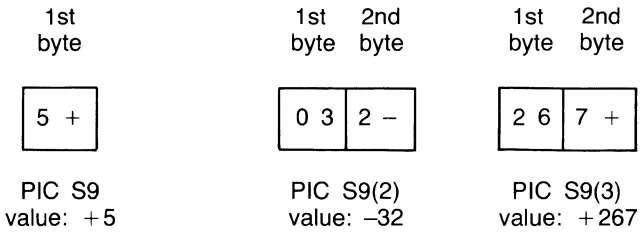
   The following signs are also valid. However, they do not result from program operations. COBOL-81 recognizes them as possible sign representations used by non-DIGITAL systems.

   Positive signs: binary 1010, hexadecimal A
   binary 1110, hexadecimal E

   Negative signs: binary 1011, hexadecimal B

   Figure 4-10 represents the storage format of COMP-3 items with one, two, and three digits.

Figure 4-10: Storage Format of COMP-3 Data Items



3. An index data item is stored as a one-word COMP item.
4. The way a data item is represented in the Data Division of a COBOL program determines whether it will be stored as an integer, packed decimal, display numeric, or character string (text) data type. The following tables:

   a. Match COBOL data description entries with their corresponding storage data types

   b. Show the allocated storage in bytes for the entry.

   Table 4-9 gives the corresponding data types for unscaled data items, and Table 4-10 gives the data types for scaled data items.

   For example, a data item described as PIC S9(4) USAGE IS DISPLAY SIGN IS TRAILING would occupy four bytes of storage as a right overpunch value.

Note

The default USAGE for a data item is DISPLAY. Therefore, you do not need to specify the USAGE clause for display numeric, alphabetic, and alphanumeric data items.

Table 4-9: Unscaled Data Items and Corresponding Storage Data Types

Unscaled Data Item			Storage Data Type
PICTURE Clause	**USAGE Clause**	**Storage Allocated in Bytes**	
PIC S9(n) [n < = 18]	USAGE IS DISPLAY	n	Right overpunch
PIC S9(n) [n < = 18]	USAGE IS DISPLAY SIGN IS TRAILING	n	Right overpunch
PIC S9(n) [n < = 18]	USAGE IS DISPLAY SIGN IS LEADING	n	Left overpunch
PIC S9(n) [n < = 18]	USAGE IS DISPLAY SIGN IS TRAILING SEPARATE	n + 1	Right separate
PIC S9(n) [n < = 18]	USAGE IS DISPLAY SIGN IS LEADING SEPARATE	n + 1	Left separate
PIC 9(n) [n < = 18]	USAGE IS DISPLAY	n	Unsigned numeric
PIC 9(n) [n < = 4]	USAGE IS COMP	2	Word integer*
PIC 9(n) [5 < = n < = 9]	USAGE IS COMP	4	Two word integer*
PIC 9(n) [10 < = n < = 18]	USAGE IS COMP	8	Four word integer*
PIC S9(n) [n < = 4]	USAGE IS COMP	2	Word integer
PIC S9(n) [5 < = n < = 9]	USAGE IS COMP	4	Two word integer
PIC S9(n) [10 < = n < = 18]	USAGE IS COMP	8	Four word integer
N/A	USAGE IS INDEX	2	One word integer
PIC S9(n) [n < = 18]	USAGE IS COMP-3	(n + 1)/2 rounded up	Packed decimal
PIC 9(n) [n < = 18]	USAGE IS COMP-3	(n + 1)/2 rounded up	Packed decimal*
PIC X(n) [n < = 65,535]	USAGE IS DISPLAY	n	ASCII Text
PIC A(n) [n < = 65,535]	USAGE IS DISPLAY	n	ASCII Text

Legend:

* The generated code treats this data type as a signed operand in all contexts except when it is a receiving-field operand. In this case, the compiler stores the absolute value of the data type.

N/A Not Applicable

USAGE
Continued

Table 4-10: Scaled Data Items and Corresponding Storage Data Types

Scaled Data Item			Storage Data Type
PICTURE Clause	USAGE Clause	Storage Allocated in Bytes	
PIC S9(n)V9(s) [(n+s) <= 18]	USAGE IS DISPLAY	n+s	Right (trailing) overpunch
PIC S9(n)V9(s) [(n+s) <= 18]	USAGE IS DISPLAY SIGN IS TRAILING	n+s	Right (trailing) overpunch
PIC S9(n)V9(s) [(n+s) <= 18]	USAGE IS DISPLAY SIGN IS LEADING	n+s	Left (leading) overpunch
PIC S9(n)V9(s) [(n+s) <= 18]	USAGE IS DISPLAY SIGN IS TRAILING SEPARATE	n+s+1	Right (trailing) separate
PIC S9(n)V9(s) [(n+s) <= 18]	USAGE IS DISPLAY SIGN IS LEADING SEPARATE	n+s+1	Left (leading) separate
PIC 9(n)V9(s) [(n+s) <= 18]	USAGE IS DISPLAY	n+s	Unsigned numeric
PIC 9(n)V9(s) [(n+s) <= 4]	USAGE IS COMP	2	Word integer*
PIC 9(n)V9(s) [5 <= (n+s) <= 9]	USAGE IS COMP	4	Two word integer*
PIC 9(n)V9(s) [10 <= (n+s) <= 18]	USAGE IS COMP	8	Four word integer*
PIC S9(n)V9(s) [(n+s) <= 4]	USAGE IS COMP	2	Word integer
PIC S9(n)V9(s) [5 <= (n+s) <= 9]	USAGE IS COMP	4	Two word integer
PIC S9(n)V9(s) [10 <= (n+s) <= 18]	USAGE IS COMP	8	Four word integer
PIC 9(n)V9(s) [(n+s) <= 18]	USAGE IS COMP-3	(n+s+1)/2 rounded up	Packed decimal*
PIC S9(n)V9(s) [(n+s) <= 18]	USAGE IS COMP-3	(n+s+1)/2 rounded up	Packed decimal

Legend:

* The generated code treats this data type as a signed operand in all contexts except when it is a receiving-field operand. In this case, the compiler stores the absolute value of the data type.

N/A Not Applicable

Additional References

Section 4.2.14 PICTURE Clause

4.2.21 VALUE IS Clause

Function

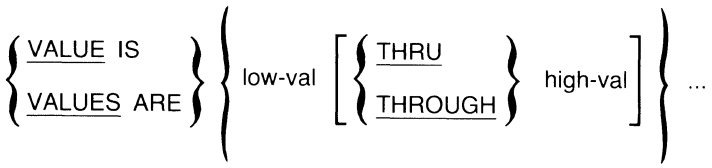
The VALUE IS clause defines the values associated with condition-names and the initial values of Working-Storage Section data items.

General Format

Format 1

```
VALUE IS lit
```

Format 2



lit
: is a numeric or nonnumeric literal.

low-val
: is a numeric or nonnumeric literal. It is the lowest value in a range of values associated with a condition-name in a level 88 data description entry.

high-val
: is a numeric or nonnumeric literal. It is the highest value in a range of values associated with a condition-name in a level 88 data description entry.

Syntax Rules

1. The words THRU and THROUGH are equivalent.
2. You can associate a signed numeric literal only with a data item that has a signed numeric PICTURE character-string.
3. If you specify a numeric literal value:
   a. It must fall in the range of values defined by the data item's PICTURE clause.
   b. It must not require truncation of nonzero digits; that is, it cannot have nonzero digits in positions represented by Ps in the item's PICTURE clause.
4. If you specify a nonnumeric literal value, it must not exceed the size defined by the data item's PICTURE clause.

General Rules

1. The VALUE IS clause must be consistent with other clauses in the data description of both the item and all subordinate items. The following rules apply:
   - If the category of the item is numeric, all literals in the VALUE IS clause must be numeric. *Lit* is aligned in the data item according to Standard Alignment Rule 1.

VALUE IS
Continued

- If the category of the item is alphabetic, alphanumeric, alphanumeric edited, or numeric edited, all VALUE IS clause literals must be nonnumeric. *Lit* is aligned in the data item as if the data item were defined as alphanumeric. Editing characters in the PICTURE clause count toward data item size but have no effect on initialization. Therefore, if *lit* applies to an edited item, it must be in an edited form; Standard Alignment Rule 3 applies.
- The BLANK WHEN ZERO and JUSTIFIED clauses do not affect initialization.

2. In the File and Linkage Sections, the VALUE IS clause can apply only to condition-name entries. That is, you can use the clause only for level 88 data items.
3. Format 2 applies only to condition-name entries.

Condition-Name Rules for Format 2

4. The VALUE IS clause is required in a condition-name entry. The condition-name entry can contain only the condition-name itself and the VALUE IS clause.
5. The characteristics of a condition-name are implicitly the same as those of its conditional variable.
6. When you use the THRU phrase, each *low-val* must be less than the corresponding *high-val*.

Rules for Other Data Description Entries

7. A Working-Storage Section VALUE IS clause takes effect only when the program enters its initial state.
8. The VALUE IS clause initializes the data item to the value of *lit*.
9. If a data item's data description entry does not have a VALUE IS clause, the initial contents of the data item are undefined. However, index-names (items defined by the INDEXED BY phrase of the OCCURS clause) are preset to occurrence number 1.
10. The VALUE IS clause cannot be used in a data description entry that: (a) has an OCCURS or REDEFINES clause or (b) is subordinate to a data description entry with an OCCURS or REDEFINES clause.
11. The VALUE IS clause can be in a data description entry for a group item. In this case:
    - *Lit* must be a figurative constant or nonnumeric literal.

      The group area is initialized as if the group were an elementary alphanumeric data item.
    - Initialization of group items is not affected by the characteristics of the group's subordinate group or elementary items.

      The VALUE IS clause cannot be used in data description entries for the group's subordinate group or elementary items.

12. The VALUE IS clause cannot be used in the data description entry for a group that contains subordinate items with any of these clauses:
    - JUSTIFIED
    - SYNCHRONIZED
    - USAGE (other than USAGE IS DISPLAY)

Additional References

Section 1.1.2.1	User-Defined Words (condition-name)
Section 2.1	PROGRAM-ID Paragraph
Section 4.1.2.2	Standard Alignment Rules
Section 4.2.14	PICTURE Clause
Section 4.2.20	USAGE Clause

Examples

1. Initializing alphanumeric data items:

```
01  ITEMA  PIC X(20) VALUE IS "12345678901234567890".
01  ITEMB  PIC XX    VALUE IS "NH".
```

2. Initializing numeric data items:

```
01  ITEMX  PIC S9999 VALUE IS -39.
01  ITEMZ  PIC 9     VALUE ZERO.
```

3. Assigning condition-name values:

```
01  ITEMC  PIC 99.
       88  VAL1      VALUE IS 4.
       88  VAL2      VALUE IS 22.
       88  VAL2      VALUE IS 5 THRU 9 12.
       88  VAL3      VALUES ARE 10 14 THRU 23 27 29 30.
       88  VAL4      VALUES ARE 0 THRU 49, 51 THRU 99.
       88  VAL5      VALUES ARE 0 10 20 30 40 50.
```

4.2.22 VALUE OF ID Clause

Function

The VALUE OF ID clause specifies, replaces, or completes a file specification.

General Format

```
VALUE OF ID IS file-spec
```

file-spec
is a nonnumeric literal or the data-name of an alphanumeric Working-Storage Section data item. It contains the full or partial file specification. *File-spec* can be qualified.

General Rules

1. Each file specification field in *file-spec* augments the specification in the SELECT clause ASSIGN phrase.
2. A file specification field in *file-spec* overrides the corresponding field in the SELECT clause. If a file specification field is either in the SELECT clause or in *file-spec* (but not in both), it becomes part of the file specification.

Technical Notes

1. *File-spec* is a complete or partial file specification in PDP-11 Record Management Services (RMS-11) format.
2. If the program opens a file in the input, I-O, or extend mode, RMS-11 uses *file-spec* to locate the existing file.
3. If the program opens a file in the output mode or opens a nonexistent file in the extend or I-O mode, RMS-11 uses *file-spec* to name the new file.
4. If the VALUE OF ID phrase is present, it supplies the contents for the RMS-11 FNA field in the FAB (File Access Block), and the file's ASSIGN clause supplies the contents of the RMS-11 DNA field of the RAB (Record Access Block).
5. If there is no VALUE OF ID phrase, the file's ASSIGN clause supplies the contents of the RMS-11 FNA field in the RAB.

Additional Reference

Part IV of the COBOL-81 User's Guide for your system	Processing Files and Records

Examples

Value of filename in SELECT clause	Value of literal or data-name in VALUE OF ID clause	Resulting file specification
MM1:FILEA.DAT	OFFICE.DAT	MM1:OFFICE.DAT
FILEB	MM0:FILEB.316	MM0:FILEB.316
FILEC.LIB	DK0:	DK0:FILEC.LIB

Chapter 5
Procedure Division

This chapter includes the general formats for all Procedure Division statements, describes their basic elements, and explains how to use them.

5.1 Verbs, Statements, and Sentences

A COBOL verb is a reserved word that expresses an action to be taken by the compiler or the object program. A verb and its operands make up a COBOL statement. One or more statements that are terminated by a separator period form a COBOL sentence.

At the statement level, actions can be further differentiated: actions taken by the object program can be conditional or unconditional. In some cases, the verb in the statement defines whether the action is conditional or unconditional. One verb, IF, always defines a conditional action. Other verbs, such as READ, always define conditional action because you must use phrases with them that make the action conditional. PERFORM and MOVE are examples of verbs that always define unconditional action. Most often, however, whether an action is conditional or unconditional depends on not only which verb, but also which phrase(s) you use in the statement.

There are three types of COBOL statements:

- *Compiler-directing* statements specify an action taken by the compiler during compilation.
- *Imperative* statements specify an unconditional action taken by the object program at run time.
- *Conditional* statements specify a conditional action taken by the object program at run time; the action depends upon a truth value that is generated by the program. (A truth value is either a "yes" or "no" answer to the question, "Is the condition true?")

Table 5-1 shows the three types of COBOL statements. It also shows that the imperative statements are further subdivided into nine categories and specifies the verbs that each category includes. When associated phrases are not specified, the verb alone defines the category. For compiler-directing and conditional statements, type and category are synonymous.

Table 5-1: Types and Categories of COBOL Statements

Type	Category	Verb
Compiler-Directing	Compiler-Directing	COPY USE
Conditional	Conditional	ACCEPT (ON EXCEPTION) ADD (SIZE ERROR) COMPUTE (SIZE ERROR) DELETE (INVALID KEY) DIVIDE (SIZE ERROR) IF MULTIPLY (SIZE ERROR) READ (AT END or INVALID KEY) RETURN REWRITE (INVALID KEY) SEARCH START (INVALID KEY) STRING (OVERFLOW) SUBTRACT (SIZE ERROR) UNSTRING (OVERFLOW) WRITE (INVALID KEY or END-OF-PAGE)
Imperative	Arithmetic	ADD (1) COMPUTE (1) DIVIDE (1) INSPECT (TALLYING) MULTIPLY (1) SUBTRACT (1)
	Data-Movement	ACCEPT (DATE, DAY, or TIME) INSPECT (REPLACING) MOVE STRING (4) UNSTRING (4)
	Ending	STOP
	Input-Output	ACCEPT (identifier) CLOSE DELETE (2) DISPLAY OPEN REWRITE (2) START (2) STOP (literal) WRITE (5)
	Interprogram-Communication	CALL
	Procedure-Branching	CALL EXIT GO TO PERFORM
	Table-Handling	SEARCH SET (TO, UP BY, or DOWN BY)
	Record-Ordering	MERGE RELEASE RETURN SORT

Legend:

(1) Without the optional SIZE ERROR phrase
(2) Without the optional INVALID KEY phrase
(3) Without the optional AT END or INVALID KEY phrase
(4) Without the optional OVERFLOW phrase
(5) Without the optional INVALID KEY or END-OF-PAGE phrase
(6) Without the optional ON EXCEPTION phrase

Like statements, COBOL sentences also can be compiler-directing, imperative, or conditional. Sentence type depends upon the type(s) of statement the sentence contains. Table 5-2 summarizes the contents of the three types of COBOL sentences. The remaining text in this section discusses each type of statement and sentence in greater detail.

Table 5-2: Contents of COBOL Sentences

Type	Contents of Sentence
Imperative	One or more consecutive imperative statements ending with a period
Conditional	One or more statements ending with a period; at least one conditional statement
Compiler-Directing	Only one compiler-directing statement ending with a period

5.1.1 Compiler-Directing Statements and Sentences

A compiler-directing statement causes the compiler to take an action during compilation. The verbs COPY or USE define a compiler-directing statement. When it is part of a sentence that contains more than one statement, the COPY or USE statement must be the last statement in the sentence.

A compiler-directing sentence is one COPY or USE statement that ends with a period.

5.1.2 Imperative Statements and Sentences

An imperative statement specifies an unconditional action for the program. It must contain a verb and the verb's operands, and cannot contain any conditional phrases. For example, the following statements are imperative:

```
OPEN INPUT FILE-A

COMPUTE C = A + B
```

However, the following statement is not imperative because it contains the phrase, ON SIZE ERROR, which makes the program's action conditional:

```
COMPUTE C = A + B ON SIZE ERROR PERFORM NUM-TOO-BIG.
```

In the Procedure Division rules, "imperative statement" can be a sequence of consecutive imperative statements. The sequence must end with: (1) a separator period or (2) any phrase associated with a statement that contains the imperative statement. For example, the following sentence contains a sequence of two imperative statements following the AT END phrase.

```
READ FILE-A AT END PERFORM NO-MORE-RECS
                   DISPLAY "No more records.".
```

An imperative sentence contains only imperative statements and ends with a separator period.

5.1.3 Conditional Statements

A conditional statement determines a condition's truth value. The statement uses the truth value to determine subsequent program action.

Conditional statements are:

- An IF, RETURN, or SEARCH statement
- A READ statement with the AT END or INVALID KEY phrase
- A WRITE statement with the INVALID KEY or END-OF-PAGE phrase
- A DELETE, REWRITE, or START statement with the INVALID KEY phrase
- An arithmetic statement (ADD, COMPUTE, DIVIDE, MULTIPLY, SUBTRACT) with the SIZE ERROR phrase
- A STRING or UNSTRING statement with the OVERFLOW phrase

A conditional sentence must contain one conditional statement and end with a separator period. It can include an imperative statement. For example, the following sentence is conditional even though it contains the imperative statement, GO TO PROC-A:

```
READ FILEA AT END GO TO PROC-A.
```

The program interprets this sentence to mean "If not at the end of the file, read the next record; otherwise, go to PROC-A."

5.1.4 Scope of Statements

A statement can be delimited by a period or by the start of the next statement that follows it in a series. For example, consider:

```
MOVE ITEMC TO ITEMB READ FILEA.
```

The statement "READ FILEA" is terminated by the period, while the statement "MOVE ITEMC TO ITEMB" is terminated by the word "READ".

When statements are contained (or nested) in other statements, the period that terminates the sentence terminates all nested statements as well.

In the following example, the period terminates the IF, READ, and PERFORM statements. The MOVE statement is terminated by the word "PERFORM".

```
IF ITEMA = ITEMB
   READ FILEA
        AT END MOVE ITEMC TO ITEMB
               PERFORM PROCA.
```

When one statement, B, is contained in another statement, A, the first phrase of A that follows B terminates B. This rule is illustrated by the following IF statement:

```
IF ITEMA = ITEMB
   READ FILEA AT END
      MOVE ITEMC TO ITEMB
ELSE
   PERFORM PROCA.
```

The READ and MOVE statements are terminated by the ELSE phrase of the IF statement.

5.2 Transfer of Program Flow

As a program executes, control transfers implicitly from one executable statement to the next in the order the statements appear in the source program. Except for the following explicit and implicit changes, the transfer occurs in this sequence until there is no next executable statement.

There is no next executable statement when:

- The last statement in a declarative is not executing under the control of another COBOL statement. This situation occurs if that declarative was reached by a GO TO from another declarative. If this occurs in a called program, an implicit EXIT PROGRAM statement executes; otherwise, an implicit STOP RUN statement executes.
- The last statement in a program is not executing under the control of another COBOL statement. If this occurs in a called program, an implicit EXIT PROGRAM statement executes; otherwise, an implicit STOP RUN statement executes.
- The program executes an explicit EXIT PROGRAM or STOP RUN statement.

5.2.1 Explicit Changes

An explicit control transfer can change the sequence just described. Procedure-branching statements and conditional statements cause explicit transfers. (Note that the procedure-branching statement EXIT must have the PROGRAM phrase to cause an explicit transfer.)

5.2.2 Implicit Changes

Three situations cause implicit changes to the transfer sequence described in the previous section.

1. If the Procedure Division has declaratives, the first statement to execute in the Procedure Division is the first statement after the declaratives.

2. When a PERFORM statement executes, program flow transfers to the first section or paragraph in the PERFORM range. At the end of the range, flow transfers back to the PERFORM statement. This procedure occurs as many times as the PERFORM specifies, or until the condition specified by the UNTIL or VARYING phrase is satisfied. (If no phrase is specified, the default is TIMES = 1.) When either the TIMES option has been fulfilled, or the UNTIL or VARYING condition has been satisfied, flow transfers to the next statement after PERFORM.
3. When a Procedure Division statement causes an error condition for which there is a USE procedure, program flow transfers to that corresponding declarative. Once the declarative has executed, flow transfers back to the statement following the one that caused the error.

5.3 Uniqueness of Reference

When Procedure Division statements refer to a word defined in your program, they must refer to a unique data item, condition, or set of procedures. (See Section 1.1.2.1, User-Defined Words.) Qualification and subscripting or indexing are methods you use in these references to avoid ambiguity that would otherwise be present. For example, more than one data item, condition, or set of procedures in your program can have the same name; a statement can qualify that name so that it is unambiguous. Also, tables contain more than one occurrence of a data item; subscripting and indexing specify a unique occurrence of that item.

5.3.1 Qualification

A reference to a user-defined word is unique if: (1) no other name has the same spelling, including hyphenation or (2) it is part of a REDEFINES clause. (The reference following the word REDEFINES is unique because of clause placement.)

A name in a hierarchy of names can occur in more than one place in your program. Unless you are redefining it, you must refer to this nonunique name using one or more higher level names in the hierarchy. These higher-level names are called *qualifiers*. Using them to achieve uniqueness of reference is called *qualification*.

To make your reference unique, you need not specify all available qualifiers for a name, only the one(s) necessary to avoid ambiguity.

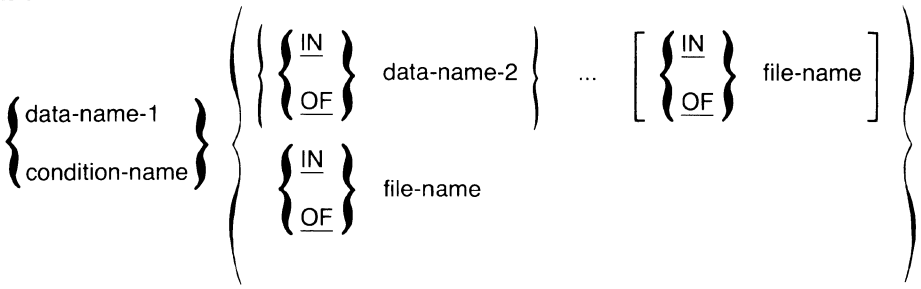
Consider the following two record descriptions:

```
01 REC1.
    05  ITEMA          PIC XX.
    05  ITEMB          PIC X(20).

01 REC2.
    05  GROUP1.
        10  ITEMA      PIC 9(5).
        10  ITEMB      PIC X(3).
    05  GROUP2.
        10  ITEMC      PIC X(4).
        10  ITEMD      PIC X(8).
```

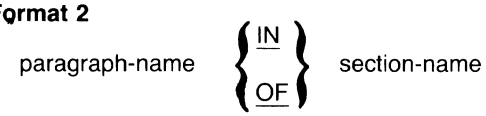
ITEMA and ITEMB appear in both record descriptions. Therefore, you must use qualifiers when you refer to these items in Procedure Division statements. For example, all of the following references to ITEMA are unique: ITEMA OF GROUP1, ITEMA OF REC1, ITEMA IN GROUP1 OF REC2.

The general formats for qualification are as follows::

Format 1

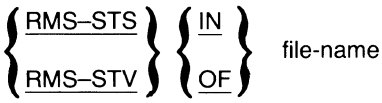


Format 2



Format 3

Format 4



The following syntax rules apply to qualification:

1. Each reference to a nonunique, user-defined name must use a sequence of qualifiers that eliminates ambiguity from the reference.
2. A name can be qualified even if it does not need qualification. If there is more than one set of qualifiers that ensures uniqueness, any set can be used.
3. IN and OF are equivalent.
4. In Format 1, each qualifier must be: (a) the name associated with a level indicator or (b) the name of a group to which the item being qualified is subordinate. Qualifiers must be ordered from least to most inclusive levels in the hierarchy.
5. In Format 1, *data-name-2* can be a record-name.
6. If the program contains explicit references to a paragraph-name, the paragraph-name cannot appear more than once in the same section. When a section-name qualifies a paragraph-name, the word SECTION cannot appear. A paragraph-name need not be qualified in a reference from within the same section.

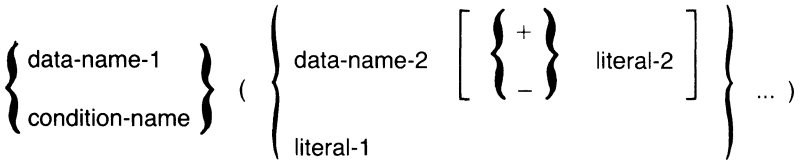
7. If the program has more than one file description entry with a LINAGE clause, every reference to LINAGE-COUNTER must be qualified.
8. If the program has more than one file description entry, every reference to RMS-STS, or RMS-STV, must be qualified.

5.3.2 Subscripts and Indexes

Occurrences of a table are not individually named. You refer to them by using a subscript or index to specify their location relative to the table's beginning. Subscripting is a general procedure; indexing is a special form of subscripting.

5.3.2.1 Subscripting — Subscripts can appear only in references to individual elements in a list, or table, of like elements that do not have individual data-names. (See Section 4.2.13, OCCURS Clause.)

The general format for subscripting is:



Note

Data-name-1 is the name of an item whose data description includes an OCCURS clause; it is not actually part of the subscript.

The rules that apply to subscripting are:

1. A subscript can be either a numeric literal or a data-name (*data-name-2*) with an optional increment (+) or decrement (–).

   If *data-name-2* is the subscript, its value must be an integer. *Data-name-2* cannot be subscripted.
2. The lowest valid subscript value is 1. This value points to the first element of the table. Subscript values 2, 3, and so on point to the next consecutive table elements.
3. The highest valid subscript value is the maximum number of occurrences specified in the OCCURS clause of *data-name-1*.
4. The subscript, or set of subscripts, that identifies the table element is delimited by a balanced pair of left and right parentheses.
5. Each reference to *data-name-1* must use subscripting unless it is:
   a. The subject of a SEARCH statement
   b. In a REDEFINES clause
   c. In the KEY IS phrase of an OCCURS clause

6. The subscript, or set of subscripts, follows *data-name-1*. *Data-name-1* is then called a subscripted data-name or an identifier.
7. The number of subscripts following *data-name-1* must equal the number of dimensions in the table; that is, there must be a subscript for each OCCURS clause in the hierarchy that contains *data-name-1* and also one for *data-name-1* itself.
8. *Data-name-1* can have up to three subscripts.
9. When *data-name-1* requires more than one subscript, its subscripts must appear in the order of successively less inclusive dimensions of the table.

Note

By default, COBOL-81 performs subscript range checking at run time. For more information, refer to the discussion of the /CHECK qualifier to the COBOL command in Part I of the COBOL-81 User's Guide for your system.

The following example defines a two dimensional table and provides sample references to elements in the table. ITEMD is the first dimension of the table; therefore, any reference to ITEMD requires one subscript. ITEME is the second dimension of the table; therefore, any reference to ITEME requires two subscripts (the first to specify the occurrence of ITEMD and the second to specify the occurrence of ITEME within that occurrence of ITEMD).

Example

```
WORKING-STORAGE SECTION.
01   ITEMA PIC 9 COMP VALUE IS 2.
01   ITEMB PIC 9 COMP VALUE IS 3.
01   ITEMC VALUE IS "ABCDEFGHIJKLMNOPQRSTUVWX".
     03   ITEMD OCCURS 4 TIMES.
          05   ITEME OCCURS 6 TIMES PIC X.
```

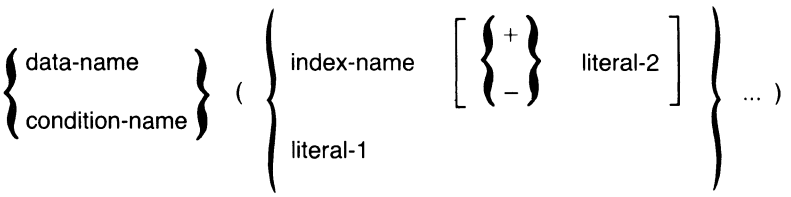
ITEMC	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
ITEMD	1						2						3						4					
ITEME	1	2	3	4	5	6	1	2	3	4	5	6	1	2	3	4	5	6	1	2	3	4	5	6

Identifier	Value
ITEMD (1)	ABCDEF
ITEMD (0)	(invalid reference; zero subscript)
ITEMD (ITEMB)	MNOPQR
ITEMD (6)	(invalid reference; outside OCCURS range)
ITEME (3, 6)	R
ITEME (1, ITEMA)	B
ITEME (ITEMB, ITEMA)	N
ITEME (1, 7)	(invalid reference; outside OCCURS range)

5.3.2.2 Indexing — Indexing is a special subscripting procedure. In indexing, you use the INDEXED BY phrase of the OCCURS clause to assign an index-name to each table level. You then refer to a table element using the index-name as a subscript.

In some table-handling procedures, you might want to store the value of an index-name for later reference. However, the size and storage format of index-name items varies according to computer architecture. You can always store the value of an index-name item in an index data item (an item described with the USAGE IS INDEX clause) and be sure the size and format match. Furthermore, if you always define index data items in the Working-Storage Section, they are compatible with index data items on other computer systems.

The general format for indexing is:



All the restrictions in the rules for subscripting apply to indexing. (See Section 5.3.2.1, Subscripting.) The following rules apply to indexing alone:

1. You must give *index-names* an initial value before using them. You can do this in:
    1. A SET statement
    2. A SEARCH statement with the ALL phrase
    3. A PERFORM statement with the VARYING phrase

    Furthermore, only the above statements can change the values of *index-names*.
2. Indexing can be either direct or relative. Direct indexing means that the value of *index-name* or *literal-1* is the occurrence number. Relative indexing means that the occurrence number is the value of *index-name* plus or minus *literal-2*. *Literal-2* must be an unsigned integer.

Note

By default, COBOL-81 performs index range checking at run time. For more information, refer to the discussion of the /CHECK qualifier to the COBOL command in Part I of the COBOL-81 User's Guide for your system.

The following example is very similar to the one that illustrates subscripting. However, this example shows: (1) use of index-names in references to the table, (2) initializing indexes with the SET statement, and (3) storing index-name values in index data items.

Example

```
WORKING-STORAGE SECTION.
01  ITEMA USAGE IS INDEX.
01  ITEMB USAGE IS INDEX.
01  ITEMC VALUE IS "ABCDEFGHIJKLMNOPQRSTUVWX".
    03  ITEMD OCCURS 4 TIMES
        INDEXED BY DX.
        05  ITEME OCCURS 6 TIMES
            INDEXED BY EX PIC X.

PROCEDURE DIVISION.
PARA.
        SET DX TO 4.
        SET EX TO 1.
        DISPLAY ITEMD (DX).
        DISPLAY ITEME (DX, EX).
        DISPLAY ITEME (DX - 3, EX)
        SET ITEMA TO DX.
        SET ITEMB TO EX.
```

Results
STUVWX
S
A

5.3.3 Identifiers

In Procedure Division rules, the term *identifier* means a data item. The term refers to all words required to make your reference to the item unique.

The general formats for identifiers are:

Format 1

data-name [qualification] [subscripting]

Format 2

data-name [qualification] [indexing]

The following sections provide more information on the methods you use to uniquely specify data items: Section 5.3.1, Qualification; Section 5.3.2.1, Subscripting; and Section 5.3.2.2, Indexing.

5.3.4 Ensuring Unique Condition-Names

If the name you use as a condition-name appears in more than one place in your program, it can be made unique through qualification, indexing, or subscripting.

The first qualifier for a condition-name can be the name of the item with which it is associated (the conditional variable). When qualifying condition-names, you must use the name of the conditional variable itself or the names of items that contain it.

References to a condition-name must have the same combination of subscripting or indexing that you use for the conditional variable.

The formats you use to ensure unique condition-names are the same as those for identifier, except that *condition-name* replaces *data-name*.

In Procedure Division rules the term *condition-name* refers to a condition-name along with any qualification and subscripting or indexing needed to avoid ambiguity.

5.4 Arithmetic Expressions

Whenever the term *arithmetic expression* appears in Procedure Division rules, it refers to one of the following:

- An identifier of a numeric elementary item
- A numeric literal
- Two or more of the above choices separated by arithmetic operators
- Two or more arithmetic expressions separated by an arithmetic operator
- An arithmetic expression enclosed in parentheses

A unary operator (a sign) can precede any arithmetic expression.

The identifiers and literals in an arithmetic expression must represent either: (1) numeric elementary items, or (2) numeric literals on which arithmetic can be performed.

5.4.1 Arithmetic Operators

Arithmetic expressions can use five binary and two unary arithmetic operators. A space must precede and follow each binary operator. A space must precede, but not follow a unary operator. The operators are:

Binary Arithmetic Operator	Meaning
+	Addition
–	Subtraction
*	Multiplication
/	Division
**	Exponentiation

Unary Arithmetic Operator	Meaning
+	The effect of multiplication by +1
–	The effect of multiplication by –1

5.4.2 Formation and Evaluation of Arithmetic Expressions

When you use them, parentheses determine the order in which elements in an arithmetic expression are evaluated. Expressions within parentheses are evaluated first. If you nest sets of parentheses, evaluation starts with the innermost set of parentheses and proceeds to the outermost set.

If the arithmetic expression contains no parentheses, your program evaluates arithmetic operators in the following hierarchical order:

First	Unary plus and minus
Second	Exponentiation
Third	Multiplication and division
Fourth	Addition and subtraction

This order also applies when all sets of parentheses to be evaluated are at the same level.

If two or more operators are at the same hierarchical level, and parentheses do not specify the sequence of operations, evaluation proceeds from left to right.

Parentheses can: (1) eliminate ambiguities in logic when there are consecutive operations at the same hierarchical level, or (2) change the normal hierarchical sequence of execution.

Consider the following expression:

```
(3 * ITEMA - 2) / ((4 + ITEMB) * -ITEMA - ITEMC ** 2)
```

The order of execution is:

1. `4 + ITEMB`
2. `-ITEMA`
3. `3 * ITEMA`
4. (The results of Step 3) - `2`
5. `ITEMC ** 2`
6. (The results of Step 1) `*` (the results of Step 2)
7. (The results of Step 6) - (the results of Step 5)
8. (The results of Step 4) `/` (the results of Step 7)

Table 5-3 shows the valid combinations of operators, variables, and parentheses in arithmetic expressions. In the table, *Yes* indicates a valid pair of symbols; *No* means an invalid pair; and *variable* means an identifier or literal.

Table 5-3: Combinations of Symbols in Arithmetic Expressions

First Symbol \ Second Symbol	Variable	* / ** + –	Unary + or –	(	)
Variable	No	Yes	No	No	Yes
* / ** + –	Yes	No	Yes	Yes	No
Unary + or –	Yes	No	No	Yes	No
(	Yes	No	Yes	Yes	No
)	No	Yes	No	No	Yes

An arithmetic expression can begin only with the symbols (, +, –, an identifier, or a literal. It can end only with the symbol), an identifier, or a literal.

Each left parenthesis in an arithmetic expression must have a matching right parenthesis, and each right parenthesis must have a matching left parenthesis.

If the first operator is unary, a (must precede it when the arithmetic expression immediately follows an identifier or another arithmetic expression.

The following rules apply to the evaluation of exponentiation:

1. An exponent can be a numeric literal, a numeric data item, or an arithmetic expression. However, the value of an exponent cannot be fractional; it must be an integer. (Zero and signed values are valid).
2. Zero raised to a negative or zero power is an undefined value. Therefore, if the value of an expression to be raised to a power is zero, the exponent value must be a positive number. Otherwise, the size error condition exists. (See Section 5.6.4, SIZE ERROR Condition.)

If the evaluation of the arithmetic expression results in an attempted division by zero, the size error condition exists.

When a statement with an arithmetic expression does not refer to a resultant identifier, the compiler stores the results of the arithmetic expression in an intermediate data item. (See Section 5.6.1, Arithmetic Operations.)

5.5 Conditional Expressions

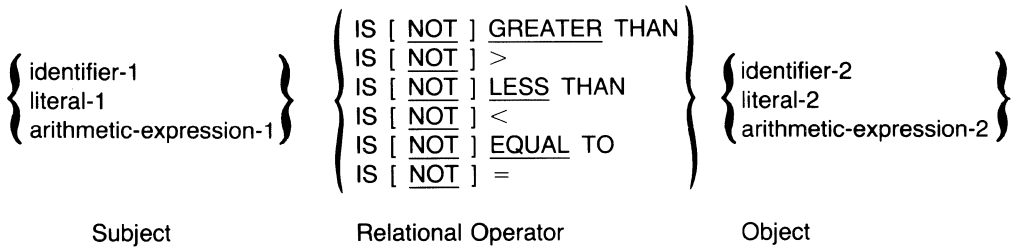
A conditional expression specifies a condition the program must evaluate to determine the path of program flow. If the condition is true, the program takes one path; if it is false, the program takes another path. The IF, PERFORM UNTIL, PERFORM VARYING, and SEARCH statements must contain conditional expressions.

A conditional expression can be either a simple or a complex condition. The types of simple conditions are the relation, class, condition-name, switch-status, and sign conditions. Complex conditions are formed by using logical operators (AND, OR, NOT) with simple conditions. You can enclose conditions within any number of paired parentheses. However, embedding conditions this way has no effect on whether they are considered simple or complex.

5.5.1 Relation Conditions

A relation condition states a relation between two operands. The program compares the operands to determine whether the stated relation is true or false. The first operand is called the condition's subject. The second operand is called its object. Either operand can be: (1) an identifier, (2) a literal, or (3) the value of an arithmetic expression. The set of words that specifies the type of comparison is called the relational operator.

The format for a relation condition is:



You can compare two numeric operands regardless of their USAGE. However, if one or both of the operands are not numeric, they must have the same USAGE. If either operand is a group item, then the comparison is treated as nonnumeric, since group items are always considered alphanumeric.

You must refer to at least one variable in a relation condition; you cannot refer only to literals.

A space must precede and follow each word in the relational operator. However, NOT and the key word or relation character that follows NOT are treated as a unit. For example, NOT EQUAL specifies an "unequal" relation condition rather than a complex condition.

Table 5-4 specifies valid "true" condition(s) that correspond to each relational operator.

Table 5-4: Relational Operators and Corresponding True Conditions

Relational Operator	"True" Condition
IS GREATER THAN IS > THAN	Subject is greater than object
IS NOT GREATER THAN IS NOT > THAN	Subject is either less than or equal to object
IS LESS THAN IS < THAN	Subject is less than object
IS NOT LESS THAN IS NOT < THAN	Subject is either greater than or equal to object
IS EQUAL TO IS = TO	Subject is equal to object
IS NOT EQUAL TO IS = TO	Subject is either greater than or less than object

The following two sections specify the rules that apply to numeric and nonnumeric comparisons in relation conditions.

5.5.1.1 Comparison of Numeric Operands — When both operands are numeric, their algebraic values are compared. The program performs the necessary conversion if the data descriptions of the operands specify different USAGE. When you use operands that are literals or arithmetic expressions, their length (in terms of the number of digits represented) is not significant.

Unsigned numeric operands are assumed to be positive for comparison. A zero value is always treated the same way, whether or not the operand contains a sign.

5.5.1.2 Comparison of Nonnumeric Operands — When one (or both) of the operands is nonnumeric, each operand is considered a string of alphanumeric characters. Therefore, the operands are compared according to the program's collating sequence. (See Section 3.1.2, OBJECT-COMPUTER Paragraph.)

If one of the operands is numeric, it must be either an integer or a data item described as an integer. The data item must be implicitly or explicitly described with USAGE DISPLAY. When a numeric operand contains a sign, its sign is part of the string only if the other operand is a group item. Otherwise, the sign is removed and is not part of the comparison.

The two operands are compared character by character, beginning at the left end of each string. When the operation finds an unequal character pair, it uses that pair to evaluate the comparison. The greater operand is the one that contains the character with the higher collating sequence position. If the operands are of unequal size, the shorter operand is treated as if it were extended on the right with spaces to make it the same size as the other. Therefore, "ABCD" is greater than "ABC".

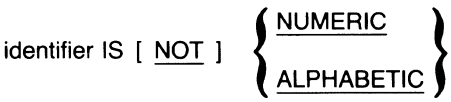
5.5.1.3 Comparisons of Index-Names or Index Data Items — A program can compare:

- Two index-names
- One index-name and one literal or data item (other than an index data item)
- One index-name and one index data item
- Two index data items

5.5.2 Class Condition

The class condition tests whether the contents of an operand are numeric or alphabetic.

The general format is:



The USAGE of the operand being tested must be DISPLAY. The following rules apply to the NUMERIC test:

1. The test is true when the operand contains only the characters 0 to 9 and the operational sign (subject to the next rule); otherwise, it is false.
2. The operand must contain an operational sign if its PICTURE clause specifies a sign. If the PICTURE clause does not specify a sign, the operand must not contain one. If the operand contains a sign that is not specified, or if a sign is specified and the operand does not contain one, the NUMERIC test is false.
3. You cannot use the test for an operand described as alphabetic or a group item containing signed elementary items.

The following rules apply to the ALPHABETIC test:

1. The test is true when the operand contains only the characters A through Z, a through z, and the space; otherwise, it is false.
2. You cannot use the ALPHABETIC test for an operand described as numeric.

NOT and the key word following it are treated as a unit. For example, NOT NUMERIC is a test for determining that the operand is nonnumeric. Using NOT in a class condition does not make the condition complex.

5.5.3 Condition-Name Condition

The condition-name condition determines if a data item contains a value assigned to one of that item's condition-names. The term *conditional variable* refers to the data item. *Condition-name* refers to a level 88 entry associated with that item.

The general format for this condition is:

condition-name

The condition is true if one of the values corresponding to *condition-name* equals the value of the associated conditional variable. The data description for a variable can associate *condition-name* with one or more ranges of values. In this case, the condition tests to determine if the value of the variable falls in the specified range (end values included).

The following example illustrates testing *condition-names* associated with both one value and a range of values.

```
WORKING-STORAGE SECTION.
01  STUDENT-REC.
    05  YEAR-ID            PIC 99.
    88  FRESHMAN                    VALUE IS 1.
    88  SOPHOMORE                   VALUE IS 2.
    88  JUNIOR                      VALUE IS 3.
    88  SENIOR                      VALUE IS 4.
    88  GRADUATE                    VALUE IS 5 THRU 10.
                          .
                          .
                          .
PROCEDURE DIVISION.
                          .
                          .
                          .
    IF FRESHMAN ...
    IF SOPHOMORE ...
    IF JUNIOR ...
    IF SENIOR ...
    IF GRADUATE ...
```

Condition-Name	Test is "true" when value of YEAR-ID equals:
FRESHMAN	1
SOPHOMORE	2
JUNIOR	3
SENIOR	4
GRADUATE	5, 6, 7, 8, 9, or 10

When your program evaluates a conditional variable and its *condition-name*, the procedure is the same as the one used with the relation condition. (See Section 5.5.1.1, Comparison of Numeric Operands, and Section 5.5.1.2, Comparison of Nonnumeric Operands.)

5.5.4 Switch-Status Condition

The switch-status condition tests the "on" or "off" setting of an external program switch. Its general format is:

```
condition-name
```

You use the SWITCH clause of the SPECIAL-NAMES paragraph to associate *condition-name* with a switch setting. (See Section 3.1.3, SPECIAL-NAMES Paragraph.) The condition is true if the switch setting in effect during program execution is the same one assigned to *condition-name*.

Technical Note

At run time, COBOL-81 prompts the operator to enter the numbers of switches to be set "ON" for that program execution.

5.5.5 Sign Condition

The sign condition determines if the algebraic value of an arithmetic expression is less than, greater than, or equal to zero.

Its general format is:

```
arithmetic-expression IS [ NOT ] { POSITIVE }
                                 { NEGATIVE }
                                 { ZERO     }
```

An operand is:

- POSITIVE, if its value is greater than zero
- NEGATIVE, if its value is less than zero
- ZERO, if its value equals zero

Arithmetic-expression must contain at least one reference to a variable.

NOT and the key word following it are treated as a unit. For example, NOT ZERO tests for a non-zero condition; the word NOT does not indicate a complex condition.

5.5.6 Complex Conditions

You form complex conditions by combining or negating other conditions. The conditions being combined or negated can be either simple or complex.

The logical operators AND and OR combine conditions. The logical operator NOT negates conditions. A space must precede and follow each logical operator in your program.

The truth value of a complex condition depends upon (1) the truth value of each condition it contains and (2) the effect of the logical operator(s). Table 5-5 shows the effect of each logical operator in complex conditions.

Table 5-5: How Logical Operators Affect Evaluation of Conditions

Logical Operator	Effect
AND	The complex condition is true if both connected conditions are true. It is false if one or both connected conditions are false.
OR	The complex condition is true if one or both connected conditions are true. It is false if both conditions are false.
NOT	The complex condition is true if the original condition is false. It is false if the original condition is true.

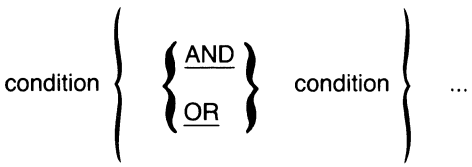
5.5.6.1 Negated Simple Conditions — The logical operator NOT negates a simple condition. The truth value of a negated simple condition is the opposite of the simple condition's truth value. Thus, the truth value of a negated simple condition is true only if the simple condition's truth value is false. It is false only if the simple condition's truth value is true.

The format for a negated simple condition is:

NOT simple-condition

5.5.6.2 Combined and Negated Combined Conditions — A combined condition results from connecting conditions with one of the logical operators AND or OR.

The general format is:



In the general format, *condition* can be:

- A simple condition
- A negated simple condition
- A combined condition
- A negated combined condition; that is, NOT followed by a combined condition enclosed in parentheses
- Valid combinations of the preceding conditions (see Table 5-6)

You can use matched pairs of parentheses in a combined condition. You do not need to write parentheses if the condition combines two or more conditions with the same logical operator (either AND or OR). In this case, the parentheses have no effect on the condition's evaluation. However, you might have to use parentheses if you use a mixture of AND, OR, and NOT logical operators. In this case, the parentheses can affect the condition's evaluation.

Table 5-6 shows the permissible combinations of conditions, logical operators, and parentheses.

Table 5-6: Combinations of Conditions, Logical Operators, and Parentheses

In a conditional expression			In a left-to-right element sequence	
Element	Can element be first?	Can element be last?	Element, when not first, can immediately follow:	Element, when not last, can immediately precede:
simple-condition	Yes	Yes	OR, NOT, AND, (	OR, AND,)
OR or AND	No	No	simple-condition,)	simple-condition, NOT, (
NOT	Yes	No	OR, AND, (	simple-condition, (
(	Yes	No	OR, NOT, AND, (	simple-condition, NOT, (
)	No	Yes	simple-condition,)	OR, AND,)

For example, Table 5-6 shows whether or not the following element pairs can occur in your program:

Element Pair	Permitted?
OR NOT	Yes
NOT OR	No
NOT (	Yes
NOT NOT	No

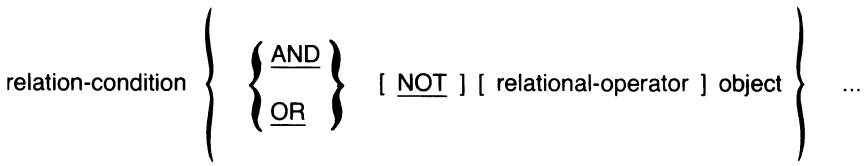
5.5.7 Abbreviated Combined Relation Conditions

When you combine simple or negated simple conditions in a consecutive sequence, you can abbreviate any of the relation conditions except the first. You do this by either:

- Omitting the subject of the relation condition
- Omitting both the subject and the relational operator of the condition

You can omit the subject or relational operator only when it is the same as that in the first relation condition of the sequence.

The general format for combined abbreviated relation conditions is:



The evaluation of a sequence of combined relation conditions proceeds as if: (1) the last preceding subject appears in place of the omitted subject, and (2) the last preceding relational operator appears in place of the omitted relational operator. The result of these substitutions must form a valid condition.

When the word NOT appears in a sequence of abbreviated conditions, its treatment depends upon the word that follows it. NOT is considered part of the relational operator when immediately followed by: GREATER, >, LESS, <, EQUAL, or =. Otherwise, NOT is considered a logical operator that negates the relation condition.

The following examples show abbreviated combined (and negated combined) relation conditions and their expanded equivalents:

Abbreviated Combined Relation Condition	Expanded Equivalent
`a > b AND NOT < c OR d`	`((a > b) AND (a NOT < c)) OR (a NOT < d)`
`a NOT = b OR c`	`(a NOT = b) OR (a NOT = c)`
`NOT a = b OR c`	`(NOT (a = b)) OR (a = c)`
`NOT (a GREATER b OR < c)`	`(NOT ((a GREATER b) OR (a < c)))`
`a / b NOT = c AND NOT d`	`((a / b) NOT = c) AND (NOT ((a / b) NOT = d))`
`NOT (a NOT > b AND c AND NOT d)`	`NOT ((((a NOT > b) AND (a NOT > c)) AND (NOT (a NOT > d))))`

5.5.8 Condition Evaluation Rules

Parentheses can specify the evaluation order in complex conditions. Conditions in parentheses are evaluated first. In nested parentheses, evaluation starts with the innermost set of parentheses. It proceeds to the outermost set.

Conditions are evaluated in a hierarchical order when there are no parentheses in a complex condition. This same order applies when all sets of parentheses are at the same level (none are nested). The hierarchy is shown in the following list:

1. Values for arithmetic expressions
2. Truth values for simple conditions, in this order:
   - Relation
   - Class
   - Condition-name
   - Switch-status
   - Sign
3. Truth values for negated simple conditions
4. Truth values for combined conditions, in this order:

   AND logical operators
   OR logical operators
5. Truth values for negated combined conditions

In the absence of parentheses, the order of evaluation of consecutive operations at the same hierarchical level is from left to right.

5.6 Common Rules and Options for Data Handling

This section describes the rules and options that apply when statements handle data.

5.6.1 Arithmetic Operations

The arithmetic statements begin with the verbs ADD, DIVIDE, COMPUTE, MULTIPLY, and SUBTRACT. When an operand in these statements is a data item, its PICTURE must be numeric and specify no more than 18 digit positions. However, operands do not have to be the same size, nor must they have the same USAGE. Conversion and decimal point alignment occur throughout the calculation.

When you write an arithmetic statement, you specify one or more data items to store the results of the operation. These data items are called *resultant identifiers*. However, the evaluation of each arithmetic statement also uses an intermediate data item. An intermediate data item is a compiler-supplied data item that the program cannot access. It stores the results of intermediate steps in the arithmetic operation before the final value is moved to the resultant identifier(s).

An intermediate data item is 18 digits long. It contains the 18 most significant digits of the intermediate result. During execution of the operation, the magnitude of this result is maintained. All truncated low-order digits are treated as zeros for the rest of the operation.

When the final value of an arithmetic operation is moved to the resultant identifier(s), it is transferred according to MOVE statement rules. Rounding and size error condition checking occur just before this final move. (See Section 5.6.3, ROUNDED Option; Section 5.6.4, ON SIZE ERROR Option; and Section 5.9.15, MOVE Statement.)

5.6.2 Multiple Receiving Fields in Arithmetic Statements

An arithmetic statement can move its final result to more than one data item. In this case, the statement is said to have *multiple receiving fields* (or *multiple results*). The statement operates as if it had been written as a series of statements. The following example illustrates these steps. The first statement in the example is equivalent to the four that follow it. (*Temp* is an intermediate data item.)

```
ADD a, b, c TO c, d (c), e

ADD a, b, c GIVING temp
ADD temp TO c
ADD temp TO d (c)
ADD temp TO e
```

5.6.3 The ROUNDED Option

The ROUNDED option allows you to specify rounding at the end of an arithmetic operation. The rounding operation adds 1 to the absolute value of the low-order digit of the resultant identifier if the absolute value of the next least significant (lower-valued) digit of the intermediate data item is greater than or equal to 5.

When the PICTURE string of the resultant identifier represents the low-order digit positions with the P character, rounding or truncation is relative to the rightmost integer position for which the compiler allocates storage. Therefore, when PIC 999PPP describes the item, the value 346711 is rounded to 347000.

If you do not use the ROUNDED phrase, any excess digits in the arithmetic result are truncated when the result is moved to the resultant identifier(s).

5.6.4 The ON SIZE ERROR Option

The ON SIZE ERROR phrase allows you to specify an action for your program to take when a size error condition exists.

A size error condition exists when the absolute value of an operation's result exceeds the largest value the resultant identifier(s) can contain. The size error condition affects the contents of only those data items storing the results of the operation that caused the size error; it does not affect the contents of all operands in the statement.

Size error checking occurs after decimal point alignment. If the statement has a ROUNDED phrase, rounding occurs before size error checking. If not, truncation of rightmost digits occurs before size error checking.

Division by zero or invalid exponentiation at any step in the arithmetic operation also causes the size error condition.

When a size error condition occurs and the statement does not contain a SIZE ERROR phrase, the value in all resultant identifiers is undefined. Also, when the size error condition results from division by zero or invalid exponentiation, the program terminates abnormally. However, when the statement does contain a SIZE ERROR phrase: (1) the values of all resultant identifiers are the same as before the operation began, and (2) the imperative statement in the SIZE ERROR phrase executes.

For the ADD and SUBTRACT statements with the CORRESPONDING phrase, any individual operation can cause a size error condition. However, the SIZE ERROR phrase imperative statement executes only after all individual additions or subtractions end.

5.6.5 CORRESPONDING Option

The CORRESPONDING option allows you to specify group items as operands in order to use their corresponding subordinate items in an operation. See also Section 5.9.2, ADD Statement; Section 5.9.29, SUBTRACT Statement; and Section 5.9.15, MOVE Statement.

The following rules apply to the identifiers of operands in a statement containing the CORRESPONDING phrase.

1. All identifiers must refer to group items.
2. The data description entries of these identifiers can contain a REDEFINES or OCCURS clause.
3. Identifiers can be subordinate to a data description entry that has a REDEFINES or OCCURS clause.
4. You cannot specify identifiers with level-number 66 or the USAGE IS INDEX clause.

The following rules describe the requirements for correspondence between data items subordinate to the identifiers. In these rules, *identifier-1* refers to the sending group item and *identifier-2* refers to the group(s) in which results of the operation are stored.

1. Data items subordinate to both identifer-1 and identifier-2 must have the same dataname.
2. All possible qualifiers for a data item contained in *identifier-1* (up to but not including identifier-1), must be identical to all possible qualifiers for the matching item in *identifer-2* (up to but not including identifier-2).
3. In an ADD or SUBTRACT statement, the CORRESPONDING phrase affects only elementary numeric data items. Other data items do not take part in the operation.
4. In a MOVE statement, either the sending or receiving subordinate item can be a group item, but both cannot be. The classes of the data items in any corresponding pair can be different.
5. The CORRESPONDING phrase ignores data items with:
   - Level-number 66
   - Level-number 88
   - A data description entry containing a REDEFINES, OCCURS, or USAGE IS INDEX clause

   A data item subordinate to one that is not eligible for correspondence is also ignored.
6. FILLER data items and their subordinates are ignored.

5.6.6 Overlapping Operands and Incompatible Data

When statements refer to data items, two conditions can occur that can make program results unpredictable.

Undefined results occur when a sending and receiving item in an arithmetic statement or an INSPECT, MOVE, SET, STRING, or UNSTRING statement share a part of their storage areas.

Procedure Division references to a data item are undefined when a data item's contents are incompatible with the class of data defined by the item's PICTURE clause. Conditional statements containing the class condition allow you to (1) determine whether or not an item contains numeric or alphabetic data and (2) specify corrective action when it does not. (See Section 5.5.2, Class Condition.)

5.7 I-O Status

If a file description entry has a FILE STATUS clause, a value is placed in the two-character FILE STATUS data item during execution of a CLOSE, DELETE, OPEN, READ, REWRITE, START, or WRITE statement. Two "keys" combine to form this value. Status Key 1 occupies the leftmost character position in the item and Status Key 2 occupies the rightmost position. In combination, the keys indicate the status of the input-output operation.

Appendix C lists all the possible values that can appear in the FILE STATUS data item, along with the I-O status condition corresponding to each value.

Any applicable USE AFTER EXCEPTION procedure executes after the FILE STATUS value is set.

Table 5-7 shows the possible combinations of Status Keys 1 and 2. In the table, "X" indicates a valid combination of keys.

Table 5-7: Possible Combinations of Status Keys 1 and 2

Status Key 1 ↓ / 2 →	No Further Information (0)	Sequence Error (1)	Duplicate Key (2)	No Record Found (3)	Boundary Violation (4)	File Not Present (5)	No Valid Next Record (6)	File Not Found (7)	Close Error (8)
Successful Completion (0)	X		X			X			
At End (1)				X		X	X		
Invalid Key (2)		X	X	X	X				
Permanent Error (3)	X				X				
DIGITAL-Defined (9)	X	X	X	X	X	X	X	X	X

Status Key 1

Status Key 1 indicates one of the following conditions when an input-output operation ends:

0 Successful Completion. The input-output statement executed successfully.

1 At End. A sequential READ statement unsuccessfully executed because:

- The file has no next logical record.
- An optional file was not present.
- The program did not establish a valid next record.

2 Invalid Key. The input-output statement executed unsuccessfully because of one of the following conditions:

- Sequence Error
- Duplicate Key
- No Record Found
- Boundary Violation
- Optional File Not Present

3 Permanent Error. The input-output statement executed unsuccessfully because of a boundary error for a sequential file. This value can also indicate an input-output error, such as data check, parity error, or transmission error.

9 DIGITAL-defined. The input-output statement executed unsuccessfully because of a condition defined by DIGITAL.

Status Key 2

Status Key 2 further describes the result of the input-output operation:

- If no further information about the input-output operation is available, Status Key 2 contains 0 .
- When Status Key 1 contains 0 (indicating successful completion), Status Key 2 can contain a 2 or a 5 :

  2 applies to a REWRITE or WRITE statement. It means that the record just written created a duplicate key value for at least one alternate record key for which duplicates are allowed.

  5 applies to OPEN statement. It means that the optional file was not present when the OPEN statement executed.

- When Status Key 1 contains 1 (indicating an at end condition), Status Key 2 describes the condition's cause:

  3 indicates that the file has no next logical record.

  5 indicates that a file you specified as optional is not present.

  6 indicates that the program did not establish a valid next record.

  The values 13 and 16 can occur for the same READ operation when a program is in an infinite loop. In this case, the FILE STATUS data item contains the following sequence of values:

  00, 00, ... , 00, 13, 16, 16, ... , 16

- When Status Key 1 contains 2 (indicating an invalid key condition), Status Key 2 describes the condition's cause:

  1 indicates a sequence error for a sequential access indexed file. This means that the program changed the prime record key value between a successfully executed READ statement and the next REWRITE statement for the file. This value can also indicate that the program violated ascending sequence requirements for successive record key values. (See Section 5.9.32, WRITE Statement.)

  2 indicates a duplicate key value. The program tried to write or rewrite a record that would have created a duplicate key in an indexed file. This value can also mean that the program tried to write a record that would have created a duplicate in a relative file.

  3 means that the program could not find a record. The program tried to access a record identified by a key, but the record does not exist in the file.

  4 indicates a boundary violation. The program tried to write beyond the boundaries defined for the file by Record Management Services (RMS-11).

- When Status Key 1 contains 3 (indicating a permanent error condition), Status Key 2 can contain a 4 to indicate a boundary violation. This means that the program tried to write beyond the boundaries defined for the file by Record Management Services (RMS-11)

  Status Key 2 can also contain a 0 . This value results from any input-output error that cannot be described by any other combination of values in Status Keys 1 and 2.

- When Status Key 1 contains 9 (indicating a DIGITAL-defined condition), Status Key 2 further describes the condition:

  0 means that the record your program is reading is also being read by another program. This condition occurs when two programs share the same record area. Because the record is available in the record area, the input operation is successful.

  1 indicates that a file is locked. The program tried to open a file that had been locked by another program.

  2 means that a record is locked. The program tried to access a record that had been locked by another program.

  In this case, the record is not available in the record area, so the input operation is unsuccessful.

  3 means that the program tried to execute a DELETE or REWRITE statement without first successfully executing a READ statement.

  4 indicates that:

  - The program tried to open a file that is: (a) already open, or (b) closed with lock.
  - The program tried to close a file that: (a) is already closed, or (b) has not been opened during the program's execution.
  - The program tried to perform an input-output operation for a file that: (a) has not been opened, or (b) is open in a mode incompatible with the operation.

  5 means that the program tried to open a file when there was too little file space on the device.

  6 means that the program tried to open a file when another SAME AREA file was open.

  7 indicates that the program tried to open a file that could not be found.

  8 indicates that an unspecified error occurred when the program attempted to close a file.

Part IV of the COBOL-81 User's Guide for your system contains information on using FILE STATUS key values. Refer to the chapter that discusses file I-O exception conditions handling.

5.7.1 The INVALID KEY Phrase

The INVALID KEY phrase specifies the action your program takes when an invalid key condition is detected for the file being processed.

The format is:

```
INVALID KEY stment
```

Stment is an imperative statement.

The invalid key condition occurs when PDP-11 Record Management Services (RMS-11) cannot complete a COBOL DELETE, READ, REWRITE, START, or WRITE statement. When the condition occurs, execution of the statement that produced it is unsuccessful, and the file is not

affected. (See Section 5.9.6, DELETE Statement; Section 5.9.19, READ Statement; Section 5.9.22, REWRITE Statement; Section 5.9.26, START Statement; and Section 5.9.32, WRITE Statement.)

When the invalid key condition is recognized, these actions occur in the following order:

1. A value that indicates the invalid key condition is placed in the FILE STATUS data item for the file.
2. If the statement causing the condition has the INVALID KEY phrase, control transfers to *stment*. Any USE AFTER EXCEPTION procedure for the file does not execute.
3. If there is no INVALID KEY phrase, control transfers to the applicable USE AFTER EXCEPTION procedure for the file.

Part IV of the COBOL-81 User's Guide for your system also contains information on using the INVALID KEY phrase. Refer to the chapter that discusses file I-O exception conditions handling.

5.7.2 The AT END Phrase

The AT END phrase specifies the action your program takes when it detects the end of an input file.

Its format is:

```
AT END stment
```

Stment is an imperative statement.

When a program detects the end of a file, the condition is called the *at end condition*. The at end condition may occur as a result of READ, RETURN, or SEARCH statement execution. (See Section 5.9.19, READ Statement; Section 5.9.21, RETURN Statement; and Section 5.9.23, SEARCH Statement.)

The following rules apply to the the AT END phrase:

1. If the at end condition occurs and the AT END phrase is present, *stment* executes.
2. If the at end condition occurs during the execution of a READ statement and there is no AT END phrase, but there is an applicable USE AFTER EXCEPTION procedure, the USE AFTER EXCEPTION procedure executes.
3. A USE procedure statement has no effect when the at end condition occurs during execution of RETURN and SEARCH statements.

5.7.3 The FROM Option

The FROM phrase implicitly moves a record from one storage area to another prior to execution of an input-output or record-ordering statement.

The format is:

```
record-name FROM identifier
```

Record-name and *identifier* must not refer to the same storage area.

The result of executing a RELEASE, REWRITE,or WRITE statement with the FROM phrase is equivalent to: (1) executing the statement "MOVE *identifier* TO *record-name*" according to the rules of the MOVE statement without the CORRESPONDING phrase, followed by (2) executing the same RELEASE, REWRITE, or WRITE statement without the FROM phrase.

After statement execution ends, the data in the area referenced by *identifier* is available to the program. The data is not available in the area referenced by *record-name*, unless there is an applicable SAME clause. (See Section 3.2.2, I-O-CONTROL Paragraph; Section 5.9.20, RELEASE Statement; Section 5.9.22, REWRITE Statement; and Section 5.9.32, WRITE Statement.)

5.7.4 The INTO Option

The INTO phrase implicitly moves a current record from the record storage area into an identifier.

The format is:

```
file-name INTO identifier
```

A READ or RETURN statement can have the INTO phrase if either of the following conditions is true:

1. Only one record description is subordinate to the file description entry
2. All *record-names* associated with *file-name* and the data item associated with *identifier* describe a group item or an elementary alphanumeric item

Executing a READ or RETURN statement with the INTO phrase is equivalent to: (1) executing the same statement without the INTO phrase then (2) moving the current record from the record area to the area specified by identifier. The move occurs according to the rules of the MOVE statement without the CORRESPONDING phrase. The move does not occur for an unsuccessful execution of the READ or RETURN statement.

Subscript or index evaluation occurs after the input operation and immediately before the move.

The record is available to the program in both the record area and the area associated with identifier.

5.8 Segmentation

The COBOL-81 segmentation facility allows you to communicate object program overlay requirements to the compiler. COBOL segmentation deals only with the segmentation of procedures. Therefore, only the Procedure Division is considered in determining segmentation requirements.

5.8.1 Organization

When segmentation is used, the Procedure Division must be written as a consecutive group of sections. Each section is composed of a series of closely related operations designed to collectively perform a particular function.

Each section must be specified as belonging to the nonoverlayable or overlayable portion of the program.

Using segmentation affects only the physical management of the object program during execution. It neither imposes any syntactic restrictions nor implies any semantic differences over the same program written without segmentation. The logical sequence of the program is the same as the physical sequence except for specific transfers of control. Transfer of control from a nonoverlayable segment to an overlayable segment, or from an overlayable segment to another overlayable segment, is accomplished by the system.

5.8.2 Using the Segmentation Facility

The COBOL-81 segmentation facility requires that you specify the SEGMENT-LIMIT clause in the OBJECT-COMPUTER paragraph (see Section 3.1.2) of the Environment Division, and that you assign segment numbers to each section of the Procedure Division.

The value specified by a segment number is used by the compiler to determine whether a segment is overlayable. That is, the value you specify in the SEGMENT-LIMIT clause is compared to the segment number you assign to each section in the Procedure Division. Sections having segment numbers less than the segment limit are not overlaid. Those having segment numbers greater than or equal to the segment limit are overlayable.

Part II of the COBOL-81 User's Guide for your system also discusses segmentation. Refer to the chapter on reducing task size.

Procedure Division
Format Entry Pages

5.9 Procedure Division General Format and Rules

Function

The Procedure Division contains the routines that process the files and data described in the Environment and Data Divisions.

General Format

Format 1

```
[ PROCEDURE DIVISION [ USING { data-name } ... ] .

[ DECLARATIVES.

{ section-name SECTION [ segment-number ] . declarative-sentence

[ paragraph-name. [ sentence ] ... ] ... } ...

  END DECLARATIVES. ]

{ section-name SECTION [ segment-number ] .

[ paragraph-name. [ sentence ] ...] ... } ... ]
```

Format 2

```
[ PROCEDURE DIVISION [ USING { data-name } ... ] .

[ paragraph-name. [ sentence ] ... ] ... ]
```

Syntax Rules

1. The Procedure Division follows the Data Division.
2. The Procedure Division must begin with the Procedure Division header.
3. A procedure consists of either:
   - One or more (successive) sections
   - One or more (successive) paragraphs

4. If one paragraph is in a section, all paragraphs must be in sections.
5. A procedure-name refers to a paragraph or section in the source program. It is either *paragraph-name* (which can be qualified) or *section-name*.
6. A section consists of a section header followed by zero or more successive paragraphs. A section ends immediately before the next section or at the end of the Procedure Division. In the declaratives part of the Procedure Division, a section can also end at the key words END DECLARATIVES.
7. A paragraph consists of *paragraph-name* followed by a separator period, and by zero or more successive sentences. A paragraph ends immediately before the next *paragraph-name* or *section-name* or at the end of the Procedure Division. In the declaratives part of the Procedure Division, a paragraph can also end at the key words END DECLARATIVES.
8. *Sentence* contains one or more statements terminated by a separator period.
9. A statement is a syntactically valid combination of words and symbols that begins with a COBOL verb.

Procedure Division Header

10. The Procedure Division header identifies and begins the Procedure Division. It consists of the reserved words PROCEDURE DIVISION and optional USING phrase followed by a separator period.
11. The USING phrase is required only if the program is invoked by a CALL statement with a USING phrase.
12. The Procedure Division header USING phrase identifies the names used in the program to refer to arguments from the calling program. In the calling program, the USING phrase of the CALL statement identifies the arguments. The data items in the two USING phrase lists correspond positionally.
13. Each *data-name* in the USING phrase must be defined in the Linkage Section with a level-01 or level-77 entry.
14. Each *data-name* cannot appear more than once in the USING phrase.

Procedure Division Body

15. The Procedure Division body consists of all Procedure Division text following the Procedure Division header.
16. *Segment-number* must be an integer from 0 through 49. If it is omitted, 00 is assumed.

General Rules

1. References to USING phrase *data-names* operate according to data descriptions in the called program's Linkage Section, regardless of the descriptions in the calling program.

2. The called program can refer, in its Procedure Division, to a Linkage Section data item only if the data item satisfies one of these conditions:
   - It is in the Procedure Division header USING phrase.
   - It is subordinate to a *data-name* that is in the Procedure Division header USING phrase.
   - Its definition includes a REDEFINES or RENAMES clause, the object of which is in the Procedure Division header USING phrase.
   - It is subordinate to an item that satisfies the previous condition.
   - It is a condition-name or index-name associated with a data item that satisfies any of the previous conditions.
3. All sections having the same *segment-number*, when *segment-number* is greater than or equal to the value specified in the SEGMENT-LIMIT clause, constitute a single program overlay. Sections with the same *segment-numbers* need not be physically contiguous in the source program.
4. Segments with *segment-numbers* less than the value specified in the SEGMENT-LIMIT clause of the OBJECT-COMPUTER paragraph belong to the nonoverlayable portion of the program.

   However, when there is more than one program in the executable image, the nonoverlayable portions of each program containing the SEGMENT-LIMIT clause share the same memory area.
5. Segments with *segment-numbers* equal to or greater than the value specified in the SEGMENT-LIMIT clause belong to the overlayable portion of the program.

Additional References

Section 5.3.3	CALL Statement
Section 5.9.31	USE Statement
Section 5.8	Segmentation

Examples

1. The Procedure Division header without the USING phrase:

   (This header can appear in: (a) a calling (main) program or (b) a called program that receives no arguments.)

```
PROCEDURE DIVISION.
```

2. Procedure Division header of a called program:

```
LINKAGE SECTION.
01      ARG1.
        03  ARG2        PIC X(6).
        03  ARG3        PIC S9(6) COMP.
01      ARG4            PIC X(4).
PROCEDURE DIVISION USING ARG1 ARG4.
```

ACCEPT

5.9.1 ACCEPT Statement

Function

The ACCEPT statement makes low-volume data available to the program. The DIGITAL extensions to the ACCEPT statement (formats 3 and 4) are COBOL language additions that facilitate video forms design and data handling.

General Format

Format 1

```
ACCEPT dest-item [ FROM input-source ]
```

Format 2

```
ACCEPT dest-item FROM { DATE }
                      { DAY  }
                      { TIME }
```

Format 3

```
ACCEPT dest-item
{ | FROM LINE NUMBER { line-num                                  }
                     { line-id [ PLUS [ plus-num ] ]             }
                     { PLUS [ plus-num ]                         }
  | FROM COLUMN NUMBER { column-num                              }
                       { column-id [ PLUS [ plus-num ] ]         }
                       { PLUS [ plus-num ]                       }
  | ERASE [ TO END OF ] { SCREEN }
                        { LINE   }
  | WITH BELL
  | UNDERLINED
  | BOLD
  | WITH BLINKING
  | PROTECTED [ SIZE protect-length ]
  | WITH CONVERSION
  | REVERSED
  | WITH NO ECHO
  | DEFAULT IS { def-src-lit  }
               { def-src-item }
  | CONTROL KEY IN key-dest-item
}
[ ON EXCEPTION stment ]
```

Format 4

```
ACCEPT CONTROL KEY IN key-dest-item

  { | FROM LINE NUMBER     { line-num                              } | }
  { |                      { line-id    [ PLUS [ plus-num ] ]      } | }
  { |                      { PLUS [ plus-num ]                     } | }
  { |                                                                | }
  { | FROM COLUMN NUMBER   { column-num                            } | }
  { |                      { column-id  [ PLUS [ plus-num ] ]      } | }
  { |                      { PLUS [ plus-num ]                     } | }
  { |                                                                | }
  { | ERASE [ TO END OF ]  { SCREEN }                                | }
  { |                      { LINE   }                                | }
  { |                                                                | }
  { | WITH BELL                                                      | }

      [ ON EXCEPTION stment ]
```

dest-item
: is the identifier of a data item into which data is accepted.

input-source
: is a mnemonic-name defined in the SPECIAL-NAMES paragraph of the Environment Division.

key-dest-item
: is the identifier of a data item that defines a control key. *Key-dest-item* must specify an alphanumeric data item at least four characters in length.

line-num
: is a numeric literal that specifies a line position on the terminal screen. *Line-num* must be a positive integer. It cannot be zero or a negative integer.

line-id
: is the identifier of a data item that provides a line position on the terminal screen.

column-num
: is a numeric literal that specifies a column position on the terminal screen. *Column-num* must be a positive integer. It cannot be zero or a negative integer.

column-id
: is the identifier of a data item that provides a column position on the terminal screen.

plus-num
: is a numeric literal that increments the current value for line or column position, or that increments the value of *line-id* or *column-id*. *Plus-num* can be zero or a positive integer. It cannot be a negative integer.

protect-length
: is a numeric literal that specifies the maximum length of the video screen field into which data can be typed. *Protect-length* must be a positive integer. It cannot be zero or a negative integer.

def-src-lit
: is a nonnumeric literal or a figurative constant. However, it cannot be the figurative constant ALL literal.

def-src-item
: is the identifier of an alphanumeric data item.

stment
: is an imperative statement executed for an on exception condition.

Syntax Rules

Format 3

1. When DIGITAL extensions to the ACCEPT statement appear, *dest-item* can be no more than 132 characters in length.
2. You cannot specify a phrase more than once for any *dest-item*.
3. When you use the PROTECTED phrase without its SIZE option, the size of *def-src-item* and *def-src-lit* must be less than, or equal to the size of *dest-item*.
4. When you use the PROTECTED phrase with its SIZE option, the size of *def-src-item* and *def-src-lit* must be less than, or equal to *protect-length*.

Format 4

5. No phrase can appear more than once for any *key-dest-item*.

General Rules

Format 1

1. The ACCEPT statement transfers data from *input-source*. The transferred data replaces the contents of *dest-item*.
2. The ACCEPT statement transfers a stream of characters with no editing or conversion. Data transfer begins with the leftmost character position of *dest-item* and continues to the right.
3. If the data does not completely fill *dest-item*, remaining character positions are filled with spaces. If the data is too long for *dest-item*, it is truncated on the right.
4. The ACCEPT statement treats *dest-item* as alphanumeric, regardless of its class.
5. If there is no FROM phrase, the ACCEPT statement transfers data from the default system input device.

Format 2

6. The ACCEPT statement transfers data to *dest-item* according to the MOVE statement rules.

7. DATE, DAY, and TIME are not actual data items. Therefore, the source program must not describe them.
8. DATE has three two-digit elements. From left to right, they are:
   - Year of century
   - Month of year
   - Day of month

   The ACCEPT statement operates as if DATE were described in the program as a six-digit, unsigned elementary numeric integer data item (PIC 9(6)).

   The date June 3, 1985 is expressed as 850603.
9. DAY has two elements. From left to right, they are:
   - Year of century (two digits)
   - Day of year (three digits)

   The ACCEPT statement operates as if DAY were described in the program as a five-digit, unsigned elementary numeric integer data item (PIC 9(5)).

   The fifteenth day of 1986 is expressed as 86015.
10. TIME represents the elapsed time since midnight using a 24-hour clock. It has four two-digit elements. From left to right, they are:
    - Hours
    - Minutes
    - Seconds
    - Hundredths of a second

    The ACCEPT statement operates as if TIME were described in the program as an eight-digit, unsigned elementary numeric integer data item (PIC 9(8)).

    The time 6:13 P.M. is expressed as 18130000. The minimum and maximum values of TIME are 00000000 and 23595999.

Formats 3 and 4

11. The ACCEPT statement transfers data from a video terminal. The data replaces the contents of *dest-item* (Format 3), or *key-dest-item* (Format 4).
12. For a Format 3 ACCEPT statement, the maximum number of characters that can be typed in at a terminal is 132.
13. The presence of either the LINE NUMBER phrase or the COLUMN NUMBER phrase implies NO ADVANCING; that is, following data input, a line feed and carriage return is not generated automatically. The cursor remains on the character position immediately following the position of the last input character. This is the default starting position of the next data item you input from or display upon the terminal.

14. If you do not use either the LINE NUMBER phrase or the COLUMN NUMBER phrase, data is accepted according to positioning rules for the Format 1 ACCEPT statement. That is, a line feed and carriage return are automatically generated following data input, and the next item displayed upon, or accepted from, the terminal will appear on the following line.

LINE NUMBER Phrase

15. The LINE NUMBER phrase positions the cursor on a specific line of the video screen for data input.
16. If the LINE NUMBER phrase does not appear, but the COLUMN NUMBER phrase does, then data is accepted from the *current* line position and specified column position.
17. You must use relative line positioning to advance the cursor position below the bottom line position of the screen. If *line-num* or the value of *line-id* is greater than the bottommost line position of the current screen, program results are undefined. (See Technical Notes.)
18. If you use *line-id* without its PLUS option, the line position is the value of *line-id*.
19. If you use *line-id* with its PLUS option, the line position is the sum of *plus-num* and the value of *line-id*.
20. If you use the PLUS option without *line-id*, the line position is the sum of *plus-num* and the value of the current line position.
21. If you use the PLUS option, but you do not specify *plus-num*, then PLUS 1 is implied.
22. Data input results are undefined if your program generates a value for *line-id* that is either zero or negative.

COLUMN NUMBER Phrase

23. The COLUMN NUMBER phrase positions the cursor on a specific column of the video screen.
24. If the COLUMN NUMBER phrase does not appear, but the LINE NUMBER phrase does appear, then data is accepted from column 1 of the specified line position.
25. If you use *column-id* without its PLUS option, the column position is the value of *column-id*.
26. If you use *column-id* with its PLUS option, the column position is the sum of *plus-num* and the value of *column-id*.
27. If you use the PLUS option without *column-id*, the column position is the sum of *plus-num* and the value of the current column position.
28. If you use the PLUS option, but do not specify *plus-num*, PLUS 1 is implied.

29. Data input results are undefined if the program generates a value for column position that is one of the following:
    - Zero
    - Negative
    - Greater than the last column position on the screen

ERASE Phrase

30. The ERASE phrase erases all, or part, of a line (or screen) before accepting data.
31. If you use its TO END option, the ERASE phrase erases the line (or screen) from the implied, or stated, cursor position to the end of the line (or screen).
32. If you do not use its TO END option, the ERASE phrase erases the entire line (or screen).

    However, if the TO END option is not used, and the output device is a VT52 terminal, then cursor position cannot be implied. In this case, you must specify both the LINE NUMBER and the COLUMN NUMBER phrases, and you must use the PLUS option only following *line-id* (or *column-id*).

    If the output device is a VT52 terminal and the TO END option is not used, the ERASE phrase produces undefined results when any of the following conditions are true:
    - The LINE NUMBER phrase is absent.
    - The COLUMN NUMBER phrase is absent.
    - The PLUS option appears, but is not preceded by either *line-id* or *column-id*.

BELL Phrase

33. The BELL phrase rings the terminal bell before accepting data.

ON EXCEPTION Phrase

34. The ON EXCEPTION phrase allows execution of an imperative statement when an exception (or error) condition occurs.
35. If you specify the ON EXCEPTION phrase in a Format 3 ACCEPT statement, typing CTRL/Z causes *stment* to execute.

Format 3

UNDERLINED Phrase

36. The UNDERLINED phrase echoes input characters to the terminal with the "underscore on" character attribute.
37. When you use the UNDERLINED phrase with the PROTECTED phrase, the input field is underlined prior to accepting data.

BOLD Phrase

38. The BOLD phrase echoes input characters to the terminal with the "bold on" character attribute.

BLINKING Phrase

39. The BLINKING phrase echoes input characters to the terminal with the "blink on" character attribute.

REVERSED Phrase

40. The REVERSED phrase echoes input characters to the terminal with the "reverse video on" character attribute.

41. When you use the REVERSED phrase with the PROTECTED phrase, the input field appears in reverse video prior to accepting data.

CONVERSION Phrase

42. The CONVERSION phrase allows you to accept data into a field and achieve the same results as you would with the MOVE statement. It enables validation of the accepted data and facilitates editing and alignment of data within *dest-item*. How the CONVERSION phrase affects data handling depends on the category of *dest-item*. (Numeric data can be described by any USAGE clause.)

43. When *dest-item* is numeric or numeric edited, the CONVERSION phrase:

    - Converts input numeric data to a numeric literal (the sign is placed in the rightmost character position)
    - Moves the result to *dest-item* (using MOVE statement rules)

44. When *dest-item* is numeric or numeric edited, and you use the CONVERSION phrase, valid input characters are as follows:

    - 0 through 9
    - period (.)
    - comma (,), if you specify DECIMAL POINT IS COMMA
    - space (leading and trailing)
    - sign (+ or –)

    The terminal operator can input space characters only as leading and trailing spaces. If this occurs, space characters are simply ignored during numeric conversion.

    However, the operator cannot input space characters *between* numeric characters, *between* numeric characters and a decimal point, or *between* a sign and any other input character. When this occurs, the input data is invalid, and an error condition results.

    The operator can input only one sign character and one decimal point character.

    When the operator inputs a sign character, it must precede or follow all numeric characters and the decimal point.

The default sign character is a plus sign (+).

The default number of decimal places is zero.

45. When you use the CONVERSION phrase and *dest-item* is numeric, data input results in an error condition if the operator enters:

    - Too many characters on either side of the decimal point (The PICTURE clause of *dest-item* determines this overflow condition.)
    - Invalid data

    When one of these error conditions occurs, and you do not specify the ON EXCEPTION phrase: (1) the contents of *dest-item* do not change, (2) the terminal bell rings, (3) the input field is erased, and (4) the ACCEPT statement executes again.

    When one of these error conditions occurs, and you do specify the ON EXCEPTION phrase: (1) the contents of *dest-item* do not change, (2) the input field is left as if no error occurred, and (3) the imperative statement of the ON EXCEPTION phrase executes.

46. When *dest-item* is not numeric, the CONVERSION phrase moves input characters to *dest-item* as an alphanumeric string (MOVE statement rules apply). Therefore, data can be accepted into an alphanumeric edited field, and the JUSTIFIED clause, if it applies to *dest-item*, can take effect.

    An overflow condition is not an error condition when *dest-item* is alphanumeric; in this case, right-end truncation occurs. However, you can specify the PROTECTED and SIZE phrases to limit the amount of input data when *dest-item* is alphanumeric.

47. When you use the CONVERSION phrase, and if the operator types the terminator key prior to any data input:

    - ZEROES are moved to a numeric or numeric edited *dest-item*, if you do not specify the DEFAULT phrase.
    - SPACES are moved to an alphanumeric or alphanumeric edited *dest-item*, if you do not specify the DEFAULT phrase.
    - However, the default value is moved to *dest-item*, if you do specify the DEFAULT phrase.

      If the default value is not a valid value for *dest-item*, an error condition results.

48. If you do not use the CONVERSION phrase, data is transferred to *dest-item* according to Format 1 ACCEPT statement rules.

PROTECTED Phrase

49. The PROTECTED phrase limits the number of characters that can be entered from the terminal.

50. If you do not specify the PROTECTED phrase, the cursor remains on the character position immediately following the position of the last input character. This is the default starting position of the next data item you input from or display upon the terminal.

However, if you use the PROTECTED phrase to delimit the field of a data item, the cursor moves to the character position immediately following the last position of the input field. In this case, the default starting position of the next data item is always to the right of the input field, as determined by the SIZE phrase or PICTURE clause.

51. When you specify the PROTECTED phrase:
    - If the operator attempts to type beyond the rightmost position of the input field: (1) the terminal bell rings, (2) the cursor remains on the rightmost position, and (3) character entry attempts beyond the rightmost position are not echoed to the terminal screen.
    - If the operator attempts to delete beyond the leftmost position of the input field: (1) the terminal bell rings, and (2) the cursor remains on the leftmost position.
52. If you use the PROTECTED phrase without the SIZE phrase, the maximum number of characters that the operator can enter is the number of characters in *dest-item*.

    However, if *dest-item* is numeric, the maximum number of characters allows for entry of sign and decimal point characters when these are implied by *dest-item's* PICTURE clause. For example, if PIC S9(4)V99 is the PICTURE clause for *dest-item*, then all of the following character strings are valid input:

    2222.22
    2222
    .22
    +2222.22
53. When you use the PROTECTED phrase, the input field is filled with spaces prior to accepting data. If you also use the UNDERLINED, REVERSED, BOLD, or BLINKING phrases, those spaces have the specified character attribute(s).
54. If you use the PROTECTED phrase on a field that extends past the last column position of the screen, the results are undefined.
55. If you do not use the PROTECTED phrase, an overflow condition is treated according to rules for the Format 1 ACCEPT statement.

SIZE Phrase

56. You can use the SIZE phrase only when you also specify the PROTECTED phrase.
57. the SIZE phrase specifies the number of characters in the input field. It allows you to specify fewer or more characters than are specified in the PICTURE clause for *dest-item*.

NO ECHO Phrase

58. The NO ECHO phrase suppresses the display of input characters on the screen.
59. When you do not use the NO ECHO phrase, input characters are displayed on the screen as they are typed.

DEFAULT Phrase

60. The DEFAULT phrase specifies default input values when no characters are entered from the terminal. Null input is signaled by entering a legal terminator key that is not preceded by data. (See the general rules for the CONTROL KEY phrase.)
61. When the null input condition occurs, *def-src-lit* or the value of *def-src-item* is moved to *dest-item*. When the move occurs, the specified default value is not displayed on the terminal screen.
62. The value of *def-src-item* cannot be the figurative constant ALL literal.

CONTROL KEY Phrase

63. If you use the CONTROL KEY phrase, the characters representing PF keys and arrow keys, as well as TAB and RETURN, are legal terminator keys and can be accepted from the terminal. (See Technical Note 6.)
64. *Key-dest-item* stores the terminator key code; unused character positions, if any, are filled with spaces. (See Technical Notes.)
65. When you use the CONTROL KEY phrase in Format 3, the operator must terminate data input with a legal terminator key.
66. When you do not use the CONTROL KEY phrase in Format 3, the operator can terminate data input only with RETURN or TAB.

Technical Notes

Format 1

1. The ACCEPT statement fills *dest-item* with spaces if the input is a carriage return only.

Formats 3 and 4

2. The DIGITAL extensions to the ACCEPT and DISPLAY statements support data input and display only on VT52 and VT100 terminal types, and on the PROFESSIONAL video terminal.
3. On RSX-11M systems ONLY:

   If both the "get multiple characteristics" and "set multiple characteristics" options were included when your system was generated, COBOL-81 automatically: (1) determines terminal type, (2) determines whether or not the terminal has been set to /NOWRAP, and (3) sets a terminal to /NOWRAP, when necessary. No manual intervention is needed for any supported terminal type. However:

   - If the "get multiple characteristics" option has not been included during system generation, then the DIGITAL extensions to the ACCEPT statement will not work on a VT52 terminal. COBOL-81 needs this option to determine terminal type. It assumes a terminal is a VT100 type if the "get characteristics" option is absent. There is no manual command you can use to solve this problem.

     The "get multiple characteristics" option also tells COBOL-81 whether a terminal (any supported type) has the /WRAP or the /NOWRAP characteristic. The /NOWRAP characteristic is needed for the DIGITAL extensions to the ACCEPT statement to work.

- If the "set multiple characteristics" option has not been included during system generation, then COBOL-81 cannot change a terminal's setting when this is necessary. Therefore, prior to program execution, you must determine if a terminal (any supported type) is set to /NOWRAP, and change the setting if it is not. Use the following MCR command to do this:

  SET /NOWRAP[=ttnn:]

- If the "set multiple characteristics" option has been included during system generation, but the "get multiple characteristics" option has not, you can still use a VT100 type terminal. But, prior to program execution, you must determine whether the terminal is set to /NOWRAP and manually change the setting when it is not. Use the preceding MCR command to do this.

4. You should only accept data from input fields that are within screen boundaries. That is, the terminal operator should see all the characters entered (assuming the NO ECHO, CONVERSION, and PROTECTED phrases are not specified). If you accept data from input fields that are outside screen boundaries, it does not result in an error condition. However, your program might not produce the results you expect.

   Values for screen boundaries depend on the terminal type and the column mode in which it is operating. Refer to the appropriate terminal user's guide for more information on screen boundaries.

5. Line positioning can be a one- or two-step process. The first (or only) step is absolute positioning, which is using the value of *line-num* or *line-id* to determine the line position. The second step is relative positioning, which is adding the value of *plus-num* to *line-id* to determine the line position. Relative positioning beyond the bottom line of the current screen results in scrolling.

   For example, suppose that the screen for which you are programming can be a maximum of 24 lines and you need to scroll the screen up one line before accepting data. The following sample statements illustrate how to use relative positioning to accomplish this (Assume ITEMA has a value of 14, and the current line position is 20):

```
ACCEPT DEST-EXAMPLE FROM LINE NUMBER PLUS 5.
ACCEPT DEST-EXAMPLE FROM LINE NUMBER ITEMA PLUS 11.
```

   The following sample statements would produce undefined results because absolute line positioning is beyond the bottom of the screen (assume ITEMB has a value of 25):

```
ACCEPT DEST-EXAMPLE FROM LINE NUMBER 25.
ACCEPT DEST-EXAMPLE FROM LINE NUMBER ITEMB.
ACCEPT DEST-EXAMPLE FROM LINE NUMBER ITEMB PLUS 0.
```

   The last ACCEPT statement illustrates that use of the PLUS option does not necessarily mean that scrolling will always occur. Absolute line positioning always occurs before the relative positioning specified by the PLUS option. In this case, *line-id* (ITEMB) has a value of 25. Therefore, the line position is outside the screen boundary *before* the PLUS option executes, and program results are undefined.

6. When you use the CONTROL KEY phrase, *key-dest-item* stores the terminator key code. Part IV of the COBOL-81 User's Guide for your system contains information on these key code values. Refer to the chapter on programming video forms.

Additional References

Section 3.1.3	SPECIAL-NAMES Paragraph
Section 5.1.4	Scope of Statements
Section 5.9.15	MOVE Statement
Part IV of the COBOL-81 User's Guide for your system	Refer to the chapter on programming video forms

Examples

In the following examples, the character *s* represents a space. The examples assume that the time is just after 2:15 PM on April 2, 1980. The Environment and Data Divisions contain the following entries:

```
SPECIAL-NAMES.
        CONSOLE IS IN-DEVICE.
DATA DIVISION.
01      ITEMA   PIC X(6).
01      ITEMB   PIC 99V99.
01      ITEMC   PIC 9(8).
01      ITEMD   PIC 9(5).
01      ITEME   PIC 9(6).
01      ITEMF   PIC 9.
```

1. `ACCEPT ITEMA.`

Input	ITEMA
COMPUTER	COMPUT
VAX	VAXsss
12.6	12.6ss

2. `ACCEPT ITEMB FROM IN-DEVICE.`

Input	ITEMB	Equivalent To
1623	1623	16.23
4	4sss	invalid data
60000	6000	60.00
–1.2	–1.2	invalid data
1.23	1.23	invalid data
COMPUTER	COMP	invalid data

	Results
3. `ACCEPT ITEME FROM DATE.`	ITEME = 800402
4. `ACCEPT ITEMC FROM TIME.`	ITEMC = 14150516
5. `ACCEPT ITEMD FROM DAY.`	ITEMD = 80093
6. `ACCEPT ITEMA FROM TIME.`	ITEMA = 141505
7. `ACCEPT ITEME FROM TIME.`	ITEME = 150516

Examples containing DIGITAL extensions to the ACCEPT statement (Formats 3 and 4) are in Part IV of the COBOL-81 User's Guide for your system. Refer to the chapter that discusses forms for video terminals.

5.9.2 ADD Statement

Function

The ADD statement adds two or more numeric operands and stores the result.

General Format

Format 1

```
ADD { num } ... TO { rsult [ ROUNDED ] } ... [ ON SIZE ERROR stment ]
```

Format 2

```
ADD { num } { num } ... GIVING { rsult [ ROUNDED ] } ... [ ON SIZE ERROR stment ]
```

Format 3

```
ADD { CORRESPONDING } grp–1 TO grp–2 [ ROUNDED ] [ ON SIZE ERROR stment ]
    { CORR          }
```

num
: is a numeric literal or the identifier of an elementary numeric item.

rsult
: is the identifier of an elementary numeric item. However, in Format 2, *rsult* can be an elementary numeric edited item. It is the resultant identifier.

grp-1, grp-2
: are the identifiers of numeric group items.

stment
: is an imperative statement.

Syntax Rule

CORR is an abbreviation for CORRESPONDING.

General Rules

1. The data descriptions of the operands need not be the same. Conversion and decimal point alignment will occur, as needed, throughout the calculation.
2. The maximum size of each operand is 18 digits.
3. Undefined results occur when operands overlap; that is, when sending fields and receiving fields share a part of their storage areas.
4. In Format 1, the values of the operands before the word TO are added. The sum is then added to the value of the first *rsult*. The process repeats for each later occurrence of *rsult*.

5. In Format 2, the values of the operands before the word GIVING are added. The sum is then moved to each *rsult*.
6. In Format 3, data items in *grp-1* are added to and stored in the corresponding data items in *grp-2*.

Additional References

Section 5.1.4	Scope of Statements
Section 5.6.1	Arithmetic Operations
Section 5.6.3	ROUNDED Option
Section 5.6.4	ON SIZE ERROR Option
Section 5.6.5	CORRESPONDING Option
Section 5.6.6	Overlapping Operands and Incompatible Data
Section 5.6.2	Multiple Receiving Fields in Arithmetic Statements

Examples

In these examples, results are shown only for data items whose values change. The examples assume the following data descriptions and beginning values:

```
                                                        Initial
                                                        Value
03  ITEMA  PIC 99 VALUE 85.                             85
03  ITEMB  PIC 99 VALUE 2.                              2
03  ITEMC  VALUE "123".
    05  ITEMD  OCCURS 3 TIMES                           123
            PIC 9.
```

Results

1. TO phrase:

```
ADD 2 ITEMB TO ITEMA.                                   ITEMA = 89
```

2. SIZE ERROR clause:

   (When the SIZE ERROR condition occurs, the value of the resultant identifier does not change.)

```
ADD 38 TO ITEMA                                         ITEMA = 85
  ON SIZE ERROR
    MOVE 0 TO ITEMB.                                    ITEMB = 0
```

3. Multiple receiving fields:

   (The operations proceed from left to right. Therefore, the subscript for ITEMD is evaluated after the addition changes its value.)

```
ADD 1 TO ITEMB ITEMD (ITEMB).                           ITEMB = 3
                                                        ITEMD (3) = 4
```

4. GIVING phrase:

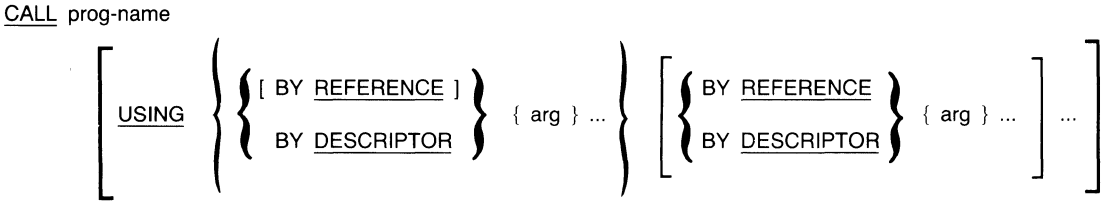
```
ADD ITEMB ITEMD (ITEMB) GIVING ITEMA.                   ITEMA = 4
```

5.9.3 CALL Statement

Function

The CALL statement transfers control to another program in the executable image.

General Format



prog-name
: must be a nonnumeric literal. It is the name of the program to which control transfers.

arg
: is the argument. It identifies the data that is available to both the calling and called programs. It is any data item described in the File Section, Working-Storage Section, or Linkage Section.

stment
: is an imperative statement.

Syntax Rules

1. *Prog-name* must be from 1 to 30 characters long. However, only the first 6 characters are significant; therefore, they must be unique. *Prog-name* can contain the characters A through Z, a through z, 0 through 9, and hyphen (-).
2. *Prog-name* is the entry-point in the called program. For COBOL programs, *prog-name* is the program-name specified in the PROGRAM-ID paragraph.
3. The same *arg* can appear more than once in the USING phrase.
4. The maximum number of arguments is 255.
5. If there is no initial argument-passing mechanism (REFERENCE, or DESCRIPTOR), BY REFERENCE is the default.
6. An argument-passing mechanism applies to every *arg* following it until a new mechanism (if any) appears.
7. The CALL statement has a USING phrase only if there is a USING phrase in the Procedure Division header of the called program. Both USING phrases must have the same number of arguments.
8. If the argument-passing mechanism is BY DESCRIPTOR, *arg* must be an alphanumeric item.

General Rules

1. The program whose name is specified by *prog-name* is the called program. The program containing the CALL statement is the calling program.
2. When the CALL statement executes, hyphens in *prog-name* are treated as periods, and lowercase letters are treated as uppercase.
3. The CALL statement transfers control to the called program.
4. The called program is in its initial state the first time it is called. On subsequent entries, the state of the called program is the same as when it was last exited.
5. Arguments correspond by position in the USING phrase – not by name. That is, the first *arg* in CALL statement corresponds to the first *data-name* in the called program's Procedure Division header, and so on.

   However, no correspondence between calling and called programs exists for index-names. If a table is passed as an argument, the index associated with that table in the called program will be the one specified in the INDEXED BY phrase in the called program, not the index specified in the calling program.
6. The arguments in the CALL statement USING phrase are made available to the called program when the CALL executes.
7. Called programs can contain CALL statements. However, a called program must not execute a CALL statement that directly or indirectly calls the calling program.
8. The CALL statement can make data available to the called program by two argument-passing mechanisms:
   - REFERENCE – The address of (pointer to) *arg* is passed to the called program. This is the default mechanism: arguments are passed BY REFERENCE if there is no explicit mechanism in the CALL statement.
   - DESCRIPTOR – The address of (pointer to) the data item's descriptor is passed to the called program.
9. If the called program is a COBOL program, the CALL statement can pass arguments only BY REFERENCE. If the called program is a non-COBOL program, the mechanism for each *arg* in the CALL statement USING phrase must be the same as the mechanism for each *data-name* in the called program's argument list.
10. If the BY REFERENCE phrase is either specified or implied for a parameter, the called program references the same storage area for the data item as the calling program. This mechanism ensures that the contents of the parameter in the calling program is identical at all times with the contents of the parameter in the called program.

CALL
Continued

Additional References

Section 1.4.1	Procedure Division
Section 2.1	PROGRAM-ID Paragraph
Section 5.1.4	Scope of Statements
Part II of the COBOL-81 User's Guide for your system	Refer to the chapter on interprogram communication

Examples

1. Passing arguments by reference:

```
CALL "DATERTN" USING ITEMA ITEMB ITEMC.
```

2. Mixing argument-passing mechanisms:

   (Reference arguments are ITEMA and ITEMD. Descriptor arguments are ITEMB, ITEMC, ITEMD, and ITEMF. ITEMD is passed twice – by reference and by descriptor. It will correspond to two different data items in the called program.)

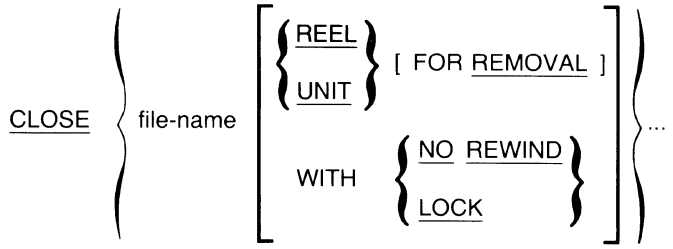
```
CALL "NEWPROG" USING ITEMA
  BY DESCRIPTOR ITEMB ITEMC
  BY REFERENCE ITEMD
  BY DESCRIPTOR ITEMD ITEMF
```

5.9.4 CLOSE Statement

Function

The CLOSE statement ends processing of reels (or units) and files. It can also perform rewind, lock, and removal operations.

General Format



file-name
is the name of a file described in the Data Division. It cannot be a sort or merge file.

Syntax Rules

1. The REEL or UNIT phrase can be used only for sequential files.
2. The words REEL and UNIT are equivalent.

General Rules

Note

In these rules, the term "reel" refers to "unit" as well.

1. A CLOSE statement can execute only for an open file.
2. To show the effects of CLOSE statements, all files are categorized as follows:
   - *Nonreel*: a file for which the concepts of rewind and reel have no meaning because of its input or output medium (for example, when a terminal device is used)
   - *Sequential single-reel*: a sequential file contained entirely on one reel
   - *Sequential multireel*: a sequential file contained on more than one reel
   - *Nonsequential*: a file with other than sequential organization, whose medium is on a mass storage device
3. Table 5-8 summarizes CLOSE statement results. Symbol definitions follow the table.

   Where definitions differ for input, output, and input-output files, separate definitions appear. Otherwise, a definition applies to files in all open modes.

Table 5-8: Effects of CLOSE Statement Formats on Files by Category

CLOSE Statement Format	File Category			
	Nonreel	Sequential Single-reel	Sequential Multireel	Nonsequential
CLOSE	C	C,G	C,G,A	C
CLOSE WITH LOCK	C,E	C,G,E	C,G,E,A	C,E
CLOSE WITH NO REWIND	X	C,B	C,B,A	X
CLOSE REEL	X	F,G	F,G	X
CLOSE REEL FOR REMOVAL	X	F,D,G	F,D,G	X

A Previous reels unaffected

For input and input-output files: All reels in the file before the current reel are processed according to the standard reel swap procedure. However, reels controlled by an earlier CLOSE REEL/UNIT statement are not affected. If other reels in the file follow the current reel, they are not processed.

For output files: All reels in the file before the current reel are processed according to the standard reel swap procedure. However, reels controlled by an earlier CLOSE REEL/UNIT statement are not affected.

B No rewind of current reel

The position of the current reel remains the same.

C Close file

The file is closed.

D Reel/unit removal

The current reel rewinds and is logically removed from the executable image. However, the executable image can access the reel again in its proper order of reels in the file. To do this, the executable image must subsequently execute: (1) a CLOSE statement without the REEL/UNIT phrase for the file and (2) an OPEN statement for the file.

E File lock

The executable image cannot open the file again in its current execution.

F Close reel/unit

For input and input-output files: If the current reel is the last or only reel for the file:

- A reel swap does not occur
- The Current Volume Pointer remains the same

- The Next Record Pointer indicates that there is no next logical record

If another reel follows the current reel for the file:

- A reel swap occurs.
- The Current Volume Pointer points to the next reel for the file.
- The Next Record Pointer points to the next record in the file. If there are no records for the current volume, another reel swap occurs.

For output files: A reel swap occurs. The Current Volume Pointer points to the new reel.

Executing the next WRITE statement for the file transfers a logical record to the new reel of the file.

G Rewind

The current reel (or device) is positioned to its physical beginning.

X Invalid

This is an invalid combination of CLOSE option and file category. It results in FILE STATUS data item value "98".

4. Executing a CLOSE statement updates the value of the FILE STATUS data item associated with the file.
5. If an optional file is not present, standard end-of-file processing does not occur.
6. The WITH NO REWIND and FOR REMOVAL phrases have no effect at execution time if they do not apply to the file's storage medium.
7. When the CLOSE statement applies to an output or extend file described with the LINAGE clause, end-of-page processing occurs before the file is closed.
8. After successful CLOSE statement execution (without the REEL or UNIT phrase), the file's record area is no longer available. After unsuccessful execution, record area availability is undefined.
9. After successful CLOSE statement execution (without the REEL or UNIT phrase), the file is no longer: (a) in the open mode or (b) associated with the file connector.
10. If the CLOSE statement has more than one *file-name*, the statement executes as if there were a separate CLOSE statement for each *file-name*.
11. In the file-sharing environment, CLOSE statement execution unlocks all record locks for *file-name*.

CLOSE
Continued

Technical Notes

1. CLOSE statement execution can result in these FILE STATUS data item values:

FILE STATUS	Meaning
00	Successful
94	File never opened or already closed
98	Any other CLOSE error

2. The RSTS/E operating system does not support multivolume tape files. Therefore, sections of the General Rules that refer to multivolume files do not apply to COBOL-81 programs that execute on a RSTS/E system.

Additional Reference

Section 5.7 I-O Status

5.9.5 COMPUTE Statement

Function

The COMPUTE statement evaluates an arithmetic expression and stores the result.

General Format

```
COMPUTE { rsult [ ROUNDED ] } ... = arithmetic-expression  [ ON SIZE ERROR stment ]
```

rsult
: is the identifier of an elementary numeric item or elementary numeric edited item. It is the resultant identifier.

stment
: is an imperative statement.

General Rules

1. The data descriptions of the operands need not be the same. Conversion and decimal point alignment will occur, as needed, throughout the calculation.
2. The maximum size of each operand is 18 digits.
3. When the arithmetic expression is evaluated, its value replaces the current value(s) of *rsult(s)*.
4. Undefined results occur when operands overlap; that is when a sending and receiving field share a part of their storage areas.
5. Exponents must be specified as integers.

Additional References

Section 5.1.4	Scope of Statements
Section 5.6.1	Arithmetic Operations
Section 5.6.3	ROUNDED Option
Section 5.6.4	ON SIZE ERROR Option
Section 5.6.6	Overlapping Operands and Incompatible Data
Section 5.6.2	Multiple Receiving Fields in Arithmetic Statements

Examples

In these examples, results are shown only for data items whose values change. The examples assume these data descriptions and beginning values:

```
                                                          Initial
                                                          Value

03  ITEMA  PIC 999V99 VALUE 2.                             2
03  ITEMB  PIC 999V99 VALUE 3.                             3
03  ITEMC  PIC 999V99 VALUE 4.                             4
03  ITEMD  PIC 999V99 VALUE 5.                             5
```

(continued on next page)

COMPUTE
Continued

Results

1. No rounding:

```
COMPUTE ITEMC =
   (ITEMA + 27) / ITEMB.
```

ITEMC = 9.66

2. With rounding:

```
COMPUTE ITEMC ROUNDED =
   (ITEMA + 27) / ITEMB.
```

ITEMC = 9.67

3. The SIZE ERROR phrase:

```
COMPUTE ITEMB = (ITEMA * ITEMD) ** 3
   ON SIZE ERROR
     MOVE 100 TO ITEMC.
```

ITEMC = 100.00

5.9.6 DELETE Statement

Function

The DELETE statement logically removes a record from a mass storage file.

General Format

```
DELETE file-name RECORD  [ INVALID KEY stment ]
```

file-name
: is the name of a relative or indexed file described in the Data Division. It cannot be the name of a sequential file or a sort or merge file.

stment
: is an imperative statement.

Syntax Rules

1. There cannot be an INVALID KEY phrase for a DELETE statement that references a file in sequential access mode.
2. There must be an INVALID KEY phrase if: (a) the file is not in sequential access mode and (b) there is no applicable USE AFTER EXCEPTION procedure.

General Rules

1. The file must be open in I-O mode when the DELETE statement executes.
2. For a file in sequential access mode, a successfully executed READ statement must be the last input-output statement executed for the file before the DELETE statement. Record Management Services (RMS-11) logically removes the record that the READ statement accessed.
3. For a relative file in random or dynamic access mode, RMS-11 logically removes the record identified by the file's RELATIVE KEY data item. If the file does not contain that record, an invalid key condition exists.
4. For an indexed file in random or dynamic access mode, RMS-11 logically removes the record identified by the file's prime record key data item. If the file does not contain that record, an invalid key condition exists.
5. After successful DELETE statement execution, the identified record has been logically removed from the file. It is no longer accessible.
6. DELETE statement execution does not affect the contents of the record area. It also does not affect the contents of the data item referred to in the DEPENDING ON phrase of the file's RECORD clause.
7. For sequential access files, DELETE statement execution does not affect the Next Record Pointer.

8. For dynamic access files, the Next Record Pointer can point to the deleted record before the DELETE. After the DELETE statement executes, the Next Record Pointer:
   - Points to a *relative* file's next existing record.
   - Points to an *indexed* file's next existing record, as established by the Key of Reference.
   - Indicates the at end condition if the file has no next record.
9. DELETE statement execution updates the value of the FILE STATUS data item for the file.
10. If there is an applicable USE AFTER EXCEPTION procedure, it executes whenever an input or output condition occurs that would result in a nonzero value in a FILE STATUS data item. However, it does not execute if the condition is invalid key, and there is an INVALID KEY phrase. If the condition is not invalid key and no applicable USE AFTER EXCEPTION Declarative procedure exists, the program run terminates abnormally.
11. When the invalid key condition is recognized, these actions occur in the following order:
    a. A value indicating the invalid key condition is placed in the FILE STATUS data item, if one is specified, for the file.
    b. If the statement causing the condition has an INVALID KEY phrase, control transfers to the associated imperative statement. Any USE AFTER EXCEPTION procedure for the file does not execute.
    c. If there is no INVALID KEY phrase, control transfers to the applicable USE AFTER EXCEPTION procedure for the file.

Technical Note

DELETE statement execution can result in the following FILE STATUS data item values:

FILE STATUS	Access Method	Meaning
00	All	Successful
23	Rand	Record not in file (invalid key)
92	All	Record locked by another program
93	Seq	No previous READ
94	All	File not open, or incompatible open mode
30	All	All other permanent errors

Additional References

Section 5.1.4	Scope of Statements
Section 5.7	I-O Status
Section 5.7.1	INVALID KEY Phrase
Section 5.9.17	OPEN Statement
Section 5.9.31	USE Statement

5.9.7 DISPLAY Statement

Function

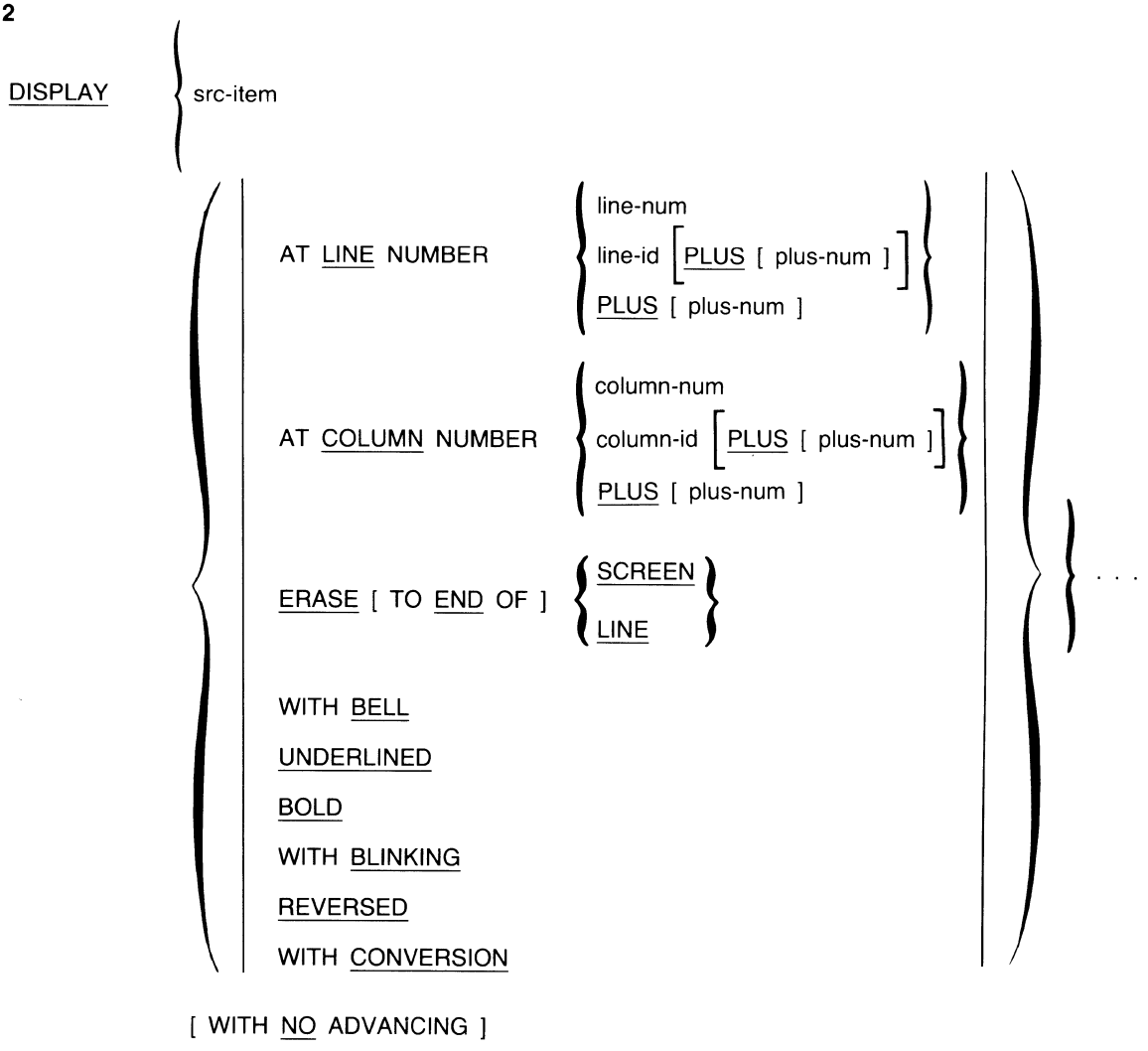
The DISPLAY statement transfers low-volume data from the program to the default system output device or to the object of a mnemonic-name.

General Format

Format 1

```
DISPLAY { src-item } ... [ UPON output-dest ] [ WITH NO ADVANCING ]
```

Format 2



src-item
: is a literal or the identifier of a data item. The literal can be any figurative constant except ALL literal.

output-dest
: is a mnemonic-name defined in the SPECIAL-NAMES paragraph of the Environment Division.

line-num
: is a numeric literal that specifies a line position on the terminal screen. *Line-num* must be a positive integer. It cannot be zero or a negative integer.

line-id
: is the identifier of a data item that provides a line position on the terminal screen.

column-num
: is a numeric literal that specifies a column position on the terminal screen. *Column-num* must be a positive integer. It cannot be zero or a negative integer.

column-id
: is the identifier of a data item that provides a column position on the terminal screen.

plus-num
: is a numeric literal that increments the current value for line or column position, or that increments the value of *line-id* or *column-id*. *Plus-num* can be zero or a positive integer. It cannot be a negative integer.

Syntax Rules

All Formats

1. In a DISPLAY statement, the number of *src-item* entries cannot exceed 254.

Format 2

2. No phrase can appear more than once for any *src-item*.

General Rules

Format 1

1. The DISPLAY statement transfers data from each *src-item* (in its order of appearance in the statement) to *output-dest*.
2. No editing or conversion occurs during DISPLAY execution.
3. If *src-item* is a figurative constant, only one occurrence is displayed.
4. When there is more than one *src-item*, sending item size is the sum of the *src-item* sizes.
5. If there is no UPON phrase, the DISPLAY statement transfers data to the default system output device.
6. If there is a WITH NO ADVANCING phrase, the DISPLAY statement does not transfer any device positioning information after the last *src-item* value.
7. If there is no WITH NO ADVANCING phrase, the DISPLAY statement transfers device positioning information. It resets the *output-dest* position to the leftmost position on the next line.

Format 2

8. The presence of either the LINE NUMBER phrase or the COLUMN NUMBER phrase implies NO ADVANCING; that is, no line feed or carriage return is generated automatically following data output. The cursor remains on the character position immediately following the position of the last character displayed. This is the default starting position for the next data item you input from or display upon the terminal.
9. If you specify neither the LINE NUMBER phrase, the COLUMN NUMBER phrase, nor the NO ADVANCING phrase, data is output according to Format 1 positioning rules for the DISPLAY statement. That is, a line feed and carriage return is generated automatically following data display.

LINE NUMBER Phrase

10. The LINE NUMBER phrase positions the cursor for output on a specific line position on the terminal screen.
11. If you do not use the LINE NUMBER phrase, but you do use the COLUMN NUMBER phrase, then data is displayed on the *current* line position and specified column position.
12. You must use relative line positioning to advance the cursor position below the bottom line position of the screen. If *line-num* or the value of *line-id* is greater than the bottommost line position of the current screen, program results are undefined. (See Technical Notes.)
13. If you use *line-id* without its PLUS option, the line position is the value of *line-id*.
14. If you use *line-id* with its PLUS option, the line position is the sum of *plus-num* and the value of *line-id*.
15. If you use the PLUS option without *line-id*, the line position is the sum of *plus-num* and the value of the current line position.
16. If you use the PLUS option, but you do not specify *plus-num*, then PLUS 1 is implied.
17. Data output results are undefined if the program generates a value for line position that is negative or zero.

COLUMN NUMBER Phrase

18. The COLUMN NUMBER phrase positions the cursor for output on a specific column of the video screen.
19. If the COLUMN NUMBER phrase does not appear, but the LINE NUMBER phrase does, then data is displayed from column 1 of the specified line position.
20. If you use *column-id* without its PLUS option, the column position is the value of *column-id*.
21. If you use *column-id* with its PLUS option, the column position is the sum of *plus-num* and the value of *column-id*.
22. If you use the PLUS option without *column-id*, the column position is the sum of *plus-num* and the value of the current column position.

23. If you use the PLUS option, but do not specify *plus-num*, then PLUS 1 is implied.
24. Data output results are undefined if the program generates a value for column position that is one of the following:
    - Zero
    - Negative
    - Greater than the rightmost column position on the screen.

ERASE Phrase

25. The ERASE phrase erases all, or part, of a line (or screen) before displaying data.
26. If you use its TO END option, the ERASE phrase erases the line (or screen) from the implied, or stated, cursor position to the end of the line (or screen).
27. If you do not use its TO END option, the ERASE phrase erases the entire line (or screen).

    However, if you do not use the TO END option and the output device is a VT52 terminal, then cursor position cannot be implied. In this case, you must specify both the LINE NUMBER and the COLUMN NUMBER phrases, and you can use the PLUS option only when it's preceded by *line-id* (or *column-id*).

    If the output device is a VT52 terminal and you do not use the TO END option, the ERASE phrase will produce unpredictable results when any of the following conditions are true:

    - The LINE NUMBER phrase is absent.
    - The COLUMN NUMBER phrase is absent.
    - The PLUS option appears, but is not preceded by either *line-id* or *column-id*.

BELL Phrase

28. The BELL phrase rings the terminal bell before displaying data.

UNDERLINED Phrase

29. The UNDERLINED phrase displays characters on the screen with the "underscore on" character attribute.

BOLD Phrase

30. The BOLD phrase displays characters on the screen with the "bold on" character attribute.

BLINKING Phrase

31. The BLINKING phrase displays characters on the screen with the "blink on" character attribute.

REVERSED Phrase

32. The REVERSED phrase display characters on the screen with the "reversed video on" character attribute.

CONVERSION Phrase

33. The CONVERSION phrase allows you to display data in a field and achieve the same results as you would with the MOVE statement. How the CONVERSION phrase affects data handling depends on the category of *dest-item*. (Numeric data can be described by any USAGE clause.)
34. The CONVERSION phrase displays without change: nonnumeric items, and numeric edited items.
35. The CONVERSION phrase displays numeric items with DISPLAY usage (for example, PIC 99, or PIC S99V99) after including space, when needed, for a decimal point and/or sign.

    If you specify the SIGN IS TRAILING clause for the data item, the sign is displayed as a trailing sign. Otherwise, the sign is displayed as a leading sign.
36. The CONVERSION phrase displays numeric items without DISPLAY usage (for example, PIC 99 COMP, or PIC S9V999 COMP SYNC) after converting them to DISPLAY usage. The conversion proceeds according to the following rules:
    - The size of the displayed field is determined from the PICTURE character-string.
    - Leading zeroes are displayed only when they immediately precede a decimal point.
    - A sign is displayed if an S is present in the PICTURE character-string; the sign appears as a leading sign.

      Only a minus sign (–) is displayed. (When a signed item with a positive value is displayed, the plus sign (+) does not appear.)

      Only a leading sign can appear when the DISPLAY statement converts a numeric item to DISPLAY usage.
37. If you do not specify the CONVERSION phrase, data is transferred to the screen according to Format 1 rules for the DISPLAY statement.

Technical Notes

Format 1

1. When there is an UPON phrase, DISPLAY transfers data to the device associated with the SPECIAL-NAMES paragraph description of *output-dest*.

Format 2

2. The DIGITAL extensions to the ACCEPT and DISPLAY statements support data input and display only on the VT52, and VT100 terminal types, and on the PROFESSIONAL video terminal.
3. The UNDERLINED, BOLD, BLINKING, and REVERSED character attributes are not available on VT52 terminals or on VT100 terminals without the advanced video option.

4. On RSX-11M systems ONLY:

   If both the "get multiple characteristics" and "set multiple characteristics" options were included when your system was generated, COBOL-81 automatically: (1) determines terminal type, (2) determines whether or not the terminal has been set to /NOWRAP, and (3) sets a terminal to /NOWRAP, when necessary. No manual intervention is needed for any supported terminal type. However:

   - If the "get multiple characteristics" option has not been included during system generation, then DIGITAL extensions to the DISPLAY statement will not work on a VT52 terminal. COBOL-81 needs this option to find out terminal type, and it will assume a terminal is a VT100 type if the "get characteristics" option is absent. There is no manual command you can use to solve this problem.

     The "get multiple characteristics" option also tells COBOL-81 whether a terminal (any supported type) has the /WRAP or the /NOWRAP characteristic. The /NOWRAP characteristic is needed for the DIGITAL extensions to the DISPLAY statement to work.

   - If the "set multiple characteristics" option has not been included during system generation, then COBOL-81 cannot change a terminal's setting when this is necessary. Therefore, prior to program execution, you must determine if a terminal (any supported type) is set to /NOWRAP, and change the setting if it is not. Use the following MCR command to do this:

     SET /NOWRAP[=ttnn:]

   - If the "set multiple characteristics" option has been included during system generation, but the "get multiple characteristics" option has not, you can still use a VT100 type terminal. But, prior to program execution, you must determine whether the terminal is set to /NOWRAP and manually change the setting when it is not. Use the preceding MCR command to do this.

5. You should only DISPLAY data on fields that are within screen boundaries. That is, the terminal operator should see all the characters entered. If data is displayed on fields that are outside screen boundaries, it does not result in an error condition. However, your program might not produce the results you expect.

   Values for screen boundaries depend on the terminal type and the column mode in which it is operating. Refer to the appropriate terminal user's guide for more information on screen boundaries.

6. Line positioning can be a one- or two-step process. The first (or only) step is absolute positioning, which is using the value of *line-num* or *line-id* to determine the line position. The second step is relative positioning, which is adding the value of *plus-num* to *line-id* to determine the line position. Relative positioning beyond the bottom line of the current screen results in scrolling.

   For example, suppose that the screen for which you are programming can be a maximum of 24 lines, and you want to scroll the screen up one line before displaying data. The following sample statements illustrate how to use relative positioning to accomplish this (Assume ITEMA has a value of 14, and the current line position is 20):

   ```
   DISPLAY SRC-EXAMPLE AT LINE NUMBER PLUS 5.
   DISPLAY SRC-EXAMPLE AT LINE NUMBER ITEMA PLUS 11.
   ```

The following sample statements would produce undefined results because they use absolute line positioning to reach a line beyond the bottom of the screen (assume ITEMB has a value of 25):

```
DISPLAY SRC-EXAMPLE AT LINE NUMBER 25.
DISPLAY SRC-EXAMPLE AT LINE NUMBER ITEMB.
DISPLAY SRC-EXAMPLE AT LINE NUMBER ITEMB PLUS 0.
```

The last DISPLAY statement illustrates that use of the PLUS option does not necessarily mean that relative positioning and scrolling will always occur. When you specify *line-id*, absolute line positioning always occurs before a PLUS option can execute. In this case, *line-id* (ITEMB) is specified, and it has a value of 25. Therefore, the line position is outside the screen boundary *before* the PLUS option executes, and program results are undefined.

Additional References

Section 3.1.3	SPECIAL-NAMES Paragraph
Part IV of the COBOL-81 User's Guide for your system	Refer to the chapter on forms for video terminals

Examples

In the example results, the character *s* represents a space. The examples assume the following Environment and Data Division entries:

```
SPECIAL-NAMES.
    LINE-PRINTER IS ERR-REPORTER.

01  ITEMA PIC X(6) VALUE "ITEMS ".
01  ITEMB PIC X(8) VALUE "VALID".
01  ITEMC PIC X(5) VALUE "TODAY".
01  ITEMD PIC 99 VALUE 2.
01  ITEME PIC X(10) VALUE "MONDAY".
```

		Results
1.	`DISPLAY ITEMC.`	`TODAY`
2.	`DISPLAY ITEMD UPON ERR-REPORTER.`	`02`
3.	`DISPLAY ITEMD ITEMA "ARE" ITEMB.`	`02ITEMSsAREVALIDsss`
4.	`DISPLAY ITEMD  SPACE ITEMA "AREs" ITEMB.`	`02sITEMSsAREsVALIDsss`

5.	`DISPLAY ITEMC "sISs" NO ADVANCING.`
	`DISPLAY ITEME.`
	`DISPLAY ITEME.`
		`TODAYsISsMONDAYssss`
		`MONDAYssss`

The COBOL-81 User's Guide for your system contains examples using the DIGITAL extensions to the DISPLAY statement (Format 2). Refer to the chapter in Part IV that discusses forms for video terminals.

5.9.8 DIVIDE Statement

Function

The DIVIDE statement divides one or more numeric data items by another. It stores the quotient and remainder.

General Format

Format 1

```
DIVIDE srcnum INTO { rsult [ ROUNDED ] } ... [ ON SIZE ERROR stment ]
```

Format 2

```
DIVIDE srcnum INTO srcnum GIVING { rsult [ ROUNDED ] } ... [ ON SIZE ERROR stment ]
```

Format 3

```
DIVIDE srcnum BY srcnum GIVING { rsult [ ROUNDED ] } ... [ ON SIZE ERROR stment ]
```

Format 4

```
DIVIDE srcnum INTO srcnum GIVING rsult [ ROUNDED ] REMAINDER remaind
       [ ON SIZE ERROR stment ]
```

Format 5

```
DIVIDE srcnum BY srcnum GIVING rsult [ ROUNDED ] REMAINDER remaind
       [ ON SIZE ERROR stment ]
```

srcnum
: is a numeric literal or the identifier of an elementary numeric item.

rsult
: is the identifier of an elementary numeric item or an elementary numeric edited item. However, in Format 1, *rsult* must be an elementary numeric item. It is the resultant identifier.

remaind
: is the identifier of an elementary numeric item or an elementary numeric edited item.

stment
: is an imperative statement.

General Rules

1. The data descriptions of the operands need not be the same. Conversion and decimal point alignment will occur, as needed, throughout the calculation.

2. The maximum size of each operand is 18 digits.
3. Undefined results occur when operands share a part of their storage areas.

Format 1

4. The value of *srcnum* is divided into the value of the first *rsult*. This quotient replaces the current value of the first *rsult*. The process repeats for each of the other occurrences of *rsult*.

Format 2

5. The value of the first *srcnum* is divided into the value of the second. This quotient replaces the current value of each *rsult*.

Format 3

6. The value of the first *srcnum* is divided by the value of the second. This quotient replaces the current value of each *rsult*.

Formats 4 and 5

7. These formats produce a remainder (*remaind*) from the division operation. The remainder is the result of subtracting the product of the quotient (*rsult*) and the divisor from the dividend.

   If *rsult* refers to a numeric edited item, the quotient is an equivalent unedited intermediate field. For example, if you describe *rsult* with the PICTURE –ZZ.99, the compiler uses an intermediate field with the implicit PICTURE S99V99.

   When the ROUNDED phrase is present, the remainder computation uses an intermediate quotient field that is truncated rather than rounded.

8. The computation described in Rule 7 determines the accuracy of *remaind*. It includes decimal point alignment and truncation (not rounding) required by the description of *remaind*.
9. When the ON SIZE ERROR phrase is present:
   - If the size error occurs on *rsult*, the contents of both *rsult* and *remaind* are unchanged.
   - If the size error occurs on *remaind*, its contents are unchanged. However, when a size error occurs in any arithmetic statement with multiple results, your program must analyze the results to determine where the size error occurred.

Additional References

Section 5.1.4	Scope of Statements
Section 5.6.1	Arithmetic Operations
Section 5.6.3	ROUNDED Option
Section 5.6.4	ON SIZE ERROR Option
Section 5.6.6	Overlapping Operands and Incompatible Data
Section 5.6.2	Multiple Receiving Fields in Arithmetic Statements

Examples

In these examples, results are shown only for data items whose values the statements change. The examples assume the following data descriptions and beginning values:

```
                                                        Initial
                                                        Value
03  ITEMA  PIC 99V99 VALUE 9.                           9.00
03  ITEMB  PIC 99V99 VALUE 24.                          24.00
03  ITEMC  PIC 99V99 VALUE 8.                           8.00
03  ITEMD  PIC 99    VALUE 12.                          12
03  ITEME  PIC 99V99 VALUE 3.                           3.00
```

1. Without GIVING phrase or rounding:

```
DIVIDE ITEMA INTO ITEMB.                    ITEMB = 2.66
```

2. With rounding:

```
DIVIDE ITEMA INTO ITEMB ROUNDED.            ITEMB = 2.67
```

3. GIVING phrase:

```
DIVIDE ITEMA INTO ITEMB                     ITEMD = 2
  GIVING ITEMD.
```

4. GIVING phrase with rounding:

```
DIVIDE ITEMA INTO ITEMB                     ITEMD = 3
  GIVING ITEMD ROUNDED.
```

5. BY phrase:

```
DIVIDE ITEMA BY ITEMB                       ITEMD = 0
  GIVING ITEMD.
```

6. REMAINDER phrase:

```
DIVIDE ITEMA INTO ITEMB                     ITEMD = 2
  GIVING ITEMD REMAINDER ITEMC.             ITEMC = 6.00
```

7. REMAINDER phrase with rounding:

```
DIVIDE ITEMA INTO ITEMB                     ITEMD = 3
  GIVING ITEMD ROUNDED REMAINDER ITEMC.     ITEMC = 6.00
```

8. Effects of decimal alignment on quotient and remainder:

```
DIVIDE ITEMA INTO ITEMB                     ITEME = 2.66
  GIVING ITEME REMAINDER ITEMC.             ITEMC = .06
```

9. Effects of decimal alignment on remainder and quotient with rounding:

```
DIVIDE ITEMA INTO ITEMB                     ITEME = 2.67
  GIVING ITEME ROUNDED REMAINDER ITEMC.     ITEMC = .06
```

5.9.9 EXIT Statement

Function

The EXIT statement provides a common logical end point for a series of procedures.

General Format

EXIT .

Syntax Rule

The EXIT statement must appear only in a sentence by itself and comprise the only sentence in the paragraph.

General Rule

The EXIT statement associates a procedure-name with a point in the program. It has no other effect on program compilation or execution.

Example

```
REPORT-INVALID-ADD.
    DISPLAY " ".
    DISPLAY "INVALID ADDITION".
    DISPLAY "RECORD ALREADY EXISTS".
    DISPLAY "UPDATE ATTEMPT: " UPDATE-REC.
    DISPLAY "EXISTING RECORD: " OLD-REC.
REPORT-INVALID-ADD-EXIT.
    EXIT.
```

EXIT PROGRAM

5.9.10 EXIT PROGRAM Statement

Function

The EXIT PROGRAM statement marks the logical end of a called program.

General Format

EXIT PROGRAM

Syntax Rule

If the EXIT PROGRAM statement is in a consecutive sequence of imperative statements, it must be the last statement in that sequence.

General Rules

1. If EXIT PROGRAM executes in a program that is not a called program, it has the same effect as a STOP RUN statement; program execution ends.
2. If the EXIT PROGRAM statement executes in a called program, execution continues with the next executable statement after the CALL statement in the calling program.

   The state of the calling program does not change; it is the same as when the program executed the CALL statement. However, the contents of data items and the positioning of data files shared by the calling and called programs may change.

   The state of the called program does not change. However, the called program is considered to have reached the ends of the ranges of all PERFORM statements it executed. Therefore, an error does not occur if the called program is entered again during image execution.

Example

```
TEST-RETURN.
    IF ITEMA NOT = ITEMB
         MOVE ITEMA TO ITEMB
         EXIT PROGRAM.
```

5.9.11 GO TO Statement

Function

The GO TO statement transfers control from one part of the Procedure Division to another.

General Format

Format 1

```
GO TO proc-name
```

Format 2

```
GO TO proc-name { proc-name } ... DEPENDING ON num
```

proc-name
is a procedure-name.

num
is the identifier of an elementary numeric item described with no positions to the right of the assumed decimal point.

Syntax Rule

A Format 1 GO TO statement that is in a consecutive sequence of imperative statements in a sentence must be the last statement in the sentence.

General Rules

Format 1

1. The GO TO statement transfers control to *proc-name*.

Format 2

2. The GO TO statement transfers control to the *proc-name* in the ordinal position indicated by the value of *num*.

   No transfer occurs, and control passes to the next executable statement if the value of *num* is one of the following:

   - Not greater than zero
   - Greater than the number of *proc-names* in the statement

Examples

1. Format 1:

```
GO TO ENDING-ROUTINE.
```

2. Format 2:

```
GO TO FRESHMAN
      SOPHOMORE
      JUNIOR
      SENIOR
        DEPENDING ON YEAR-LEVEL.
MOVE ...
```

Sample Results

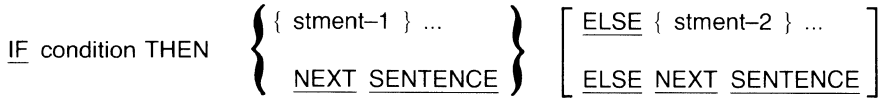
YEAR-LEVEL	Transfers to
1	FRESHMAN
3	JUNIOR
5	MOVE statement
0	MOVE statement
–10	MOVE statement

5.9.12 IF Statement

Function

The IF statement evaluates a condition. The condition's truth value determines the program action that follows.

General Format



stment-1
stment-2
: are imperative or conditional statements. An imperative statement can precede a conditional statement.

Syntax Rule

The ELSE NEXT SENTENCE phrase is optional if it immediately precedes a separator period.

General Rules

1. The scope of an IF statement ends with any of the following:
   - A separator period
   - An ELSE phrase associated with an IF statement at a higher nesting level
2. If the condition is true, the following control transfers occur:
   - If there is a *stment-1*, it executes.

     Stment-1 can contain a procedure branching or conditional statement. Control then transfers according to the rules of the statement.

     Otherwise, the ELSE phrase (if any) is ignored. Control passes to the end of the IF statement.
   - If you use NEXT SENTENCE instead of *stment-1*, the ELSE phrase (if any) is ignored. Control passes to the next executable sentence.
3. If the condition is false, the following control transfers occur:
   - *Stment-1* or its substitute NEXT SENTENCE is ignored. If *stment-2* is used, it executes.

     Stment-2 can contain a procedure branching or conditional statement. Control then transfers according to the rules of the statement. Otherwise, control passes to the end of the IF statement.
   - If there is no ELSE phrase, *stment-1* is ignored. Control passes to the end of the IF statement.
   - If the ELSE NEXT SENTENCE phrase is present, *stment-1* is ignored. Control passes to the next executable sentence.

4. An IF statement can appear in either or both *stment-1* and *stment-2*. In this case, the IF statement is considered nested, because its scope is entirely within the scope of another IF statement.
5. IF statements within IF statements are paired combinations, beginning with IF and ending with ELSE. This pairing proceeds from left to right. Thus, an ELSE applies to the first preceding unpaired IF.

Additional References

Section 5.1	Verbs, Statements, and Sentences
Section 5.1.4	Scope of Statements
Section 5.5	Conditional Expressions

Examples

1. No ELSE phrase:

```
IF ITEMA < 20
  MOVE "X" TO ITEMB.
```

ITEMA	ITEMB
4	"X"
35	?
19	"X"

2. With ELSE phrase:

```
IF ITEMA > 10
  MOVE "X" TO ITEMB
ELSE
  GO TO PROC-A.
ADD ...
```

ITEMA	Next Statement	ITEMB
96	ADD	"X"
8	PROC-A	?

3. With NEXT SENTENCE phrase:

   (In each case, the next executable statement is the ADD statement.)

```
IF ITEMA < 10 OR > 20
  NEXT SENTENCE
ELSE
  MOVE "X" TO ITEMB.
ADD ...
```

ITEMA	ITEMB
5	?
17	"X"
35	?

4. Nested IF statements:

```
IF ITEMA > 10
  IF ITEMA = ITEMC
    MOVE "X" TO ITEMB
  ELSE
    MOVE "Y" TO ITEMB
ELSE
  GO TO PROC-A.
ADD ...
```

ITEMA	ITEMC	Next Statement	ITEMB
12	6	ADD	"Y"
12	12	ADD	"X"
8	8	PROC-A	?

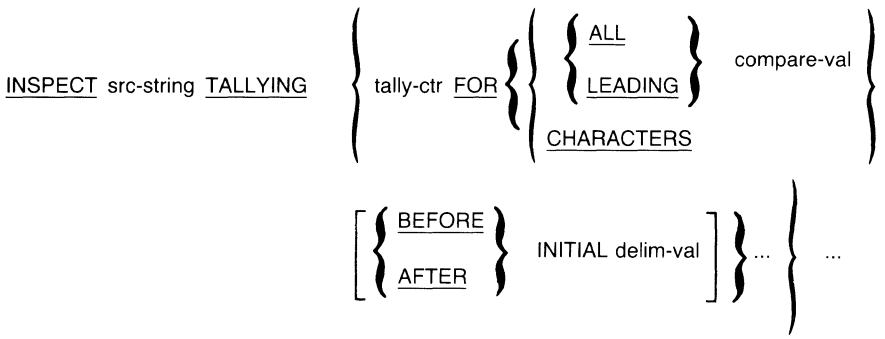
5.9.13 INSPECT Statement

Function

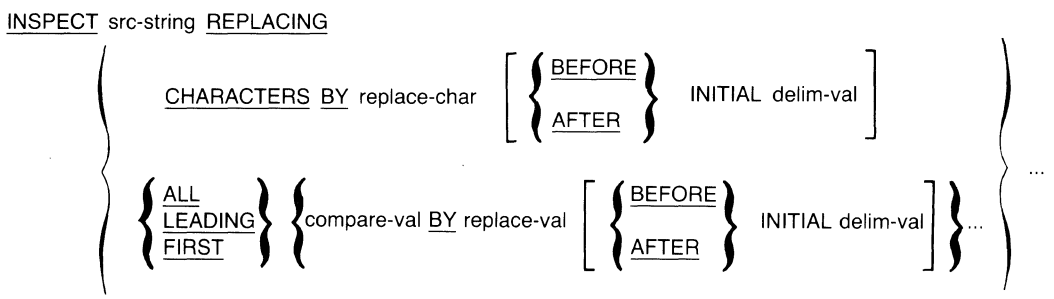
The INSPECT statement counts or replaces occurrences of single characters or groups of characters in a data item.

General Format

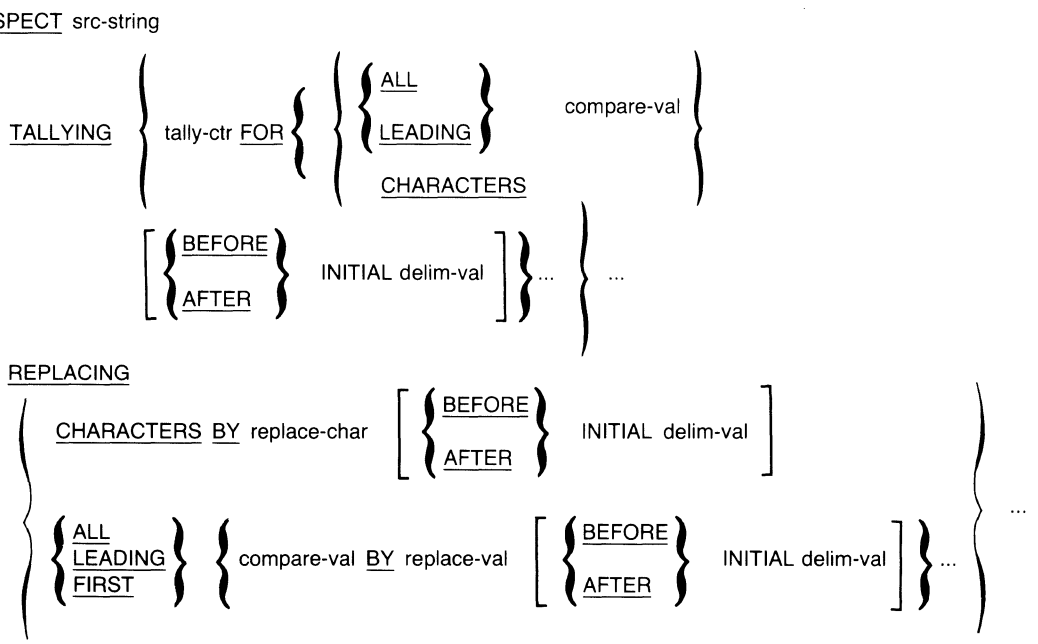
Format 1



Format 2



Format 3



src-string
: is the identifier of a group item or an elementary data item with DISPLAY usage. INSPECT operates on the contents of this data item.

tally-ctr
: is the identifier of an elementary numeric data item.

compare-val
: is the character-string INSPECT uses for comparison. It is a nonnumeric literal (or figurative constant other than ALL literal) or the identifier of an elementary alphabetic, alphanumeric, or numeric data item with DISPLAY usage.

delim-val
: is the character-string that delimits the INSPECT operation. Its content restrictions are the same as those for *compare-val*.

replace-char
: is the one-character item that replaces all characters. Its content restrictions are the same as those for *compare-val*.

replace-val
: is the character-string that replaces occurrences of *compare-val*. Its content restrictions are the same as those for *compare-val*.

Syntax Rules

All Formats

1. If *compare-val*, *delim-val*, or *replace-char* is a figurative constant, the figurative constant represents a one-character data item.
2. An ALL, LEADING, CHARACTERS, or FIRST phrase can have either a BEFORE phrase or an AFTER phrase following it, but not both.

Format 2

3. The sizes of the data referred to by *replace-val* and *compare-val* must be equal. When *replace-val* is a figurative constant, its size equals that of the data referred to by *compare-val*.
4. When there is a CHARACTERS phrase, the size of the data referred to by *delim-val* must be one character.

Format 3

5. A Format 3 INSPECT statement is equivalent to a Format 1 statement followed by a Format 2 statement. Therefore, Syntax Rules 3 and 4 apply to the REPLACING clause of Format 3.

General Rules

All Formats

1. Inspection includes: (a) comparison, (b) setting boundaries for the BEFORE and AFTER phrases, and (c) tallying and/or replacing. Inspection starts at the leftmost character position of the *src-string* data item. It proceeds to the rightmost character position, as described in General Rules 3 through 5.

2. If *src-string*, *compare-val*, *delim-val*, or *replace-val* refers to a data item, the INSPECT statement treats the contents of the item according to the category implied by its data description.

   a. For an alphabetic or alphanumeric item – INSPECT treats the data item as a character-string.

   b. For an alphanumeric edited, numeric edited, or unsigned numeric item – INSPECT treats the data item as though:

      - The data item were redefined as alphanumeric
      - The INSPECT statement were written to refer to the redefined data item. (See General Rule 2a.)

   c. For a signed numeric item – INSPECT treats the data item as though it were moved to an unsigned numeric data item of the same length. It then applies General Rule 2b.

3. During inspection of *src-string*, each matched occurrence of *compare-val* is:

   a. Tallied (Formats 1 and 3)

   b. Replaced by *replace-char* or *replace-val* (Formats 2 and 3)

4. The comparison operation determines which occurrences of *compare-val* are tallied and/or replaced:

   a. INSPECT processes the operands of the TALLYING and REPLACING phrases in the order they appear, from left to right. The first *compare-val* is compared to the same number of contiguous characters, starting with the leftmost character position in *src-string*. *Compare-val* and the compared characters in *src-string* match if they are equal, character for character. Otherwise, they do not match.

   b. If the comparison of the first *compare-val* does not produce a match, the comparison repeats for each successive *compare-val* until either:

      - A match results
      - There is no next *compare-val*

      When there is no next *compare-val*, INSPECT determines the leftmost character position in *src-string* for the next comparison. This position is to the immediate right of the *leftmost* character position for the preceding comparison. The comparison cycle starts again with the first *compare-val*.

   c. Each time there is a match, tallying and/or replacing occur, as described in General Rules 8 through 16. INSPECT determines the leftmost character position in *src-string* for the next comparison. This position is to the immediate right of the rightmost character position that matched in the preceding comparison. The comparison cycle starts again with the first *compare-val*.

   d. Inspection ends when the rightmost character position of *src-string* has either:

      - Participated in a match
      - Served as the leftmost character position

e. When the CHARACTERS phrase is present, INSPECT does not perform any comparison on the contents of *src-string*. The cycle described in General Rules 4a through 4d operates as if:

- Inspection compares a one-character data item to each character in *src-string*
- A match occurs for each comparison

5. The BEFORE phrase determines the final character position in *src-string* that will be used in the comparison operation.

   The position of the first occurrence of *delim-val* in *src-string* is determined before the operation begins. Comparison then occurs on *src-string* only from its leftmost character position to, but not including, the first occurrence of *delim-val*.

   If *delim-val* does not occur in *src-string*, the comparison operation proceeds as if there were no BEFORE phrase.

6. The AFTER phrase determines the first character position in *src-string* that will be used in the comparison operation.

   The position of the first occurrence of *delim-val* in *src-string* is determined before the operation begins. Comparison then occurs on *src-string* beginning with the character position to the immediate right of the rightmost character position of *delim-val*'s first occurrence.

   If *delim-val* is not in *src-string*, no match occurs, and inspection causes no tallying or replacement.

7. Undefined results occur when operands overlap; that is, when sending fields and receiving fields share a part of their storage areas.

Format 1

8. Executing the INSPECT statement does not initialize the value of *tally-ctr*.
9. If the ALL phrase is present, the value of *tally-ctr* is incremented by one for each occurrence of *compare-val* in *src-string*.
10. If the LEADING phrase is present, the value of *tally-ctr* is incremented by one for each contiguous occurrence of *compare-val* in *src-string*. The leftmost occurrence of *compare-val* must be at the position where comparison begins in the first comparison cycle. Otherwise, no tallying occurs.
11. If the CHARACTERS phrase is present, the value of *tally-ctr* is incremented by one for each character matched in *src-string* (see General Rule 4e).

Format 2

12. The adjectives ALL, LEADING, and FIRST apply to succeeding BY phrases until the next adjective appears.
13. If the CHARACTERS phrase is present, each character matched in *src-string* is replaced by *replace-char* (see General Rule 4e).
14. When ALL is present, each occurrence of *compare-val* in *src-string* is replaced by *replace-val*.

15. When LEADING is present, each contiguous occurrence of *compare-val* in *src-string* is replaced by *replace-val*. The leftmost occurrence of *compare-val* must be at the position where comparison begins in the first comparison cycle. Otherwise, no replacement occurs.
16. When FIRST is present, the leftmost occurrence of *compare-val* in *src-string* is replaced by *replace-val*.

Format 3

17. A Format 3 INSPECT statement executes as if there were two successive INSPECT statements with the same *src-string*. Execution proceeds as if:
    - The first statement were a Format 1 statement with TALLYING phrases identical to those in the Format 3 statement
    - The second statement were a Format 2 statement with REPLACING phrases identical to those in the Format 3 statement

    The General Rules for Formats 1 and 2 apply to the corresponding phrases in the Format 3 statement.

Additional References

Section 5.9.15	MOVE Statement
Part II of the COBOL-81 User's Guide for your system	Refer to the chapter on nonnumeric character handling

Examples

In the following examples, the initial values of COUNT1 and COUNT2 are zero.

1. TALLYING phrase with BEFORE option:

```
INSPECT ITEMA TALLYING COUNT1 FOR LEADING "L" BEFORE "A",
        COUNT2 FOR LEADING "A" BEFORE "L".
```

ITEMA	COUNT1	COUNT2
LARGE	1	0
ANALYST	0	1

2. TALLYING phrase and REPLACING LEADING phrase with AFTER option:

```
INSPECT ITEMA TALLYING COUNT1 FOR ALL "L"
        REPLACING LEADING "A" BY "E" AFTER INITIAL "L".
```

ITEMA	COUNT1	ITEMA
CALLAR	2	CALLAR
SALAMI	1	SALEMI
LATTER	1	LETTER

3. REPLACING ALL phrase with BEFORE option:

```
INSPECT ITEMA REPLACING ALL "A" BY "G" BEFORE "X".
```

ITEMA	ITEMA
ARXAX	GRXAX
HANDAX	HGNDGX
HANDAA	HGNDGG

4. TALLYING and REPLACING ALL phrases:

```
INSPECT ITEMA TALLYING COUNT1 FOR CHARACTERS AFTER "J"
       REPLACING ALL "A" BY "B".
```

ITEMA	COUNT1	ITEMA
ADJECTIVE	6	BDJECTIVE
JACK	3	JBCK
JUJMAB	5	JUJMBB

5. REPLACING ALL phrase:

```
INSPECT ITEMA REPLACING ALL "X" BY "Y", "B" BY "Z",
       "W" BY "Q" AFTER "R".
```

ITEMA	ITEMA
RXXBQWY	RYYZQQY
YZACDWBR	YZACDWZR
RAWRXEB	RAQRYEZ

6. REPLACING CHARACTERS phrase:

```
INSPECT ITEMA REPLACING CHARACTERS BY "B" BEFORE "A".
```

ITEMA	ITEMA
12RXZABCD	BBBBBABCD
12RXZBBCD	BBBBBBBBB

7. REPLACING ALL phrase:

```
INSPECT ITEMA REPLACING ALL "A" BY "X" ALL "R" BY "X"
             AFTER "XXL".
```

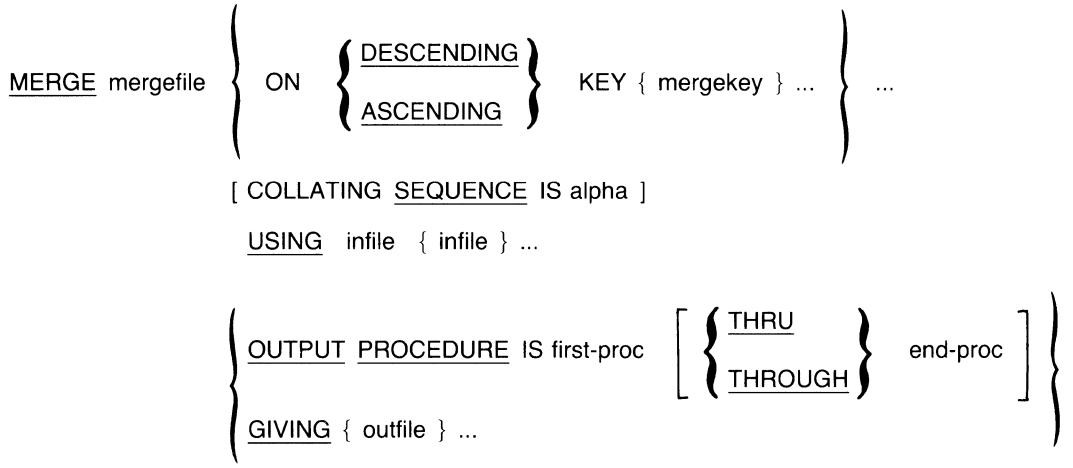
ITEMA	ITEMA
AALRRRA	XXLRRRX
AXXLRRR	XXXLXXX

5.9.14 MERGE Statement

Function

The MERGE statement takes two or more identically sequenced files and combines them according to the key values you specify. During the process, it makes records available, in merged order, to routines in OUTPUT PROCEDURE or to an output file.

General Format



mergefile
: is a file-name described in a sort-merge file description (SD) entry in the Data Division.

mergekey
: is the data-name of a data item in a record associated with mergefile.

alpha
: is an alphabet-name defined in the SPECIAL-NAMES paragraph of the Environment Division.

infile
: is the file-name of an input file. It must be described in a file description (FD) entry in the Data Division.

first-proc
: is the section-name of the output procedure's first section.

end-proc
: is the section-name of the output procedure's last section.

outfile
: is the file-name of an output file. It must be described in a file description (FD) entry in the Data Division.

Syntax Rules

1. MERGE statements can appear anywhere in the Procedure Division except in:
   - DECLARATIVES
   - Sections of a SORT or MERGE statement's INPUT or OUTPUT PROCEDURE
2. If *mergefile* contains variable length records, *infile* records must not be smaller than the smallest record in *mergefile* nor larger than the largest.
3. If *mergefile* contains fixed length records, *infile* records must not be larger than the largest record described for *mergefile*.
4. If *outfile* contains variable length records, *mergefile* records must not be smaller than the smallest record in *outfile* nor larger than the largest.
5. If *outfile* contains fixed length records, *mergefile* records must not be larger than the largest record described for *outfile*.
6. Each *mergekey* must be described in records associated with *mergefile*.
7. *Mergekey* can be qualified.
8. *Mergekey* cannot be a group that contains variable occurrence data items.
9. The description of *mergekey* cannot contain an OCCURS clause or be subordinate to one that does.
10. *Mergefile* can have more than one record description. However, *mergekey* need not be described in more than one of the record descriptions. The character positions referenced by *mergekey* are used as the key for all the file's records.
11. The words THRU and THROUGH are equivalent.
12. If *outfile* is an indexed file, the first *mergekey* must be in the ASCENDING phrase. It must specify the same character positions in its record as the prime record key for outfile.
13. *Mergekey* cannot be larger than 255 characters.
14. The total number of characters in all *mergekeys* for the file cannot be more than 512 characters.
15. Each MERGE statement can specify no more than 16 *mergekeys*.
16. Neither *infile* nor *outfile* can describe an indexed file in random access mode.

General Rules

1. The MERGE statement merges all records in the *infile* files.
2. If *mergefile* contains fixed length records, any shorter *infile* records are space filled on the right after the last character. Space filling occurs before the *infile* record is released to *mergefile*.

3. The leftmost *mergekey* is the major key, and the next *mergekey* is the next most significant key. The significance of *mergekey* data items is not affected by how they are divided into KEY phrases. Only left-to-right order determines significance.
4. The ASCENDING phrase causes the merged sequence to be from the lowest *mergekey* value to the highest.
5. The DESCENDING phrase causes the merged sequence to be from the highest *mergekey* value to the lowest.
6. Merge sequence follows the rules for relation condition comparisons.
7. When the contents of all key data items of one record equals the contents of the corresponding key data items in another record, the order of return from the merge:
   - Follows the order of the associated input files in the MERGE statement
   - Causes all records with equal key values from one input file to be returned before any are returned from another.
8. The MERGE statement determines the comparison collating sequence for nonnumeric *mergekey* items when it begins execution. If there is a COLLATING SEQUENCE phrase in the MERGE statement, MERGE uses that sequence. Otherwise, it uses the collating sequence that was established for the program as a whole in the PROGRAM COLLATING SEQUENCE clause of the OBJECT-COMPUTER paragraph. If you do not specify the collating sequence in either the MERGE statement or the OBJECT-COMPUTER paragraph, the program uses the NATIVE collating sequence.
9. The results of the merge are undefined unless the records in the *infile* files are ordered as described in the MERGE statement's ASCENDING or DESCENDING KEY clause.
10. The MERGE statement transfers all records in *infile* to *mergefile*. When the MERGE statement executes, *infile* must not be open.
11. For each *infile*, the MERGE statement:
    - Begins file processing as if the program had executed an OPEN statement with the INPUT phrase
    - Gets the logical records and releases them to the merge operation. MERGE obtains each record as if the program had executed a READ statement with the NEXT and AT END phrases.
    - Terminates file processing as if the program had executed a CLOSE statement with no optional phrases.

    These implicit OPEN, READ, and CLOSE operations cause associated USE procedures to execute if an exception condition occurs.
12. OUTPUT PROCEDURE consists of one or more sections that are:
    - Contiguous in the source program
    - Not a part of any other procedure

13. When the MERGE statement enters the OUTPUT PROCEDURE range, it is ready to select the next record in merged order. Statements in the OUTPUT PROCEDURE range must execute at least one RETURN statement to make records available for processing.
14. The program must not pass control to any statements in the OUTPUT PROCEDURE range except during execution of a related MERGE statement.
15. The OUTPUT PROCEDURE range cannot include SORT or MERGE statements. It must not explicitly transfer control outside the range. However, its statements can cause implied control transfers to DECLARATIVES.
16. The remainder of the Procedure Division must not transfer control to statements in the OUTPUT PROCEDURE range.
17. If OUTPUT PROCEDURE is used, control passes to its sections during execution of the MERGE statement. When control passes to the last statement in the OUTPUT PROCEDURE range, the MERGE statement ends. Control then transfers to the next executable statement after the MERGE statement.
18. During execution of statements in the OUTPUT PROCEDURE range — or any USE AFTER EXCEPTION procedure implicitly invoked during the MERGE statement — no statement outside the range can manipulate the files or record areas associated with *infile* or *outfile*.
19. If there is a GIVING phrase, the MERGE statement writes all merged records to each *outfile*. This transfer is an implied MERGE statement OUTPUT PROCEDURE. Therefore, when the MERGE statement executes, *outfile* must not be open.
20. The MERGE statement begins *outfile* processing as if the program had executed an OPEN statement with the OUTPUT phrase.
21. The MERGE statement gets the merged logical records and writes them to each *outfile*. MERGE writes each record as if the program had executed a WRITE statement with no optional phrases.

    For relative files, the value of the relative key data item is 1 for the first returned record, 2 for the second, and so on. When the MERGE statement ends, the value of the relative key data item indicates the number of *outfile* records.
22. The MERGE statement terminates *outfile* processing as if the program had executed a CLOSE statement with no optional phrases.
23. These implicit OPEN, WRITE, and CLOSE operations cause associated USE procedures to execute if an exception condition occurs. If the MERGE statement tries to write beyond the boundaries of *outfile*, the applicable USE AFTER EXCEPTION procedure executes. If control returns from the USE procedure, or if there is no USE procedure, *outfile* processing terminates as if the program had executed a CLOSE statement with no optional phrases.
24. If *outfile* contains fixed length records, any shorter *mergefile* records are space filled on the right after the last character. Space filling occurs before the *mergefile* record is released to *outfile*.

MERGE
Continued

Additional References

Section 3.1.2	OBJECT-COMPUTER Paragraph
Section 3.1.3	SPECIAL-NAMES Paragraph
Section 3.2.2	I-O-CONTROL Paragraph
Section 5.9.31	USE Statement
Part IV of the COBOL-81 User's Guide for your system	Refer to the chapter on sorting records and merging files

5.9.15 MOVE Statement

Function

The MOVE statement transfers data to one or more data areas. The editing rules control data transfer.

General Format

Format 1

Format 2

src-item
: is an identifier that represents the sending area.

lit
: is a literal that represents the sending area.

dest-item
: is an identifier that represents the receiving area.

Syntax Rules

1. CORR is an abbreviation for CORRESPONDING.
2. In the CORRESPONDING phrase, both *src-item* and *dest-item* must be group items, and there can be only one *dest-item*.
3. If any *dest-item* is numeric or numeric edited, *lit* cannot be any of these: HIGH-VALUE(S), LOW-VALUE(S), SPACE(S), or QUOTE(S).
4. No operand can be an index data item.

General Rules

1. When the CORRESPONDING phrase is present, selected items in *src-item* are moved to selected items in *dest-item*. The rules for the CORRESPONDING option control these moves. The results are the same as if the MOVE statement were replaced by separate MOVE statements for each pair of corresponding items in *src-item* and *dest-item*.
2. The MOVE statement moves the sending area to the first *dest-item*, then to each additional *dest-item*, in the same order in which they appear in the statement.
3. Subscript or index evaluation for *src-item* occurs once, immediately before the move to the first *dest-item*.

4. Subscript or index evaluation for a *dest-item* occurs immediately before the move to that item.
5. The length of *src-item* is evaluated once, immediately before the move to the first *dest-item*.
6. The length of *dest-item* is assumed to be the maximum in its data description.
7. The result of the first of the following MOVE statements is equivalent to the three that follow. The word *temp* represents an intermediate result item supplied by the compiler.

```
MOVE ITEMA (ITEMB) TO ITEMB, ITEMC (ITEMB).

MOVE ITEMA (ITEMB) TO temp.
MOVE temp TO ITEMB.
MOVE temp TO ITEMC (ITEMB).
```

8. Undefined results occur when a sending item and a receiving item overlap; that is, when they share a part of their storage areas.

Elementary Moves

9. A move is elementary when *dest-item* is an elementary item, and the sending area is either an elementary data item or a literal.
   a. An elementary item belongs to one of these categories, depending on its PICTURE clause:
      - Numeric
      - Alphabetic
      - Alphanumeric
      - Numeric edited
      - Alphanumeric edited
   b. Numeric literals are numeric. Nonnumeric literals are alphanumeric.
   c. The figurative constant ZERO is numeric when moved to a numeric or numeric edited item. Otherwise, it is alphanumeric.
   d. The figurative constant SPACE is alphabetic.
   e. All other figurative constants are alphanumeric.
10. These rules apply to elementary moves between categories:
    a. The figurative constant SPACE or a numeric edited, alphanumeric edited, or alphabetic data item cannot be moved to a numeric or numeric edited data item.
    b. A numeric literal, the figurative constant ZERO, or a numeric or numeric edited data item cannot be moved to an alphabetic data item.
    c. A noninteger numeric literal or data item cannot be moved to an alphanumeric or alphanumeric edited data item.
    d. All other elementary moves are valid.

Editing and Data Conversion

11. Editing, or other required internal data conversions occur during elementary moves. They are controlled by the description of *dest-item*.

12. When *dest-item* is alphanumeric or alphanumeric edited, alignment and space-filling occur according to the Standard Alignment Rules.

    If *lit* or *src-item* is signed numeric, the operational sign is not moved. If the operational sign occupies a separate character position:

    a. The sign character is not moved

    b. The size of *lit* or *src-item* is considered to be one less than its actual size (in terms of Standard Data Format characters)

13. When *dest-item* is numeric or numeric edited, decimal point alignment and zero-filling occur according to the Standard Alignment Rules.

    a. When *dest-item* is a signed numeric item, the sign from *lit* or *src-item* is placed in it. If the sending item is unsigned, a positive sign is placed in *dest-item*.

    b. When *dest-item* is an unsigned numeric item, the absolute value of *lit* or *src-item* is moved.

    c. When *lit* or *src-item* is alphanumeric, the move occurs as if the item were described as an unsigned numeric integer.

14. When *dest-item* is alphabetic, justification and space-filling occur according to the Standard Alignment Rules.

Group Moves

15. Any nonelementary move is considered a group move. A group move occurs as if it were an alphanumeric-to-alphanumeric elementary move. However, there is no internal data conversion. The move is not affected by individual elementary or group items in either *src-item* or *dest-item*, except as noted in the General Rules for the OCCURS clause.

16. Table 5-9 summarizes the valid types of MOVE statements.

Table 5-9: Valid MOVE Statements

Category of Sending Data Item (lit or src-item)	Category of Receiving Data Item (dest-item)		
	Alphabetic	Alphanumeric Edited Alphanumeric	Numeric Integer Numeric Non-Integer Numeric Edited
Alphabetic	Yes	Yes	No
Alphanumeric	Yes	Yes	Yes
Alphanumeric Edited	Yes	Yes	No
Numeric Integer	No	Yes	Yes
Numeric Non-Integer	No	No	Yes
Numeric Edited	No	Yes	No

MOVE
Continued

Additional References

Section 4.1.2.2	Standard Alignment Rules
Section 4.2.13	OCCURS Clause
Section 4.2.14	PICTURE Clause
Section 4.2.18	SIGN Clause
Section 5.6.5	CORRESPONDING Option

Examples

The following examples show the result of executing the statement:

```
MOVE ITEMA TO ITEMB.
```

An "s" indicates a space character.

1. Numeric edited receiving item:

   (The PICTURE of ITEMA is S9999V99.)

	ITEMA Value	ITEMB PICTURE	ITEMB Contents
a.	+0023.00	ZZZZ.99	ss23.00
b.	−0036.93	++++.99	s–36.93
c.	+1234.56	Z,ZZZ.99	1,234.56
d.	+1234.56	Z,ZZZ.99−	1,234.56s
e.	+1234.56	Z,ZZZ.99+	1,234.56+
f.	−1234.56	$,$$$,$$$.99DB	sss$1,234.56DB
g.	−1234.56	$,$$$.99−	s$234.56−
h.	+0001.25	$,$$$.99	sss$1.25
i.	−0001.25	$,$$$.99	sss$1.25
j.	+0000.00	$,$$9.99	sss$0.00
k.	+0000.00	$,$$$.$$	ssssssss

2. Alphanumeric receiving item:

   (The PICTURE of ITEMA is X(4).)

	ITEMA Value	ITEMB Description	ITEMB Contents
a.	ABCD	PIC X(4)	ABCD
b.	ABCD	PIC X(6)	ABCDss
c.	ABCD	PIC X(6) JUST	ssABCD
d.	ABCs	PIC X(6) JUST	ssABCs
e.	ABCD	PIC XXX	ABC
f.	ABCD	PIC XX JUST	CD

3. Alphanumeric edited receiving item:

   (The PICTURE of ITEMA is X(7).)

	ITEMA Value	ITEMB Description	ITEMB Contents
a.	063080s	XX/99/XX	06/30/80
b.	30JUN80	99BAAAB99	30sJUNs80
c.	6374823	XXXBXXX/XX/X	637s482/3s/s
d.	123456s	0XB0XB0XB0XB	01s02s03s04s

5.9.16 MULTIPLY Statement

Function

The MULTIPLY statement multiplies two numeric operands and stores the result.

General Format

Format 1

```
MULTIPLY srcnum BY { rsult [ ROUNDED ] } ... [ ON SIZE ERROR stment ]
```

Format 2

```
MULTIPLY srcnum BY srcnum GIVING { rsult [ ROUNDED ] } ... [ ON SIZE ERROR stment ]
```

srcnum
is a numeric literal or the identifier of an elementary numeric item.

rsult
is the identifier of an elementary numeric item. However, in Format 2, *rsult* can be an elementary numeric edited item. It is the resultant identifier.

stment
is an imperative statement.

General Rules

1. The data descriptions of the operands need not be the same. Conversion and decimal point alignment will occur, as needed, throughout the calculation.
2. The maximum size of each operand is 18 digits.
3. In Format 1, the value of *srcnum* is multiplied by the value of the first *rsult*. The product replaces the current value of the first *rsult*. The process repeats for each later occurrence of *rsult*.
4. In Format 2, the values of the operands before the word GIVING are multiplied together. The product replaces the current value of each *rsult*.

Additional References

Section 5.1.4	Scope of Statements
Section 5.6.1	Arithmetic Operations
Section 5.6.3	ROUNDED Option
Section 5.6.4	ON SIZE ERROR Option
Section 5.6.6	Overlapping Operands and Incompatible Data
Section 5.6.2	Multiple Receiving Fields in Arithmetic Statements

MULTIPLY
Continued

Examples

The examples assume these data descriptions and beginning values:

```
                                                          Initial
                                                          Value

03  ITEMA  PIC S99 VALUE 4.                                4
03  ITEMB  PIC S99 VALUE -35.                             -35
03  ITEMC  PIC S99 VALUE 10.                              10
03  ITEMD  PIC S99 VALUE 5.                                5
```

Results

1. Without GIVING phrase:

```
MULTIPLY 2 BY ITEMB.
```

ITEMB = –70

2. SIZE ERROR phrase:

   (When the SIZE ERROR condition occurs, the values of resultant identifiers do not change.)

```
MULTIPLY 3 BY ITEMB
  ON SIZE ERROR
    MOVE 0 TO ITEMC.
```

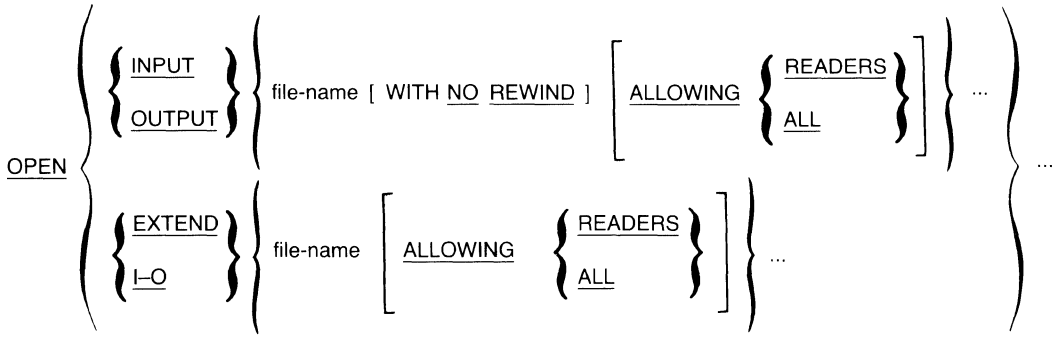
ITEMA = 4
ITEMB = –35
ITEMC = 0
ITEMD = 5

5.9.17 OPEN Statement

Function

The OPEN statement makes the file available to the program, begins the processing of a file, and specifies file sharing.

General Format



file-name
is the name of a file described in the Data Division. It cannot be the name of a sort or merge file.

Syntax Rules

1. The NO REWIND phrase can be used only for files with sequential organization.
2. The I-O phrase can be used only for mass storage files. In other words, the I-O phrase applies only to files that can be accessed randomly, as well as sequentially.
3. The EXTEND phrase can be used only for files with sequential organization.

General Rules

1. Successful OPEN statement execution:
   - Makes the file available to the program
   - Puts the file in an open mode
   - Associates the file with *file-name* through the file connector
   - Makes the file's record area available to the program

2. An executable image can open a *file-name* more than once with the INPUT, OUTPUT, I-O, and EXTEND phrases. After the first OPEN statement, each later OPEN for the same *file-name* must follow the execution of a CLOSE statement for the *file-name*. However, the CLOSE statement must not have a REEL, UNIT, or LOCK phrase.
3. The OPEN statement does not get or release the first data record.
4. For an OPEN statement with the INPUT, I-O, or EXTEND phrases, *file-name's* file description entry must be equivalent to that used when the file was created.
5. The NO REWIND phrase has no effect if the file is not on magnetic tape.
6. If the file is on magnetic tape, and:
   - There is neither an EXTEND nor NO REWIND phrase, then OPEN statement execution positions the file at its beginning.
   - There is a NO REWIND phrase, then the OPEN statement does not reposition the file. The file must already be positioned at its beginning before the OPEN statement executes.
7. Successful execution of an OPEN statement sets the Current Volume Pointer to:
   - The first or only reel/unit for an available input or input-output file
   - The reel/unit containing the last logical record for an extend file
   - The new reel/unit for an unavailable output, input-output, or extend file

   Multivolume tape files are not supported by all DIGITAL operating systems. Refer to Technical Notes for more information.
8. If there is more than one *file-name* in the OPEN statement, execution is the same as if there were separate OPEN statements, one for each *file-name*.
9. A file's maximum record size is established at the time the file is created and must not subsequently be changed.
10. A file is available if it is both:
    - Physically present
    - Recognized by Record Management Services (RMS-11)

    Table 5-10 shows the result of opening available and unavailable sequential, relative, and indexed files.

Table 5-10: Opening Available and Unavailable Sequential, Relative and Indexed Files

Open Mode	File is Available	File is Unavailable
INPUT	Normal open	Error
INPUT (Optional File)	Normal open	Normal open The first read causes the at end condition or invalid key condition
I-O (RANDOM or DYNAMIC access)	Normal open	The open creates the file
I-O (SEQUENTIAL access)	Normal open	Error
OUTPUT	*	The open creates the file
EXTEND	Normal open	The open creates the file

* On an RSX-11M/M-PLUS system, opening an existing file for output creates a new version of the file.

The RSTS/E operating system does not support multiple versions of the same file. When a file already exists, and it is opened for output on a RSTS/E system, the OPEN statement creates a new file that supersedes the existing one.

11. When a file is not in an open mode, no statement that references the file can execute either implicitly or explicitly, except for:
    - A MERGE statement
    - An OPEN statement
    - A SORT statement with the USING or GIVING phrase
12. An OPEN statement for a file must successfully execute before any allowable input-output statement executes for the file. Table 5-11 shows allowable input-output statements by file organization, access mode, and open mode for sequential, relative, and indexed files.

Table 5-11: Allowable Input-Output Statements for Sequential, Relative, and Indexed Files

File Organization	Access Mode	Statement	Open Mode			
			INPUT	OUTPUT	I-O	EXTEND
SEQUENTIAL	SEQUENTIAL	READ	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		WRITE	No	Yes	No	Yes
RELATIVE	SEQUENTIAL	DELETE	No	No	Yes	No
		READ	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		START	Yes	No	Yes	No
		WRITE	No	Yes	No	No
	RANDOM	DELETE	No	No	Yes	No
		READ	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		WRITE	No	Yes	Yes	No
	DYNAMIC	DELETE	No	No	Yes	No
		READ	Yes	No	Yes	No
		READ NEXT	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		START	Yes	No	Yes	No
		WRITE	No	Yes	Yes	No
INDEXED	SEQUENTIAL	DELETE	No	No	Yes	No
		READ	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		START	Yes	No	Yes	No
		WRITE	No	Yes	No	No
	RANDOM	DELETE	No	No	Yes	No
		READ	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		WRITE	No	Yes	Yes	No
	DYNAMIC	DELETE	No	No	Yes	No
		READ	Yes	No	Yes	No
		READ NEXT	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		START	Yes	No	Yes	No
		WRITE	No	Yes	Yes	No

13. If the file opened with the INPUT phrase is an optional file that is not present, the OPEN statement sets the Next Record Pointer to indicate the at end condition on the next READ. The Status Key is set to 05.

14. For indexed files opened with the INPUT or I-O phrase, the OPEN statement sets the Next Record Pointer to the first record existing in the file when it is opened. The prime record key is established as the Key of Reference. It determines the first record to be accessed. If the file has no records, the OPEN statement sets the Next Record Pointer to cause an at end condition on the next sequential READ statement for the file.
15. An OPEN statement with the EXTEND phrase positions the file immediately after its last logical record (the last record written in the file).
16. The I-O phrase opens a mass storage file for both input and output operations.
17. Successful execution of an OPEN statement with the EXTEND or I-O phrase creates the file if it is not available. Successful execution of an OPEN statement with the OUTPUT phrase also creates the file. In each case, the created file contains no data records.
18. The ALLOWING phrase specifies a file sharing option for the file.
19. The READERS phrase allows access only by the READ statement.
20. The ALL phrase specifies unlimited file sharing. In other words, any other program can execute READ, START, DELETE, WRITE, REWRITE, OPEN, and CLOSE statements on the file during program execution.
21. If there is no ALLOWING phrase, the default is ALLOWING READERS.

Technical Notes

1. OPEN statement execution can result in these FILE STATUS data item values:

FILE STATUS	Meaning
00	Successful
05	Optional file not present
91	File is locked by another program
94	File is already open, or closed with lock
95	No file space on device
96	Same Area busy
97	File not found
30	All other permanent errors

2. The RSTS/E operating system does not support multivolume tape files. Therefore, sections of the rules that refer to reels (or units) of a file do not apply to COBOL-81 programs that execute on a RSTS/E system.

Additional References

Section 5.9.4	CLOSE Statement
Section 5.9.31	USE Statement
Part IV of the COBOL-81 User's Guide for your system	Processing Files and Records

PERFORM

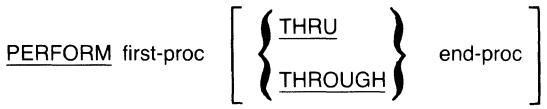
5.9.18 PERFORM Statement

Function

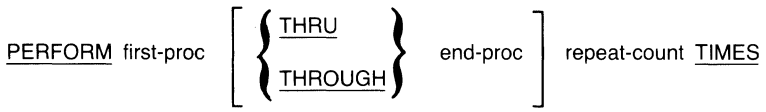
The PERFORM statement executes one or more procedures. It returns control to the end of the PERFORM statement when procedure execution ends.

General Format

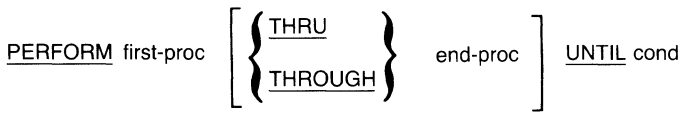
Format 1



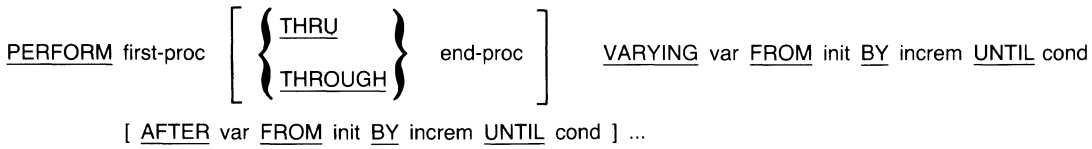
Format 2



Format 3



Format 4



first-proc
: is a procedure-name that identifies a paragraph or section in the Procedure Division. The set of statements in *first-proc* are the first (or only) set of statements in the PERFORM range.

end-proc
: is a procedure-name that identifies a paragraph or section in the Procedure Division. The set of statements in *end-proc* are the last set of statements in the PERFORM range.

repeat-count
: is a numeric integer literal or the identifier of a numeric integer elementary item. It controls how many times the statement set (or sets) executes.

cond
: can be any conditional expression.

var
: is an index-name or the identifier of a numeric elementary data item. Its value is changed by *increm* each time all statements in the PERFORM range execute.

init
: is a numeric literal, index-name, or the identifier of a numeric elementary data item. It specifies the value of *var* before any statement in the PERFORM range executes.

increm
: is a nonzero numeric literal or the identifier of a numeric elementary data item. It systematically changes the value of *var* each time the program executes all statements in the PERFORM range.

Syntax Rules

All Formats

1. If either *first-proc* or *end-proc* is in the Declaratives part of the Procedure Division, both must be in the same section of DECLARATIVES.
2. The words THRU and THROUGH are equivalent and interchangeable.

Format 4

3. If *var* is an index-name:
   - *Init* must be an integer data item or a positive integer literal
   - *Increm* must be an integer data item or a nonzero integer literal
4. If *init* is an index-name:
   - *Var* must be an integer data item
   - *Increm* must be an integer data item or a positive integer literal

General Rules

All Formats

1. When the PERFORM statement executes, control transfers to the first statement of *first-proc*. However, control might not transfer to the statement set, depending on condition evaluation, when a Format 2, 3, or 4 PERFORM statement begins.

   Unlike the GO TO statement, control transfers back to the PERFORM statement after the set(s) of statements in the PERFORM range executes. When control transfers, an implicit control transfer to the end of the PERFORM statement is established according to the following rules:

   - If *first-proc* is a paragraph-name and there is no *end-proc*, the return is after the last statement of *first-proc*.
   - If *first-proc* is a section-name and there is no *end-proc*, the return is after the last statement of the last paragraph of *first-proc*.
   - If *end-proc* is a paragraph-name, the return is after the last statement of *end-proc*.
   - If *end-proc* is a section-name, the return is after the last statement of the last paragraph of *end-proc*.

2. *First-proc* and *end-proc* need not be related. However, they are the beginning and end of a consecutive sequence of operations.

   GO TO and PERFORM statements can occur between *first-proc* and *end-proc*. If there is more than one logical path to the return point, *end-proc* can be a paragraph, consisting of the EXIT statement, to which all these paths must lead.

3. A statement other than PERFORM can transfer control to statements inside the PERFORM range. When this happens, control does not transfer back to the end of the PERFORM statement after execution of statements in the range.

4. The range of a PERFORM statement consists of all statements executed as a result of executing the PERFORM. It continues through execution of the implicit control transfer to the end of the PERFORM statement.

   The range includes all statements:

   - Executed as the result of a control transfer by CALL, EXIT (without the PROGRAM phrase), GO TO and PERFORM statements in the PERFORM statement range
   - In Declarative procedures executed as a result of the execution of statements in the PERFORM statement range The statements in the PERFORM range need not be in consecutive order in the source program.

5. Statements executed as the result of a control transfer caused by an EXIT PROGRAM statement in the PERFORM range are not part of the range.

6. A PERFORM statement range can contain another PERFORM statement. In that case, the included PERFORM statement's sequence of procedures must be either totally included in, or excluded from, the logical sequence of the first PERFORM statement.

   Thus:

   - An active PERFORM statement whose execution point is in the range of another active PERFORM statement must not allow control to pass to the exit of the other active PERFORM.
   - Two or more active PERFORM statements cannot have a common exit.

   Figure 5-3 shows valid and invalid nested PERFORM statements.

Format 1

7. Format 1 is the basic PERFORM statement. The statement set(s) in the PERFORM range executes once. Control then passes to the end of the PERFORM statement.

Format 2

8. The statement set(s) executes the number of times specified by *repeat-count*. If the value of *repeat-count* is zero or negative when the PERFORM statement executes, control passes to the end of the PERFORM statement.

   During PERFORM statement execution, changing the value of *repeat-count* does not change the number of times the statement set(s) executes.

Formats 3 and 4

9. The PERFORM statement tests to determine if *cond* is true before procedure execution.

Format 3

10. The statement set(s) executes until *cond* is true. Control then transfers to the end of the PERFORM statement.
11. If *cond* is true when the PERFORM statement executes, there is no transfer to *first-proc*; control passes to the end of the PERFORM statement.

Format 4

12. The Format 4 PERFORM statement systematically changes the value of *var* during its execution.
13. If *var* is an index-name, its value, when PERFORM statement execution begins, must equal the occurrence number of an element in its table.
14. If *init* is an index-name, *var* must equal the occurrence number of an element in the table associated with *init*. As the value of the *var* index changes during PERFORM execution, it cannot contain a value outside the range of its table. However, when the PERFORM statement ends, the *var* index can contain a value outside the range of the table by one increment or decrement value.
15. *Increm* must not be zero.
16. *Init* must be positive when *var* is an index-name and *init* is an identifier.
17. If one *var* is varied (See Figure 5-1):
    - *Var* is set to the value of *init* when PERFORM statement execution begins.
    - If *cond* is false, the statement set executes once. The value of *var* changes by the increment or decrement value (*increm*), and *cond* is evaluated again. This cycle continues until *cond* is true. Control then transfers to the end of the PERFORM statement.
    - If *cond* is true when the PERFORM statement begins executing, control transfers to the end of the PERFORM statement.
18. If the PERFORM statement has two *vars* (See Figure 5-2):
    - The first and second *vars* are set to the value of the first and second *init* when PERFORM statement execution begins.
    - If the first *cond* is true, control transfers to the end of the PERFORM statement.
    - If the second *cond* is false, the statement set executes once. The second *var* changes by the value of *increm*, and the second *cond* is evaluated again. This cycle continues until the second *cond* is true.
    - When the second *cond* is true, the second *var* is set to the value of the second *init*, and the value of the first *var* changes by the value of the first *increm*. The first *cond* is reevaluated. The PERFORM statement ends if the first *cond* is true. Otherwise, the cycle continues until *cond* is true.

19. At the end of a PERFORM statement:
    - The value of the first *var* exceeds the last-used value by one increment or decrement value. However, if *cond* was true when the PERFORM statement began, *var* contains the current value of *init*.
    - The value of each other *var* equals the current value of its associated *init*.
20. During execution of the set(s) of statements in the range, any change to *var*, *increm*, or *init* affects PERFORM statement operation.
21. When there is more than one *var*, *var* in each AFTER phrase goes through a complete cycle each time *var* in the preceding AFTER (or VARYING) phrase is varied.

Figure 5-1: PERFORM ... VARYING with One Condition

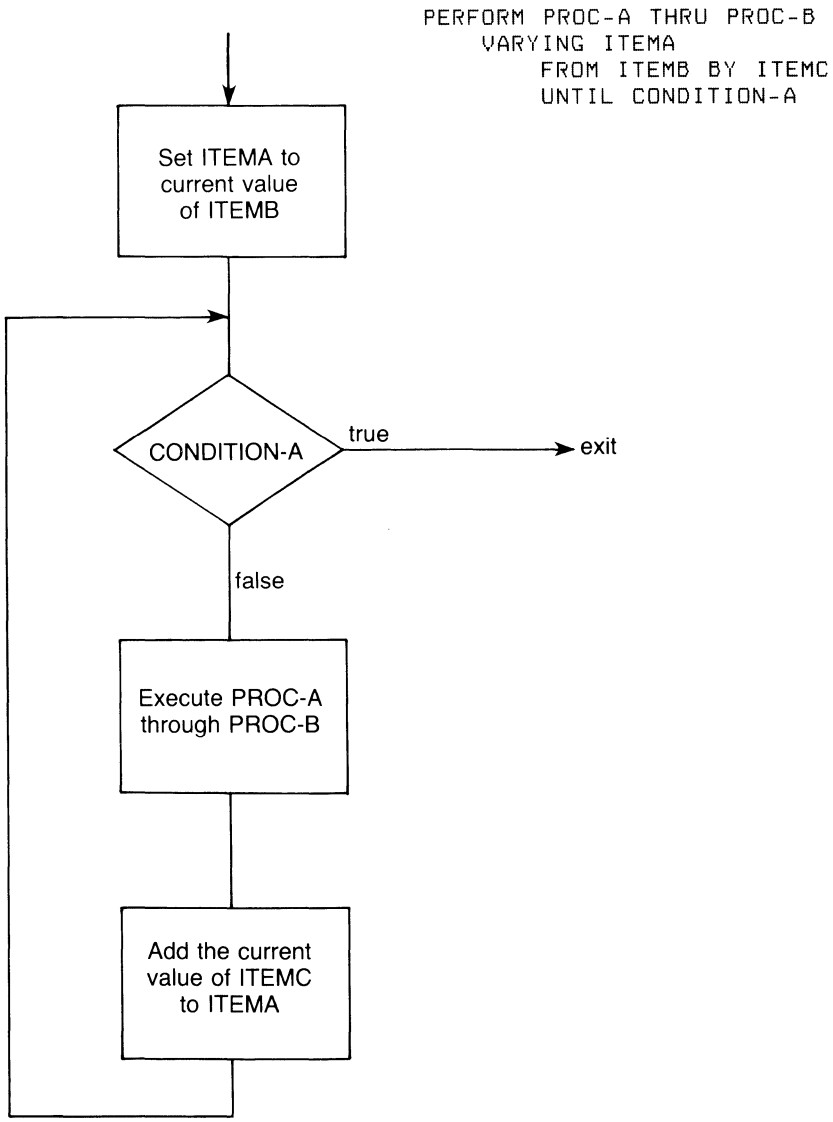
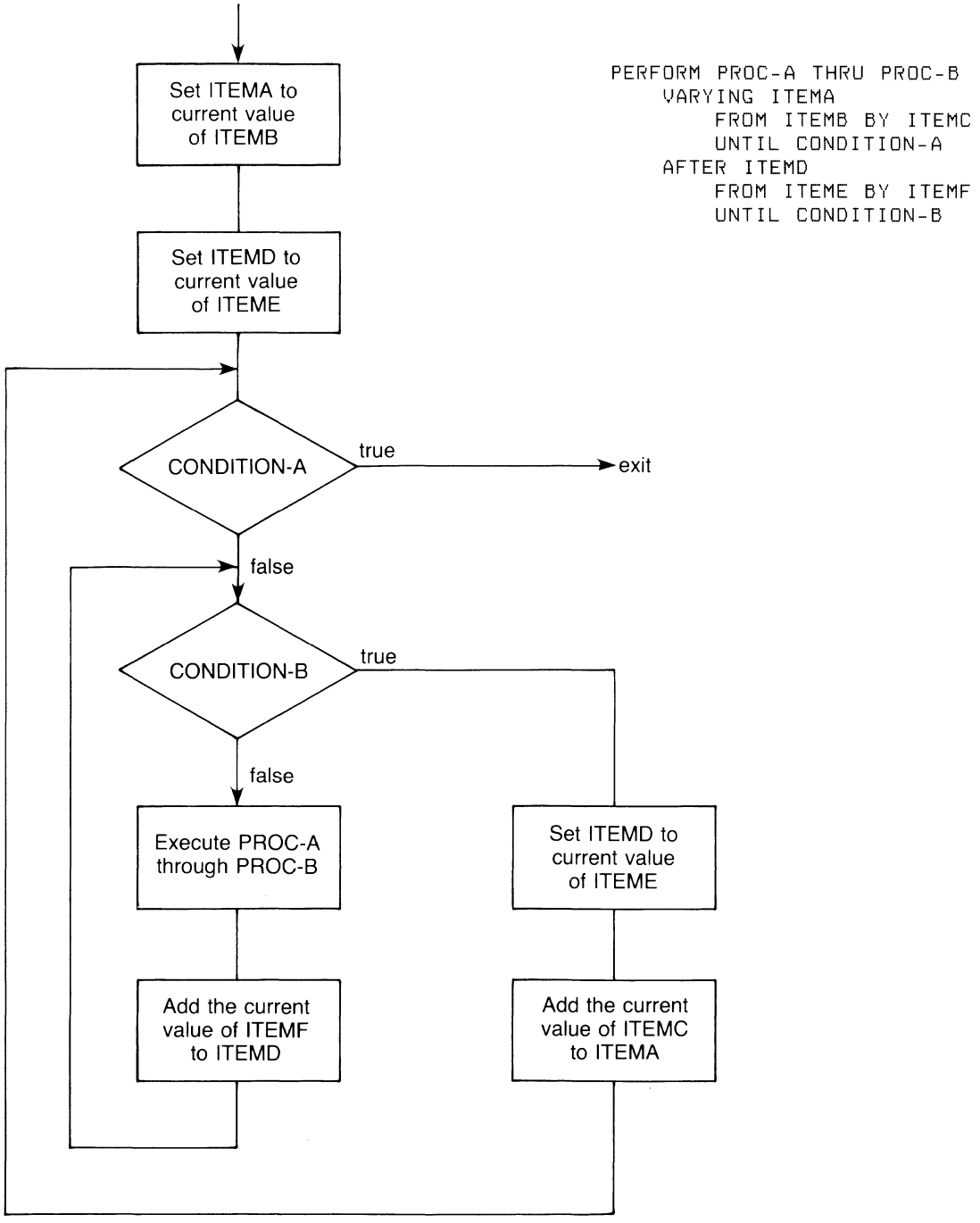


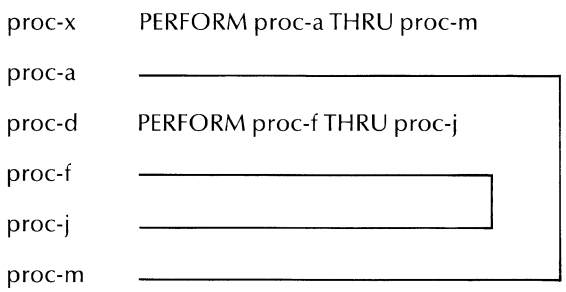
Figure 5-2: PERFORM ... VARYING with Two Conditions



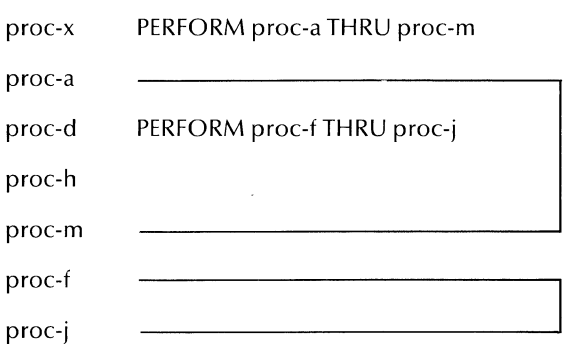

```
PERFORM PROC-A THRU PROC-B
    VARYING ITEMA
        FROM ITEMB BY ITEMC
        UNTIL CONDITION-A
    AFTER ITEMD
        FROM ITEME BY ITEMF
        UNTIL CONDITION-B
```

Figure 5-3: Valid and Invalid Nested PERFORM Statements

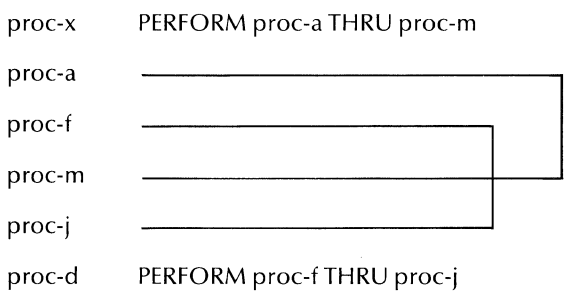
This combination is valid. The range of the second PERFORM is totally contained in the range of the first.



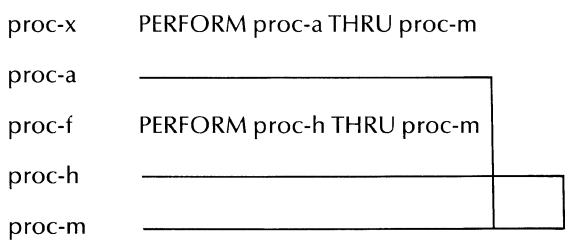
This combination is valid. The range of the second PERFORM is totally outside the range of the first PERFORM.



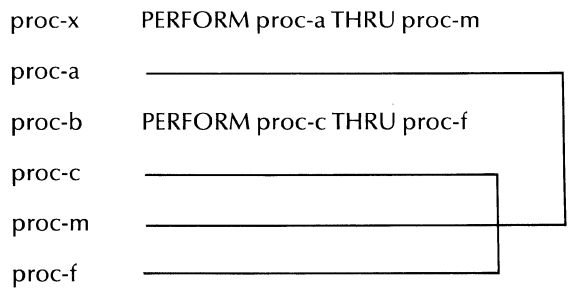
This combination is valid. The first PERFORM is no longer active when the second PERFORM executes.



This combination is invalid. The second active PERFORM has the same exit as the first active PERFORM.



This combination is valid if proc-m is not in the *logical* range of the second PERFORM. For example, proc-c may transfer control to proc-f without executing proc-m. Otherwise, this combination is invalid.



Additional References

Section 5.1.4 Scope of Statements
Section 5.5 Conditional Expressions

Examples

In the examples' results, *s* represents a space. The examples assume these Data Division and Procedure Division entries:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  ITEMA  VALUE "ABCDEFGHIJ".
    03  CHARA OCCURS 10 TIMES PIC X.
01  ITEMB  VALUE SPACES.
    03  CHARB OCCURS 10 TIMES PIC X.
01  ITEMC   PIC 99  VALUE 1.
01  ITEMD   PIC 99  VALUE 7.
01  ITEME   PIC 99  VALUE 4.
01  ITEMF VALUE SPACES.
    03  ITEMG OCCURS 4 TIMES.
        05  ITEMH OCCURS 5 TIMES.
            07  ITEMI PIC 99.

PROCEDURE DIVISION.

    ...

PROC-A.
    MOVE CHARA (ITEMC) TO CHARB (ITEMC).
PROC-B.
    MOVE CHARA (ITEMC) TO CHARB (10).
PROC-C.
    ADD 2 TO ITEMC.
PROC-D.
    MULTIPLY ITEMC BY ITEMD
                GIVING ITEMI (ITEMC, ITEMD).
```

1. Performing one procedure (Format 1):

```
PERFORM PROC-A.
```

Result
ITEMB = Asssssssss

2. Performing a range of procedures (Format 1):

```
PERFORM PROC-A THRU PROC-B.
```

Result
ITEMB = AsssssssssA

3. Performing a range of procedures (Format 2):

```
PERFORM PROC-A THRU PROC-C
  3 TIMES.
```

Result
ITEMB = AsCsEssssE
ITEMC = 07

4. Performing a range of procedures (Format 4):

```
PERFORM PROC-A THRU PROC-B
  VARYING ITEMC FROM 1 BY 1
  UNTIL ITEMC > 5.
```

Result
ITEMB = ABCDEssssE
ITEMC = 06

5. Performing a range of procedures varying a data item by a negative amount (Format 4):

```
PERFORM PROC-A THRU PROC-B
  VARYING ITEMC FROM ITEMD BY -1
  UNTIL ITEMC < ITEME.
```

Result
ITEMB = sssDEFGssD
ITEMC = 03

6. Varying two data items (Format 4):

```
PERFORM PROC-D
  VARYING ITEMC FROM 1 BY 1 UNTIL ITEMC > 4
  AFTER ITEMD FROM 1 BY 1 UNTIL ITEMD > 5.
```

Result
ITEMG (1) = 01s02s03s04s05s
ITEMG (2) = 02s04s06s08s10s
ITEMG (3) = 03s06s09s12s15s
ITEMG (4) = 04s08s12s16s20s

5.9.19 READ Statement

Function

For sequential access files, the READ statement makes the next logical record available. For random access files, READ makes a specified record available.

General Format

Format 1

```
READ file-name [ NEXT ] RECORD [ INTO dest-item ]  [ AT END stment ]
```

Format 2

```
READ file-name RECORD [ INTO dest-item ] [ KEY IS key-name ] [ INVALID KEY stment ]
```

file-name
: is the name of a file described in the Data Division. It cannot be a sort or merge file.

dest-item
: is the identifier of a data item that receives the record accessed by the READ statement.

stment
: is an imperative statement executed for an at end or invalid key condition.

key-name
: is the data-name of a data item specified as a record key for *file-name*. It can be qualified.

Syntax Rules

1. Format 1 must be used for a sequential access mode file.
2. There must be a NEXT phrase for dynamic access mode files to retrieve records sequentially.
3. Format 2 can be used for random or dynamic access mode files to retrieve records randomly.
4. The KEY phrase can be used only for indexed files.
5. There must be an INVALID KEY or AT END phrase when there is no applicable USE AFTER EXCEPTION procedure for the file.
6. The storage area associated with *dest-item* and the record area associated with *file-name* cannot be the same storage area.

General Rules

1. The file must be open in the INPUT or I-O mode when the READ statement executes.
2. For sequential access mode files, the NEXT phrase is optional. It has no effect on READ statement execution.

3. Executing a Format 1 READ statement can cause the following to occur:
   - The record pointed to by the Next Record Pointer becomes available in the file's record area.
   - For sequential and relative files, the Next Record Pointer points to the file's next existing record.
   - For indexed files, the Next Record Pointer points to the next existing record established by the file's Key of Reference.
   - If the file has no next record, the Next Record Pointer indicates that no next logical record exists and the next attempt to read the file will cause the at end condition.
4. The READ statement updates the value of the FILE STATUS data item for the file.
5. A record is available before any statement executes after the READ.
6. More than one record description can describe a file's logical records. The records then share the same record area in storage. Sharing a record area is equivalent to implicit redefinition.

   READ statement execution does not change the contents of data items in the record area beyond the range of the current data record. The contents of those items are undefined.
7. A Format 1 READ statement can recognize the end of reel/unit during its execution. If it has not reached the logical end of the file, the READ statement performs a reel/unit swap. The Current Volume Pointer points to the file's next reel/unit.
8. During execution of a Format 2 READ statement, the Next Record Pointer can indicate that an optional file is not present. The invalid key condition then exists, and READ statement execution is unsuccessful.
9. When a Format 1 READ statement executes, the Next Record Pointer can indicate that:
   - There is no next logical record
   - No valid next record has been established
   - An optional file is not present

   When the READ statement detects one of these conditions:
   a. It updates the FILE STATUS data item for the file to indicate the at end condition.
   b. If the READ statement has an AT END phrase, control transfers to *stment*. Even if a USE AFTER EXCEPTION procedure is specifed for the file, it does not execute.
   c. If there is no AT END phrase, a USE AFTER EXCEPTION procedure must be associated with the file. Control transfers to that procedure. Control returns from the USE AFTER EXCEPTION procedure to the next executable statement after the end of the READ statement.

   When the at end condition occurs, execution of the READ statement is unsuccessful.

10. After the unsuccessful execution of a READ statement, the contents of the file's record area are undefined. If an optional file is not present, the Next Record Pointer is unchanged; otherwise, it indicates that no valid next record has been established. For indexed files, the Key of Reference is undefined.

11. For a relative or indexed file in dynamic access mode, a Format 1 READ statement with the NEXT phrase retrieves the file's next logical record.

12. For a relative file, a Format 1 READ statement updates the contents of the file's RELATIVE KEY data item. The data item contains the relative record number of the available record.

13. For a relative file, a Format 2 READ statement sets the Next Record Pointer to the record whose relative record number is in the file's RELATIVE KEY data item. Execution then continues as specified in General Rule 3.

    If the record is not in the file, the invalid key condition exists, and READ statement execution is unsuccessful.

14. When your program sequentially accesses an indexed file for records with duplicate alternate record key values, those records are made available to your program in the same order in which they were created. The duplicate values can be created by execution of WRITE or REWRITE statements.

15. For an indexed file, a Format 2 READ statement with the KEY phrase establishes *key-name* as the Key of Reference for the retrieval. For a dynamic access mode file, the same Key of Reference applies to later retrievals by Format 1 READ statement executions for the file. The Key of Reference continues in effect until a new Key of Reference is established.

16. For an indexed file, a Format 2 READ statement without the KEY phrase establishes the prime record key as the Key of Reference for the retrieval. For a dynamic access mode file, the same Key of Reference applies to later retrievals by Format 1 READ statement executions for the file. The Key of Reference continues in effect until a new Key of Reference is established.

17. For an indexed file, a Format 2 READ statement compares the value in the Key of Reference with the value in the corresponding data item in the file's records. The comparison continues until the READ statement finds the first record with an equal value. For an alternate key with duplicate values, the first record found is the first of a sequence of duplicates released to RMS. The READ statement sets Next Record Pointer to the record. Execution then continues as specified in General Rule 3.

    If the READ statement cannot identify a record with an equal value, the invalid key condition exists. READ statement execution is then unsuccessful.

    If the size of the retrieved record exceeds the maximum size specified for the file, READ statement execution is unsuccessful.

18. If there is an applicable USE AFTER EXCEPTION procedure, it executes whenever an input condition occurs that would result in a nonzero value in a FILE STATUS data item. However, it does not execute if: (a) the condition is invalid key, and there is an INVALID KEY phrase or (b) the condition is at end, and there is an AT END phrase.
19. Executing a READ statement with the INTO phrase is equivalent to executing the same statement without the phrase, then moving the current record from the record area to the area specified by *dest-item*.

Technical Note

READ statement execution can result in the FILE STATUS data item values summarized in the following table:

FILE STATUS	File Organization	Access Method	Meaning
00	All	All	Successful
13	All	Seq	No next logical record (at end)
15	All	Seq	Optional file not present (at end)
16	All	Seq	No valid next record (at end)
23	Ind, Rel	Rand	Record not in file (invalid key)
90	All	All	Record locked by another program; record available in record area
92	All	All	Record locked by another program; record not available
94	All	All	File not open, or incompatible open mode
30	All	All	All other permanent errors

Additional References

Section 3.2.2	I-O-CONTROL Paragraph, SAME Clause
Section 4.1.1.3	Multiple Record Descriptions
Section 5.7	I-O Status
Section 5.7.1	INVALID KEY Phrase
Section 5.7.2	AT END Phrase
Section 5.7.4	INTO Option
Section 5.9.17	OPEN Statement
Section 5.9.31	USE Statement

5.9.20 RELEASE Statement

Function

The RELEASE statement transfers records to the initial phase of a sort operation.

General Format

```
RELEASE rec [ FROM src-area ]
```

rec
: is the name of a logical record in a sort-merge file description (SD) entry. It can be qualified.

src-area
: is the identifier of the data item that contains the data.

Syntax Rules

1. A RELEASE statement can be used only in an input procedure. The input procedure must be associated with a SORT statement for the sort or merge file that contains *rec*.
2. *Rec* and *src-area* cannot refer to the same storage area.

General Rules

1. The rules for the FROM phrase appear in Section 5.7.3.
2. The RELEASE statement transfers the contents of *rec* to the first phase of the sort.
3. After the RELEASE statement executes, the record is no longer available in *rec* unless the associated sort or merge file-name is in a SAME RECORD AREA clause. In that case, the record is available to the program as a record of the sort-merge file-name. It is also available as a record of all other file-names in the same SAME RECORD AREA clause.

Additional References

Section 5.7.2	I-O-CONTROL Paragraph, SAME Clause
Section 5.7.3	FROM Option
Part IV of the COBOL-81 User's Guide for your system	Refer to the chapter on sorting records and merging files

5.9.21 RETURN Statement

Function

The RETURN statement gets sorted records from a sort operation. It also returns merged records in a merge operation.

General Format

```
RETURN smrg-file RECORD [ INTO dest-area ] AT END stment
```

smrg-file
is the name of a file described in a sort-merge file description (SD) entry.

dest-area
is the identifier of the data item to which the returned *smrg-file* record is moved.

stment
is an imperative statement.

Syntax Rules

1. A RETURN statement can be used only in an output procedure. The output procedure must be associated with a SORT or MERGE statement for *smrg-file*.
2. The storage area associated with *dest-area* and the record area associated with *smrg-file* cannot be the same storage area.

General Rules

1. The rules for the INTO phrase appear in Section 5.7.4.
2. When more than one record description describes the logical records for *smrg-file*, the records share the same storage area. The contents of storage positions beyond the range of the returned record are undefined when the RETURN statement ends.
3. Before the output procedure executes, the Next Record Pointer is updated. It points to the record whose key values make it first in the file. If there are no records, the Next Record Pointer indicates the at end condition.
4. The RETURN statement makes the next record (pointed to by the Next Record Pointer) available in the record area for *smrg-file*.
5. The Next Record Pointer is updated to point to the next record in *smrg-file*. The key values in the SORT or MERGE statement determine which is the next record.
6. If *smrg-file* has no next record, the Next Record Pointer is updated to indicate the at end condition.
7. If the Next Record Pointer indicates the at end condition when the RETURN statement executes, control transfers to *stment*. The contents of the *smrg-file* record areas are then undefined.

8. When the at end condition occurs:
   - RETURN statement execution is unsuccessful.
   - The Next Record Pointer is not changed.

Additional References

Section 3.2.2	I-O-CONTROL Paragraph, SAME Clause
Section 5.7.2	AT END Phrase
Section 5.7.4	INTO Option
Part IV of the COBOL-81 User's Guide for your system	Refer to the chapter on sorting records and merging files

5.9.22 REWRITE Statement

Function

The REWRITE statement logically replaces a mass storage file record.

General Format

```
REWRITE rec-name [ FROM src-item ] [ INVALID KEY stment ]
```

rec-name
: is the name of a logical record in the Data Division File Section. It can be qualified.

src-item
: is the identifier of the data item that contains the data.

stment
: is an imperative statement.

Syntax Rules

1. The INVALID KEY phrase cannot be used in a REWRITE statement that refers to a sequential file or to a relative file with sequential access mode.
2. For a relative file with random or dynamic access mode, or for an indexed file, the REWRITE statement must have an INVALID KEY phrase when there is no applicable USE AFTER EXCEPTION procedure for the file.
3. *Rec-name* and *src-item* cannot share the same storage area.

General Rules

All Files

1. The file associated with *rec-name* must be a mass storage file. It must be open in the I-O mode when the REWRITE statement executes.
2. For sequential access mode files, the last input-output statement executed for the file before the REWRITE statement must be a successfully executed READ statement. The REWRITE statement logically replaces the record accessed by the READ.
3. The record is no longer available in *rec-name* after a REWRITE statement successfully executes unless the associated file-name is in a SAME RECORD AREA clause. In this case, also available in the record areas of other file-names in the same SAME RECORD AREA clause.
4. The REWRITE statement does not affect the Next Record Pointer.
5. The REWRITE statement updates the value of the FILE STATUS data item for the file.
6. The rules for the FROM phrase appear in Section 5.7.3, FROM Option.

Sequential Files

7. The record named by *rec-name* must be the same size as the record being replaced.

Relative Files

8. For a random or dynamic access mode file, the REWRITE statement logically replaces the record specified in the RELATIVE KEY data item for *rec-name's* file. If the record is not in the file, the invalid key condition exists. The update does not occur, and the data in the record area is not affected.

Indexed Files

9. For a sequential access mode file, the prime record key specifies the record to be replaced. The values of the prime record keys in the record to be replaced and the last record read from (or positioned in) the file must be equal.
10. For a random or dynamic access mode file, the prime record key specifies the record to replace.
11. For a record with an alternate record key:
    - When the REWRITE does not change the value of an alternate record key, the order of retrieval is unchanged when the key is the Key of Reference.
    - When duplicate key values are allowed, and the value of an alternate record key changes, the later retrieval order of the record changes when the key is the Key of Reference. The record's logical position is last in the group of records with the same value in the alternate record key that changed.
12. Any of the following occurrences cause the invalid key condition:
    - The access mode is sequential, and the values in the prime record keys of the record to replace and the last record read from (or positioned in) the file are not equal.
    - The value in the prime record key does not equal that of any record in the file.
    - The value in an alternate record key whose definition does not have a DUPLICATES clause equals that of a record already in the file.

    The update does not occur, and the data in the record area is not affected.
13. If there is an applicable USE AFTER EXCEPTION procedure, it executes whenever an input or output condition occurs that would result in a nonzero value in a FILE STATUS data item. However, the INVALID KEY phrase (if present) supersedes a USE AFTER EXCEPTION procedure when there is an invalid key condition. In this case, the USE AFTER EXCEPTION procedure does not execute.

REWRITE
Continued

Technical Note

REWRITE statement execution can result in the FILE STATUS data item values summarized in the following table:

FILE STATUS	File Organization	Access Method	Meaning
00	All	All	Successful
02	Ind	All	Created duplicate alternate key
21	Ind	Seq	Primary key changed after READ
22	Ind	All	Duplicate alternate key (invalid key)
23	Ind, Rel	Rand	Record not in file (invalid key)
92	Ind, Rel	All	Record locked by another program
93	All	Seq	No previous READ
94	All	All	File not open, or incompatible open mode
30	All	All	All other permanent errors

Additional References

Section 3.2.2	I-O-CONTROL Paragraph, SAME Clause
Section 4.1.1.3	Multiple Record Descriptions
Section 5.1.4	Scope of Statements
Section 5.7	I-O Status
Section 5.7.1	INVALID KEY Phrase
Section 5.7.3	FROM Option
Section 5.9.17	OPEN Statement
Section 5.9.19	READ Statement
Section 5.9.31	USE Statement

5.9.23 SEARCH Statement

Function

The SEARCH statement searches for a table element that satisfies a condition. It sets the value of the associated index to point to the table element.

General Format

Format 1

```
SEARCH src-table [ VARYING pointr ] [ AT END stment ]
    { WHEN cond { stment          } } ...
                { NEXT SENTENCE   }
```

Format 2

```
SEARCH ALL src-table [ AT END stment ] WHEN { elemnt { IS EQUAL TO } arg }
                                            { IS =  }
                                            { cond-name             }

    [ AND { elemnt { IS EQUAL TO } arg } ] ... { stment          }
          { IS =   }                           { NEXT SENTENCE }
          { cond-name             }
```

src-table
: is a table identifier.

pointr
: is an index-name or the identifier of a data item described as USAGE INDEX, or an elementary numeric data item with no positions to the right of the assumed decimal point.

cond
: is any conditional expression.

stment
: is an imperative statement.

elemnt
: is an indexed data-name. It refers to the table element against which the argument is compared.

arg
: is the argument tested against each *elemnt* in the search. It is an identifier, a literal, or an arithmetic expression.

cond-name
: is a condition-name.

SEARCH
Continued

Syntax Rules

Both Formats

1. *Src-table* must not be subscripted or indexed. However, its description must contain an OCCURS clause with the INDEXED BY phrase.

Format 2

2. *Src-table* must contain the KEY IS phrase in its OCCURS clause.
3. Each *cond-name* must be defined as having only one value. The data-name associated with *cond-name* must be in the KEY IS phrase of the OCCURS clause for *src-table*.
4. Each *elemnt*:
   - Can be qualified
   - Must be indexed by the first index-name associated with *src-table*, in addition to other indexes or literals required for uniqueness
   - Must be in the KEY IS phrase of the OCCURS clause for *src-table*
5. Neither *arg* nor any identifier in its arithmetic expression can either:
   - Be used in the KEY IS phrase of the OCCURS clause for *src-table*
   - Be indexed by the first index-name associated with *src-table*
6. When *elemnt* or the data-name associated with *cond-name* is in the KEY phrase of the OCCURS clause for *src-table*, each preceding data-name (or associated *cond-name*) in that phrase must also be referenced.

General Rules

Both Formats

1. After executing a *stment* that does not end with a GO TO statement, control passes to the end of the SEARCH statement.
2. *Src-table* can be subordinate to a data item that contains an OCCURS clause. In that case, an index-name must be associated with each dimension of the table through the INDEXED BY phrase of the OCCURS clause. The SEARCH statement modifies the setting of only the index-name for *src-table* (and *pointr*, if there is one).

   A SEARCH statement must execute several times to search a multidimensional table. Before each execution, SET statements must execute to change the values of index-names that need adjustment.

Format 1

3. The Format 1 SEARCH statement searches a table serially, starting with the current index setting.
   a. The index-name associated with *src-table* can contain a value that indicates a higher occurrence number than is allowed for *src-table*. If the SEARCH statement execution starts when this condition exists, the search terminates immediately. If there is an AT END phrase, *stment* then executes. Otherwise, control passes to the end of the SEARCH statement.

b. If the index-name associated with *src-table* indicates a valid *src-table* occurrence number, the SEARCH statement evaluates the conditions in the order they appear. It uses the index settings to determine the occurrence numbers of items to test.

   If no condition is satisfied, the index-name for *src-table* is incremented to refer to the next occurrence. The condition evaluation process repeats using the new index-name settings. However, if the new value of the index-name for *src-table* indicates a table element outside its range, the search terminates as in General Rule 3a.

   When a condition is satisfied:

   - The search terminates immediately
   - The *stment* associated with the condition executes
   - The index-name remains set at the occurrence that satisfied the condition

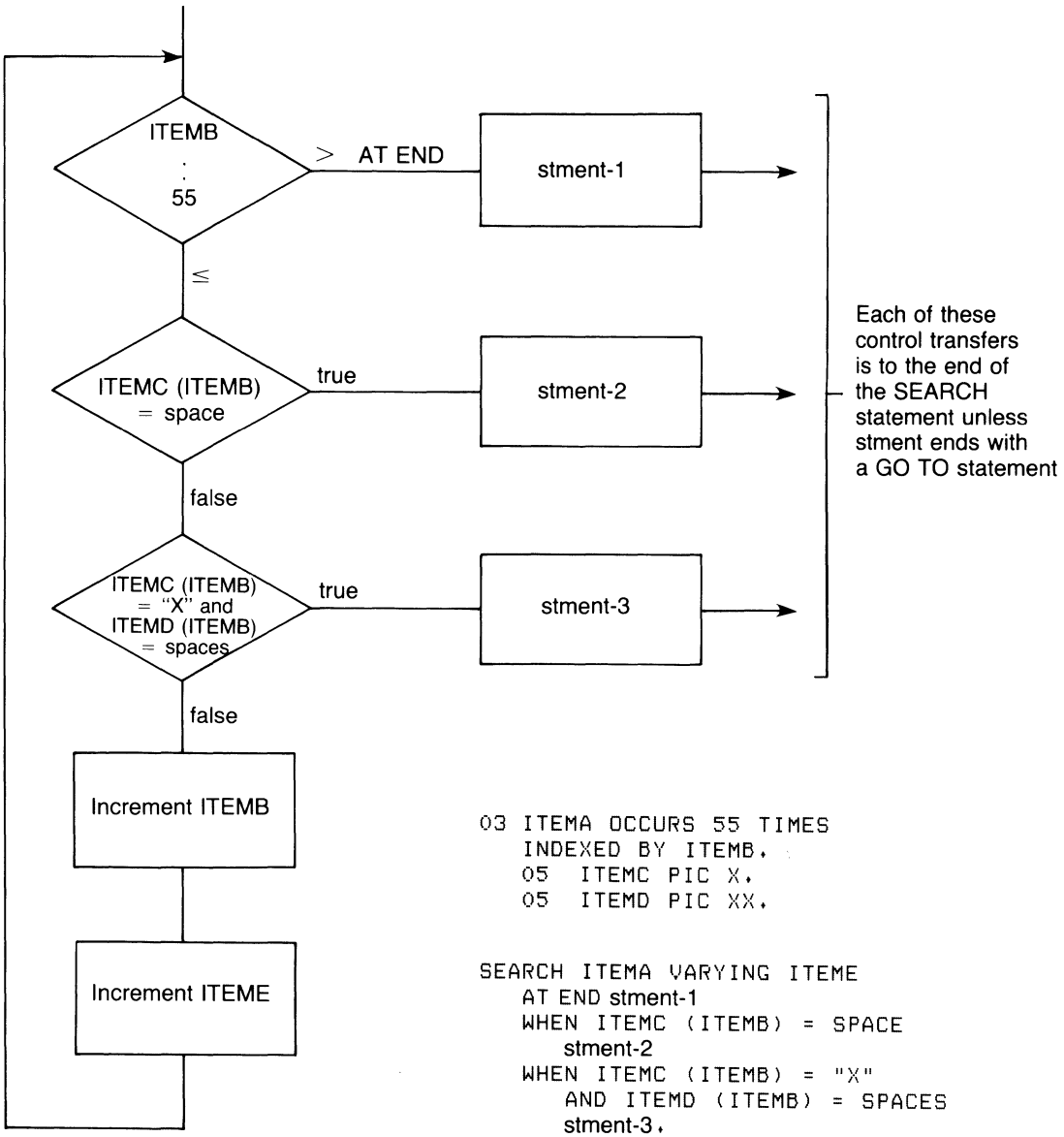
4. If there is no VARYING phrase, the index-name used for the search is the first index-name in the OCCURS clause for *src-table*. Other *src-table* index-names are unchanged.
5. If there is a VARYING phrase, *pointr* can be an index-name for *src-table*. (*Pointr* is named in the INDEXED BY phrase of the OCCURS clause for *src-table*.) The search then uses that index-name. Otherwise, it uses the first index-name in the INDEXED BY phrase.
6. *Pointr* also can be an index-name for another table. (*Pointr* is named in the INDEXED BY phrase in the OCCURS clause for that table entry.) In this case, the search increments the occurrence number represented by *pointr* by the same amount, and at the same time, as it increments the occurrence number represented by the *src-table* index-name.
7. If *pointr* is an index data item rather than an index-name, the search increments it by the same amount, and at the same time, as it increments the *src-table* index-name. If *pointr* is not an index data item or an index-name, the search increments it by one when it increments the *src-table* index-name.
8. Figure 5-4 describes the operation of a Format 1 SEARCH statement with two WHEN phrases.

Format 2

9. A SEARCH ALL operation yields predictable results only when:
   - The data in the table has the same order as described in the KEY IS phrase of the OCCURS clause for *src-table*
   - The contents of the keys in the WHEN phrase identify a unique table element
10. SEARCH ALL causes a nonserial (or binary) search. It ignores the initial setting of the *src-table* index-name and varies its setting during execution.
11. If the WHEN phrase conditions are not satisfied for any index setting in the allowed range, control passes to the AT END phrase *stment*, if there is one, or to the end of the SEARCH statement. In either case, the setting of the *src-table* index-name is not predictable.

12. If all the WHEN phrase conditions are satisfied for an index setting in the allowed range, control passes to either *stment* or the next sentence, whichever is in the statement. The *src-table* index-name then indicates the occurrence number that satisfied the conditions.
13. The index-name used for the search is the first index-name in the OCCURS clause for *src-table*. Other *src-table* index-names are unchanged.

Figure 5-4: Format 1 SEARCH Statement with Two WHEN Phrases




```
03 ITEMA OCCURS 55 TIMES
   INDEXED BY ITEMB.
   05  ITEMC PIC X.
   05  ITEMD PIC XX.

SEARCH ITEMA VARYING ITEME
   AT END stment-1
   WHEN ITEMC (ITEMB) = SPACE
      stment-2
   WHEN ITEMC (ITEMB) = "X"
      AND ITEMD (ITEMB) = SPACES
      stment-3.
```

Additional References

Section 4.2.13 OCCURS Clause
Section 5.5 Conditional Expressions

Examples

The examples assume these Data Division entries:

```
01  CUSTOMER-REC.
    03  CUSTOMER-USPS-STATE  PIC XX.
    03  CUSTOMER-REGION      PIC X.
    03  CUSTOMER-NAME        PIC X(15).
01  STATE-TAB.
    03  FILLER  PIC X(153)
          VALUE
          "AK3AL5AR5AZ4CA4CO4CT1DC1DE1FL5GA5HI3
-         "IA2ID3IL2IN2KS2KY5LA5MA1MD1ME1MI2MN2
-         "MO5MS5MT3NC5ND3NE2NH1NJ1NM4NV4NY1OH2
-         "OK4OR3PA1RI1SC5SD3TN5TX4UT4VA5VT1WA3
-         "WI2WV5WY4".
01  STATE-TABLE REDEFINES STATE-TAB.
    03  STATES OCCURS 51 TIMES
        ASCENDING KEY IS STATE-USPS-CODE
        INDEXED BY STATE-INDEX.
        05  STATE-USPS-CODE  PIC XX.
        05  STATE-REGION     PIC X.
01  STATE-NUM   PIC 99.
01  STATE-ERROR PIC 9.
01  NAME-TABLE VALUE SPACES.
    03  NAME-ENTRY OCCURS 8 TIMES
        INDEXED BY NAME-INDEX.
        05  LAST-NAME    PIC X(15).
        05  NAME-COUNT   PIC 999.
```

1. Binary search:

   (The correctness of this statement's operation depends on the ascending order of key values.)

```
SEARCH ALL STATES
  AT END
    MOVE 1 TO STATE-ERROR
  WHEN STATE-USPS-CODE (STATE-INDEX) = CUSTOMER-USPS-STATE
    MOVE 0 TO STATE-ERROR
    MOVE STATE-REGION (STATE-INDEX) TO CUSTOMER-REGION.
```

Results			
CUSTOMER-STATE	**CUSTOMER-REGION**	**STATE-INDEX**	**STATE-ERROR**
NH	1	31	0
CA	4	5	0
DM		10	1
WY	4	51	0

2. Serial search with two WHEN phrases:

```
INITIALIZE-SEARCH.
    MOVE 1 TO CUSTOMER-REGION.
    MOVE "NH" TO CUSTOMER-USPS-STATE.

    DISPLAY "States in customer's region:".

SEARCH-LOOP.
    SEARCH STATES
        AT END
                GO TO SEARCH END
        WHEN STATE-REGION (STATE-INDEX) = CUSTOMER-USPS-STATE
                SET STATE-NUM TO STATE-INDEX
        WHEN STATE-REGION (STATE-INDEX) = CUSTOMER-REGION
                DISPLAY STATE-USPS-CODE (STATE-INDEX)
                    " " WITH NO ADVANCING.
    SET STATE-INDEX UP BY 1.
    GO TO SEARCH-LOOP.

SEARCH-END.
    DISPLAY " "
    DISPLAY "Customer state index number = " STATE-NUM.
```

Results

```
IA   13  0
IL   15  0
IN   16  0
KS   17  0
MI   23  0
MN   24  0
NE   30  0
NH   31  3
OH   36  3
WI   49  3
     52  1
```

3. Updating a table in a SEARCH statement:

```
GET-NAME.
    DISPLAY "Enter name: " NO ADVANCING.
    ACCEPT CUSTOMER-NAME.
    SET NAME-INDEX TO 1.
    SEARCH NAME-ENTRY
      AT END
        DISPLAY "   Table full"
       SET NAME-INDEX TO 1
        PERFORM SHOW-TABLE 8 TIMES
        STOP RUN
     WHEN LAST-NAME (NAME-INDEX) = CUSTOMER-NAME
        ADD 1 TO NAME-COUNT (NAME-INDEX)
      WHEN LAST-NAME (NAME-INDEX) = SPACES
         MOVE CUSTOMER-NAME TO LAST-NAME (NAME-INDEX)
         MOVE 1 TO NAME-COUNT (NAME-INDEX).
    GO TO GET-NAME.
SHOW-TABLE.

DISPLAY LAST-NAME (NAME-INDEX) " " NAME-COUNT (NAME-INDEX).
    SET NAME-INDEX UP BY 1.
```

Results

```
Enter name: CRONKITE
Enter name: GEORGE
Enter name: PHARES
Enter name: CRONKITE
Enter name: BELL
Enter name: SMITH
Enter name: FRANKLIN
Enter name: HENRY
Enter name: GEORGE
Enter name: ROBBINS
Enter name: BELL
Enter name: FRANKLIN
Enter name: SMITH
Enter name: BELL
Enter name: SMITH
   Table full
CRONKITE  002
GEORGE  002
PHARES  001
BELL  003
SMITH  003
FRANKLIN  002
HENRY  001
ROBBINS  001
```

5.9.24 SET Statement

Function

The SET statement sets values of indexes associated with table elements.

General Format

Format 1

```
SET { rsult } ... TO val
```

Format 2

```
SET { indx } ... { UP BY   } increm
                 { DOWN BY }
```

rsult
: is an index-name, the identifier of an index data item, or an elementary numeric data item described as an integer.

val
: is a positive integer, which may be signed. It can also be an index-name (or the identifier of an index data item) or an elementary numeric data item described as an integer.

indx
: is an index-name.

increm
: is an integer, which may be signed. It can also be the identifier of an elementary numeric data item described as an integer.

General Rules

1. Index-names are associated with a table in the table's OCCURS clause INDEXED BY phrase.
2. Undefined results occur when operands overlap; that is, when they share a part of their storage areas.
3. If *rsult* is an index-name, its value after SET statement execution must correspond to an occurrence number of an element in the associated table.
4. If *val* is an index-name, its value before SET statement execution must correspond to an occurrence number of an element in the table associated with *rsult*.
5. The value of *indx*, both before and after SET statement execution, must correspond to an occurrence number of an element in the table associated with *indx*.

Format 1

6. The SET statement sets the value of *rsult* to refer to the table element whose occurrence number corresponds to the table element referred to by *val*. If *val* is an index data item, no conversion occurs.
7. If *rsult* is an index data item, *val* cannot be an integer. No conversion occurs when *rsult* is set to the value of *val*.
8. If *rsult* is not an index data item or an index-name, *val* can only be an index-name.
9. When there is more than one *rsult*, SET uses the original value of *val* in each operation. Subscript or index evaluation for *rsult* occurs immediately before its value changes.
10. Table 5-12 shows the validity of operand combinations. An asterisk (*) means that no conversion occurs during the SET operation.

Table 5-12: Validity of Operand Combinations in Format 1 SET Statements

Sending Item	Receiving Item		
	Integer Data Item	Index	Index Data Item
Integer Literal	Invalid/Rule 7	Valid/Rule 5	Invalid/Rule 6
Integer Data Item	Invalid/Rule 7	Valid/Rule 5	Invalid/Rule 6
Index	Valid/Rule 7	Valid/Rule 5	Valid/Rule 6*
Index Data Item	Invalid/Rule 7	Valid/Rule 5*	Valid/Rule 6*

Format 2

11. The SET statement increments (UP) or decrements (DOWN) *indx* by a value that corresponds to the number of occurrences *increm* represents.
12. When there is more than one *indx*, SET uses the original value of *increm* in each operation.

Additional References

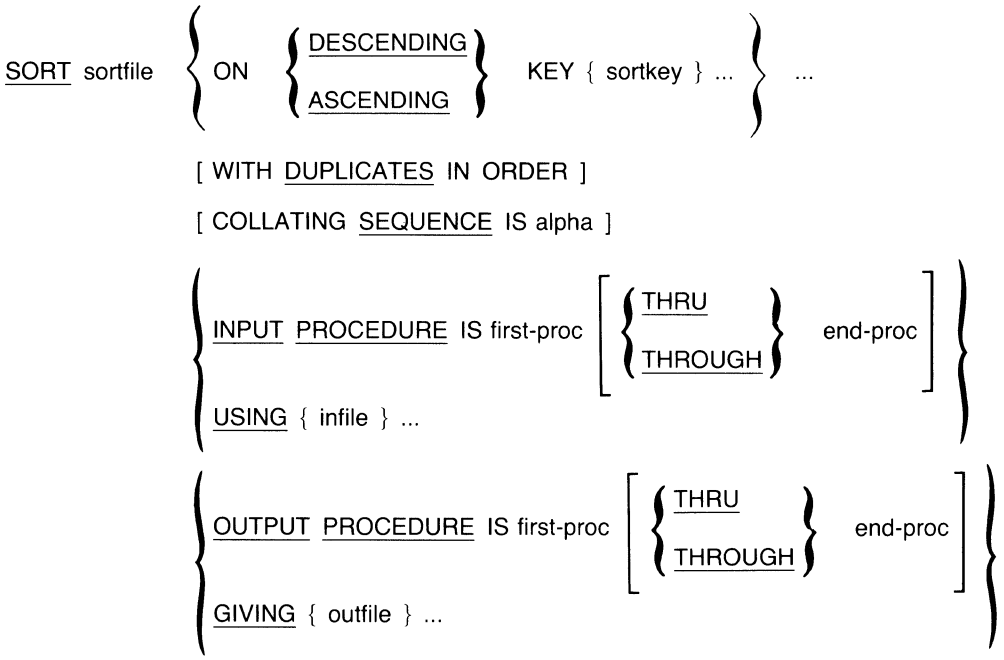
Section 5.9.15 MOVE Statement
Section 5.9.23 SEARCH Statement

5.9.25 SORT Statement

Function

The SORT statement creates a sort file by executing input procedures or transferring records from an input file. It sorts the records in the sort file using one or more keys that you specify. Finally, it returns each record from the sort file, in sorted order, to output procedures or an output file.

General Format



sortfile
: is a file-name described in a sort-merge file description (SD) entry in the Data Division.

sortkey
: is the data-name of a data item in a record associated with *sortfile*.

alpha
: is an alphabet-name defined in the SPECIAL-NAMES paragraph of the Environment Division.

first-proc
: is the section-name of the first (or only) section of the INPUT or OUTPUT procedure range.

infile
: is the file-name of the input file. It must be described in a file description (FD) entry in the Data Division.

end-proc
is the section-name of the last section of the INPUT or OUTPUT procedure range.

outfile
is the file-name of the output file. It must be described in a file description (FD) entry in the Data Division.

Syntax Rules

1. You can use SORT statements anywhere in the Procedure Division except in:
   - Declaratives
   - A SORT or MERGE statement input or output procedure
2. If *sortfile* contains variable length records, *infile* records must not be smaller than the smallest in *sortfile* nor larger than the largest.
3. If *sortfile* contains fixed length records, *infile* records must not be larger than the largest record described for *sortfile*.
4. If *outfile* contains variable length records, *sortfile* records must not be smaller than the smallest in *outfile* nor larger than the largest.
5. If *outfile* contains fixed length records, *sortfile* records must not be larger than the largest record described for *outfile*.
6. *Sortkey* can be qualified.
7. *Sortkey* cannot be in a group item that contains variable occurrence data items.
8. The *sortkey* description cannot contain an OCCURS clause or be subordinate to a data description entry that does.
9. *Sortfile* can have more than one record description. However, *sortkey* need be described in only one of the record descriptions. The character positions referenced by *sortkey* are used as the key for all the file's records.
10. The words THRU and THROUGH are equivalent.
11. If *outfile* is an indexed file, the first *sortkey* must be in the ASCENDING phrase. It must specify the same character positions in its record as the prime record key for *outfile*.
12. Each *sortkey* cannot be larger than 255 characters.
13. The total number of characters in all *sortkeys* for the file cannot be more than 512 characters.
14. Each SORT statement can specify no more than 16 *sortkeys*.
15. Neither *infile* nor *outfile* can be an indexed file in random access mode.

General Rules

1. If *sortfile* contains fixed length records, any shorter *infile* records are space filled on the right, following the last character. Space filling occurs before the *infile* record is released to *sortfile*.
2. The first *sortkey* you specify is the major key, the next *sortkey* you specify is the next most significant key, and so forth. The significance of *sortkey* data items is not affected by how you divide them into KEY phrases. Only first-to-last order determines significance.
3. The ASCENDING phrase causes the sorted sequence to be from the lowest to highest *sortkey* value.
4. The DESCENDING phrase causes the sorted sequence to be from the highest to the lowest *sortkey* value.
5. Sort sequence follows the rules for relation condition comparisons.
6. The DUPLICATES phrase affects the return order of records whose corresponding *sortkey* values are equal.
   - When there is a USING phrase, return order is the same as the order of appearance of *infile* names in the SORT statement.
   - When there is an INPUT PROCEDURE, return order is the same as the order in which the records were released.
7. If there is no DUPLICATES phrase, the return order for records with equal corresponding *sortkey* values is unpredictable.
8. The SORT statement determines the comparison collating sequence for nonnumeric *sortkey* items when it begins execution. If there is a COLLATING SEQUENCE phrase in the SORT statement, SORT uses that sequence. Otherwise, it uses the program collating sequence described in the OBJECT-COMPUTER paragraph.
9. The INPUT PROCEDURE range consists of one or more sections that:
   - Appear contiguously in the source program
   - Do not form a part of an OUTPUT PROCEDURE range
10. The statements in the INPUT PROCEDURE range must include at least one RELEASE statement to transfer records to *sortfile*.
11. The program must not pass control to any statement in the INPUT PROCEDURE range except during execution of a related SORT statement.
12. The INPUT PROCEDURE range cannot contain SORT or MERGE statements. Its statements must not transfer control explicitly outside the INPUT PROCEDURE section(s). However, its statements can cause implied control transfers to Declaratives.

13. The remainder of the Procedure Division must not transfer control to statements in the INPUT PROCEDURE range.
14. If there is an INPUT PROCEDURE phrase, control passes to the first statement in its range before the SORT statement sequences the *sortfile* records. When control passes the last statement in the INPUT PROCEDURE range, the records released to *sortfile* are sorted.
15. During execution of the INPUT or OUTPUT procedures, or any USE AFTER EXCEPTION procedure implicitly invoked during the SORT statement, no outside statement can manipulate the files or record areas associated with *infile* or *outfile*.
16. If there is a USING phrase, the SORT statement transfers all records in *infile* to *sortfile*. This transfer is an implied SORT statement input procedure. When the SORT statement executes, *infile* must not be open.
17. For each *infile*, the SORT statement:
    - Initiates file processing as if the program had executed an OPEN statement with the INPUT phrase.
    - Gets the logical records and releases them to the sort operation. SORT obtains each record as if the program had executed a READ statement with the NEXT and AT END phrases.
    - Terminates file processing as if the program had executed a CLOSE statement with no optional phrases. The SORT statement ends file processing before it executes any output procedure.

    These implicit OPEN, READ, and CLOSE operations cause associated USE procedures to execute when an on exception condition exists.
18. OUTPUT PROCEDURE consists of one or more sections that:
    - Appear contiguously in the source program
    - Do not form a part of an INPUT PROCEDURE range
19. When the SORT statement begins OUTPUT PROCEDURE, it is ready to select the next record in sorted order. The statements in the OUTPUT PROCEDURE range must include at least one RETURN statement to make records available for processing.
20. The program must not pass control to any statement in the OUTPUT PROCEDURE range except during execution of a related SORT statement.
21. The OUTPUT PROCEDURE range cannot contain SORT or MERGE statements. Its statements must not transfer control explicitly outside the range. However, its statements cause implied control transfers to Declaratives if an on exception condition occurs.

22. The remainder of the Procedure Division must not transfer control to statements in the OUTPUT PROCEDURE range.

23. If there is an OUTPUT PROCEDURE phrase, control passes to the first statement in its range after the SORT statement sequences the records in *sortfile*. When control passes the last statement in the OUTPUT PROCEDURE range, the SORT statement ends. Control then transfers to the next executable statement after the SORT statement.

24. If there is a GIVING phrase, the SORT statement writes all sorted records to each *outfile*. This transfer is an implied SORT statement output procedure. When the SORT statement executes, *outfile* must not be open.

25. The SORT statement initiates *outfile* processing as if the program had executed an OPEN statement with the OUTPUT phrase. The SORT statement does not initiate *outfile* processing until after INPUT PROCEDURE execution.

26. The SORT statement gets the sorted logical records and writes them to each *outfile*. SORT writes each record as if the program had executed a WRITE statement with no optional phrases.

    For relative files, the value of the relative key data item is 1 for the first returned record, 2 for the second, and so on. When the SORT statement ends, the value of the relative key data item indicates the number of *outfile* records.

27. The SORT statement terminates *outfile* processing as if the program had executed a CLOSE statement with no optional phrases.

28. These implicit OPEN, WRITE, and CLOSE operations can cause associated USE procedures to execute. For example, if the SORT statement tries to write beyond the boundaries of *outfile*, the applicable USE AFTER EXCEPTION procedure executes. If control returns from a USE procedure, or if there are none, *outfile* processing terminates as if the program had executed a CLOSE statement with no optional phrases.

29. If *outfile* contains fixed length records, any shorter *sortfile* records are space filled on the right, after the last character. Space filling occurs before the *sortfile* record is released to *outfile*.

Additional References

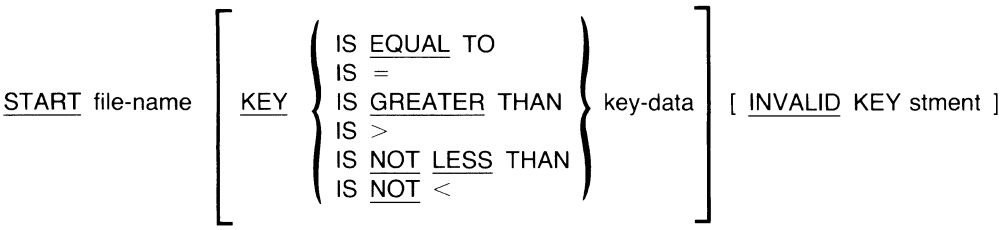
Section 3.1.2	OBJECT-COMPUTER Paragraph
Section 3.1.3	SPECIAL-NAMES Paragraph
Section 3.2.2	I-O-CONTROL Paragraph
Section 5.5.1	Relation Conditions
Section 5.8	Segmentation
Section 5.9.31	USE Statement
Part IV of the COBOL-81 User's Guide for your system	Refer to the chapter on sorting records and merging files

5.9.26 START Statement

Function

The START statement establishes the logical position of the Next Record Pointer in an indexed or relative file. The logical position affects subsequent sequential record retrieval.

General Format



file-name
: is the name of an indexed or relative file with sequential or dynamic access. It cannot be the name of a sort or merge file.

key-data
: is the data-name of a record key, the leftmost part of a record key, or the relative key for *file-name*. It can be qualified.

stment
: is an imperative statement.

Syntax Rules

1. There must be an INVALID KEY phrase if *file-name* does not have an applicable USE AFTER EXCEPTION procedure.
2. For a relative file, *key-data* must be the file's RELATIVE KEY data item.
3. For an indexed file, *key-data* can be either:
   - A record key for the file.
   - An alphanumeric data item subordinate to the description of the file's record key. The leftmost character position of *key-data* must correspond to that of the record key data item.

General Rules

All Files

1. The file must be open in the INPUT or I-O mode when the START statement executes.
2. If there is no KEY phrase, the implied relational operator is EQUAL.
3. START statement execution does not change: (a) the contents of the record area or (b) the contents of the data item referred to in the DEPENDING ON phrase of the file's RECORD clause.

4. The comparison specified by the KEY phrase relational operator occurs between a key for a record in the file and a data item. (See General Rules 11, 12, and 13.) If the file is indexed, and the operand sizes are unequal, the comparison operates as if the longer one was truncated on the right to the size of the shorter. All other numeric or nonnumeric comparison rules apply.

   The Next Record Pointer is set to the first logical record in the file whose key satisfies the comparison.

   If no record in the file satisfies the comparison:

   - The invalid key condition exists
   - START statement execution is unsuccessful
   - The Next Record Pointer indicates that no valid next record is established

5. The START statement updates the FILE STATUS data item for the file.
6. If the Next Record Pointer indicates that an optional file is not present when the START statement executes, the invalid key condition exists. START statement execution is then unsuccessful.

Relative Files

7. The comparison described in General Rule 4 uses the data item referred to by the RELATIVE KEY phrase in the file's ACCESS MODE clause.

Indexed Files

8. The START statement establishes a Key of Reference as follows:

   - If there is no KEY phrase, the file's prime record key becomes the Key of Reference.
   - If there is a KEY phrase, and *key-data* is a record key for the file, that record key becomes the Key of Reference.
   - If there is a KEY phrase, and *key-data* is not a record key for the file, the record key whose leftmost character corresponds to the leftmost character of *key-data* becomes the Key of Reference.

   The Key of Reference establishes the record ordering for the START statement. (See General Rule 4.) If the execution of the START statement is successful, later sequential READ statements use the same Key of Reference.

9. If there is a KEY phrase, the comparison described in General Rule 4 uses the contents of *key-data*.
10. If there is no KEY phrase, the comparison described in General Rule 4 uses the data item referred to in the file's RECORD KEY clause.
11. If START statement execution is not successful, the Key of Reference is undefined.
12. If there is an applicable USE AFTER EXCEPTION procedure, it executes whenever an input or output condition occurs that would result in a nonzero value in a FILE STATUS data item. However, it does not execute if the condition is invalid key and there is an INVALID KEY phrase.

13. When the invalid key condition is recognized, these actions occur in the following order:

    a. A value indicating the invalid key condition is placed in the FILE STATUS data item, if one is specified, for the file.

    b. If the statement causing the condition has an INVALID KEY phrase, control transfers to the associated imperative statement. Any USE AFTER EXCEPTION procedure for the file does not execute.

    c. If there is no INVALID KEY phrase, control transfers to the applicable USE AFTER EXCEPTION procedure for the file.

    d. The Next Record Pointer is set to the beginning of the file.

Technical Note

START statement execution can result in these FILE STATUS data item values:

FILE STATUS	Meaning
00	Successful
23	Record not in file (invalid key)
90	Record locked by another user, record is available in record area
92	Record locked by another program
94	File not open, or incompatible open mode
30	All other permanent errors

Additional References

Section 5.5.1.1	Comparison of Numeric Operands
Section 5.5.1.2	Comparison of Nonnumeric Operands
Section 5.7	I-O Status
Section 5.7.1	INVALID KEY Phrase
Section 5.9.17	OPEN Statement
Section 5.9.19	READ Statement
Section 5.9.31	USE Statement

5.9.27 STOP Statement

Function

The STOP statement permanently ends or temporarily suspends image execution.

General Format

```
STOP  { RUN  }
      { disp }
```

disp
is any literal, or any figurative constant except ALL literal.

Syntax Rule

If a STOP RUN statement is in a consecutive sequence of imperative statements in a sentence, it must be the last statement in that sequence.

General Rules

1. STOP RUN ends image execution.
2. STOP *disp* temporarily suspends the image. It displays the value of *disp* on the user's standard display device. If the user continues the image, execution resumes with the next executable statement.

Technical Notes

1. STOP RUN causes all open files to be closed.
2. STOP *disp* suspends execution of the image without terminating it.

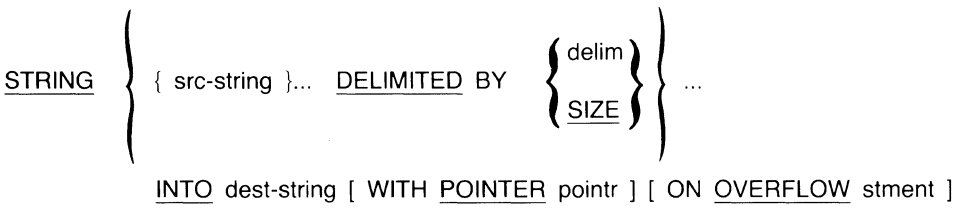
   The user can continue image execution by typing in anything but "STOP". Then execution resumes with the next executable statement.

5.9.28 STRING Statement

Function

The STRING statement concatenates the partial or complete contents of one or more data items into a single data item.

General Format



src-string
: is a nonnumeric literal or identifier of a DISPLAY data item. It is the sending area.

delim
: is a nonnumeric literal or the identifier of a DISPLAY data item. It is the delimiter of *src-string*.

dest-string
: is the identifier of a DISPLAY data item. *Dest-string* is the receiving area that contains the result of the concatenated *src-strings*.

pointr
: is an elementary numeric data item described as an integer. It points to the position in *dest-string* to contain the next character moved.

stment
: is an imperative statement.

Syntax Rules

1. *Pointr* cannot define the assumed decimal scaling position character (P) in its PICTURE clause.
2. Literals can be any figurative constant other than ALL literal.
3. The description of *dest-string* cannot: (a) have a JUSTIFIED clause or (b) indicate an edited data item.
4. The size of *pointr* must allow it to contain a value one greater than the size of *dest-string*.

General Rules

1. *Delim* specifies the character(s) to delimit the move.
2. If *src-string* is a variable length item, SIZE refers to the number of characters currently defined for it.

3. When *src-string* or *delim* is a figurative constant, its size is one character.
4. The STRING statement moves characters from *src-string* to *dest-string* according to the rules for alphanumeric to alphanumeric moves. However, no space-filling occurs.
5. When the DELIMITED phrase contains *delim*:
   - The contents of each *src-string* are moved to *dest-string* in the sequence they appear in the statement
   - Data movement begins with the leftmost character and continues to the right, character by character
   - Data movement ends when the STRING operation completes any of the following:
     a. Reaches the end of *src-string*
     b. Reaches the end of *dest-string*
     c. Detects the characters specified by *delim*
6. When the DELIMITED phrase contains the SIZE phrase:
   - The entire contents of each *src-string* is moved to *dest-string* in the same sequence as they appear in the statement
   - Data movement begins with the leftmost character and continues to the right, character by character
   - Data movement ends when the STRING operation either:
     a. Has transferred all data in each *src-string*
     b. Reaches the end of *dest-string*
   - If *src-string* is a variable length data item, the STRING statement moves the number of characters currently defined for the data item
7. When the POINTER phrase appears, the program must set *pointr* to an initial value greater than zero before executing the STRING statement. The PICTURE for *pointr* cannot contain Ps.
8. When there is no POINTER phrase, the STRING statement operates as if *pointr* were set to an initial value of 1.
9. When the STRING statement transfers characters to *dest-string*, the moves operate as if:
   - The characters were moved one at a time from *src-string*
   - Each character were moved to the position in *dest-string* indicated by *pointr* (if *pointr* does not exceed the length of *dest-string*)
   - The value of *pointr* were increased by one before moving the next character
10. When the STRING statement ends, only those parts of *dest-string* referenced during statement execution change. The rest of *dest-string* contains the same data as before the STRING statement executed.

11. Before it moves each character to *dest-string*, the STRING statement tests the value of *pointr*. If it is less than one or greater than the number of character positions in *dest-string*, the STRING statement:
    - Moves no further data to *dest-string*
    - Executes the ON OVERFLOW phrase *stment*
    - Transfers control to the end of the STRING statement if there is no ON OVERFLOW phrase
12. Subscripting or indexing evaluation for *src-string* and *delim* occur just before the STRING statement examines *src-string* for its delimiters.
13. Subscripting or indexing evaluation for *pointr* occurs just before STRING statement execution.
14. Undefined results occur when operands overlap; that is, when sending fields and receiving fields share a part of their storage areas.

Additional References

Section 5.1.4 Scope of Statements
Section 5.9.15 MOVE Statement

Examples

The examples assume the following data description entries:

```
WORKING-STORAGE SECTION.
01  TEXT-STRING              PIC X(30).
01  INPUT-MESSAGE            PIC X(60).
01  NAME-ADDRESS-RECORD.
    03  CIVIL-TITLE          PIC X(5).
    03  LAST-NAME            PIC X(10).
    03  FIRST-NAME           PIC X(10).
    03  STREET               PIC X(15).
    03  CIT                  PIC X(15).
* Assume CITY ends with "/"
    03  STATE                PIC XX.
    03  ZIP                  PIC 9(5).
01  PTR                      PIC 99.
01  HOLD-PTR                 PIC 99.
01  LINE-COUNT               PIC 99.
```

1. Using both delimiters and SIZE:

```
DISPLAY " ".
DISPLAY NAME-ADDRESS-RECORD.
MOVE SPACES TO TEXT-STRING.
STRING CIVIL-TITLE DELIMITED BY " "
  " " DELIMITED BY SIZE
  FIRST-NAME DELIMITED BY " "
  " " DELIMITED BY SIZE
  LAST-NAME DELIMITED BY SIZE
    INTO TEXT-STRING.
DISPLAY TEXT-STRING.
DISPLAY STREET.
```

(continued on next page)

```
MOVE SPACES TO TEXT-STRING.
STRING CITY DELIMITED BY "/"
  ", " DELIMITED BY SIZE
  STATE DELIMITED BY SIZE
  " " DELIMITED BY SIZE
  ZIP DELIMITED BY SIZE
    INTO TEXT-STRING.
DISPLAY TEXT-STRING.
```

Results

```
Mr. Smith      Irwin      603 Main St.    Merrimack/      NH03054
Mr. Irwin Smith
603 Main St.
Merrimack, NH 03054
Miss Lambert   Alice      1229 Exeter St.Boston/          MA03102
Miss Alice Lambert
1229 Exeter St.
Boston, MA 03102

Mrs. Gilbert   Rose       8 State Street New York/       NY10002
Mrs. Rose Gilbert
8 State Street
New York, NY 10002

Mr. Cowherd    Owen       1064 A St.      Washington/    DC20002
Mr. Owen Cowherd
1064 A St.
Washington, DC 20002
```

2. Using the POINTER phrase:

```
    MOVE 0 TO LINE-COUNT.
    MOVE 1 TO PTR.
GET-WORD.
    IF LINE-COUNT NOT < 4
      DISPLAY "      " TEXT-STRING
      GO TO GOT-WORDS.
    ACCEPT INPUT-MESSAGE.
    DISPLAY INPUT-MESSAGE.
SAME-WORD.
    MOVE PTR TO HOLD-PTR.
    STRING INPUT-MESSAGE DELIMITED BY SPACE
      ", " DELIMITED BY SIZE
      INTO TEXT-STRING
      WITH POINTER PTR
      ON OVERFLOW
        STRING "                  " DELIMITED BY SIZE
          INTO TEXT-STRING
          WITH POINTER HOLD-PTR
        DISPLAY "    " TEXT-STRING
    MOVE SPACES TO TEXT-STRING
        ADD 1 TO LINE-COUNT
        MOVE 1 TO PTR
        GO TO SAME-WORD.
    GO TO GET-WORD.
GOT-WORDS.
    EXIT.
```

Results

```
This
example
demonstrates
how
    This, example, demonstrates,
the
STRING
statement
can
    how, the, STRING, statement,
construct
text
strings
    can, construct, text,
using
the
POINTER
phrase
    strings, using, the, POINTER,
    phrase,
```

5.9.29 SUBTRACT Statement

Function

The SUBTRACT statement subtracts one, or the sum of two or more, numeric items from one or more items. It stores the result in one or more items.

General Format

Format 1

```
SUBTRACT { num } ... FROM { rsult [ ROUNDED ] } ... [ ON SIZE ERROR stment ]
```

Format 2

```
SUBTRACT { num } ... FROM num GIVING { rsult [ ROUNDED ] } ... [ ON SIZE ERROR stment ]
```

Format 3

```
SUBTRACT { CORRESPONDING } grp-1 FROM grp-2 [ ROUNDED ] [ ON SIZE ERROR stment ]
         { CORR          }
```

num
: is a numeric literal or the identifier of an elementary numeric item.

rsult
: is the identifier of an elementary numeric item. However, in Format 2, *rsult* can be an elementary numeric edited item. It is the resultant identifier.

grp
: is the identifier of a group item.

stment
: is an imperative statement.

Syntax Rule

CORR is an abbreviation for CORRESPONDING.

General Rules

1. The data descriptions of the operands need not be the same. Conversion and decimal point alignment will occur, as needed, throughout the calculation.
2. The maximum size of each operand is 18 digits.
3. Undefined results occur when operands share a part of their storage areas.
4. In Format 1, the values of the operands before the word FROM are summed. This total is subtracted from the value of the first *rsult*. The process repeats for each later occurrence of *rsult*.

5. In Format 2, the values of the operands before the word FROM are summed. This total is subtracted from the *num* following the word FROM. The result replaces the current value of each *rsult*.
6. In Format 3, data items in *grp-1* are subtracted from and stored into the corresponding data items in *grp-2*.

Additional References

Section 5.1.4	Scope of Statements
Section 5.6.1	Arithmetic Operations
Section 5.6.3	ROUNDED Option
Section 5.6.4	ON SIZE ERROR Option
Section 5.6.5	CORRESPONDING Option
Section 5.6.6	Overlapping Operands and Incompatible Data
Section 5.6.2	Multiple Receiving Fields in Arithmetic Statements

Examples

In these examples, results are shown only for data items whose values change. The examples assume these data descriptions and beginning values:

```
                                                          Initial
                                                          Value

03  ITEMA  PIC S99 VALUE -85.                             -85
03  ITEMB  PIC 99 VALUE 2.                                2
03  ITEMC  VALUE "123".
    05  ITEMD  OCCURS 3 TIMES                             1 2 3
           PIC 9.
03  ITEME  PIC S99 VALUE -95.                             -95
```

Results

1. Without GIVING phrase:

```
SUBTRACT 2 ITEMB FROM ITEMA.                              ITEMA = -89
```

2. SIZE ERROR clause:

   (When the SIZE ERROR condition occurs, the value of the resultant identifier does not change.)

```
SUBTRACT 38 FROM ITEMA                                    ITEMA = -85
  ON SIZE ERROR                                           ITEMB = 0
    MOVE 0 TO ITEMB.
```

3. Multiple receiving fields:

   (The operations proceed from left to right. Therefore, the subscript for ITEMB is evaluated after the subtraction changes its value.)

```
SUBTRACT 1 FROM ITEMB ITEMD (ITEMB).
```

ITEMB = 1
ITEMD (1) = 0

4. GIVING phrase:

```
SUBTRACT ITEME ITEMD (ITEMB) FROM ITEMA
     GIVING ITEMB.
```

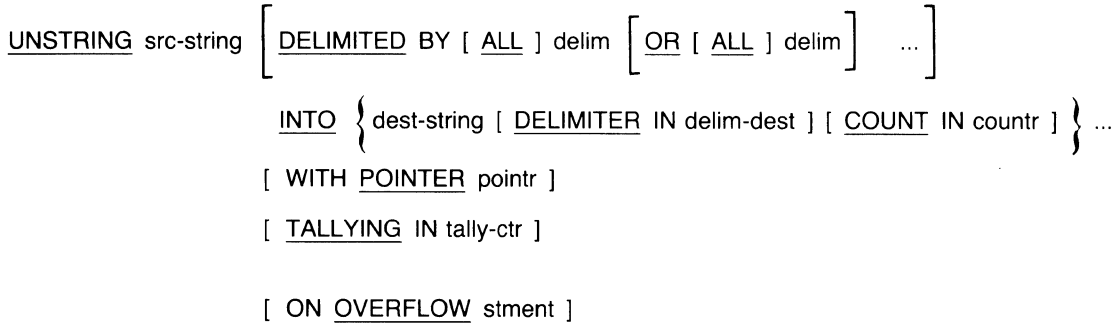
ITEMB = 8

5.9.30 UNSTRING Statement

Function

The UNSTRING statement separates contiguous data in a sending field and stores it in one or more receiving fields.

General Format



src-string
: is the identifier of an alphanumeric class data item. *Src-string* is the sending field.

delim
: is a nonnumeric literal or the identifier of an alphanumeric data item. It is the delimiter for the UNSTRING operation.

dest-string
: is the identifier of an alphanumeric, alphabetic, or numeric DISPLAY data item. It is the receiving field for the data from *src-string*.

delim-dest
: is the identifier of an alphanumeric data item. It is the receiving field for delimiters.

countr
: is the identifier of an elementary numeric data item described as an integer. It contains the count of characters moved.

pointr
: is the identifier of an elementary numeric data item described as an integer. It points to the current character position in *src-string*.

tally-ctr
: is the identifier of an elementary numeric data item described as an integer. It counts the number of *dest-string* fields accessed during the UNSTRING operation.

stment
: is an imperative statement.

UNSTRING
Continued

Syntax Rules

1. Literals can be any figurative constant other than ALL literal.
2. *Pointr* must be large enough to contain a value one greater than the size of *src-string*.
3. The DELIMITER IN and COUNT IN phrases can appear only if there is a DELIMITED BY phrase.
4. *Countr*, *pointr*, *dest-string*, and *tally-ctr* cannot define the assumed decimal scaling position character P in its PICTURE clause.

General Rules

1. *Countr* represents the number of characters in *src-string* isolated by the delimiters for the move to *dest-string*. The count does not include the delimiter characters.
2. When *delim* is a figurative constant, its length is one character.
3. When the ALL phrase is present:
   - One occurrence, or two or more contiguous occurrences, of *delim* (whether or not they are figurative constants) is treated as only one occurrence
   - One occurrence of *delim* is moved to *delim-dest* when there is a DELIMITER IN phrase
4. When any examination finds two contiguous delimiters, the current *dest-string* is filled with:
   - Spaces, if its class is alphabetic or alphanumeric
   - Zeros, if its class is numeric
5. *Delim* can contain any characters in the computer character set.
6. Each *delim* is one delimiter. When *delim* contains more than one character, all its characters must be in *src-string* (in contiguous positions and the given order) to qualify as a delimiter.
7. When the DELIMITED BY phrase contains an OR phrase, an "OR" condition exists between all occurrences of *delim*. Each *delim* is compared to *src-string*. If a match occurs, the character(s) in *src-string* is a single delimiter. No character(s) in *src-string* can be part of more than one delimiter.
8. Each *delim* applies to *src-string* in the order it appears in the UNSTRING statement.
9. When execution of the UNSTRING statement begins, the current receiving field is the first *dest-string*.
10. If there is a POINTER phrase, the string of characters in *src-string* is examined, beginning with the position indicated by *pointr*. Otherwise, examination begins with the leftmost character position.
11. If there is a DELIMITED BY phrase, examination proceeds to the right until the UNSTRING statement detects *delim*. (See General Rule 6.)

12. If there is no DELIMITED BY phrase, the number of characters examined equals the size of the current *dest-string*. However, if the sign of *dest-string* is defined as occupying a separate character position, UNSTRING examines one less character than the size of *dest-string*. If *dest-string* is a variable-length data item, its current size determines the number of characters examined.
13. If the UNSTRING statement reaches the end of *src-string* before detecting the delimiting condition, examination ends with the last character examined.
14. The characters examined (excluding *delim*) are:
    - Treated as an elementary alphanumeric data item
    - Moved to the current *dest-string* according to the MOVE statement rules
15. When there is a DELIMITER IN phrase, the delimiter is:
    - Treated as an elementary alphanumeric data item
    - Moved to *delim-dest* according to the MOVE statement rules

    If the delimiting condition is the end of *src-string*, *delim-dest* is space-filled.
16. The COUNT IN phrase causes the UNSTRING statement to:
    - Count the number of characters examined (excluding the delimiter)
    - Move the count to *countr* according to the elementary move rules
17. When there is a DELIMITED BY phrase, UNSTRING continues examining characters immediately to the right of the delimiter. Otherwise, examination continues with the character immediately to the right of the last one transferred.
18. After data transfers to *dest-string*, the next *dest-string* becomes the current receiving field.
19. The process described in General Rules 12 through 18 repeats until either:
    - There are no more characters in *src-string*
    - The last *dest-string* has been processed
20. The UNSTRING statement does not initialize *pointr* or *tally-ctr*. The program must set their initial values before executing the UNSTRING statement.
21. The UNSTRING statement adds one to *pointr* for each character it examines in *src-string*. When UNSTRING execution ends, *pointr* contains a value equal to its beginning value plus the number of characters the statement examined in *src-string*.
22. At the end of an UNSTRING statement with the TALLYING phrase, *tally-ctr* contains a value equal to its beginning value plus the number of *dest-string* fields the statement accessed.
23. An overflow condition can arise from either of these conditions:
    - When the UNSTRING statement begins, the value of *pointr* is less than one or greater than the number of characters in *src-string*.
    - During UNSTRING execution, all *dest-string* fields have been processed, and there are unexamined *src-string* characters.

24. When an overflow condition occurs, the UNSTRING operation ends. If there is an ON OVERFLOW phrase, *stment* executes. Otherwise, control passes to the end of the UNSTRING statement.
25. Subscripting or indexing evaluation for *src-string*, *pointr*, and *tally-ctr* occurs only once, just before the statement transfers any data.
26. Subscripting or indexing evaluation for *delim* occur only once, just before the statement examines *src-string* for its set of delimiters.
27. Subscripting or indexing evaluation for *dest-string*, *delim-dest*, and *countr* occur just before the statement transfers data to any of these data items.
28. Undefined results occur when operands overlap; that is, when sending fields and receiving fields share a part of their storage areas.

Additional References

Section 5.1.4 Scope of Statements
Section 5.9.15 MOVE Statement

Examples

The examples assume these data descriptions:

```
WORKING-STORAGE SECTION.
01      INMESSAGE PIC X(20).
01      THEDATE.
        03  THEYEAR  PIC XX JUST RIGHT.
        03  THEMONTH PIC XX JUST RIGHT.
        03  THEDAY   PIC XX JUST RIGHT.
01      HOLD-DELIM   PIC XX.
01      PTR PIC 99.
01      FIELD-COUNT PIC 99.
01      MONTH-COUNT PIC 99.
01      DAY-COUNT   PIC 99.
01      YEAR-COUNT  PIC 99.
```

1. With OVERFLOW phrase:

```
DISPLAY "Enter a date: " NO ADVANCING.
ACCEPT INMESSAGE.
UNSTRING INMESSAGE
  DELIMITED BY "-" OR "/" OR ALL " "
    INTO THEMONTH DELIMITER IN HOLD-DELIM
         THEDAY DELIMITER IN HOLD-DELIM
         THEYEAR DELIMITER IN HOLD-DELIM
  ON OVERFLOW MOVE ALL "0" TO THEDATE.
INSPECT THEDATE REPLACING ALL " " BY "0".
DISPLAY THEDATE.
```

(continued on next page)

Results

```
Enter a date: 6/13/80
800613

Enter a date: 6-13-80
800613

Enter a date: 6-13 80
800613

Enter a date: 6/13/80/2
000000

Enter a date: 1-2-3
030102
```

2. With POINTER and TALLYING phrases:

```
       DISPLAY "Enter two dates in a row: " NO ADVANCING.
       ACCEPT INMESSAGE.
       MOVE 1 TO PTR.
       PERFORM DISPLAY-TWO 2 TIMES.
       GO TO DISPLAYED-TWO.
DISPLAY-TWO.
       MOVE SPACES TO THEDATE.
       MOVE 0 TO FIELD-COUNT.
       UNSTRING INMESSAGE
         DELIMITED BY "-" OR "/" OR ALL " "
           INTO THEMONTH DELIMITER IN HOLD-DELIM
                THEDAY DELIMITER IN HOLD-DELIM
                THEYEAR DELIMITER IN HOLD-DELIM
           WITH POINTER PTR
           TALLYING IN FIELD-COUNT.
       INSPECT THEDATE REPLACING ALL " " BY "0".
       DISPLAY THEDATE "   " PTR "   " FIELD-COUNT.
DISPLAYED-TWO.
       EXIT.
```

Results

```
Enter two dates in a row: 6/13/80 8/15/80
800613   09   03
800815   21   03

Enter two dates in a row: 10 15 80-1 1 81
801015   10   03
810101   21   03

Enter two dates in a row: 6/13/80-12/31/80
800613   09   03
801231   21   03

Enter two dates in a row: 6/13/80-12/31
800613   09   03
001231   21   02

Enter two dates in a row: 6/13/80/12/31/80
800613   09   03
801231   21   03
```

3. With COUNT phrase:

```
       DISPLAY "Enter two dates in a row: " NO ADVANCING.
       ACCEPT INMESSAGE.
       MOVE 1 TO PTR.
       PERFORM DISPLAY-TWO 2 TIMES.
       GO TO DISPLAYED-TWO.
DISPLAY-TWO.
       MOVE SPACES TO THEDATE.
       MOVE 0 TO FIELD-COUNT MONTH-COUNT DAY-COUNT YEAR-COUNT.
       UNSTRING INMESSAGE
         DELIMITED BY "-" OR "/" OR ALL " "
         INTO THEMONTH DELIMITER IN HOLD-DELIM COUNT MONTH-COUNT
              THEDAY DELIMITER IN HOLD-DELIM COUNT DAY-COUNT
              THEYEAR DELIMITER IN HOLD-DELIM COUNT YEAR-COUNT
           WITH POINTER PTR
           TALLYING IN FIELD-COUNT.
       INSPECT THEDATE REPLACING ALL " " BY "0".
       DISPLAY THEDATE "   " PTR "   " FIELD-COUNT
       " : " MONTH-COUNT "-" DAY-COUNT "-" YEAR-COUNT.
DISPLAYED-TWO.
       EXIT.
```

Results

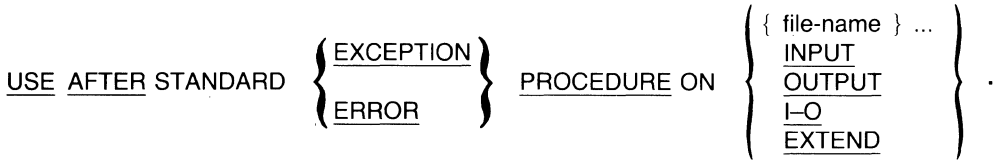
```
Enter two dates in a row: 6/13/80 8/15/80
800613   09   03 : 01-02-02
800815   21   03 : 01-02-02

Enter two dates in a row: 10 15 80-1 1 81
801015   10   03 : 02-02-02
810101   21   03 : 01-01-02

Enter two dates in a row: 6/13/80-12/31/80
800613   09   03 : 01-02-02
801231   21   03 : 02-02-02

Enter two dates in a row: 6/13/80-12/31
800613   09   03 : 01-02-02
001231   21   02 : 02-02-00

Enter two dates in a row: 6/13/80/12/31/80
800613   09   03 : 01-02-02
801231   21   03 : 02-02-02
```

5.9.31 USE Statement

Function

The USE statement specifies Declarative procedures to handle input-output errors. These procedures supplement the standard procedures in the COBOL-81 Object Time System (OTS) and PDP-11 Record Management Services (RMS-11).

General Format



file-name
is the name of a file connector described in a file description entry in a Data Division. It cannot refer to a sort or merge file.

Syntax Rules

1. A USE statement can be used only in a sentence immediately after a section header in the Procedure Division Declaratives area. It must be the only statement in the sentence. The rest of the section can contain zero, one, or more paragraphs to define the Declarative procedures.
2. The USE statement itself does not execute. It defines the conditions that cause execution of the Declarative.
3. ERROR and EXCEPTION are equivalent and interchangeable.

General Rules

1. A Declarative executes automatically:
   - After standard input-output error processing end
   - When an invalid key or at end condition results from an input-output statement that has no INVALID KEY or AT END clause
2. If there is an applicable USE AFTER EXCEPTION procedure, it executes whenever an input or output condition occurs that would result in a nonzero value in a FILE STATUS data item. However, it does not execute if: (a) the condition is invalid key and there is an INVALID KEY phrase, or (b) the condition is at end, and there is an AT END phrase.
3. A procedure in the Declarative Section cannot refer to a procedure that is not in the Declarative Section, and vice versa.
4. After a Declarative executes, control returns to the next executable statement in the invoking routine, if one is defined. Otherwise, control transfers according to the rules for explicit and implicit transfers of control.

5. One input-output error cannot cause more than one USE AFTER EXCEPTION procedure to execute.
6. More than one USE AFTER EXCEPTION procedure can apply to an input-output operation when there is one procedure for *file-name* and another for the applicable open mode. In this case, only the procedure for *file-name* executes.
7. If an input-output error occurs and there is no applicable USE AFTER EXCEPTION procedure, the image terminates abnormally unless the condition is invalid key or at end and the input-output statement has the appropriate phrase.
8. One USE AFTER EXCEPTION procedure can invoke another. However, a USE AFTER EXCEPTION procedure must return control to the routine that invoked it before it can be invoked again.

Additional References

Section 5.2	Transfer of Program Flow
Part IV of the COBOL-81 User's Guide for your system	Refer to the chapter on file I-O exception conditions handling

Example

(The notes following this example explain execution of the USE procedures shown.)

```
PROCEDURE DIVISION.
DECLARATIVES.
000-FILEA-PROBLEM SECTION.
    USE AFTER STANDARD ERROR PROCEDURE ON FILEA.
001-PROCA.
    IF FILEA-STATUS ...
010-ALL-EXTEND-PROBLEM SECTION.
    USE AFTER EXCEPTION PROCEDURE ON EXTEND.
011-PROCA.
    DISPLAY ...
020-I-O-PROBLEM SECTION.
    USE AFTER ERROR PROCEDURE ON I-O.
021-PROCA.
    DISPLAY ...
END DECLARATIVES.
```

Notes

1. If any input-output statement for FILEA results in an error, 000-FILEA-PROBLEM executes.
2. If an error occurs because of an input-output statement for any file open in the extend mode except FILEA, 010-ALL-EXTEND-PROBLEM executes.
3. If an error occurs because of an input-output statement for any file open in the I-O mode except FILEA, 020-I-O-PROBLEM executes.

5.9.32 WRITE Statement

Function

The WRITE statement releases a logical record to an output or input-output file. It can also position lines vertically on a logical page.

General Format

Format 1

```
WRITE rec-name [ FROM src-item ]

[ { BEFORE } ADVANCING { advance-num [ LINE ] }  ]
  { AFTER  }           { PAGE        [ LINES ] }

[ AT { END-OF-PAGE } stment ]
     { EOP         }
```

Format 2

```
WRITE rec-name [ FROM src-item ] [ INVALID KEY stment ]
```

rec-name
: is the name of a logical record described in the Data Division File Section. It cannot be qualified. The logical record cannot be in a sort-merge file description entry.

src-item
: is the identifier of the data item that contains the data.

advance-num
: is an integer or the identifier of an unsigned data item described as an integer. Its value can be zero.

stment
: is an imperative statement.

Syntax Rules

1. Format 1 must be used for sequential files.
2. Format 2 must be used for relative and indexed files.
3. If there is an END-OF-PAGE phrase, the file description entry containing *rec-name* must have a LINAGE clause.

4. The words END-OF-PAGE and EOP are equivalent.
5. In Format 2, there must be an INVALID KEY phrase if there is no applicable USE AFTER EXCEPTION procedure for the file.
6. *Rec-name* and *scr-item* must not refer to the same storage area.

General Rules

All Files

1. The record is no longer available in *rec-name* after a WRITE statement successfully executes. However, if the associated file-name is in a SAME RECORD AREA clause, the record is available in *rec-name*. In this case, the record is also available in the record areas of other file-names in the same SAME RECORD AREA clause.
2. Rules for the FROM phrase appear in Section 5.7.3, FROM Option.
3. For mass storage files, the WRITE statement does not affect the Next Record Pointer.
4. The WRITE statement updates the value of the FILE STATUS data item for the file.
5. A file's maximum record size is set when it is created. It cannot be changed later.
6. On a mass storage device, the number of characters required to store a logical record in a file depends on file organization and record type.
7. WRITE statement execution releases a logical record to PDP-11 Record Management Services (RMS-11).

Sequential Files

8. The file must be open in the OUTPUT, I-O, or EXTEND mode when the WRITE statement executes.
9. The sequence of records in a sequential file is set by the order of WRITE statement executions that create the file. The relationship does not change, except when records are added to the end of the file.
10. For a sequential file open in the extend mode, the WRITE statement adds records to the end of the file as if the file were open in the output mode. If the file has records, the first record written after execution of an OPEN statement with the EXTEND phrase is the successor of the file's last record.
11. When a program tries to write beyond a sequential file's externally defined boundaries, an exception condition exists as follows:
    - The contents of the record area are unaffected.
    - The value of the FILE STATUS data item for the file indicates a boundary violation.
    - If a USE AFTER EXCEPTION procedure applies to the file, it executes.
    - If there is no applicable USE AFTER EXCEPTION procedure, the program terminates abnormally.

12. If the end of a reel/unit is recognized, and the WRITE does not exceed the externally defined file boundaries:
    - A reel/unit swap occurs
    - The Current Volume Pointer points to the file's next reel/unit
13. The ADVANCING and END-OF-PAGE phrases control the vertical positioning of each line on a logical representation of a printed page. If there is no ADVANCING phrase, the default is AFTER ADVANCING 1 LINE.

    If there is an ADVANCING phrase:
    - The WRITE statement advances the logical page by the number of lines specified by the value of *advance-num*
    - The BEFORE phrase causes the statement to write the line before advancing the logical page
    - The AFTER phrase causes the statement to write the line after advancing the logical page
    - The PAGE phrase writes the line before or after (depending on the phrase) positioning the device to the next logical page

      If the associated file description entry has a LINAGE clause, the device is positioned to the first line that can be written on the next logical page. (The LINAGE clause specifies which line is first.)

      If there is no associated LINAGE clause, the device is positioned to the first line on the next logical page.

      If page has no meaning for the associated device, PAGE is the same as ADVANCING 1 LINE. However, the BEFORE and AFTER phrases affect operation sequence.
14. If the program reaches end of the logical page during execution of a WRITE statement with the END-OF-PAGE phrase, *stment* executes. The LINAGE clause associated with the file specifies the logical end.
15. An end-of-page condition is reached when a WRITE statement with the END-OF-PAGE phrase causes printing or spacing in the footing area of the page body.

    This condition occurs when the WRITE causes the LINAGE-COUNTER to equal or exceed the value in the LINAGE clause FOOTING phrase. *Stment* then executes after *rec-name* is written to the file.
16. An automatic page overflow condition occurs when the page body cannot fully accommodate a WRITE statement (with or without the END-OF-PAGE phrase).

    This condition occurs when WRITE statement execution would cause the LINAGE-COUNTER to exceed the number of lines in the page body specified in the LINAGE clause. When this happens, the line is presented on the logical page before or after (depending on the phrase) device positioning. The device is positioned to the first line that can be written on the next logical page (as described in the LINAGE clause). *Stment* then executes after *rec-name* is written to the file.

17. If there is no LINAGE clause FOOTING phrase, the WRITE statement operates as if the FOOTING phrase value were the same as the number of lines on the logical page. That is, the end-of-page condition occurs when the WRITE statement causes the LINAGE-COUNTER to equal the number of lines on the logical page.
18. If there is a FOOTING phrase, and a WRITE statement would cause the LINAGE-COUNTER to exceed both the number of lines in a logical page and the value in the LINAGE clause FOOTING phrase, the WRITE statement operates as if there were no FOOTING phrase.

Relative Files

19. The file must be open in the OUTPUT or I-O mode when the WRITE statement executes.
20. When a relative file with sequential access mode is open in the output mode, the WRITE statement releases a record to RMS-11. The first record has a relative record number of 1. Subsequent records have relative record numbers of 2, 3, 4, and so on. If *rec-name* has an associated RELATIVE KEY data item, the WRITE places the relative record number of the released record into it.
21. When a relative file with random or dynamic access mode is open in the output mode, the program must place a value in the RELATIVE KEY data item before executing the WRITE statement. The value is the relative record number to associate with the record in *rec-name*. The WRITE statement releases the record to RMS-11.
22. When a relative file is open in the I-O mode and the access mode is random or dynamic, the program must place a value in the RELATIVE KEY data item before executing the WRITE statement. The value is the relative record number to associate with the record in *rec-name*. Executing a Format 2 WRITE statement releases the record to RMS-11.
23. The invalid key condition exists when either:
    - The access mode is random or dynamic, and the RELATIVE KEY data item specifies a record that already exists in the file
    - The program tries to write a record beyond the externally defined file boundaries.
24. When the program detects an invalid key condition, WRITE statement execution is unsuccessful. The following results occur:
    - The contents of the current record area are not affected.
    - The WRITE statement sets the FILE STATUS data item for the file to indicate the cause of the condition.
    - Program execution continues according to the rules for the invalid key condition.

Indexed Files

25. The file must be open in the OUTPUT or I-O mode when the WRITE statement executes.
26. Executing a Format 2 WRITE statement releases a record to RMS-11. The contents of the record keys enable later record access based on any defined key.

27. The value of the prime record key must be unique in the file's records.
28. The program must set the value of the prime record key data item before executing the WRITE statement.
29. If the file is open in the sequential access mode, the program must release records in ascending order of prime record key values.
30. If the file is open in the random or dynamic access mode, the program can release records in any order.
31. When the file description entry has an ALTERNATE RECORD KEY clause, the alternate record key value can be nonunique only if there is a DUPLICATES phrase. When a program later accesses these records sequentially, the retrieval order is the same as the order in which they were written.
32. The invalid key condition is caused by any of the following:
    - The file is open in the sequential access mode *and* in the output mode, and the prime record key value is not greater than the prime record key value of the previous record.
    - The file is open in the output or I-O mode, and the prime record key value duplicates an existing record's prime record key value.
    - The file is open in the output or I-O mode, and the value of an alternate record key (for which duplicates are not allowed) duplicates the value of the corresponding data item in an existing record.
    - The program tries to write a record beyond the externally defined file boundaries.
33. When the program detects an invalid key condition, WRITE statement execution is unsuccessful. The following results occur:
    - The contents of the current record area are not affected.
    - The WRITE statement sets the FILE STATUS data item for the file to indicate the cause of the condition.
    - Program execution continues according to the rules for the invalid key condition.
34. If there is an applicable USE AFTER EXCEPTION procedure, it executes whenever an input or output condition occurs that would result in a nonzero value in a FILE STATUS data item. However, it does not execute if: (a) the condition is invalid key, and (b) there is an INVALID KEY phrase.

Technical Note

WRITE statement execution can result in the FILE STATUS data item values summarized in the following table:

FILE STATUS	File Organization	Access Method	Meaning
00	All	All	Successful
02	Ind	All	Created duplicate Alternate Key
21	Ind	Seq	Attempted non-ascending key value (invalid key)
22	Ind, Rel	All	Duplicate key (invalid key)
24	Ind, Rel	All	Boundary violation (invalid key)
34	Seq	Seq	Boundary violation
92	Ind, Rel	All	Record locked by another program
94	All	All	File not open, or incompatible open mode
30	All	All	All other permanent errors

Additional References

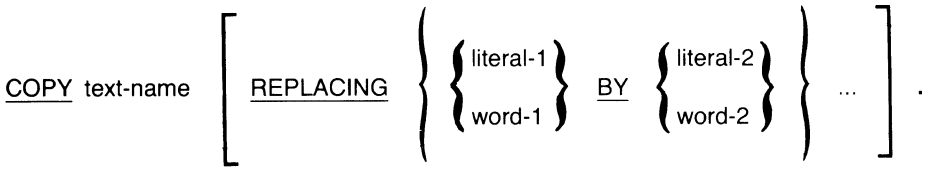
Section 3.2.2	I-O-CONTROL Paragraph, SAME Clause
Section 4.1.1.3	Multiple Record Descriptions
Section 5.7	I-O Status
Section 5.7.1	INVALID KEY Phrase
Section 5.7.3	FROM Option
Section 5.9.17	OPEN Statement

Chapter 6
The Copy Statement

Function

The COPY statement includes text from a library file in a COBOL source program.

General Format



text-name
: is the name of a COBOL library file available during compilation. It must be a nonnumeric literal or a user-defined word representing the complete or partial specification of the library file. If it is a user-defined word, the compiler treats it as if it were enclosed in quotation marks.

literal-1
word-1
: are arguments that the compiler compares against character-strings in the library text.

literal-2
word-2
: are replacement items that the compiler inserts into the source program.

Syntax Rules

1. A COPY statement can be used anywhere that a character-string or separator (other than a closing quotation mark) can be used in a program.
2. A space must precede the word COPY.
3. The COPY statement must be terminated by the separator period.
4. *Word-1* or *word-2* can be any single COBOL word.

General Rules

1. The COPY statement causes the compiler to copy the library text associated with *text-name* into the program. The library text logically replaces the COPY statement, beginning with the word COPY and ending with the separator period (inclusive).
2. The compiler evaluates the source program after processing all COPY statements.
3. The COPY statement does not change the original source program text file. Rather, it causes the compiler to create a temporary source file that is a composite of the original source text and the library text that replaces the COPY statement. The compiler performs syntax evaluation on this temporary source file.
4. Library text must follow the source reference format rules. Library text and source program formats must be the same; that is, both must be ANSI format or both must be terminal format.
5. If there is no REPLACING phrase, the compiler copies the library text without change.
6. If there is a REPLACING phrase, the compiler changes the library text as it copies it. The compiler replaces each successfully matched occurrence of an argument (*literal-1* or *word-1*) in the library text by the corresponding replacement item (*literal-2* or *word-2*).
7. If there is a REPLACING phrase, COPY statement execution proceeds as follows:
   a. The compiler compares the first character-string in the library text with each REPLACING phrase argument until (a) a match occurs or (b) there are no more arguments.
   b. If no match occurs after the compiler compares all REPLACING phrase arguments with the character-string, the compiler copies the character-string from the library text into the source program.

      The compiler then resumes the matching operation, using the *next* character-string in the library text, and matching it to each REPLACING phrase argument, in turn.
   c. If a match occurs between a REPLACING phrase argument and a library text character-string, then the compiler copies the replacement item (*word-2* or *literal-2*) into the source program.

      The compiler then resumes the matching operation, as described in b.
   d. The copy operation is complete when each successive character-string in the library text has been either copied to the source program or replaced in the source program by *word-2* or *literal-2*.
8. The compiler copies comment lines or blank lines in the library text into the source program unchanged (see Example 1).

   Comment lines that are contained within the bounds of *literal-1* are copied, but they appear immediately before the literal in the resultant source file.

9. The source program cannot contain a COPY statement after the compiler processes a COPY statement. In other words:
   - The library text cannot contain a COPY statement unless the replacement operation changes the word COPY in the resultant source file.
   - The replacement item in the REPLACING phrase must not insert a COPY statement into the resultant source file.
10. When the compiler copies a character-string from the library text, it places the character-string in the source program beginning in the same area as in the library text. That is, a character-string that begins in Area A in the library text begins in Area A of the source program after the copy operation. Similarly, a character-string that begins in Area B in the library text begins somewhere in Area B of the source program.

    Tab and space characters are copied to the source program as they appear in the library text. However, commas and semicolons are not copied to the source program.

Technical Notes

1. When the COPY statement executes, Record Management Services (RMS-11):
   - Removes leading and trailing spaces and tab characters from the file specification
   - Translates lowercase letters in the file specification to uppercase
2. The default file type for *text-name* is LIB. For example, "CUSTFILE" becomes "CUSTFILE.LIB".

Examples

The examples that follow copy library text from two library files:

Contents of "CUSTFILE.LIB":

```
01⇥CUSTOMER-REC.
⇥03  CUST-REC-KEY⇥PIC X(03) VALUE "KEY".
⇥03  CUST-NAME⇥PIC X(25).
⇥03  CUST-ADDRESS.
⇥    05  CUST-CUST-STREET⇥PIC X(20).
⇥    05  CUST-CITY⇥PIC X(20).
⇥    05  CUST-STATE⇥PIC XX.
⇥    05  CUST-ZIP⇥PIC 9(5).
* THE COMPILER IGNORES COMMENT LINES AND BLANK LINES

* FOR MATCHING PURPOSES
⇥03  CUST-ORDERS OCCURS XYZ TIMES.
⇥    05  CUST-ORDER⇥PIC 9(6).
⇥    05  CUST-ORDER-DATE⇥PIC 9(6).
```

Contents of "CPROC01.LIB":

```
⇥ADD CUST-ORDER-AMT (X) TO TOTAL-ORDERS.
⇥COMPUTE AVERAGE-ORDER = (TOTAL-ORDERS - CANCELLED-ORDERS)
⇥  / NUMBER-ORDERS.
⇥MOVE CUST-REC-KEY
⇥  OF CUSTOMER-REC TO CUST-ID (X).
⇥MOVE CUST-REC-KEY
⇥  OF KEY-HOLD TO NEW-KEY.
```

The original source program text is in lowercase. The extracts from resulting source program listings show how replacements occur.

1. No REPLACING phrase:

   (The compiler copies the library text without change. In this example, syntax errors result from invalid library text.)

```
1          identification division.
2          program-id. cust01.
3          data division.
4          working-storage section.
5          copy custfile.
6L          01  CUSTOMER-REC.
7L              03  CUST-REC-KEY         PIC X(03) VALUE "KEY".
8L              03  CUST-NAME   PIC X(25).
9L              03  CUST-ADDRESS.
10L                 05  CUST-CUST-STREET         PIC X(20).
11L                 05  CUST-CITY                PIC X(20).
12L                 05  CUST-STATE               PIC XX.
13L                 05  CUST-ZIP                 PIC 9(5).
14L        * THE COMPILER IGNORES COMMENT LINES AND BLANK LINES
15L
16L        * FOR MATCHING PURPOSES
17L             03  CUST-ORDERS OCCURS XYZ TIMES.

                                          ^        ^
*** F  223   This is not a valid clause in a record description.
*** I  501   *Compilation resumed at this point.

18L                   05  CUST-ORDER      PIC 9(6).
19L                   05  CUST-ORDER-DATE PIC 9(6).
```

2. Replacing a word by a literal:

```
22           copy custfile replacing xyz by 6.
23L          01  CUSTOMER-REC.
24L              03  CUST-REC-KEY         PIC X(03) VALUE "KEY".
25L              03  CUST-NAME   PIC X(25).
26L              03  CUST-ADDRESS.
27L                  05  CUST-CUST-STREET         PIC X(20).
28L                  05  CUST-CITY        PIC X(20).
29L                  05  CUST-STATE       PIC XX.
30L                  05  CUST-ZIP         PIC 9(5).
31L         * THE COMPILER IGNORES COMMENT LINES AND BLANK LINES
32L
33L         * FOR MATCHING PURPOSES
34L              03  CUST-ORDERS OCCURS 6   TIMES.
35L                  05  CUST-ORDER       PIC 9(6).
36L                  05  CUST-ORDER-DATE  PIC 9(6).
```

3. Copying Procedure Division text without change:

```
37              move cust-rec-key
38                of key-hold to new-key, copy cproc01.
39L             ADD CUST-ORDER-AMT (X) TO TOTAL-ORDERS.
40L             COMPUTE AVERAGE-ORDER = (TOTAL-ORDERS - CANCELLED-ORDERS)
41L               / NUMBER-ORDERS.
42L             MOVE CUST-REC-KEY
43L               OF CUSTOMER-REC TO CUST-ID (X).
44L             MOVE CUST-REC-KEY
45L               OF KEY-HOLD TO NEW-KEY.
46              add 1 to cust-transactions.
```

4. Replacing a word by a longer item:

```
48              move cust-rec-key
49                of key-hold to new-key, copy cproc01
50              replacing x by cust-sub.
51L             ADD CUST-ORDER-AMT (
52L                                  cust-sub
53L                                   ) TO TOTAL-ORDERS.
54L             COMPUTE AVERAGE-ORDER = (TOTAL-ORDERS - CANCELLED-ORDERS)
55L               / NUMBER-ORDERS.
56L             MOVE CUST-REC-KEY
57L               OF CUSTOMER-REC TO CUST-ID (
58L                                           cust-sub
59L                                            ).
60L             MOVE CUST-REC-KEY
61L               OF KEY-HOLD TO NEW-KEY.
62              add 1 to cust-transactions.
```

5. COPY statement in the middle of a source program line:

   (This example shows the appearance of the source listing when the COPY statement both follows and precedes other text on the same line.

   In the source program, line 75 is:

```
add 1 to x, copy cproc01, subtract 1 from x.
```

   This is the resulting source program listing.

```
75              add 1 to x, copy cproc01.
76L             ADD CUST-ORDER-AMT (X) TO TOTAL-ORDERS.
77L             COMPUTE AVERAGE-ORDER = (TOTAL-ORDERS - CANCELLED-ORDERS)
78L               / NUMBER-ORDERS.
79L              MOVE CUST-REC-KEY
80L                OF CUSTOMER-REC TO CUST-ID (X).
81L              MOVE CUST-REC-KEY
82L                OF KEY-HOLD TO NEW-KEY.
83                                       subtract 1 from x.
```

Appendix A
COBOL-81/VAX–11 COBOL Reserved Words

Some of the following reserved words pertain to features only available in VAX–11 COBOL. However, to ensure upward compatibility between the two COBOL compilers, all words reserved in VAX–11 COBOL programs are also reserved in COBOL-81 programs.

ACCEPT
ACCESS
ADD
ADVANCING
AFTER
ALL
ALLOWING
ALPHABET
ALPHABETIC
ALPHABETIC-LOWER
ALPHABETIC-UPPER
ALPHANUMERIC
ALPHANUMERIC-EDITED
ALSO
ALTER
ALTERNATE
AND
ANY
APPLY
ARE
AREA
AREAS
ASCENDING
ASSIGN
AT
AUTHOR

BATCH
BEFORE
BEGINNING
BELL
BIT
BITS
BLANK
BLINKING
BLOCK
BOLD
BOOLEAN
BOTTOM
BY

CALL
CANCEL
CD
CF
CH
CHARACTER
CHARACTERS
CLOCK-UNITS
CLOSE
COBOL
CODE
CODE-SET
COLLATING
COLUMN
COMMA
COMMIT
COMMON
COMMUNICATION
COMP
COMP-1
COMP-2
COMP-3
COMP-4
COMP-5
COMP-6
COMPUTATIONAL
COMPUTATIONAL-1
COMPUTATIONAL-2
COMPUTATIONAL-3
COMPUTATIONAL-4
COMPUTATIONAL-5
COMPUTATIONAL-6
COMPUTE
CONCURRENT
CONFIGURATION
CONNECT
CONTAINS
CONTENT
CONTINUE
CONTROL

CONTROLS
CONVERSION
CONVERTING
COPY
CORR
CORRESPONDING
COUNT
CURRENCY
CURRENT

DATA
DATE
DATE-COMPILED
DATE-WRITTEN
DAY
DAY-OF-WEEK
DB
DB-ACCESS-CONTROL-KEY
DB-CONDITION
DB-CURRENT-RECORD-ID
DB-CURRENT-RECORD-NAME
DB-EXCEPTION
DB-RECORD-NAME
DB-SET-NAME
DB-STATUS
DE
DEBUG-CONTENTS
DEBUG-ITEM
DEBUG-LENGTH
DEBUG-LINE
DEBUG-NAME
DEBUG-NUMERIC-CONTENTS
DEBUG-SIZE
DEBUG-START
DEBUG-SUB
DEBUG-SUB-1
DEBUG-SUB-2
DEBUG-SUB-3
DEBUG-SUB-ITEM
DEBUG-SUB-N

DEBUG-SUB-NUM
DEBUGGING
DECIMAL-POINT
DECLARATIVES
DEFAULT
DELETE
DELIMITED
DELIMITER
DEPENDING
DESCENDING
DESCRIPTOR
DESTINATION
DETAIL
DICTIONARY
DISABLE
DISCONNECT
DISPLAY
DISPLAY-6
DISPLAY-7
DISPLAY-9
DIVIDE
DIVISION
DOWN
DUPLICATE
DUPLICATES
DYNAMIC

ECHO
EGI
ELSE
EMI
EMPTY
ENABLE
END
END-ACCEPT
END-ADD
END-CALL
END-COMMIT
END-COMPUTE
END-CONNECT
END-DELETE
END-DISCONNECT
END-DIVIDE
END-ERASE
END-EVALUATE
END-FETCH
END-FIND
END-FINISH
END-FREE
END-GET
END-IF
END-KEEP
END-MODIFY
END-MULTIPLY
END-OF-PAGE
END-PERFORM
END-READ
END-READY
END-RECEIVE
END-RECONNECT
END-RETURN
END-REWRITE
END-ROLLBACK
END-SEARCH
END-START
END-STORE
END-STRING
END-SUBTRACT
END-UNSTRING
END-WRITE
ENDING
ENTER
ENVIRONMENT
EOP
EQUAL
EQUALS
ERASE
ERROR
ESI
EVALUATE
EVERY
EXCEEDS
EXCEPTION
EXCLUSIVE
EXIT
EXOR
EXTEND
EXTERNAL

FAILURE
FALSE
FD
FETCH
FILE
FILE-CONTROL
FILLER
FINAL
FIND
FINISH
FIRST
FOOTING
FOR
FREE
FROM

GENERATE
GET
GIVING
GLOBAL
GO
GREATER
GROUP

HEADING
HIGH-VALUE
HIGH-VALUES
I-O
I-O-CONTROL
IDENTIFICATION
IF
IN
INCLUDING
INDEX
INDEXED
INDICATE
INITIAL
INITIALIZE
INITIATE
INPUT
INPUT-OUTPUT
INSPECT
INSTALLATION
INTO
INVALID
IS

JUST
JUSTIFIED

KEEP
KEY

LABEL
LAST
LD
LEADING
LEFT
LENGTH
LESS
LIMIT
LIMITS
LINAGE
LINAGE-COUNTER
LINE
LINE-COUNTER
LINES
LINKAGE
LOCALLY
LOCK
LOW-VALUES

MATCHES
MEMBER
MEMBERSHIP
MEMORY
MERGE
MESSAGE
MODE
MODIFY
MODULES
MOVE
MULTIPLE
MULTIPLY

NATIVE
NEGATIVE
NEXT
NO
NON-NULL
NOT
NULL
NUMBER
NUMERIC
NUMERIC-EDITED

OBJECT-COMPUTER
OCCURS
OF
OFF
OFFSET
OMITTED
ON
ONLY
OPEN
OPTIONAL
OR
ORDER
ORGANIZATION
OTHER
OTHERS
OUTPUT
OVERFLOW
OWNER

PADDING
PAGE
PAGE-COUNTER
PERFORM
PF
PH
PIC
PICTURE
PLUS
POINTER
POSITION
POSITIVE
PRINTING
PRIOR
PROCEDURE
PROCEDURES
PROCEED
PROGRAM
PROGRAM-ID
PROTECTED
PURGE

QUEUE
QUOTE
QUOTES

RANDOM
RD
READ
READERS
READY
REALM
REALMS
RECEIVE
RECONNECT
RECORD
RECORD-NAME
RECORDS
REDEFINES
REEL
REFERENCE
REFERENCE-MODIFIER
REFERENCES
REGARDLESS
RELATIVE
RELEASE
REMAINDER
REMOVAL
RENAMES
REPLACE
REPLACING
REPORT
REPORTING
REPORTS
RERUN
RESERVE
RESET
RETAINING
RETRIEVAL
RETURN
REVERSED
REWIND
REWRITE
RF
RH
RIGHT
RMS-FILENAME
RMS-STS
RMS-STV
ROLLBACK
ROUNDED
RUN

SAME
SCREEN
SD
SEARCH
SECTION
SECURITY
SEGMENT
SEGMENT-LIMIT
SELECT
SEND
SENTENCE
SEPARATE
SEQUENCE
SEQUENCE-NUMBER
SEQUENTIAL
SET
SETS
SIGN
SIZE
SORT
SORT-MERGE
SOURCE
SOURCE-COMPUTER
SPACE
SPACES
SPECIAL-NAMES
STANDARD
STANDARD-1
STANDARD-2
START
STATUS
STOP
STORE
STRING
SUB-QUEUE-1
SUB-QUEUE-2
SUB-QUEUE-3
SUB-SCHEMA
SUBTRACT
SUCCESS
SUM
SUPPRESS
SYMBOLIC
SYNC
SYNCHRONIZED

TABLE
TALLYING
TAPE
TENANT
TERMINAL
TERMINATE
TEST
TEXT
THAN
THEN
THROUGH
THRU
TIME
TIMES
TO
TOP
TRAILING
TRUE
TYPE

UNDERLINED
UNEQUAL
UNIT
UNLOCK
UNSTRING
UNTIL
UP
UPDATE

UPDATERS
UPON
USAGE
USAGE-MODE
USE
USING

VALUE
VALUES
VARYING

WAIT
WHEN
WHERE
WITH
WITHIN
WORDS
WORKING-STORAGE
WRITE
WRITERS

ZERO
ZEROES
ZEROS

Appendix B
Computer Character Set

Table B-1 shows the characters of the computer (ASCII) character set with each character's decimal and octal equivalent.

Characters belonging to set "C" constitute the COBOL character set. Set "L" contains those characters that can appear in nonnumeric literals. The characters in set "X" delimit lines of the source text.

Table B-1: ASCII Character Set

Decimal	Octal	Character	Set	Decimal	Octal	Character	Set
000	000	NUL	L	032	040	space	C, L
001	001	SOH	L	033	041	!	L
002	002	STX	L	034	042	"	C, L
003	003	ETX	L	035	043	#	L
004	004	EOT	L	036	044	$	C, L
005	005	ENQ	L	037	045	%	L
006	006	ACK	L	038	046	&	L
007	007	BEL	L	039	047	'	L
008	010	BS	L	040	050	(	C, L
009	011	HT	C	041	051	)	C, L
010	012	LF	X	042	052	*	C, L
011	013	VT	X	043	053	+	C, L
012	014	FF	X	044	054	,	C, L
013	015	CR	X	045	055	–	C, L
014	016	SO	L	046	056	.	C, L
015	017	SI	L	047	057	/	C, L
016	020	DLE	L	048	060	0	C, L
017	021	DC1	L	049	061	1	C, L
018	022	DC2	L	050	062	2	C, L
019	023	DC3	L	051	063	3	C, L
020	024	DC4	L	052	064	4	C, L
021	025	NAK	L	053	065	5	C, L
022	026	SYN	L	054	066	6	C, L
023	027	ETB	L	055	067	7	C, L
024	030	CAN	L	056	070	8	C, L
025	031	EM	L	057	071	9	C, L
026	032	SUB	L	058	072	:	L
027	033	ESC	L	059	073	;	C, L
028	034	FS	L	060	074	<	C, L
029	035	GS	L	061	075	=	C, L
030	036	RS	L	062	076	>	C, L
031	037	US	L	063	077	?	L

(continued on next page)

Table B-1: ASCII Character Set (Cont.)

Decimal	Octal	Character	Set	Decimal	Octal	Character	Set
064	100	@	L	096	140	`	L
065	101	A	C, L	097	141	a	L
066	102	B	C, L	098	142	b	L
067	103	C	C, L	099	143	c	L
068	104	D	C, L	100	144	d	L
069	105	E	C, L	101	145	e	L
070	106	F	C, L	102	146	f	L
071	107	G	C, L	103	147	g	L
072	110	H	C, L	104	150	h	L
073	111	I	C, L	105	151	i	L
074	112	J	C, L	106	152	j	L
075	113	K	C, L	107	153	k	L
076	114	L	C, L	108	154	l	L
077	115	M	C, L	109	155	m	L
078	116	N	C, L	110	156	n	L
079	117	O	C, L	111	157	o	L
080	120	P	C, L	112	160	p	L
081	121	Q	C, L	113	161	q	L
082	122	R	C, L	114	162	r	L
083	123	S	C, L	115	163	s	L
084	124	T	C, L	116	164	t	L
085	125	U	C, L	117	165	u	L
086	126	V	C, L	118	166	v	L
087	127	W	C, L	119	167	w	L
088	130	X	C, L	120	170	x	L
089	131	Y	C, L	121	171	y	L
090	132	Z	C, L	122	172	z	L
091	133	[	L	123	173	{	L
092	134	\	L	124	174	|	L
093	135	]	L	125	175	}	L
094	136	^	L	126	176	~	L
095	137	_	L	127	177	DEL	L

Appendix C
FILE STATUS Key Values

This appendix summarizes the values that can appear in FILE STATUS data items. The entry for each statement describes specific causes for each condition.

Technical Note

If you use the /FIPS:74 switch when compiling your program, values for some FILE STATUS data items are different than those in this table. Part I of the COBOL-81 User's Guide for your system discusses the /FIPS:74 switch. Refer to Appendix D, MCR (or CCL) Commands for COBOL-81.

FILE STATUS	Input-Output Statements	File Organization	Access Method	Meaning
00	All	All	All	Successful
02	REWRITE WRITE	Ind	All	Created duplicate alternate key
05	OPEN	Seq	Seq	Optional file not present
13	READ	All	Seq	No next logical record (at end)
15	READ	Seq	Seq	Optional file not present (at end)
16	READ	All	Seq	No valid next record (at end)
21	REWRITE	Ind	Seq	Primary key changed after READ
21	WRITE	Ind	Seq	Attempted nonascending key value (invalid key)
22	REWRITE	Ind	All	Duplicate alternate key (invalid key)
22	WRITE	Ind,Rel	All	Duplicate key (invalid key)
23	DELETE READ REWRITE START	Ind,Rel	Ran	Record not in file (invalid key)
24	WRITE	Ind,Rel	All	Boundary violation (invalid key)
30	All	All	All	All other permanent errors

(continued on next page)

FILE STATUS	Input-Output Statements	File Organization	Access Method	Meaning
34	WRITE	Seq	Seq	Boundary violation
90	READ	All	All	Record locked by another user; record is available in record area
91	OPEN	All	All	File locked by another program; record is not available
92	DELETE READ REWRITE START WRITE	All	All	Record locked by another program
93	DELETE REWRITE	All	Seq	No previous READ
94	CLOSE	All	All	File never opened or already closed
94	OPEN	All	All	File already open, or closed with lock
94	DELETE READ REWRITE START WRITE	All	All	File not open, or incompatible open mode
95	OPEN	All	All	No file space on device
96	OPEN	All	All	Same area busy
97	OPEN	All	All	File not found
98	CLOSE	All	All	Any other CLOSE error

Appendix D
Ensuring COBOL-81 Compatibility with VAX–11 COBOL

COBOL-81 has some incompatibilities with VAX–11 COBOL due to the differences in architecture between the PDP-11 and the VAX–11 computers. When you compile your COBOL-81 program using the /STA:VAX switch, the compiler issues a diagnostic message whenever it processes a definite or possible incompatibility with VAX–11 COBOL. If you plan to transfer a COBOL-81 program or its corresponding data files to a VMS system, you must resolve any VAX–11 COBOL incompatibilities flagged by the compiler.

Note

The /STA:VAX switch is available only when you use the C81 command. Part I of the COBOL-81 User's Guide for your system explains how to use this command and its options. Refer to Appendix D, Using MCR (or CCL) Commands for COBOL-81.

The following sections describe each incompatibility and suggest ways to design your program so that it is compatible with both COBOL products.

D.1 Size of INDEX Data Items

INDEX data items in COBOL-81 are two bytes long. In VAX–11 COBOL, they are four bytes long. Therefore, INDEX data items will always cause an incompatibility when they are stored in files. Such files are not directly transferable between COBOL-81 and VAX–11 COBOL.

This means that you can define USAGE IS INDEX items only for use within your program. You must:

- Describe items with the USAGE IS INDEX clause only in the Working-Storage Section of your program
- Not READ a record INTO or WRITE a record FROM records that you define in the Working-Storage Section when these records contain USAGE IS INDEX items.

D.2 Alignment of COMP Data Items

COBOL-81 aligns COMPUTATIONAL data items only on word boundaries whereas VAX–11 COBOL aligns COMPUTATIONAL items on the next available byte boundary.

There are two ways you can create files that contain COMP data items and that are still transferable between COBOL-81 and VAX–11 COBOL:

- Automatically, by writing the SYNCHRONIZED (or SYNC) clause for all COMP items in your program.
- Manually, by inserting FILLER data items before any items that: (1) are COMP items or groups containing COMP items, (2) you include in a record for a file, and (3) would align on an odd byte boundary using VAX–11 COBOL.

DIGITAL recommends that you specify the SYNCHRONIZED clause to ensure COMP item compatibility. This is by far the easiest method to use. The following discussion of record allocation is important only to those who choose the manual method.

When you use the SYNCHRONIZED clause for a COMP item, you specify a storage format that is common to both COBOL-81 and VAX–11 COBOL. In other words, both products allocate storage for a COMP SYNC item in the same way. The SYNCHRONIZED clause causes the compiler to insert fill bytes before a COMP item (and before any group item containing it) so that it aligns on a predetermined boundary. Boundaries are predetermined by the size range in which the item falls. Section 4.2.19, SYNCHRONIZED Clause, explains the correspondence between size and alignment for COMP SYNC items.

In most cases, the fill bytes inserted before COMP SYNC items will not have a significant effect on the size of a record or table in your program. Even large COMP items often align just before the boundary on which their COMP SYNC counterparts would begin. However, you might want to consider using the manual method if fill bytes cause your program to create a very large file or table, and your system has limited disk space.

If you use the manual method, remember that you must force VAX–11 COBOL to align on a word boundary even a COMP item in a Working-Storage Section record if you use that record when writing to or reading from a file.

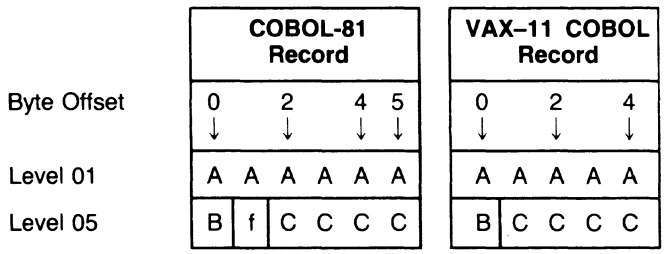
Example D-1 shows how you can change a simple record description so that both COBOL-81 and VAX–11 COBOL use the same storage format.

In the figures for examples, *f* represents an implicit fill byte and *F* represents a FILLER data item.

Example D-1: Changing a Simple Record to Ensure COMP item Compatibility

Incompatible record:

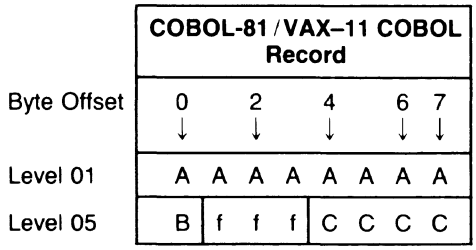
```
FD  FILE-1.
01  ITEM-A.
    03   ITEM-B              PIC X.

    03   ITEM-C              PIC 9(9) COMP.
```



(continued on next page)

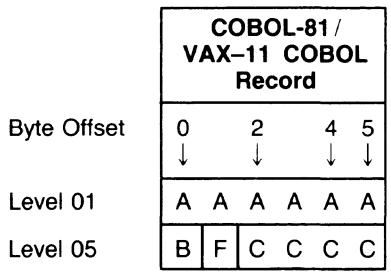
Compatible record using SYNCHRONIZED clause:

```
FD  FILE-2.
01  ITEM-A.
    03    ITEM-B              PIC X.
    03    ITEM-C              PIC 9(9) COMP SYNC.
```



Compatible record using FILLER insertion:

```
FD  FILE-3.
01  ITEM-A.
    03    ITEM-B              PIC X.
    03    FILLER              PIC X.
    03    ITEM-C              PIC 9(9) COMP.
```



FILE-1 specifies a record that cannot be transferred to a VAX–11 COBOL program because each COBOL compiler has a different storage format. COBOL-81 aligns ITEM-C OF FILE-1, a COMP data item, on a word boundary; however, VAX–11 COBOL aligns the same item on the next available byte boundary.

Both FILE-2 and FILE-3 specify records that can be transferred to a VAX–11 COBOL program. In FILE-2, this is achieved by including the SYNCHRONIZED clause for the COMP item, ITEM-C. In FILE-3, this is achieved manually by inserting a FILLER data item before ITEM-C.

ITEM-C is the same size in all files; however, its alignment requirements are different in each file. In FILE-1, one fill byte precedes ITEM-C so that it aligns on a word boundary. In FILE-2, three fill bytes precede ITEM-C because a nine-character COMP SYNC item must align either on the boundary where the record begins or on a 4-byte offset from that boundary. In FILE-2, the insertion of the FILLER data item ensures that ITEM-C naturally aligns on a word boundary and no fill bytes are required.

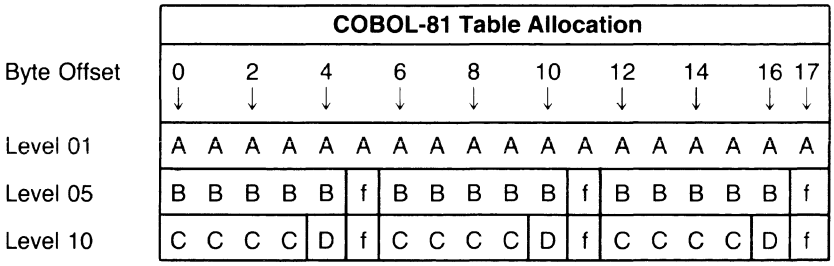
The size of the record, ITEM-A, is not the same in all files. ITEM-A of FILE-1 occupies six bytes in a COBOL-81 file and five bytes in a VAX–11 COBOL file. ITEM-A of FILE-1 contains a fill byte only in a COBOL-81 file. ITEM-A of FILE-2 is eight bytes in length and contains three fill bytes. ITEM-A of FILE-3 is the same size as the COBOL-81 ITEM-A of FILE-1 (six bytes); however, it contains no implicit fill bytes.

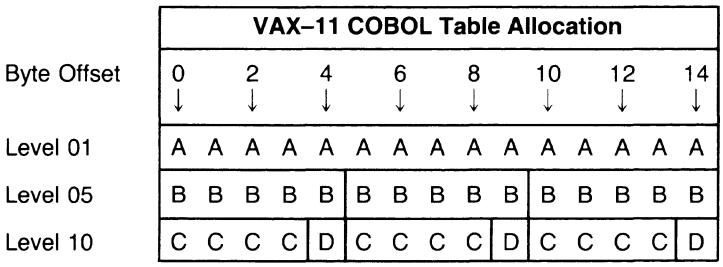
Example D-2 shows how you can change a table to ensure COMP item compatibility.

Example D-2: Changing a Table to Ensure COMP Item Compatibility

Incompatible table:

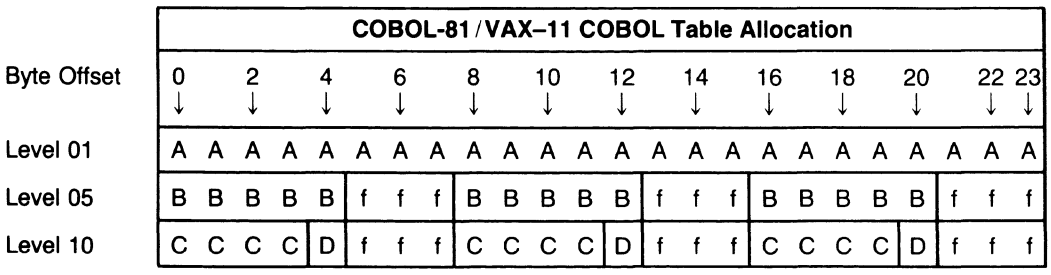
```
01   ITEM-A.
     05   ITEM-B OCCURS 3 TIMES.
          10   ITEM-C              PIC 9(9) COMP.
          10   ITEM-D              PIC X.
```





Compatible table using SYNCHRONIZED clause:

```
01   ITEM-A.
     05   ITEM-B OCCURS 3 TIMES.
          10   ITEM-C              PIC 9(9) COMP SYNC.
          10   ITEM-D              PIC X.
```

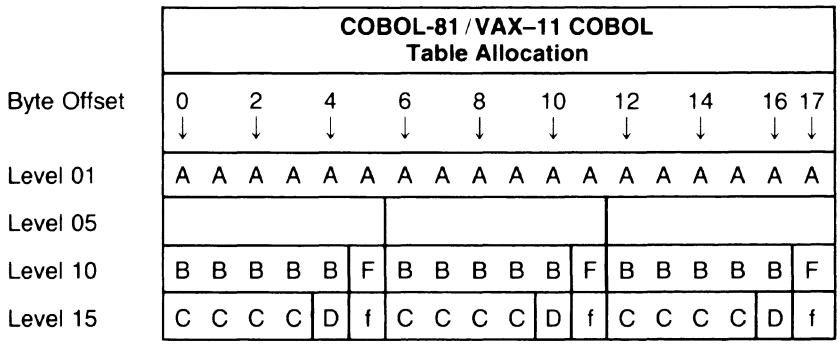


(continued on next page)

Compatible table using FILLER insertion:

(Note that the OCCURS clause is no longer on the same level as ITEM-B. This is to ensure that no fill bytes are included in ITEM-B's length. Also note that you insert the FILLER item on the same level as the group item, ITEM-B, rather than as an item subordinate to ITEM-B. This is the method to use whenever a COMP item is subordinate to a group item and is on an even byte offset from the beginning of the group. In all cases, the storage allocation for a compatible table should mirror the COBOL-81 storage allocation for an incompatible table–except for the fact that insertion of the FILLER item forces word alignment that otherwise would not occur.)

```
01  ITEM-A.
    05  OCCURS 3 TIMES.
        10  ITEM-B.
            15  ITEM-C              PIC 9(9) COMP.
            15  ITEM-D              PIC X.
        10  FILLER                  PIC X.
```

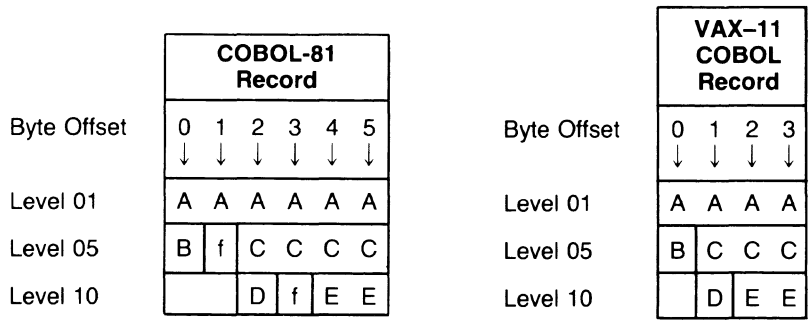


Example D-3 shows how to change a more complex record than appears in the previous examples. In this example, the COMP item aligns on an odd byte offset from the beginning of the group. For this reason, one fill byte (or FILLER item) must precede the COMP item and adds to the size of the group item.

Example D-3: Changing a Complex Record to Ensure COMP Item Compatibility

Incompatible record:

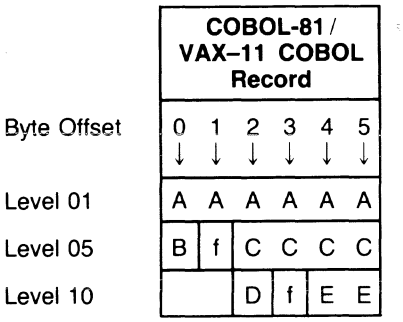
```
01  ITEM-A.
    05  ITEM-B                  PIC X.
    05  ITEM-C.
        10  ITEM-D              PIC X.
        10  ITEM-E              PIC 9(4) COMP.
```



(continued on next page)

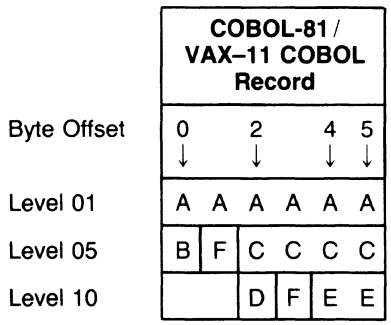
Compatible record using SYNCHRONIZED clause:

```
01   ITEM-A.
     05   ITEM-B                  PIC X.
     05   ITEM-C.
          10   ITEM-D             PIC X.
          10   ITEM-E             PIC 9(4) COMP SYNC.
```



Compatible record using FILLER insertion:

```
01   ITEM-A.
     05   ITEM-B                  PIC X.
     05   FILLER                  PIC X.
     05   ITEM-C.
          10   ITEM-D             PIC X.
          10   FILLER             PIC X.
          10   ITEM-E             PIC 9(4) COMP.
```



Data item alignment is a complex subject. You can find a more detailed explanation of implicit fill byte insertion and record allocation in Chapter 4 of this manual and also in Chapter 1 in Part III of the COBOL-81 User's Guide for your system.

D.3 Detection of Invalid Decimal Data

The VAX processor traps many cases of invalid data in a numeric item while the PDP-11 processor does not.

The following program creates invalid data that the PDP-11 processor cannot detect:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INV-DEC-DATA.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  ITEM-A.
    05  ITEM-B                      PIC XX.
    05  ITEM-C                      PIC AA.
01  ITEM-D REDEFINES ITEM-A         PIC 9999.
PROCEDURE DIVISION.
P-NAME.
    MOVE ZEROS TO ITEM-D.
    MOVE "XX" TO ITEM-B.
    MOVE "AA" TO ITEM-C.
    ADD 2 TO ITEM-D.
    DISPLAY ITEM-D.
    STOP RUN.
```

After the moves to ITEM-B and ITEM-C, ITEM-D no longer contains numeric data. If this program were run on a VAX–11 system, it would terminate following the statement "ADD 2 to ITEM-D". However, on a PDP-11 system, it would run to completion and display the invalid contents of ITEM-D. It is always good programming practice to test an operand for numeric contents before using it in an arithmetic operation. Keep this in mind when debugging COBOL-81 programs.

D.4 Size of Special Registers (RMS-STS, RMS-STV, and LINAGE-COUNTER)

COBOL-81 special registers are one word in length. VAX–11 COBOL special registers are two words in length. Therefore, if you move the contents of these registers to a data item you define in your program, you must define the item to be two words in length in your COBOL-81 program. (Your item description should be PIC S9(9) COMP.)

D.5 RMS-STS and RMS-STV Values

Record Management Services for PDP-11 systems returns different values to the RMS-STS and RMS-STV registers than does Record Management Services for VAX–11 systems.

This is an incompatibility only if your exception handling procedures state RMS-STS and RMS-STV values to determine which logical path your program should take. You still can display the contents of these special registers for informational purposes. Part IV of the COBOL-81 User's Guide for your system contains a sample program that illustrates display use of RMS-STS and RMS-STV that is compatible with VAX–11 COBOL.

An alternative way to get around this problem is to create a library file that contains RMS-STS and RMS-STV values and include these values in your program with the COPY statement. In this case, only the library file would need modification when you transfer your program to a VMS system.

The program in Example D-4 illustrates one error condition, an incorrect device specification for an input file. Following the program is the library source code appropriate for each COBOL compiler.

Example D-4: Including an RMS-STS Value Using the COPY Statement

Program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. STSEXAMPLE.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT DATA-FILE
           ASSIGN TO "INPUT.DAT".
DATA DIVISION.
FILE SECTION.
FD  DATA-FILE
           VALUE OF ID IS WRONG-DEV.
01  DATA-REC                    PIC X(80).
WORKING-STORAGE SECTION.
COPY "RMSVAL.LIB".
01  WRONG-DEV                   PIC X(4) VALUE IS "XYZ:".
PROCEDURE DIVISION.
DECLARATIVES.
DATA-FILE-ERROR SECTION.
    USE AFTER EXCEPTION PROCEDURE ON DATA-FILE.
DATA-FILE-ERROR-PARAG.
    MOVE RMS-STS OF DATA-FILE TO RMS-ERROR.
    IF BAD-DEVICE
                         .
                         .
                         .
END DECLARATIVES.
FIRST-SECT SECTION.
FIRST-PARAGRAPH.
    OPEN INPUT DATA-FILE.
                         .
                         .
                         .
```

Contents of RMSVAL.LIB on PDP-11 system:

```
01  RMS-ERROR      PIC S9(5) COMP VALUE IS ZEROS.
    88  BAD-DEVICE VALUE IS -448.
```

Contents of RMSVAL.LIB on VAX–11 system:

```
01  RMS-ERROR      PIC S9(9) COMP.
    88  BAD-DEVICE VALUE IS EXTERNAL RMS$_DEV.
```

D.6 Not Allowing Duplicate Keys in Indexed Files

A VAX–11 COBOL program does not execute correctly when accessing COBOL-81 indexed files that should not contain duplicate alternate record keys.

When a COBOL-81 program creates an indexed file that contains at least one alternate key for which duplicates are not allowed, PDP-11 Record Management Services does not store this information with the file. Rather, the COBOL-81 OTS (Object Time System) must check each program that accesses the file to find out whether or not the file can contain duplicate keys.

However, VAX–11 Record Management Services stores duplicate key information with the file and the VAX–11 COBOL OTS relies on this information. In effect, the information contained in the file identifier overrides what the VAX–11 COBOL program specifies. Since COBOL-81-generated indexed files do not contain this information, the VAX–11 COBOL OTS always treats the file as if duplicate keys were allowed. This means that a VAX–11 COBOL program can never execute an exception condition when your program attempts to create a record with a duplicate key.

In this case, you must recreate the indexed file that you want to transfer to a VAX–11 system. You can use a VAX–11 COBOL program to do this. The VAX–11 COBOL program must:

- Open the file for input
- Read each record in sequential order
- Write each record to another file with identical SELECT clause, file description entry, and record description entry

Remember that you do not need to recreate indexed files in which duplicate alternate record keys are allowed. These files are transferable.

D.7 Value for ESCape Character (RSTS/E Only)

On RSTS/E systems, the value for the escape character is 155 decimal. However, on VMS systems, its value is 27 decimal. When the 155 value is used on a VMS system, the Terminal Driver cannot adjust its character count to the terminal screen width. As a result, it wraps lines and generates space characters in unexpected places.

If you have a program that includes terminal-handling procedures using the 155 value, you must change that value to 27 when transferring the program to a VMS system.

D.8 Program-Names

COBOL-81 uses only the first six characters of the program-name. However, VAX–11 COBOL uses all characters in the program-name. If you specify program-names that are longer than six characters, you might create compile-time or link-time problems when you transfer a COBOL-81 program to a VAX/VMS system.

To ensure that a COBOL-81 program will compile, link and run on a VAX/VMS system, limit all program-names to six characters.

Glossary

abbreviated combined relation condition

The combined condition that results from the explicit omission of a common subject and common relational operator in a consecutive sequence of relation conditions.

abnormal termination

The premature end of an image that occurs when the operating system detects a condition that prevents further successful execution.

access mode

The way a program operates on a file's records.

access stream

A serial sequence of I-O operations on records in a sequential, relative, and indexed file. Successful OPEN statement execution creates an access stream. A successful explicit or implicit CLOSE statement terminates an access stream.

actual decimal point

The physical representation of the decimal point position in a data item. The characters comma (,) or period (.) represent the decimal point.

alphabet-name

A user-defined word in the SPECIAL-NAMES paragraph of the Environment Division. Alphabet-name assigns a name to a specific character set and/or collating sequence.

alphabetic character

A character that belongs to the set that includes the uppercase letters (A-Z) and the space. For the contents of data items, the set also includes the lowercase letters (a-z).

alphanumeric character

Any character in the computer character set.

alternate record key

A key, other than the prime record key, whose contents identify a record in an indexed file.

ANSI

*A*merican *N*ational *S*tandard *I*nstitute. An organization concerned with the standards to which many products in the marketplace must adhere. This includes the COBOL language upon which VAX–11 COBOL is based. The ANSI X3J4 Technical Committee specifies the standard for COBOL.

application program

A sequence of instructions and routines, not part of the basic operating system, designed to serve the specific needs of a user.

argument

One of the independent variables that determine the value of an expression.

arithmetic expression

- An identifier of a numeric elementary item
- A numeric literal
- Two of the items just described, separated by arithmetic operators
- Two arithmetic expressions separated by an arithmetic operator
- An arithmetic expression enclosed in parentheses

arithmetic operation

The process that results in a mathematically correct solution during:

- The execution of an arithmetic statement
- The evaluation of an arithmetic expression

arithmetic operator

Any of the following one- or two-character symbols:

Operator	Meaning
+	Addition
–	Subtraction
*	Multiplication
/	Division
**	Exponentiation

arithmetic statement

The arithmetic statements are: ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT.

ascending key

A key whose values determine the ordering of data. Ascending order starts with the lowest key value and ends with the highest, according to the rules for comparing data items.

assumed decimal point

A decimal point position in a data item. The assumed decimal point has logical meaning but no physical representation. It does not occupy a character position in the data item.

at end condition

A condition caused during:

- READ statement execution for a sequentially accessed file, when no next logical record exists or has been established for the file, or when an optional file is not present.
- RETURN statement execution, when no next logical record exists for the associated sort or merge file.
- SEARCH statement execution, when the search operation ends without satisfying the condition specified in the statement's WHEN phrase.

bit

The smallest unit in the computer's storage structure. A bit can express two distinct alternatives.

block

A physical unit of data that usually consists of one or more logical records. For mass storage files, a block can contain part of a logical record. The size of a block has no direct relationship to the size of: (1) the file that contains the block, or (2) the logical record(s) that are either contained in or overlap the block. Block is synonymous with Physical Record.

bottom margin

An empty area that follows the page body.

byte

An eight-bit unit of physical storage. In COBOL-81, a byte stores one character position.

called program

A program that is the object of a CALL statement. A called program is linked with the calling program to produce an executable image, or run unit.

calling program

A program that executes a CALL to another program.

character

The basic, indivisible unit of the COBOL language.

character data item

A data item that consists entirely of Standard Data Format characters.

character position

The amount of physical storage needed to store one Standard Data Format character whose usage is DISPLAY. In COBOL-81, a character position requires one byte of storage.

character-string

A sequence of contiguous characters that forms a COBOL word, literal, or PICTURE character-string.

class condition

A test of whether the content of a data item is either wholly alphabetic or wholly numeric.

clause

A subdivision of a COBOL sentence; an ordered set of consecutive COBOL character-strings that specifies an attribute of an entry.

COBOL character set

The set of characters used in the COBOL language, exclusive of the contents of nonnumeric literals, comment-entries, and comment lines:

Character	Meaning
0, 1, ..., 9	digit
A, B, ... , Z	letter
a, b, ... , z	lowercase letter (equivalent to letter)
	space
TAB	horizontal tab (equivalent to space)
+	plus sign
–	minus sign (hyphen)
*	asterisk
/	slash (stroke)
=	equal sign
$	currency sign
,	comma (decimal point)
;	semicolon
.	period (decimal point, full stop)
"	quotation mark
(	left parenthesis
)	right parenthesis
>	greater than symbol
<	less than symbol
:	colon
_	underline or underscore

See also *computer character set*.

COBOL word

See *word*.

CODASYL

An acronym for the Conference on Data Systems Languages, the committee that develops the COBOL language. The CODASYL committee produces a document titled *CODASYL COBOL Journal of Development*. This document serves as the basis for the standardization of the COBOL language.

collating sequence

The character-ordering sequence for sorting, merging and comparing.

column

A character position within a line on a video terminal screen. The columns are numbered from 1, by 1, starting at the leftmost character position of the line and extending to the rightmost position of the line.

combined condition

A condition that results from connecting two or more conditions with the AND or the OR logical operator.

comment line

A source program line with an asterisk in the Indicator Area. Both Area A and Area B can contain any characters from the computer character set. The comment line is for documentation only. A special form of comment line contains a stroke (/) in the Indicator Area instead of an asterisk (*). It causes the display device to advance to the top of the next page before printing the comment on the source listing.

compile time

When the compiler translates a COBOL source program to an object program.

compiler-directing statement

A statement that begins with a compiler-directing verb. A compiler-directing statement causes the compiler to take a specific action during compilation.

complex condition

A condition in which one or more logical operators act on one or more conditions. See *negated simple condition; combined condition; negated combined condition*.

computer-name

A system-name (PDP-11) that identifies the computer on which the program is to be compiled or run.

computer character set

The set of characters available on the computer. Most elements of a COBOL program can contain characters only from a subset of the computer character set. (See *COBOL character set.*) However, comment lines, comment-entries, and nonnumeric literals can contain characters from the full computer character set.

For COBOL-81, the computer character set is identical to the ASCII character set.

condition

The status of an executing program for which a truth value can be determined. When "condition" refers to language specifications or general formats, it is a conditional expression that consists of:

- A simple condition (with or without enclosing parentheses)
- A combined condition consisting of a combination of simple conditions, logical operators, and parentheses

condition-name

A data item (at level 88) that assigns a name to one value that a conditional variable can assume.

In general formats, condition-name represents a unique data item reference. The reference is: (1) a condition-name, and (2) qualification, subscripting, or indexing needed for uniqueness of reference.

condition-name condition

A proposition that tests whether or not a conditional variable's value is the same as that represented by condition-name.

conditional expression

A simple condition or a complex condition specified in an IF, PERFORM, or SEARCH statement. See *simple condition;* and *complex condition*.

conditional statement

A statement that determines the truth value of a condition. Subsequent program action depends on the truth value.

conditional variable

A data item to which a condition-name applies.

configuration section

An Environment Division section that usually describes the source and object computers.

connective

A reserved word used to:

- Associate a data-name, paragraph-name, condition-name or text-name with its qualifier
- Link two or more operands written in a series
- Form conditions (logical connectives)

See *logical operator*.

counter

A data item used for storing numbers or number representations, permitting them to be: (1) increased or decreased by another number, or (2) changed or reset to zero or an arbitrary positive or negative value.

currency sign

The character $ of the COBOL character set.

currency symbol

The character defined by the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph. If there is no CURRENCY SIGN clause, the currency symbol is identical to the currency sign.

current record

In sequential, relative, or indexed file processing, the record that is available in the file's record area.

data clause

A clause in a data description entry that describes an attribute of a data item.

data description entry

An entry in the Data Division that consists of a level-number followed by a data-name, if required, and continuing with a set of data clauses, as required.

data item

A unit of data (excluding literals) defined in a COBOL program.

data-name

A user-defined word that names a data item described in a data description entry. In general formats, *data-name* represents a word that must not be subscripted, indexed or qualified unless specifically allowed by rules of the format.

declarative sentence

A compiler-directing sentence consisting of a single USE statement.

declaratives

A set of one or more special-purpose sections at the beginning of the Procedure Division. The key word DECLARATIVES precedes the first of these sections, and the key words END DECLARATIVES follow the last. A declarative consists of a section header, followed, in order, by a USE sentence and zero, one, or more paragraphs.

delimiter

A character or sequence of contiguous characters that mark the end of a string of characters. A delimiter separates a string of characters from the following string. A delimiter is not part of the string of characters that it delimits.

descending key

A key whose values determine the ordering of data. Descending order starts with the highest key value and ends with the lowest, according to the rules for comparing data items.

digit position

The amount of physical storage needed to store one digit. This amount can vary depending on the usage specified in the data description entry that defines the data item. When the data description entry specifies that usage is DISPLAY, a digit position equals one character position.

division

A collection of zero, one, or more sections or paragraphs. Each of the four divisions consists of a division header and division body. The divisions are:

IDENTIFICATION
ENVIRONMENT
DATA
PROCEDURE

division header

A combination of words, followed by a separator period, that indicates the beginning of a division. The division headers are:

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

dynamic access

An access mode in which a program can randomly or sequentially obtain records from, or randomly place records into, a mass storage file. A program can use both types of access during the scope of the same OPEN statement.

editing character

A PICTURE clause character used to format data for output. Editing characters can be any of the following set of single characters or fixed two-character combinations:

Character	Meaning
B	space
0	zero
+	plus
–	minus
CR	credit
DB	debit
Z	zero suppress
*	check protect
$	currency sign
,	comma (decimal point)
.	period (decimal point)
/	slash (stroke)

elementary item

A data item that is not further subdivided.

end of Procedure Division

The physical position in a source program after which no further procedures appear.

entry

Any descriptive set of consecutive clauses, terminated by a separator period, in the Identification, Environment, or Data Division.

environment clause

A clause that is part of an Environment Division entry.

executable image

An image that is capable of being run in a process. When run, an executable image is read from a file for execution in a process. *Executable image* and *task image* are equivalent terms.

execution time

See *object time*.

expression

An arithmetic or conditional expression.

extend mode

The state of a file after a program opens it with the EXTEND phrase and before the program closes it without the REEL or UNIT phrase.

external switch

A software device that indicates that one of two alternate states exists.

figurative constant

A compiler-generated value that a program can refer to with specific reserved words.

file

A collection of logical records stored as a unit.

file clause

A clause that is part of a file description (FD) or sort-merge file description (SD).

file connector

A storage area that contains information about a file. It links:

- A file-name and a physical file
- A file-name and its associated record area

FILE-CONTROL

An Environment Division paragraph that declares the program's data files.

FILE-CONTROL entry

A SELECT clause and all its subordinate clauses. A File-Control entry declares a file's physical attributes.

file description entry

An entry in the File Section of the Data Division that starts with the level indicator FD. The level indicator is followed, in order, by: (1) a file-name, and (2) a set of file clauses, as required.

file-name

A user-defined word that names a file connector described in a file description entry or a sort-merge file description entry.

file organization

The permanent logical file structure defined when a file is created.

FILE SECTION

A Data Division section. The File Section contains file description and sort-merge file description entries and their associated record descriptions.

fixed length record

A record of a file whose file description or sort-merge description entry requires that all records contain the same number of character positions.

footing area

The position of the page body adjacent to the bottom margin.

format

A specific arrangement of a set of data.

group item

A data item that contains subordinate data items.

high order end

The leftmost character of a string of characters.

I-O mode

The state of a file after a program opens it with the I-O phrase, and before the program closes it without the REEL or UNIT phrase.

I-O-CONTROL

An Environment Division paragraph. The I-O-CONTROL paragraph specifies input-output techniques and sharing of the same areas by several data files.

I-O-CONTROL entry

An entry in the I-O-CONTROL paragraph.

identifier

The combination of a data-name, qualifiers, subscripts, and indexes required for uniqueness of reference. However, the rules for an identifier associated with a general format may specifically prohibit qualification, subscripting, or indexing. An identifier names a data item.

imperative statement

A statement that specifies an unconditional action. An imperative statement begins with an imperative verb and can consist of a sequence of imperative statements.

implementor-name

A system-name that refers to a feature available in COBOL-81.

implicit scope terminator

- A separator period that ends the scope of any preceding unterminated statement
- A phrase of a statement that, by its occurrence, ends the scope of any statement in the preceding phrase

index

A computer storage area or register whose contents represent the identification of an element in a table.

index data item

A data item in which a program can store the values associated with an index-name. The USAGE IS INDEX clause defines an index data item.

index-name

A user-defined word that names an index associated with a specific table.

indexed data-name

An identifier that consists of a data-name followed by one or more index-names in parentheses.

indexed file

A file with indexed organization.

indexed organization

The permanent logical file structure in which each record contains one or more keys whose values identify it.

input file

A file opened in input mode.

input mode

The state of a file after a program opens it with the INPUT phrase, and before the program closes it without the REEL or UNIT phrase.

input procedure

A set of statements that receives control during SORT statement execution. An input procedure controls the release of records to the sort.

INPUT-OUTPUT SECTION

An Environment Division section. The Input-Output Section names the program's files and their external media. It also provides information required for transmission and handling of data during program execution.

input-output file

A file opened in I-O mode.

integer

A numeric literal or a numeric data item that has no character positions to the right of the assumed decimal point. When the term *integer* appears in a general format, or in its associated rules, integer must not be a numeric data item. Also, it cannot be signed or zero unless explicitly allowed by the rules of the format.

intermediate data item

A signed numeric data item provided by the compiler. It contains the results developed during an arithmetic operation before moving the final result to the resultant-identifier, if any. The default size of an intermediate data item is 18 digits.

invalid key condition

At run time, a condition caused when the value of the key associated with an indexed or relative file is determined to be invalid.

key

A data item that identifies a record's location.

Key of Reference

The key, either prime or alternate, currently being used to access records in an indexed file.

key word

A reserved word required by a general format.

level indicator

Two alphabetic characters that identify a specific type of file.

level-number

A user-defined word, expressed as a one- or two-digit number, that indicates: (1) the hierarchical position of a data item, or (2) the special properties of a data description entry. Level-numbers from 1 through 49 indicate a data item's position in the hierarchical structure of a logical record. Level-numbers 66 and 88 identify special properties of a data description entry. Level-number 77 identifies noncontiguous items in the Working-Storage and Linkage Sections.

library text

The part of a source program that is included by the execution of a COPY statement. It is not part of the original source program file.

LINAGE-COUNTER

A special register whose value points to the current position in the page body.

line number

An integer that denotes the vertical position of a line on a video terminal screen, or on a page within a print file.

LINKAGE SECTION

The section in the Data Division of a called program that describes data items available from the calling program. Both the calling and the called programs can refer to these data items.

literal

A character-string whose value is implied by the ordered set of characters comprising the string.

logical operator

One of the reserved words AND, OR, or NOT. In the formation of a condition, either AND or OR, or both, can be used as logical connectives. NOT indicates logical negation.

logical page

A conceptual entity consisting of the top margin, page body, and bottom margin.

logical record

The most inclusive data item. The level-number for a logical record is 01 or 77. A record can be either an elementary or a group item.

low order end

The rightmost character of a string of characters.

mass storage

A storage medium in which data can be organized and maintained both sequentially and nonsequentially.

mass storage file

A collection of records assigned to a mass storage medium.

merge file

A collection of records to be merged by a MERGE statement. The merge file is created and can be used only by the merge function.

mnemonic-name

A user-defined word associated in the Environment Division with a specific implementor-name.

NATIVE character set

The 256-character set that starts with the 128 characters of the ASCII character set.

NATIVE collating sequence

The collating sequence of the ASCII character set.

negated combined condition

The "NOT" logical operator immediately followed by a combined condition enclosed in parentheses.

negated simple condition

The "NOT" logical operator immediately followed by a simple condition.

next executable sentence

The next sentence to which control transfers after execution of the current statement is complete.

next executable statement

The next statement to which control transfers after execution of the current statement is complete.

next record

The record that logically follows the current record of a file.

next record pointer

A conceptual entity for sequential, relative, and indexed files that points to the next logical record. The next record pointer can also indicate that: (1) no next logical record exists or has been established, or (2) an optional file is not present.

nonnumeric item

A data item whose description permits it to contain any combination of characters from the computer character set. Certain categories of nonnumeric items can contain only more restricted character sets.

nonnumeric literal

A literal bounded by quotation marks. The string of characters can include any character in the computer character set.

numeric character

A character that belongs to the set of digits 0 through 9.

numeric item

A data item whose description allows it to contain only digits. A signed numeric item can also contain a "+", "–", or other representation of an operational sign.

numeric literal

A literal consisting of one or more numeric characters. A numeric literal can contain either a decimal point or an algebraic sign, or both. The decimal point must not be the rightmost character. The algebraic sign must be the leftmost character.

OBJECT-COMPUTER

An Environment Division paragraph that describes the computer environment in which the program runs.

OBJECT-COMPUTER entry

An entry in the OBJECT-COMPUTER paragraph of the Environment Division. The entry contains clauses that describe the computer environment in which the program runs.

object of entry

A set of operands and reserved words in a Data Division entry that immediately follows the subject of the entry.

object program

A set of executable machine-language instructions and other material designed to interact with data to solve problems. Where there is no danger of ambiguity, "program" means "object program."

object time

When a program runs.

open mode

The state of a file after a program opens it and before the program closes it without the REEL or UNIT phrase. The OPEN statement specifies the open mode as INPUT, OUTPUT, I-O , or EXTEND.

operand

A component that is operated on. Any lowercase word(s) in a statement or entry format can be considered: (1) an operand, and (2) an implied reference to the data indicated by the operand.

operational sign

An algebraic sign associated with a numeric data item or numeric literal to indicate whether its value is positive or negative.

optional file

An input file whose presence is not necessary each time the program executes. The program checks for the presence or absence of the file.

optional word

An optional reserved word included in a specific format only to improve the source program's readability.

output file

A file opened in either output or extend mode.

output mode

The state of a file after a program opens it with the OUTPUT or EXTEND phrase, and before the program closes it without the REEL or UNIT phrase.

output procedure

A set of statements that receives control during the execution of a SORT statement after the sort function ends. Also, a set of statements that receives control during the execution of a MERGE statement after the merge function has selected the next record in merged order.

padding character

An alphanumeric character used to fill the unused character positions in a physical record.

page body

The part of the logical page in which lines can be written and/or spaced.

paragraph

In the Identification and Environment Divisions, a paragraph header followed by zero, one, or more entries. In the Procedure Division, a paragraph-name followed by a separator period and by zero, one, or more sentences.

paragraph header

A reserved word, followed by a separator period, that indicates the beginning of a paragraph in the Identification and Environment Divisions. Allowable paragraph headers are:

In the Identification Division:

PROGRAM-ID.
AUTHOR.
INSTALLATION.
DATE-WRITTEN.
DATE-COMPILED.
SECURITY.

In the Environment Division:

SOURCE-COMPUTER.
OBJECT-COMPUTER.
SPECIAL-NAMES.
FILE-CONTROL.
I-O-CONTROL.

paragraph-name

A user-defined word that identifies and begins a paragraph in the Procedure Division.

phrase

An ordered set of one or more consecutive COBOL character-strings that forms part of a clause or procedural statement.

physical record

See *block*.

pointer

A place marker that identifies the record address of a storage segment.

prime record key

A key that uniquely identifies a record in an indexed file.

procedure

A paragraph (or section) or group of logically successive paragraphs (or sections) in the Procedure Division.

procedure-name

A user-defined word that names a paragraph or section in the Procedure Division. It consists of a paragraph-name (which can be qualified), or a section-name.

program-name

A user-defined word that identifies a COBOL program.

punctuation character

A character from the following set:

Character	Meaning
,	comma
;	semicolon
.	period (full stop)
"	quotation mark
(	left parenthesis
)	right parenthesis
	space
=	equal sign

qualified data-name

An identifier that consists of one data-name followed by one or more OF (or IN) phrases containing a another data-name.

qualifier

A procedural reference that uniquely identifies an element in a COBOL source program. Qualifiers consist of the words IN or OF followed by:

1. A data-name or file-name when referring to:
   - Another data-name representing an item subordinate to the qualifier
   - A condition-name
   - LINAGE-COUNTER
   - RMS-STS
   - RMS-STV
2. A section-name when referring to a paragraph-name appearing in that section

random access

An access mode in which the program-specified value of a key data item identifies the logical record in a relative or indexed file.

record

See *logical record*.

record access stream

See *access stream*.

record area

A storage area allocated to process the sequential, relative, and indexed record described in a record description entry in the File Section.

record description entry

The total set of data description entries associated with a record.

record key

A key whose contents identifies a record in an indexed file. Within an indexed file, record key is either the prime record key or an alternate record key.

record-name

A user-defined word that names a record described in a record description entry.

reference format

A standard way to describe COBOL source programs.

relation

See *relational operator*.

relation character

A character from the following set:

Character	Meaning
>	Greater than
<	Less than
=	Equal to

relation condition

A comparison of the value of an arithmetic expression or data item to the value of another arithmetic expression or data item. See *relational operator*.

relational operator

In a relation condition, the reserved words used to compare the values of two operands. Valid relational operators are:

Relational Operator	Meaning
IS [NOT] GREATER THAN IS [NOT] >	Greater or not greater than
IS [NOT] LESS THAN IS [NOT] <	Less or not less than
IS [NOT] EQUAL TO IS [NOT] =	Equal or not equal to

relative file

A file with relative organization.

relative key

A key whose contents identifies a logical record in a relative file.

relative organization

The permanent logical file structure in which each record is uniquely identified by positive integer value. The integer value specifies the record's logical ordinal position in the file.

repeating group

A group data item whose description contains an OCCURS clause. Also, a group data item subordinate to a data item whose description contains an OCCURS clause.

reserved word

A COBOL word specified in the list of words that can appear in a COBOL program. (See Section 1.1.2.3, Reserved Words.) A reserved word cannot appear in a program as a user-defined word or system-name.

resource

A facility or service controlled by the operating system that can be used by an executing program.

resultant identifier

A user-defined data item that is to contain the result of an arithmetic operation.

RMS-STS

A Record Management Services (RMS-11) exception condition register. It contains the primary RMS-11 status value of an I-O operation. RMS-STV is the secondary value.

RMS-STV

A Record Management Services (RMS-11) exception condition register. It contains the secondary RMS-11 status value of an I-O operation. RMS-STS is the primary value.

section

A set of zero, one, or more paragraphs or entries (called a section body) that follows a section header. Each section consists of the section header and related section body.

section header

A combination of words (followed by a separator period) that indicates the beginning of a section in the Environment, Data, and Procedure Divisions.

In the Environment and Data Divisions, a section header consists of reserved words followed by a separator period. Valid section headers are:

In the Environment Division:

CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.

In the Data Division:

FILE SECTION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.

In the Procedure Division, a section header consists of:

- A section-name
- The reserved word SECTION
- A segment-number (optional)
- A separator period

section-name

A user-defined word that names a section in the Procedure Division.

segment-number

A user-defined word that classifies sections in the Procedure Division to enable segmentation. Segment-numbers can contain only the characters 0 through 9. They can be one- or two-digit numbers.

sentence

A sequence of one or more statements, the last of which ends with a separator period.

separately compiled program

A program that is compiled separately from all other programs.

separator

A character or two contiguous characters that delimit character-strings.

sequential access

An access mode in which logical records are obtained from or placed into a file in consecutive sequence. The order in which records were written to the file determines the logical record sequence.

sequential file

A file with sequential organization.

sequential organization

The permanent logical file structure established by the order in which records are written to the file.

sign condition

A test to determine whether the algebraic value of a data item or an arithmetic expression is either less than, greater than, or equal to zero.

simple condition

Any single relation, switch-status, condition-name, class, or sign condition that cannot be reduced into two or more conditions. See also *complex condition*.

sort file

A collection of records to be sorted by a SORT statement. The sort file is created and can be used only by the sort function.

sort key

The data item or items whose values determine how records are ordered in a sort or merge file.

sort-merge file description entry

An entry in the File Section that consists of the level indicator SD, followed by: (1) a file-name, and (2) a set of file clauses, as required.

SOURCE-COMPUTER

An Environment Division paragraph that describes the computer environment in which the source program is compiled.

SOURCE-COMPUTER entry

An entry in the SOURCE-COMPUTER paragraph that contains clauses describing the computer environment in which the source program is compiled.

source program

A syntactically correct set of COBOL statements. A source program begins with the Identification Division or a COPY statement. It ends with the end of the Procedure Division. When there is no danger of ambiguity, "program" means "source program."

special character

A character from the set:

Character	Meaning
+	plus sign
–	minus sign (hyphen)
*	asterisk
/	slash (stroke)
=	equal sign
$	currency sign
,	comma (decimal point)
;	semicolon
.	period (decimal point, full stop)
"	quotation mark
(	left parenthesis
)	right parenthesis
>	greater than symbol
<	less than symbol
!	exclamation point
#	number sign
%	percent
&	ampersand
'	apostrophe
:	colon
?	question mark
@	at sign
_	underline (underscore)
\	backslash

special character word

A reserved word that is an arithmetic operator or relation character.

SPECIAL-NAMES

The name of an Environment Division paragraph that relates implementor-names to user-specified mnemonic-names.

SPECIAL-NAMES entry

An entry in the SPECIAL-NAMES paragraph that contains clauses to: (1) specify the currency sign, (2) choose the decimal point, (3) relate implementor-names to user-specified mnemonic-names, and (4) relate alphabet-names to character sets or collating sequences.

special registers

Compiler-generated storage areas primarily used to store information relating to specific COBOL features.

Standard Data Format

A method of describing data as if the data appears on a printed page of infinite length and breadth. It does not relate to the way that data is stored internally or on an external medium.

statement

In the Procedure Division, a syntactically valid combination of words and symbols that begins with a verb.

subject of entry

An operand or reserved word that appears immediately after the level indicator or level-number in a Data Division entry.

subprogram

See *called program*.

subscript

An integer whose value identifies a table element.

subscripted data-name

An identifier that consists of a data-name followed by one or more subscripts enclosed in parentheses.

switch-status condition

The proposition that an external switch has been set to an "on" or "off" status.

system-name

A COBOL word that has already been defined by the implementor to refer to the program's operating environment.

table

A set of logically consecutive data items defined with an OCCURS clause in the Data Division.

table element

A data item that belongs to the set of repeated items comprising a table.

terminal operator

The individual who, at run time, enters data in response to program prompts.

text-name

A user-defined word that identifies a COBOL library.

text-word

A character (or a sequence of characters) in a COBOL library or source program, that is:

1. A literal, including the opening and closing quotation marks for nonnumeric literals
2. A separator other than a space or the opening and closing quotation marks of a nonnumeric literal

   The right and left parentheses are text-words, regardless of their context in a library or source program.
3. Any other sequence of contiguous characters, bounded by separators, except:
   - Comment lines
   - Separators

top margin

An empty area that precedes the page body.

truth value

The result of determining whether a condition is true or false.

unary operator

A plus (+) or a minus (–) sign that precedes a variable or left parenthesis in an arithmetic expression. It has the effect of multiplying the expression by +1 or –1.

unsuccessful execution

The attempted execution of a statement that does not result in the execution of all its operations. The unsuccessful execution of a statement does not affect any data referenced by that statement. However, it can affect status indicators.

user-defined word

A COBOL word that must appear in the source program to satisfy the format of a clause or statement.

variable

A data item whose value can be changed by program execution. A variable used in an arithmetic expression must be a numeric elementary item.

variable length record

A record associated with a file whose file description or sort-merge description entry permits records to contain a varying number of character positions.

variable occurrence data item

A table element that is repeated a variable number of times. It must contain an OCCURS DEPENDING ON clause in its data description entry or be subordinate to an item that does.

verb

A word that causes the compiler or object program to take an action.

word

A character-string of not more than 30 characters that forms a user-defined word, a system-name, or a reserved word.

WORKING-STORAGE SECTION

A Data Division section describing records and subordinate data items that are not parts of files.

Master Index

This Master Index contains a complete list of the references to subjects in the COBOL-81 Language Reference Manual and the four parts of the COBOL-81 User's Guide.

The index uses the following conventions:

Example	Explanation
1-8t	A page number followed by a t indicates a table.
4-6f	A page number followed by an f indicates a figure.

Entries in the Master Index are also preceded by an acronym indicating which manual, and part to a manual, the page number refers to:

Acronym	Title
LRM	COBOL-81 Language Reference Manual
RSTS/E UG I	COBOL-81 User's Guide, Part I for RSTS/E
RSX UG I	COBOL-81 User's Guide, Part I for RSX-11M/M-PLUS
UG II	COBOL-81 User's Guide, Part II
UG III	COBOL-81 User's Guide, Part III
UG IV	COBOL-81 User's Guide, Part IV

Where a subject references more than one manual and/or parts, references to the COBOL-81 Language Reference Manual appear first, followed in order by the COBOL-81 User's Guide Part I, then Part II, Part III, and Part IV.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

HOW TO ORDER ADDITIONAL DOCUMENTATION

DIRECT TELEPHONE ORDERS

In Continental USA and Puerto Rico call **800–258–1710**

In Canada call **800–267–6146**

In New Hampshire, Alaska or Hawaii call **603–884–6660**

DIRECT MAIL ORDERS (U.S. and Puerto Rico*)

DIGITAL EQUIPMENT CORPORATION
P.O. Box CS2008
Nashua, New Hampshire 03061

DIRECT MAIL ORDERS (Canada)

DIGITAL EQUIPMENT OF CANADA LTD.
940 Belfast Road
Ottawa, Ontario, Canada K1G 4C2
Attn: A&SG Business Manager

INTERNATIONAL

DIGITAL EQUIPMENT CORPORATION
A&SG Business Manager
c/o Digital's local subsidiary
or approved distributor

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Northboro, Massachusetts 01532

*Any prepaid order from Puerto Rico must be placed with the Local Digital Subsidiary:
809–754–7575

COBOL–81
Language Reference Manual
AA-J434B-TC

Reader's Comments

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement. _____

_____

_____

_____

_____

_____

_____

Did you find errors in this manual? If so, specify the error and the page number. _____

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

— — — Do Not Tear - Fold Here and Tape —

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

BSSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03061

— — — Do Not Tear - Fold Here —

digital

DIGITAL EQUIPMENT CORPORATION

Printed in U.S.A.