VMS Software

VSI COBOL
Reference Manual

Operating System and Version: VSI OpenVMS IA-64 Version 8.4-1H1 or higher
VSI OpenVMS Alpha Version 8.4-2L1 or higher

Software Version: VSI COBOL Version 3.1-7 for OpenVMS

VMS Software, Inc. (VSI)
Boston, Massachusetts, USA

VSI COBOL Reference Manual

VMS Software

Legal Notice

Table of Contents

Preface

This book describes the constructs and rules of the VSI COBOL for OpenVMS programming language, which is a VSI Company implementation of COBOL (COmmon Business-Oriented Language) for the OpenVMS and UNIX platforms. It includes information about language syntax and semantics, as well as information about adherence and extensions to various COBOL standards.

This documentation set also includes the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/] and, optionally, the *VSI COBOL DBMS Database Programming Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-dbms-database-programming-manual/].

VSI COBOL is the new name for what has formerly been known as HP COBOL, Compaq COBOL, DEC COBOL, DIGITAL COBOL. VSI COBOL, unmodified, refers to the following products:

VSI COBOL for OpenVMS Industry Standard 64
VSI COBOL for OpenVMS Alpha
VSI COBOL for UNIX

Any references to the former names in product documentation or other components should be construed as references to the VSI COBOL names.

1. About VSI

VMS Software, Inc. (VSI) is an independent software company licensed by Hewlett Packard Enterprise to develop and support the OpenVMS operating system.

2. Intended Audience

This manual is intended for experienced applications programmers who have a thorough understanding of the COBOL language and some familiarity with their operating system. This is not a tutorial manual.

If you are a new COBOL user, you may need to read introductory COBOL textbooks or take COBOL courses.

3. Document Structure

This manual is organized as follows:

- Chapter 1 presents the elements of the COBOL language, describes two format options for a COBOL program, and explains how the remaining chapters organize and present the COBOL general formats.
- Chapter 2 describes the organization of a COBOL program. It presents the general format for the four COBOL divisions and introduces the concept of contained programs. This chapter shows the relationship between a program name and a source file name.
- Chapter 3 describes the general format and contents of the Identification Division. It explains how to identify a COBOL program and its source listing.
- Chapter 4 describes the general format and contents of the Environment Division. It explains how to describe the program's physical environment.
- Chapter 5 describes the general format and contents of the Data Division. It explains how to describe data the program receives, creates, manipulates, and produces as output.

- Chapter 6 describes the general format and contents of the Procedure Division. It describes COBOL verbs, which process the files and data in the Environment and Data Divisions.
- Chapter 7 describes the general format and use of the intrinsic functions.
- Chapter 8 describes the general format of the COPY and REPLACE statements.
- Appendix A lists the VSI COBOL for OpenVMS reserved words, which are words that cannot be used as system names or user-defined names.
- Appendix B lists the ASCII, EBCDIC, and NATIVE character sets.
- Appendix C lists the exception condition values that can appear in File Status data items.
- Appendix D contains individual presentation rules and tables for each type of report group.
- Appendix E describes RTL routines for accessing the RAB and FAB structures on OpenVMS systems.

4. Related Documents

The following documents contain additional information directly related to various topics covered in this manual:

- Release Notes

  Consult the VSI COBOL for OpenVMS release notes for your installed version for late corrections and new features.

  On the OpenVMS Alpha, I64 operating system, the release notes are in:

  SYS$HELP:COBOL*nnn*.RELEASE_NOTES (ASCII text)
  SYS$HELP:COBOL*nnn*_RELEASE_NOTES.PS

  Where *nnn* is the version and release number.

  On the UNIX, the release notes are in:

  /usr/lib/cmplrs/cobol/relnotes

- *VSI COBOL User Manual [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/]*

  This manual describes how to use features of the VSI COBOL for OpenVMS language to develop programs on the OpenVMS Alpha, I64, and VAX or the UNIX operating systems.

- *VSI COBOL Installation Guide [https://docs.vmssoftware.com/vsi-cobol-for-openvms-installation-guide/]*

  This manual provides instructions for installing VSI COBOL for OpenVMS on the OpenVMS Alpha and OpenVMS I64 operating systems.

- *VSI COBOL DBMS Database Programming Manual [https://docs.vmssoftware.com/vsi-cobol-for-openvms-dbms-database-programming-manual/]*

  This manual provides information on using VSI COBOL for OpenVMS for database programming with Oracle CODASYL DBMS on the OpenVMS Alpha, the OpenVMS I64, or OpenVMS VAX operating systems.

- **The OpenVMS Documentation Set**

  This set contains information about using the features of the OpenVMS I64 and OpenVMS Alpha operating systems and their tools.

- **The UNIX Documentation Set**

  This set contains introductory and detailed information about using the features of the UNIX operating system and its tools.

5. OpenVMS Documentation

The full VSI OpenVMS documentation set can be found on the VMS Software Documentation webpage at https://docs.vmssoftware.com.

6. VSI Encourages Your Comments

You may send comments or suggestions regarding this manual or any VSI document by sending electronic mail to the following Internet address: `<docinfo@vmssoftware.com>`. Users who have VSI OpenVMS support contracts through VSI can contact `<support@vmssoftware.com>` for help with this product.

7. Conventions

The following product names may appear in this manual:

- VSI OpenVMS for Integrity servers
- OpenVMS I64
- I64

All three names — the longer form and the two abbreviated forms — refer to the version of the OpenVMS operating system that runs on the Intel ® Itanium ® architecture.

The following typographic conventions may be used in this manual:

Convention	Meaning
RECORD KEY IS	Underlined uppercase words are required when used in a general format. Uppercase words not underlined are optional.
sortfile	Lowercase words used in a general format are generic terms that indicate entries you must provide.
{ | | }	Braces used in a general format enclose lists from which you must choose only one item. For example: { SEQUENTIAL | RANDOM | DYNAMIC }
{ | | }	Brackets used in a general format enclose optional items from which you can choose none or one. For example: { RECORD | ALL RECORDS }
{ { | | } }	Choice indicators, vertical lines inside a set of braces, used in a general format enclose lists from which you must choose one or more items, using each item chosen only once. For example:

Convention	Meaning
	{ { COMMON | INITIAL } }
. . .	A horizontal ellipsis indicates that the item preceding the ellipsis can be repeated. For example: { switch-name ... }
. . .	A vertical ellipsis indicates that not all of the statements are shown.
Format	Program examples are shown in terminal format, rather than in ANSI standard format.
special-character words	The following symbols, when used in a general format, constitute required special-character words: Plus sign (+) Minus sign (-) Single (=) and double (==) equal signs Less than (<) or greater than (>) symbols Less than or equal to (<=) and greater than or equal to (>=) symbols Period (.) Colon (:) Single (*) and double (**) asterisks Slash (/) Left parenthesis (( ) or right parenthesis ( ))
quotation mark	The term quotation mark is used to refer to the double quotation mark character ('').
apostrophe	The term apostrophe is used to refer to the single quotation mark character (’).
`user input`	In examples, user input (what you enter) is shown as **`monospaced text`**.
report file	Bold type indicates a new term.

full-file-name	This syntax term refers to the name of a file and the device and directory, or path, in which it is located. For example:
	
	`DISK2$:[HOME.PUBLIC]FILENAME.TXT; (OpenVMS file specification)`
	`/disk2/home/public/filename.txt (UNIX file specification)`

compiler option	This term refers to command-line qualifiers (OpenVMS Alpha and I64 systems) or flags (UNIX systems). For example:
	
	`/LIST (OpenVMS qualifier specification)`
	`-list (UNIX flag specification)`

COBOL	This term refers to language information common to ANSI-85 COBOL, VSI COBOL for OpenVMS, and VSI COBOL.
Enter	A boxed symbol indicates that you must press a key on the terminal; for example, **Enter** indicates that you press the Enter key.
Tab	This symbol indicates a nonprinting tab character.
Ctrl/x	The symbol **Ctrl/x** indicates that you hold down the key labeled CTRL while you press another key, for example, **Ctrl C** or **Ctrl O**.

Convention	Meaning
$	The dollar sign ($) represents the OpenVMS system prompt.
%	The percent sign (%) represents the UNIX system prompt.

Chapter 1. Overview of the COBOL Language

This chapter provides information about the structure and language of COBOL source programs. It describes the elements of the COBOL language, reference formats, and language organization.

The COBOL language consists of the following components:

- Programs
- Divisions
- Sections
- Paragraphs
- Sentences
- Statements
- Clauses
- Entries
- Words
- Characters

A **separately compiled COBOL program** is a program that, together with its contained programs (if present), is compiled separately from all other programs. Each COBOL program is divided into four parts, called **divisions**: the Identification Division, Environment Division, Data Division, and Procedure Division. Divisions can contain sections, which in turn can contain paragraphs. Paragraphs can contain sentences, clauses, statements, or entries.

The building blocks of these language components include the COBOL character set, character-strings, separators, punctuation, and literals.

A COBOL program is a string of characters that is syntactically correct according to the COBOL language rules.

1.1. The COBOL Character Set

The **COBOL character set**, shown in Table 1.1, is used to form character-strings and separators.

The only components of a COBOL 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 elements can use.

Table 1.1. The COBOL Character Set

Character	Meaning
0, 1, …, 9	digit

Character	Meaning
A, B, …, Z	letter
a, b, …, z	lowercase letter (equivalent to letter)
+	plus sign
-	minus sign (hyphen)
*	asterisk
/	slash (stroke, virgule)
\	backslash
=	equal sign
$	currency sign
>	greater than symbol
<	less than symbol
:	colon
_	underline (underscore)
	space
Tab	horizontal tab
(	left parenthesis
)	right parenthesis
,	comma (decimal point)
;	semicolon
.	period (decimal point, full stop)
"	quotation mark (double quotation mark)
’	apostrophe (single quotation mark)
{	left brace
}	right brace
[	left bracket
]	right bracket
<<	double left-angle brackets
>>	double right-angle brackets

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
```

1.2. Character Strings

A **character-string** is a character or a sequence of contiguous characters that form a COBOL word, a literal, a PICTURE character-string, or a comment-entry. Separators delimit character-strings. The following sections describe these topics in detail.

1.2.1. COBOL Words

A **COBOL word** is a character-string of not more than 31 characters that forms one of the following:

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

A user-defined word or system-name 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.

1.2.1.1. 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 to Z, 0 to 9, the currency sign ($), underline (_), and hyphen (-). Throughout this manual, and except where specific rules apply, the hyphen (-) and the underline (_) are treated as the same character in a user-defined word. The underline (_), however, can begin or end a user-defined word, and the hyphen (-) cannot. By convention, names containing a currency sign ($) are reserved for VSI.

Table 1.2 provides brief descriptions of the COBOL user-defined words.

Table 1.2. COBOL User-Defined Words

User-Defined Word	Purpose
Alphabet-Name	Assigns a name to a character set, collating sequence, or both. Alphabet-names must be defined in the SPECIAL-NAMES paragraph. (See SPECIAL-NAMES in Chapter 4.)
Class-Name	Relates a name to a specified set of characters listed in that clause. (See SPECIAL-NAMES in Chapter 4.)
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 reference modified, subscripted, indexed, or qualified unless specifically allowed by the rules for that format.
File-Name	Names a file connector. A **file connector** is a storage area that contains information about a file and is the link between: ● A file-name and a physical file ● A file-name and its associated storage area File description entries and sort-merge file description entries describe file connectors.

User-Defined Word	Purpose
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 Sections 5.1.1 and 5.1.2.)
Library-Name	Names a COBOL library used in a source program compilation. (See the COPY statement in Chapter 8.)
Mnemonic-Name	Associates a name with a system-name, such as CONSOLE, SYSERR, ARGUMENT-NUMBER, ENVIRONMENT-NAME, C01, OR SWITCH-8. (See SPECIAL-NAMES in Chapter 4.)
Paragraph-Name	Names a Procedure Division paragraph. (See the section called “Paragraph, Paragraph Header, Paragraph-Name”.) Paragraph-names are equivalent only if they are identical; that is, if 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. (See the PROGRAM-ID paragraph in Chapter 3, and the section on CALL in Chapter 6, for a description of case-sensitivity on the UNIX. Also refer to the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/] for a description of the `-names lowercase`, `-names uppercase`, and `-names as_is` flags.)
Record-Name	Names a data item described with level-number 01 or 77.
Report-Name	Names a report produced by the Report Writer Control System (RWCS). (See the REPORT clause in Chapter 5.)
Screen-Name (Alpha, I64)	Names a screen item defined in the SCREEN SECTION of a program. (See the Screen Description (Alpha, I64) section of Chapter 5.)
Section-Name	Names a Procedure Division section. Section-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. (See the section called “Section Header”.)
Segmented-Key-Name	Identifies a segmented key, which is a concatenation of one or more (up to eight) data items (segments) within a record associated with an indexed file. A segmented key is a form of primary or alternate key. It offers flexibility in defining record description entries for indexed files. (Refer to the section on segmented keys in the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/].)
Segment-Number	Is a 1- or 2-digit number that classifies a Procedure Division section for segmentation. In VSI COBOL for OpenVMS programs, segment-numbers specify independent and fixed segments. (See Section 6.7.)
Symbolic-Character	Identifies a user-defined figurative constant.

User-Defined Word	Purpose
Text-Name	Identifies library text in a COBOL library. (See the COPY statement in Chapter 8.)

Within a given program, but excluding any contained program, the user-defined words are grouped into the following disjoint sets:

alphabet-names
class-names
condition-names, data-names, and record-names
file-names
index-names
library-names
mnemonic-names
paragraph-names
program-names
report-names
screen-names
section-names
segmented-key-names
symbolic-characters
text-names

All user-defined words in a program, except segment-numbers and level-numbers, can belong to only one of these sets. User-defined words in each set must be unique, except as described in the rules for uniqueness of reference. (See Section 6.2).

Except for section-names, paragraph-names, segment numbers, and level-numbers, all user-defined words must contain at least one alphabetic character. Segment-numbers and level-numbers need not be unique. Any segment-number or level-number can be the same as any other segment-number or level-number.

1.2.1.2. System-Names

System-names are COBOL words that refer to the program's operating environment. The same COBOL word can be used in a program as both a user-defined word and a system-name. The compiler determines the word's class from its context.

The system-names are as follows:

ALPHA
ASCII
CARD-READER
CONSOLE
CONTIGUOUS
CONTIGUOUS-BEST-TRY
C01
DEFERRED-WRITE
EBCDIC
EXTENSION
FILL-SIZE
I64
LINE-PRINTER

LOCK-HOLDING
MASS-INSERT
OPERATOR
PAPER-TAPE-PUNCH
PAPER-TAPE-READER
PREALLOCATION
PRINT-CONTROL
SWITCH
WINDOW

1.2.1.3. 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 for a list of reserved words.)

The three types of reserved words follow:

- Required words
- Optional words
- Special-purpose words

Required Word

A **required word** must be used when its format is used in a program.

The two types of required words are **keywords** and **special character words**. In general formats, keywords are uppercase and underlined. Arithmetic operators and relation characters are special character words; they are not underlined in the general format.

In the following sample format, the keywords are COMPUTE, ROUNDED, SIZE, ERROR, NOT, and END-COMPUTE. The equal sign (=) is a special-character word.

```
COMPUTE { rsult [ ROUNDED ] } . . . = arithmetic-expression

[ ON SIZE ERROR stment ]
[ NOT ON SIZE ERROR stment2 ]
[ END-COMPUTE ]
```

Optional Words

In general formats, uppercase words that are not underlined are **optional words**. They can make a program more human-readable, but have no semantic effect. In the previous sample format, ON is an optional word.

Special-Purpose Words

The two types of **special-purpose words** are **figurative constants** and **special registers**. Figurative constants name and refer to specific constant values and are described in detail in Section 1.2.3. Special registers name and refer to special storage areas that the compiler provides.

The VSI COBOL for OpenVMS special registers are primarily used to store information related to or produced by specific VSI COBOL for OpenVMS features. Table 1.3 shows the special registers, their usage, and their descriptions.

Table 1.3. Special Registers

Special Register	Usage—Description
RETURN-CODE (Alpha, I64)	X/OPEN—Names an VSI COBOL for OpenVMS special register that may be used to set a return value for a calling program or to retrieve the value returned from a called program. It is represented by PIC S9(9) USAGE IS COMP. It is implicitly defined with GLOBAL scope. The RETURN-CODE register is initialized with the platform-specific success code. On OpenVMS Alpha and OpenVMS I64, it is initialized to one. On UNIX it is initialized to zero. The RETURN-CODE special register can be set by a called program, prior to the execution of a STOP RUN or EXIT PROGRAM statement, to pass a value to the calling program or the execution environment. For a calling program, it can be read, subsequent to the CALL, to obtain the value of the RETURN-CODE set by the called program. On UNIX the main program sets the shell variable `status` to the value of the RETURN-CODE. On OpenVMS Alpha and OpenVMS I64 the main program sets the symbol $STATUS to the value of the RETURN-CODE. If you use the GIVING phrase on the CALL statement or on the Procedure Division header, specifying a data item as its argument, this data item (instead of RETURN-CODE) receives the return value. Note that you can specify the special register RETURN-CODE as the argument to GIVING, in which case RETURN-CODE receives the return value. For more information on the relationship between the GIVING phrase and the RETURN-CODE special register, see Table 6.7 in Chapter 6. Because the reserved word RETURN-CODE is one of the X/Open reserved words, you cannot use the `noxopen` keyword in the `reserved_words` compiler option if you want to use the RETURN-CODE special register. For related information, see Section 6.8 for the syntax and description of the GIVING phrase of the Procedure Divison header; and the CALL statement for the syntax and description of CALL GIVING.
LINAGE-COUNTER	LINAGE files—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 the LINAGE clause in Chapter 5.) The implicit size of LINAGE-COUNTER is nine decimal digits represented by PIC S9(9) COMP. You can qualify LINAGE-COUNTER with a file-name. Procedure Division statements and the SOURCE clause of the Report Section can access the value of LINAGE-COUNTER but cannot change its value. LINAGE-COUNTER is global if file-name is global and external if file-name is external.

Special Register	Usage—Description
PAGE-COUNTER	REPORT WRITER—A page counter that the compiler provides for each report in the Report Section of the Data Division. You can qualify PAGE-COUNTER with a report-name. Its value is the number of the current page within a report. The implicit size of PAGE-COUNTER is six unsigned decimal digits represented by PIC 9(6) COMP. The Report Writer Control System (RWCS) maintains the value of PAGE-COUNTER and uses this value to number the pages of a report. The SOURCE clause of the Report Section can reference PAGE-COUNTER. The values in PAGE-COUNTER range from 1 to 999999 and can be altered by Procedure Division statements.
LINE-COUNTER	REPORT WRITER—A line counter that the compiler generates for each report in the Report Section of the Data Division. It may be qualified by a report-name. Its value is the number of the current line within a page. (See PAGE-COUNTER.) The implicit size of LINE-COUNTER is six unsigned decimal digits represented by PIC 9(6) COMP. The Report Writer Control System (RWCS) maintains the value of LINE-COUNTER and uses this value to determine the vertical positioning of a report. The SOURCE clause of the Report Section can reference LINE-COUNTER. The values in LINE-COUNTER range from 0 to 999999. Procedure Division statements can access the values in LINE-COUNTER; however, only the RWCS can change its value.
RMS-STS[1](OpenVMS)	RMS—Contains the primary RMS status value of an I/O operation. (RMS-STV contains the secondary value.) RMS-STS provides additional information on COBOL File Status values resulting from I/O operations.[2]It is represented by PIC S9(9) USAGE IS COMP. You must qualify RMS-STS with a file-name. If the file-name is global, RMS-STS is also global. If the file-name is external, RMS-STS is also external. 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 (FAB). After executing a READ, WRITE, REWRITE, DELETE, START, or UNLOCK statement, RMS-STS is set to the value of the STS field in the associated record access block (RAB).
RMS-STV[1](OpenVMS)	RMS—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 represented by PIC S9(9) USAGE IS COMP. You must qualify RMS-STV with a file-name. If the file-name is global, RMS-STV is also global. If the file-name is external, RMS-STV is also external. 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 FAB. After executing a READ, WRITE, REWRITE,

Special Register	Usage—Description
	DELETE, or START statement, RMS-STV is set to the value of the STV field in the associated RAB.
RMS-FILENAME[1](OpenVMS)	RMS—Names the complete RMS filename. It consists of 255 alphanumeric characters represented by PIC X(255) USAGE IS DISPLAY. You must qualify it with a file-name. If the file-name is global, RMS-FILENAME is also global. If the file-name is external, RMS-FILENAME is also external. Before the program opens the file for the first time, the value of RMS-FILENAME is undefined. For each COBOL OPEN statement, RMS-FILENAME is set to the complete RMS file specification string of file-name: for example, DBB1:[COBOL]MASTER.DAT.
RMS-CURRENT-STS[1](OpenVMS)	RMS—Names an VSI COBOL for OpenVMS exception condition register. It contains the primary RMS status value of the most recent RMS I/O operation, regardless of the file operated on. (RMS-CURRENT-STV contains the secondary value.) It is represented by PIC S9(9) USAGE IS COMP. Since this register can contain the primary RMS status value for any file, you must not qualify it with a file-name. After your program executes any RMS I/O operation, it sets RMS-CURRENT-STS to the value contained in RMS-STS for that file.
RMS-CURRENT-STV[1](OpenVMS)	RMS—Names an VSI COBOL for OpenVMS exception condition register. It contains the secondary RMS status value of the most recent RMS I/O operation, regardless of the file operated on. (RMS-CURRENT-STS contains the primary value.) It is represented by PIC S9(9) USAGE IS COMP. Since this register can contain the secondary RMS status value for any file, you must not qualify it with a file-name. After your program executes any RMS I/O operation, it sets RMS-CURRENT-STV to the value contained in RMS-STV for that file.
RMS-CURRENT-FILENAME[1](OpenVMS)	RMS—Names an VSI COBOL for OpenVMS exception condition register. It contains the complete RMS file specification string of the file most recently operated on by an I/O statement. It consists of 255 alphanumeric characters represented by PIC X(255) USAGE IS DISPLAY. Since this register can contain the file-name for any file, you must not qualify it with a file-name. After your program executes any I/O operation, it sets RMS-CURRENT-FILENAME to the string contained in RMS-FILENAME for that file.

[1]Procedure Division statements can the values or strings stored in the RMS special registers; however, only the RMS facility can the contents of the registers. Refer to the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/] for programming examples. For an explanation and a listing of RMS STS and STV values, refer to the *OpenVMS System Messages and Recovery Procedures Reference Manual*, an archived manual available on the OpenVMS Documentation CD-ROM, or the online OpenVMS Help Message utility. Refer to the *VSI OpenVMS Record Management Services Reference Manual* for information on RMS. (RMS is on OpenVMS systems only.)

[2]The FILE STATUS data item (see Section 6.6.8) provides the primary source of status information for the file I-O verbs, and RMS-STS and RMS-STV provide supplementary information.

1.2.1.4. Function-Names

A **function-name** is the name of a function as shown in Table 7.1. Note that function-names are not reserved words and may appear in a different context in a program as a user-defined word or a system-name.

1.2.2. Literals

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.

VSI COBOL for OpenVMS provides two types of literals: numeric and nonnumeric. Numeric literals include floating-point literals and nonnumeric literals include hexadecimal and national literals. Floating-point, hexadecimal, and national literals are VSI extensions. The following two sections describe literals in detail.

1.2.2.1. Numeric Literals

A **numeric literal** is a character string of 1 to 33 characters on Alpha and I64 selected from the digits 0 to 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.

Syntax Rules

1. A numeric literal must contain at least 1 digit and not more than 31 digits on Alpha and I64.

2. A numeric literal must not contain more than one sign character, which must be the leftmost character. If the literal is unsigned, its value is positive.

3. A numeric literal must not contain more than 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.

Table 1.4 provides examples of numeric literals.

Table 1.4. Numeric Literals

Literal	Value
12	12
0.12000	0.12
-123456789012345678	-123456789012345678
000000003	3
-34.455445555	-34.455445555
0	0
+0.000000000001	+0.000000000001
+0000000000001	+1

Floating-Point Literals

A **floating-point literal**, a VSI extension to numeric literals, is a character-string whose value is specified by 4 to 37 characters on Alpha and I64, selected from the digits 0 to 9, the plus sign (+), the minus sign (-), the decimal point (.), and the letter E (uppercase or lowercase).

You can use floating-point literals to achieve a wider range of numeric literal values.

Syntax Rules

1. A floating-point literal must be between 4 and 37 (Alpha, I64) characters in length.
2. A floating-point literal must contain the following characters:
   - At least 1 digit to the left of the E
   - A decimal point to the left of the E
   - An E (uppercase or lowercase)
   - At least 1 digit to the right of the E
3. The maximum number of characters to the left of the E is 33 (Alpha, I64) of which no more than 31 can be digits.
4. The maximum number of characters to the right of the E is 4 (Alpha, I64) of which no more than 3 can be digits.
5. A floating-point literal must not contain more than two sign characters as follows:
   - The first character of the literal
   - The first character following the E
6. If the first character of the literal is not a sign character, the literal is positive.
7. If the first character following the E is not a sign character, the value of the numeric component following the E is positive.
8. A floating-point literal must contain only one decimal point that can appear only to the left of the E.
9. A comma must be used in place of the decimal point, if the DECIMAL POINT IS COMMA clause is specified.

The value of a floating-point literal is the algebraic quantity represented by the characters in the literal that precede the E multiplied by ten raised to the power of the algebraic quantity represented by the characters in the literal following the E.

Table 1.5 provides a few examples of floating-point literals.

Table 1.5. Floating-Point Literals

Literal	Value
1.6e5	160000.0
3.2E-3	0.0032
-1.e4	-10000.0

Literal	Value
0.002e+6	2000.0
-.8E-2	-0.008

1.2.2.2. Nonnumeric Literals

A **nonnumeric literal** is a character-string of 0 to 256 characters. It is delimited on both ends by quotation marks (") or apostrophes (’). A nonnumeric literal delimited by apostrophes is treated in the same manner as a nonnumeric literal delimited by quotation marks.

The value of a nonnumeric literal is the value of the characters in the character-string. It does not include the quotation marks (or apostrophes) 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.

Syntax Rules

1. A space, left parenthesis, or pseudo-text delimiter (==) must immediately precede the opening quotation mark (or apostrophe).
2. The closing quotation mark (or apostrophe) must be immediately followed by one of the following:
   - Space
   - Comma
   - Semicolon
   - Period
   - Right parenthesis
   - Pseudo-text delimiter
3. If a nonnumeric literal is delimited by quotation marks ("), two consecutive quotation mark characters in the literal represent one quotation mark character.
4. If a nonnumeric literal is delimited by apostrophes (’), two consecutive apostrophes in the literal represent one apostrophe (’).

Table 1.6 provides examples of nonnumeric literals. In these examples, *s* represents a space character.

Table 1.6. Nonnumeric Literals

Literal	Value
"ABC "	ABC
"01 "	01
"s01 "	s01
"D " "E " "F "	D "E "F
"a.b "	a.b
’GHI ’	GHI
’02 ’	02

Literal	Value
’s02 ’	s02
’c.d ’	c.d
" " " "	"
’ " " ’	" "
’ ’ ’ ’	’
" ’ ’ "	’ ’
’J " "K ’	J " "K
"J " " " "K "	J " "K
’J ’ ’ ’ ’K ’	J ’ ’K
"J ’ ’K "	J ’ ’K
’L ’ ’M ’ ’N ’	L ’M ’N
"L ’M ’N "	L ’M ’N
’O "P "Q ’	O "P "Q
"O " "P " "Q "	O "P "Q
’R " "S " "T ’	R " "S " "T
"R " " " "S " " " "T "	R " "S " "T
’U ’ ’ ’ ’V ’ ’ ’ ’W ’	U ’ ’V ’ ’W
"U ’ ’V ’ ’W "	U ’ ’V ’ ’W

Hexadecimal Literals

A **hexadecimal literal** (a VSI extension to nonnumeric literals) is a character string of 2 to 256 hexadecimal digits. On the left it is delimited by the separator X (or x) immediately followed by a quotation mark (") or apostrophe ('); on the right it is delimited by a matching quotation mark or apostrophe. For example:

```
03 HEX_VAL PIC X VALUE X"00".
```

The character string consists only of pairs of hexadecimal digits representing a byte value ranging from 00 to FF; hence, only the characters 0 to 9, A to F, and a to f are valid.

The value of a hexadecimal literal is the composite value of the paired hexadecimal representations. The compiler truncates hexadecimal literals to a maximum of 128 hexadecimal representations (pairs of hexadecimal digits).

A hexadecimal literal can be used interchangeably wherever a nonnumeric literal can appear in VSI COBOL for OpenVMS syntax. (Thus, hexadecimal literals cannot be used as operands in arithmetic statements.)

Syntax Rules

1. A space, left parenthesis, or pseudo-text delimiter (==) must immediately precede the opening character X (or x).

2. The closing quotation mark or apostrophe must be immediately followed by one of the following:

   - Space

- Comma
- Semicolon
- Period
- Right parenthesis
- Pseudo-text delimiter

Table 1.7 provides examples of hexadecimal literals.

Table 1.7. Hexadecimal Literals

Literal	Value
X "00 "	NUL
x "0D "	CR
x "2424 "	$$
X ’7b7a ’	{z

National Literals

National literals can be from 0 to 128 2-byte characters (hence 256 bytes). The syntax is:

```
VALUE N"".
```

National literals are made available when /NATIONALITY=JAPAN or `-nationality japan` is specified.

1.2.3. 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. Table 1.8 lists the figurative constants.

Table 1.8. Figurative Constants

Figurative Constant	Value

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 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. For example, HIGH-VALUE for the native collating sequence is hexadecimal FF.

Figurative Constant	Value
	The value of HIGH-VALUE depends on the collating sequence specified by clauses in the OBJECT-COMPUTER and SPECIAL-NAMES paragraphs. For example, if the program collating sequence is ASCII, HIGH-VALUE is hexadecimal 7F (hexadecimal FF for EBCDIC). For more information, see OBJECT-COMPUTER and SPECIAL-NAMES sections in Chapter 4.
LOW-VALUE, LOW-VALUES	Represent one or more occurrences of the character with the lowest ordinal position in the program collating sequence (hexadecimal 00 for the native collating sequence). The value of LOW-VALUE depends on the program collating sequence specified by clauses in the OBJECT-COMPUTER and SPECIAL-NAMES paragraphs. For more information, see the OBJECT-COMPUTER and SPECIAL-NAMES sections in Chapter 4.

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 bound 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 making up the literal. The literal must be either nonnumeric, a symbolic-character, or a figurative constant other than ALL literal. For a figurative constant, the word ALL is redundant and serves only to enhance readability.[1]
Symbolic-character	Represents one or more occurrences of the character specified as the value of symbolic-character. (See SPECIAL-NAMES in Chapter 4.)

[1]The reserved word ALL, not followed by a literal, can be a subscript of an identifier that is a function argument. (The function must allow a variable number of arguments in this argument position; see Chapter 7.)

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

- 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.

- 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 or ZEROES, plural) is the only valid figurative constant for literals restricted to numeric characters.

The actual characters associated with HIGH-VALUE, HIGH-VALUES, LOW-VALUE, and LOW-VALUES depend on the program collating sequence. For more information, see OBJECT-COMPUTER and SPECIAL-NAMES in Chapter 4.

1.2.4. 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 PICTURE.)

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.

1.2.5. Separators

A **separator** delimits character-strings. It can be one character or two contiguous characters formed according to the rules in Table 1.9.

Table 1.9. Separators

Separator	Usage Rules
Space	The space can be a separator or part of a separator. ● Where a space is used as a separator or part of a separator, more than one space can be used. ● A space can immediately precede any separator except: • As specified by the rules for reference formats (see Section 1.3) • The closing quotation mark of a nonnumeric literal; the space is then considered part of the nonnumeric literal rather than a separator ● 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: ● Statement and sentence structure definitions (see Section 6.1) ● Reference format rules (see Section 1.3)
Parentheses	Parentheses can be used only in balanced pairs of left and right parentheses to delimit: ● Subscripts

Separator	Usage Rules
	● Indexes ● Arithmetic expressions ● Conditions ● Reference modification ● Boolean expressions ● Intrinsic function argument lists
Quotation Marks Apostrophes	An opening quotation mark or apostrophe must be immediately preceded by a separator space or a left parenthesis. A closing quotation mark (") or apostrophe (') 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.)
Pseudo-Text Delimiter	The pseudo-text delimiter is two contiguous equal signs (==), both of which must be on the same source line. A space must immediately precede an opening pseudo-text delimiter. One of the following separators must immediately follow a closing pseudo-text delimiter: spaces, commas, semicolons, or periods. Pseudo-text delimiters can be used only in balanced pairs. They delimit pseudo-text. (See Chapter 8.)
Colon	The separator colon delimits operands in reference modification. It is required when shown in a general format. (See Section 6.2.3.)

1.3. Source Reference Format

The VSI COBOL for OpenVMS compiler recognizes two source program formats: ANSI and terminal.

- ANSI format conforms to the American National Standard COBOL reference format.
- Terminal format is a concise VSI specified format. It shortens source program lines by allowing horizontal tab characters and carriage returns. In terminal format, you do not use the ANSI format sequence numbers or identification area.

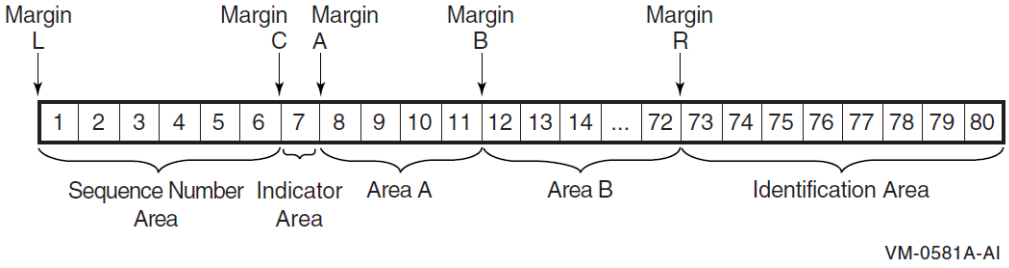
By default, the compiler expects terminal-format source lines. The compiler expects ANSI format only when the command line includes the `ansi` compiler option.

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

1.3.1. ANSI Format

The **ANSI source reference format** describes COBOL programs in terms of character positions on an input line. A source program line has 80 character positions as shown in Figure 1.1.

Figure 1.1. Source Program Line



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 characters from the computer character set.

The compiler does not check the uniqueness of the contents. However, the compiler does check for the ascending sequence of the contents if the compiler command line includes the `sequence` compiler option.

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. However, the source line appears on the program listing.

Character	Source Line Interpretation
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.
A-Z, a-z	Conditional compilation lines. The compiler processes the line as normal COBOL text if you specify the DEBUGGING MODE clause in the SOURCE-COMPUTER paragraph, or if you specify the `conditionals` compiler option in the command line. If you do not specify either, the compiler processes this line as a comment line.

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. However, the contents appear on the source program listing.

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 ends with 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. All spaces at the end of the continued line are part of the literal. Area A of the continuation line must be blank.

If the indicator area is blank:

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

ANSI Format Example

```
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.

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 library text.

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 characters from the computer character set. Comment lines can be anywhere in a source program or library text.

Conditional Compilation Lines

A conditional compilation line is any source line after the OBJECT COMPUTER paragraph that includes one of these uppercase or lowercase alphabetic characters in its indicator area: A to Z, a to z. The compiler processes the line as normal COBOL text if you specify the DEBUGGING MODE clause in the SOURCE COMPUTER paragraph.

The compiler processes the line as normal COBOL text if you include the appropriate `conditionals` compiler option in the command line.

If you specify neither, the compiler processes this line as a comment line.

Lines conditioned by one letter can be compiled or treated as comments independently of other conditional compilation lines. On OpenVMS systems, for instance, if you compile with /CONDITIONALS=(A,B), lines conditioned with A and B compile while those conditioned by other letters are treated as comments.

See Chapter 8 for additional information on the interaction between conditional compilation lines and the COPY statement.

Pseudo-Text

Pseudo-text character-strings and separators can start in either Area A or Area B. However, if there is a hyphen in the indicator area of a line that follows the opening pseudo-text delimiter, Area A of the line must be blank.

The normal rules for line continuation apply to the formation of text-words.

Pseudo-text is described in Chapter 8.

Short Lines and Tab Characters

If the source program input medium is not punched cards, carriage return and horizontal tab characters can shorten source lines.

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, 12, 20, 28, 36, 44, 52, 60, 68, and 76.

The following example shows how the compiler interprets carriage return and horizontal tab characters in a source program:

Shortened ANSI Format Source Line

```
000100*The following record description shows the source line format Return
000110 01 Tab RECORD-A. Return
000120 Tab Tab03 GROUP-A. Return
000130 Tab Tab Tab05 ITEM-A Tab PIC X(10). Return
000140* Tab The tab character in the nonnumeric literal Return
000150* Tab on the next line is stored as one character Return
000160 Tab Tab Tab05
ITEM-B Tab PIC X VALUE IS " Tab". Return
000170 Tab Tab03 ITEM-C Tab Tab PIC X(10). Return
000180D01 Tab RECB REDEFINES RECORD-A Tab PIC X(21). Return
```

Source Line as Interpreted by the 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 " Tab".
000170 03 ITEM-C PIC X(10).
000180D01 RECB REDEFINES RECORD-A PIC X(21).
```

Use more tab characters only when necessary. Compiler error diagnostics result if you use tab characters beyond the permissible character positions for a COBOL statement or entry. The following example shows how the compiler treats source program lines 000004 and 000005. Line 000004: contains one too many tab characters, which places paragraph-name P0 out of Area A.

Shortened ANSI Format Source Line

```
000001 TabIDENTIFICATION DIVISION.
000002 TabPROGRAM-ID. ANSI-TEST.
000003 TabPROCEDURE DIVISION.
000004 Tab TabP0.
000005 Tab TabSTOP RUN.
```

Listing File Result on OpenVMS Alpha, I64

```
000001 IDENTIFICATION DIVISION.
000002 PROGRAM-ID. ANSI-TEST.
000003 PROCEDURE DIVISION.
000004 P0.
.........^
%COBOL-F-UNDEFSYM, Undefined name
at line number 4 in file DISK:[DIRECTORY]ANSI.COB;1
000005 STOP RUN.
........^
%COBOL-W-SYN6, Missing paragraph header
at line number 5 in file DISK:[DIRECTORY]ANSI.COB;1
```

Listing File Result on UNIX

```
000001 IDENTIFICATION DIVISION.
000002 PROGRAM-ID. ANSI-TEST.
000003 PROCEDURE DIVISION.
cobol: Severe: dwork/t.cob, line 4: Undefined name
000004 P0.
--------^
cobol: Warning: dwork/t.cob, line 5: Missing paragraph header
000005 STOP RUN.
--------^
```

Note

The previous error messages have no additional online explanations. If a diagnostic message has a further explanation, an asterisk (*) is displayed (to the left of the error message). On OpenVMS Alpha and I64 systems, the VSI COBOL for OpenVMS online help file lists and describes error messages that have further explanations.

1.3.2. Terminal Format

The VSI COBOL for OpenVMS **terminal format** shortens program preparation time and reduces storage space for source programs. This format eliminates the sequence number and identification areas. It also combines the indicator area with Area A. Except for the differences described in this section, the rules for ANSI format also apply to terminal-format source programs.

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

(-) hyphen
(*) asterisk
(/) slash

The compiler also recognizes the following conditional compilation line characters as valid indicator area characters in the first and second character positions:

(\x) backslash and x

where x can be any uppercase or lowercase alphabetic character.

Area A then begins in character position 2 (or 3 if using \x). Otherwise, Area A begins in the first character position.

Area B begins four character positions to the right of the beginning of Area A. It ends when the compiler detects a carriage return, or at Margin R.

The maximum length of a terminal-format source line is 256 characters. The compiler's tab stops are immediately to the right of Margin B, and every eight character positions to the right, until the end of the line.

Note

The maximum length of the source line on the program listing is 125 characters, including the sequence field. The compiler processes the complete source line but displays only the first 125 characters on the

listing. It also replaces all nonprintable ASCII characters with periods (or other symbols depending on the device) in the listing file. (Refer to the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/].)

The following example shows source lines in terminal format. It is equivalent to the ANSI-format source line examples in the previous section.

```
*The following record description shows the source line format Return
01 Tab RECORD-A. Return
Tab03 GROUP-A. Return
Tab Tab05 ITEM-A Tab PIC X(10). Return
* Tab The tab character in the nonnumeric literal Return
* Tab on the next line is stored as one character Return
Tab Tab05 ITEM-B Tab PIC X VALUE IS " Tab". Return
Tab03 ITEM-C Tab Tab PIC X(10). Return
\D01 Tab RECB REDEFINES RECORD-A Tab PIC X(21). Return
```

1.4. Sample Entry Format

The following format is used to describe most entries in this manual. Each COBOL division or major topic begins a new chapter and each entry begins on a new page. The entries are in functional or alphabetical order.

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 appear in the rules in *italic type*.

Syntax Rules

Syntax rules define or clarify the arrangement of words or elements. They can also impose further restrictions or relax restrictions implied by the general format.

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.

Technical Notes

Technical notes describe, in system-specific terms, any system-specific behavior, and any other VSI COBOL for OpenVMS behavior of note not described in the rules. They define relationships between the COBOL program and the operating system and its components.

Additional References

Additional references point to other relevant information in this manual, the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/], and other VSI documentation sets.

Examples

Examples show the use of a statement, clause, or other entry. The *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/] contains other examples in application contexts.

The following example shows a general format:

General Format

```
  identification-division
[ environment-division ]
[ data-division ]
[ procedure-division ]
[ source-program ] ...
[ end-program-header ]
```

Additional References

- Chapter 3
- Chapter 4
- Chapter 5
- Chapter 6

Chapter 2. Organization of a COBOL Program

A COBOL source program is a syntactically correct set of COBOL statements that:

- Mark the beginning of the program
- Describe its physical environment
- Describe the data the program creates, receives as input, manipulates, and produces as output
- Specify the processing of the program's files and data

General Format

```
  identification-division
[ environment-division ]
[ data-division ]
[ procedure-division ]
[ source-program ]...
[ end-program-header ]
```

`identification-division`

represents a COBOL Identification Division.

`environment-division`

represents a COBOL Environment Division.

`data-division`

represents a COBOL Data Division.

`procedure-division`

represents a COBOL Procedure Division.

`source-program`

represents a contained (nested) COBOL source program. A COBOL source program may be nested; more than one source program may be present in a single source file.

`end-program-header`

represents a COBOL END PROGRAM header.

Syntax Rule

The end-program-header must be present if either:

1. The COBOL source program contains one or more other COBOL source programs.
2. The COBOL source program is contained within another COBOL source program.
3. The COBOL source program precedes another separately compiled program.

General Rules

1. The appropriate division header indicates the beginning of a division.
2. The following indicates the end of a division:
   a. Another division header
   b. An Identification Division header that indicates the start of another source program
   c. The end-program-header
   d. The physical position at which no further source lines occur
3. A COBOL source program may contain other COBOL source programs.
4. A COBOL source program that is directly or indirectly contained within another program is called a contained or nested program. It may reference certain resources in the containing program.
5. A separately compiled program has a nesting level number of 1. If this program contains other source-programs, it is the outermost containing program.
6. A contained program has a nesting level number greater than 1.

Additional References

- Identification Division
- Environment Division
- Data Division
- Procedure Division
- END PROGRAM Header

Program Structure

Figure 2.1 shows the basic structure of a COBOL program, which is organized in divisions, sections, paragraphs, sentences, and entries.

Figure 2.1. Structure of a COBOL Program

```
IDENTIFICATION DIVISION.
PROGRAM-ID. program-name.
AUTHOR.
INSTALLATION.
DATE-WRITTEN.
DATE-COMPILED.
SECURITY.
OPTIONS.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.
OBJECT-COMPUTER.
SPECIAL-NAMES.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
I-O-CONTROL.
DATA DIVISION.
SUBSCHEMA SECTION.
    subschema entries and keeplist entries
FILE SECTION.
    file and record description entries
    report file description entries
    sort-merge file and record description entries
WORKING-STORAGE SECTION.
    record description entries
LINKAGE SECTION.
    record description entries
REPORT SECTION.
    report and report group description entries.
SCREEN SECTION. (Alpha, I64)
    screen description entries (Alpha, I64)
PROCEDURE DIVISION.
DECLARATIVES.
    sections
        paragraphs
            sentences
END DECLARATIVES.
.
.
.
sections
    paragraphs
        sentences
.
.
.
END PROGRAM header
```

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 and REPLACE statements, and the END PROGRAM header (see END PROGRAM in Chapter 6), the statements, entries, paragraphs, and sections of a COBOL source program are grouped into four divisions in this order:

1. IDENTIFICATION DIVISION.
2. ENVIRONMENT DIVISION.
3. DATA DIVISION.
4. PROCEDURE DIVISION.

The end of a COBOL source program is indicated either by the END PROGRAM header (END PROGRAM) or by the end of that program's Procedure Division.

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
- A DECLARATIVES header for the USE procedure sections (after the PROCEDURE DIVISION header only)
- A PROGRAM-ID paragraph (after the IDENTIFICATION DIVISION header only)

Only this item can immediately follow a DECLARATIVES header:

- A section header for a USE procedure

Note

The PROCEDURE DIVISION header can contain a USING and GIVING phrase. (See Section 6.8.)

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.
```

REPORT SECTION.
SCREEN SECTION. (Alpha, I64)

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)

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 as follows:

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

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

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 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

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, file connector, or screen item
3. A sequence of independent descriptive clauses
4. A separator period

The level indicators are as follows:

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

Level indicators can begin anywhere to the right of Area A.

Entries that begin with level-numbers are called either data description or screen description entries, depending on their context. The level-number values are 01 to 49, 66, 77, and 88 for data description items and 01 to 49 for screen description entries. Level-numbers 01 to 09 can be represented as one- or two-digit numbers.

All data description entries and screen description entries can begin anywhere to the right of Margin A. However, indentation has no effect on level-number magnitude; it merely enhances readability.

Declaratives

Declaratives specify USE procedures to be executed only when certain conditions occur. You must write USE 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 be on a line by itself, starting in Area A; and be followed by a separator period. For example:

```
PROCEDURE DIVISION.
DECLARATIVES.
IOERROR SECTION.
USE AFTER ... .
PAR-1.
.
.
.
END DECLARATIVES.
```

When you specify USE procedures, you must divide the remainder of the Procedure Division into sections.

Chapter 3. 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      [ IS { COMMON | INITIAL } PROGRAM ]
      [ WITH IDENT ident-string ] .
  [ AUTHOR. [ comment-entry ] . . . ]
* [ INSTALLATION. [ comment-entry ] . . . ]
* [ DATE-WRITTEN. [ comment-entry ] . . . ]
  [ DATE-COMPILED. [ comment-entry ] . . . ]
* [ SECURITY. [ comment-entry ] . . . ]
  [ OPTIONS. [ ARITHMETIC IS { NATIVE | STANDARD } ] ] (Alpha, I64)
```

* 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.

PROGRAM-ID

PROGRAM-ID — The PROGRAM-ID paragraph identifies a program and assigns selected program attributes.

General Format

```
PROGRAM-ID. program-name
    [ IS { COMMON | INITIAL } PROGRAM ]
    [ WITH IDENT ident-string ] .
```

[program-name]

is a user-defined word that names the program.

Syntax Rules

1. The PROGRAM-ID paragraph must be present in every program.
2. *program-name* must contain 1 to 31 characters and follow the rules for user-defined words.
3. Programs contained within a separately compiled program must have a unique *program-name*.
4. The optional COMMON clause may be used only if the program is contained within another program.
5. *ident-string* must be a nonnumeric literal 1 to 31 characters in length.
6. The optional IDENT clause cannot be used in a contained program.

General Rules

1. *program-name* is a user-defined word that identifies a COBOL program and its source listing. It 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, each separately compiled program must have a unique *program-name*.
4. The COMMON clause specifies a common program. A common program is contained within another program but may be called from programs other than that directly containing it.
5. Files associated with a called program's internal file connectors are not in the open mode:

   a. The first time the program is called

   b. The first time the program is called after execution of a CANCEL statement referring to the program

   c. Every time the program is called, if it has the INITIAL attribute

   On all other entries, the status and positioning of files in a called program are the same as when the program last exited.
6. The INITIAL clause specifies an initial program. Whenever the program is called, it and any programs contained within it are placed in their initial state, and the internal data in each program is initialized.
7. On OpenVMS, the IDENT clause specifies a literal string that is used for identification purposes. This string is written to the object file as the "module version."

   When the /ANALYSIS_DATA qualifier is included on the COBOL command, the string is written to the analysis data file as the module ident.
8. On UNIX systems, *program-name* is case-sensitive. By default, *program-name* is converted to lowercase for all separately compiled program units. Any calls from other programs (VSI COBOL for OpenVMS as well as other languages) must specify the routine to be called in lowercase.

   However, if the `names` option is set to `uppercase` on the command line, calls from other programs must specify the routine to be called in uppercase. If the `names` option is set to `as_is`,

the effect on *program-name* is as if `uppercase` were specified. (The `as_is` setting is used for calling non-COBOL programs with mixed case.)

Examples

```
PROGRAM-ID. PROGA.
PROGRAM-ID. SUBR1 INITIAL.
PROGRAM-ID. COMPUTE-PAY WITH IDENT "JOB6a-V1.1". (OpenVMS)
PROGRAM-ID.
WRITEMASTERREPORT.
PROGRAM-ID. PAYROLL IS COMMON.
Identification
```

Additional Reference

See Section 6.2.6: Scope of Names.

AUTHOR

AUTHOR — The AUTHOR paragraph is for documentation only.

General Format

AUTHOR [comment-entry] […]

[comment-entry]

is a user-supplied comment about the program's author.

Syntax Rules

1. *comment-entry* can consist of any combination of characters from the computer character set.
2. *comment-entry* 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. JOHN SMITH.

AUTHOR. This program was written by John Smith
                                    1226 Main St.
                                    Merrimack, NH 03054


AUTHOR.
```

DATE-COMPILED

DATE-COMPILED — The DATE-COMPILED paragraph provides the compilation date in the source program listing file.

General format

DATE-COMPILED [comment-entry] […]

[comment-entry]

is user-supplied information about the date compiled.

Syntax Rules

1. *comment-entry* can consist of any combination of characters from the computer character set.
2. *comment-entry* can span several lines in Area B. However, it 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.

General Rule

The paragraph-name DATE-COMPILED causes the current date to be inserted in your source program listing during compilation. Therefore, if a DATE-COMPILED paragraph is present in your source program, it will be replaced with a paragraph of the following form:

```
DATE-COMPILED. dd-mmm-yyyy.
```

OPTIONS

OPTIONS — The OPTIONS paragraph specifies information for use by the compiler in generating executable code for a source unit.

General Format

OPTIONS. [arithmetic-clause]

[arithmetic-clause]

specifies the method used in developing the intermediate results. The format is:

ARITHMETIC IS { NATIVE | STANDARD }

Syntax Rule

The period appearing in the general format after the arithmetic-clause may be omitted if the arithmetic-clause is not specified.

General Rules

1. The ARITHMETIC clause in the OPTIONS paragraph applies to the source element in which it is specified and to all source elements contained in that source element unless overridden by an ARITHMETIC clause in an OPTIONS paragraph in a contained source element.

2. If the NATIVE phrase is specified, the techniques used in handling arithmetic expressions and arithmetic statements shall be those specified for native arithmetic in the appendix on compatibility in the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/].

3. If the STANDARD phrase is specified, the techniques used in handling arithmetic expressions and arithmetic statements shall be those specified for standard arithmetic in the ANSI Standard for COBOL. (Refer to the appendix on compatibility in the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/].)

4. If the ARITHMETIC clause is not specified in this source element or a containing source element, it is as if the ARITHMETIC clause were specified with the NATIVE phrase.

Chapter 4. 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.

```
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 ] ] ]
```

The Environment Division can contain two sections:

- Configuration Section (see Section 4.1)
- Input-Output Section (see the section called “INPUT-OUTPUT”)

Syntax Rules

1. The Environment Division follows the Identification Division.
2. The general format defines the order of appearance of Environment Division entries.
3. A contained program cannot include a Configuration Section.

General Format

Explicit or implicit Configuration Section entries in a program containing other programs apply to each contained program.

4.1. CONFIGURATION

The *Configuration Section* can contain three paragraphs:

- SOURCE-COMPUTER paragraph (see SOURCE-COMPUTER)
- OBJECT-COMPUTER paragraph (see OBJECT-COMPUTER)
- SPECIAL-NAMES paragraph (see SPECIAL-NAMES)

The Configuration Section must not be stated in a program that is contained within another program. If Configuration Section entries are stated in a program that contains other programs, they apply to each contained program.

SOURCE-COMPUTER

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

Format

```
SOURCE-COMPUTER.
    [ { ALPHA
        I64
        VAX
        computer-type } [ WITH DEBUGGING MODE ] . ]
```

computer-type

is a user-defined word that names the computer.

Syntax Rule

ALPHA and *I64* are system-names. They are not reserved words, and are for documentation only.

General Rules

1. If the WITH DEBUGGING MODE clause is not used, this paragraph is for documentation only.
2. All clauses of the SOURCE-COMPUTER paragraph apply to the program that specifies them. They also apply to any program contained within that program.
3. If you include the WITH DEBUGGING MODE clause in a program, or if you specify the `conditionals` command-line option, all conditional compilation lines are compiled. Otherwise, the compiler treats all conditional compilation lines as comment lines. (See Section 1.3.1 for additional information about source line interpretation.)

OBJECT-COMPUTER

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

Format

```
OBJECT-COMPUTER.
[ { ALPHA
    I64
    VAX
    computer-type }
  [ MEMORY SIZE integer { WORDS
                          CHARACTERS
                          MODULES } ]
  [ PROGRAM COLLATING SEQUENCE IS alpha-name ]
  [ SEGMENT-LIMIT IS segment-number ] . ]
```

computer-type

is a user-defined word that names the computer.

integer

is a numeric literal that has no digits to the right of the assumed decimal point.

alpha-name

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

`segment-number`

is an integer from 1 to 49.

Syntax Rule

ALPHA and *I64* are system-names. They are not reserved words, and are for documentation only.

General Rules

1. All clauses of the OBJECT-COMPUTER paragraph apply to the program that explicitly or implicitly specifies them. They also apply to any program contained within that program.
2. The MEMORY SIZE clause is for documentation only. It has no effect on program execution.
3. The PROGRAM COLLATING SEQUENCE clause causes the program to use the collating sequence of *alpha-name* to determine the truth value of nonnumeric comparisons in:
   - Relation conditions
   - Condition-name conditions
   - Report description entries, the CONTROL clause
4. 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.
5. If there is no PROGRAM COLLATING SEQUENCE clause, the program uses the NATIVE collating sequence.
6. The SEGMENT-LIMIT clause is for documentation only.

Additionally, refer to the information on SORT and MERGE statements in the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/].

Examples

1. Computer name only:

   ```
   OBJECT-COMPUTER. Alpha.
   ```

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

   ```
   OBJECT-COMPUTER.
   ```

3. With PROGRAM COLLATING SEQUENCE clause:

   ```
   OBJECT-COMPUTER. Alpha
   PROGRAM COLLATING SEQUENCE IS ALPH-A.
   ```

   The SPECIAL-NAMES paragraph must define ALPH-A.

4. With PROGRAM COLLATING SEQUENCE clause:

   ```
   OBJECT-COMPUTER. Alpha
   ```

```
SEQUENCE IS EBCDIC.
```

The SPECIAL-NAMES paragraph must define EBCDIC.

If EBCDIC refers to the EBCDIC collating sequence, the SPECIAL-NAMES paragraph must contain the following clause:

```
ALPHABET EBCDIC IS EBCDIC
```

Additional References

- SPECIAL-NAMES Paragraph
- SPECIAL-NAMES Paragraph
- Section 6.5.1
- Section 6.5.3
- Section 6.7

SPECIAL-NAMES

SPECIAL-NAMES — The *SPECIAL-NAMES* paragraph: (1) associates compiler features and logical names (on OpenVMS systems) or environment variables (on UNIX systems) with user-defined mnemonic-names, (2) provides a way to reference command-line arguments and (on UNIX) environment variables or (on OpenVMS) logical names with user-defined mnemonic names, (3) defines symbolic-characters, (4) specifies the currency sign, (5) selects the decimal point, (6) relates alphabet-names to character sets or collating sequences, (7) relates class-names to character sets, (8) provides for cursor positioning for an ACCEPT (Format 5) statement, and (9) provides information on the cause of termination of an ACCEPT (Format 5) statement.

Format

```
SPECIAL-NAMES . [

[ { CARD-READER
    PAPER-TAPE-READER
    CONSOLE
    LINE-PRINTER
    PAPER-TAPE-PUNCH
    SYSIN (Alpha, I64)
    SYSOUT (Alpha, I64)
    SYSERR (Alpha, I64) } IS device-name

  [ ARGUMENT-NUMBER IS argument-number (Alpha, I64)
    ARGUMENT-VALUE IS argument-value (Alpha, I64)
    ENVIRONMENT-NAME IS environment-name (Alpha, I64)
    ENVIRONMENT-VALUE IS environment-value (Alpha, I64) ]

  C01 IS top-of-page-name

  SWITCH switch-num

  { IS switch-name
      [ ON STATUS IS cond-name ]
      [ OFF STATUS IS cond-name]

    IS switch-name
      [ OFF STATUS IS cond-name]
      [ ON STATUS IS cond-name ]

    ON STATUS IS cond-name [ OFF STATUS IS cond-name ]
    OFF STATUS IS cond-name [ ON STATUS IS cond-name ] } ]
```

```
[ ALPHABET alpha-name IS
  { ASCII
    STANDARD-1
    STANDARD-2
    NATIVE
    EBCDIC
    { first-literal [ { THRU } last-literal ] } ...
                    { THROUGH }
                    { ALSO lit } ... ] } ... ]
[ SYMBOLIC CHARACTERS
  { { { symbol-char } ... { IS } { char-val } ... } ... [ IN alpha-name ] } ... ] ...
                          { ARE }
[ CLASS class-name IS
  { first-literal [ { THRU } last-literal ] } ... ] ...
                    { THROUGH }
[ CURRENCY SIGN IS char ]
[ CURRENCY SIGN IS { char
                     literal-7 [WITH PICTURE SYMBOL literal-8] } ] ... (Alpha, I64)
[ DECIMAL-POINT IS COMMA ]
[ CURSOR IS cursor-position] (Alpha, I64)
[ CRT STATUS IS crt-status-code ] (Alpha, I64) . ]
```

`device-name`

is a user-defined word for a device. Only the ACCEPT and DISPLAY statements can refer to it.

`argument-number`

is a user-defined word that contains the current argument position indicator number when used with DISPLAY, or the count of command line arguments when used with ACCEPT. Only the ACCEPT and DISPLAY statements can refer to it.

`argument-value`

is a user-defined word that contains the value of the current command line argument as indicated by the current ARGUMENT-NUMBER. Only the ACCEPT and DISPLAY statements can refer to it.

`environment-name`

is a user-defined word that contains the name of an environment variable or system logical. Only the ACCEPT and DISPLAY statements can refer to it.

`environment-value`

is a user-defined word that contains the value of the environment variable or logical named by the current ENVIRONMENT-NAME. Only the ACCEPT and DISPLAY statements can refer to it.

`top-of-page-name`

is a user-defined word for the top of a page. Only the WRITE statement can refer to it.

`switch-num`

is the number of a program switch. Its value can range from 1 to 16.

`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. It always possesses the global attribute. Its truth value is true when the STATUS phrase matches the status of the switch, false when it does not.

`alpha-name`

is the user-defined word for a character set, collating sequence, or both. It always possesses the global attribute.

`first-literal`

is a literal. It specifies either: (1) the value of one or more alphabet characters, or (2) the first in a range of values.

`last-literal`

is a literal. It specifies the last in a range of values.

`lit`

is a literal. It specifies an alphabet character value.

`symbol-char`

is a user-defined word that names the symbolic-character. It always possesses the global attribute. The same *symbol-char* cannot appear more than once in the SYMBOLIC CHARACTERS clause.

`char-val`

is an integer that indicates the ordinal position of a character in the native character set.

`class-name`

is the user-defined word for a class. It always possesses a global attribute.

`char`

is a one-character nonnumeric literal that specifies the currency symbol. It cannot be a symbolic-character or figurative constant.

`literal-7 (Alpha, I64)`

is an alphanumeric literal. It cannot be a figurative constant.

`literal-8 (Alpha, I64)`

is an alphanumeric literal consisting of a single character. It cannot be a figurative constant. No two occurrences of *literal-8* can have the same value.

`cursor-position (Alpha, I64)`

is a data item declared in the Working-Storage Section of the program. It is either an elementary unsigned numeric integer either four or six characters in length, described as USAGE IS DISPLAY, or a group item either four or six characters in length, consisting of two elementary unsigned data items.

`crt-status-code (Alpha, I64)`

is a group data item three characters in length, declared in the Working-Storage Section of the program.

Syntax Rules

1. In the first-literal phrase of the ALPHABET or CLASS clauses:
   - If *alpha-name* is in the PROGRAM COLLATING SEQUENCE clause, the ALPHABET clause cannot specify any character more than once.
   - If the ALSO or THRU phrase appears, *first-literal* must be one character long.
   - Numeric literals must be unsigned integers from 1 to 256.
   - If *last-literal* or *lit* is nonnumeric, it must be one character long.
   - THRU and THROUGH are equivalent.
2. If the *first-literal* phrase appears, *alpha-name* cannot be referenced in a CODE-SET clause.
3. The following are accessible only by ACCEPT and DISPLAY statements:

   argument-count
   argument-value
   environment-name
   environment-value

General Rules

1. All clauses of the SPECIAL-NAMES paragraph apply to the program defining them and to all programs contained within that program.

device-name Clause

2. The device-name clause associates a device with a user-defined word (*device-name*).

   On UNIX, the device name is derived from an environment variable, if that environment variable exists. Otherwise, the defaults are as follows:

System-Name	UNIX Environment Variable	UNIX Default File Name
CARD-READER	COBOL_CARDREADER	`stdin`
PAPER-TAPE-READER	COBOL_PAPERTAPEREADER	`stdin`
CONSOLE	COBOL_CONSOLE	`stderr`
LINE-PRINTER	COBOL_LINEPRINTER	`stdout`
PAPER-TAPE-PUNCH	COBOL_PAPERTAPEPUNCH	`stdout`
SYSIN	COBOL_INPUT	`stdin`

System-Name	UNIX Environment Variable	UNIX Default File Name
SYSOUT	COBOL_OUTPUT	`stdout`
SYSERR	COBOL_ERROR	`stderr`

The input device for the ACCEPT statement is derived from COBOL_INPUT, if defined, and defaults to `stdin`. The output device for the DISPLAY statement is derived from COBOL_OUTPUT, if defined, and defaults to `stdout`.

On OpenVMS, the file-name is derived from a logical name if that logical name exists. Otherwise, the defaults are as follows:

System-Name	OpenVMS Logical Name	OpenVMS Default File Name
CARD-READER	COB$CARDREADER	SYS$INPUT
PAPER-TAPE-READER	COB$PAPERTAPEREADER	SYS$INPUT
CONSOLE	COB$CONSOLE	SYS$ERROR
LINE-PRINTER	COB$LINEPRINTER	SYS$OUTPUT
PAPER-TAPE-PUNCH	COB$PAPERTAPEPUNCH	SYS$OUTPUT
SYSIN (Alpha, I64)	COB$INPUT	SYS$INPUT
SYSOUT (Alpha, I64)	COB$OUTPUT	SYS$OUTPUT
SYSERR (Alpha, I64)	COB$ERROR	SYS$ERROR

The input device for the ACCEPT statement is derived from COB$INPUT, if defined, and defaults to SYS$INPUT. The output device for the DISPLAY statement is derived from COB$OUTPUT, if defined, and defaults to SYS$OUTPUT. (See the ACCEPT and DISPLAY statements in Chapter 6, and refer to the *VSI COBOL User Manual [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/]* for more information.)

top-of-page-name Clause

3. The system-name C01 refers to the first line of a logical page. Only the ADVANCING phrase of the WRITE statement can refer to the top-of-page-name equated to C01. (See the WRITE statement in Chapter 6.)

SWITCH Clause

4. 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.

   Switches can also be read from the OpenVMS logical name COB$SWITCHES or the UNIX environment variable COBOL_SWITCHES.

   The compiler interprets SWITCH *n* and SWITCH- *n* (where *n* represents a number from 1 to 8) as identical clauses. For example, SWITCH 1 is equivalent to SWITCH-1.

   Refer to the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/] for more information on using switches.

ALPHABET Clause

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

The ALPHABET clause specifies:

- A character code set, when *alpha-name* is in a CODE-SET clause in the FILE-CONTROL paragraph or file description entry.
- A collating sequence, when *alpha-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.

6. ASCII refers to the character set defined in *American National Standard X3.4-1968,* "Code for Information Interchange."
7. STANDARD-1 refers to the ASCII character set.
8. STANDARD-2 refers to the international version of the ISO 7-bit code. It is defined in *International Standard 646,* "7-Bit Coded Character Set for Information Processing Interchange. "
9. 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 VSI.
10. EBCDIC refers to the EBCDIC character set or collating sequence. It is defined in Appendix B.
11. The character with the highest ordinal position in the program collating sequence equals the figurative constant HIGH-VALUE, except when this figurative constant is specified as a literal in the SPECIAL-NAMES paragraph. If more than one character has the highest position, HIGH-VALUE is the last character you specify.
12. The character with the lowest ordinal position in the program collating sequence equals the figurative constant LOW-VALUE, except when this figurative constant is specified as a literal in the SPECIAL-NAMES paragraph. If more than one character has the lowest position, LOW-VALUE is the first character you specify.

Literals in the ALPHABET Clause

13. The value of each numeric literal specifies the ordinal number of a character in the native character set. For example, 66 refers to the ASCII character A.
14. The value of each nonnumeric literal specifies the actual character in the native character set.
15. If the literal contains more than one character, the compiler interprets each character from left to right. It assigns each a successive ascending position in the collating sequence or character code set.
16. The order of appearance of literals in the ALPHABET clause specifies each character's ordinal number in ascending sequence. If the ALPHABET clause defines a character code set, the ordinal number identifies the character's relative position in the set.
17. Any unspecified characters in the native collating sequence have higher positions in the new collating sequence than all specified characters. The relative order of the unspecified characters is the same as in the native collating sequence.

    For example, the following clauses are equivalent:

```
ALPHABET XYZ IS 2 4
ALPHABET XYZ IS 2 4 1 3 5 6 7
ALPHABET XYZ IS 2 4 1
```

THROUGH Phrase

18. The THROUGH phrase specifies a set of contiguous characters in the native character set. The first character is *first-literal*; the last character is *last-literal*.

19. The compiler assigns each character in the set a successive ascending position in the collating sequence or character code set.

20. The THROUGH phrase can specify the set of contiguous characters in either ascending or descending order. For example, “L” THRU “H” assigns successively higher numbers to L, K, J, I, and H.

21. The ALSO phrase assigns *first-literal* and each *lit* to the same position in the collating sequence or character code set. For example, “A” ALSO “$” causes the characters A and $ to be equivalent in comparisons when the associated *alpha-name* is in the PROGRAM COLLATING SEQUENCE clause.

SYMBOLIC CHARACTERS Clause

22. Each *symbol-char* corresponds to the *char-val* in the same relative position. In the following example, CARRIAGE-RET corresponds to 14 and ESCAPE to 28:

```
SYMBOLIC CHARACTERS CARRIAGE-RET ESCAPE ARE 14 28
```

23. If the IN phrase is not specified, *symbol-char* represents the character, in the native character set, that has the ordinal position specified by *char-val*.

Note

The ordinal position is one greater than the internal representation of the character. For example, the character A is in ordinal position 66. Its internal representation is decimal 65 (hexadecimal 41).

---

24. If the IN phrase is specified, *char-val* represents the character that has the ordinal position specified by the IN *alpha-name* phrase.

CLASS Clause

25. The CLASS clause relates a name to a specified set of characters in that clause. *class-name* can be referenced only in a class condition. The characters specified by the values of the literals in this clause define the set of characters of which this *class-name* consists.

26. The value of each numeric literal specifies the ordinal number of a character in the native character set. This value must not exceed the value that represents the number of characters in the native character set.

27. The value of each nonnumeric literal specifies the actual character in the native character set. If the nonnumeric literal contains multiple characters, each character in the literal is included in the set of characters identified by *class-name*.

28. The THROUGH phrase specifies a set of contiguous characters in the native character set. The first character is *first-literal*; the last character is *last-literal*. The characters specified by a given THROUGH phrase can be specified in ascending or descending order.

CURRENCY SIGN Clause

29. 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, the lowercase characters a to z, or the space
- Asterisk (*), plus sign (+), minus sign (-), comma (,), period (.), semicolon (;), quotation mark ("), equal sign (=), slash (/), left parenthesis (( ), or right parenthesis ( ))

30. The CURRENCY SIGN clause cannot contain a symbolic-character or figurative constant.

31. If there is no CURRENCY SIGN clause, the default currency sign used for the PICTURE clause is the "$" symbol.

    On OpenVMS, if you define the logical name SYS$CURRENCY at DCL command level prior to compilation, the quoted character string to which you define it will be the currency string. To do this, prior to compiling the COBOL program, issue the following DCL command:

    ```
    $ DEFINE SYS$CURRENCY "quoted-character-string"
    ```

    The COBOL compiler will utilize the first character of this string as the currency symbol for the program.

    Subsequently, the system default value of SYS$CURRENCY can be restored for the process with the following DCL command:

    ```
    $ DEASSIGN SYS$CURRENCY
    ```

    The default currency sign can also be established based on the `nationality` compiler option, depending on the keyword, as follows:

US (default)	The default currency sign and symbol are the dollar sign ($), and Japanese language support features are disabled.
JAPAN	The default currency sign and symbol are the Yen sign (¥) (which is not overridden by a SYS$CURRENCY definition), and Japanese language support features are enabled, including national character user-defined-words, data items (PIC N), and literals (N"").

CURRENCY SIGN Clause (Alpha, I64)

32. To use CURRENCY SIGN IS *literal-7*, you must compile the program with the /RESERVED_WORDS=200X qualifier. Without that qualifier, you can specify only CURRENCY SIGN IS *char*, and specify it only once.

33. The CURRENCY SIGN IS *literal-7* clause specifies a currency string that is placed into numeric-edited data items when they are used as receiving items and de-edited from a data item when the data item is used as a sending item that has a numeric or numeric-edited receiving item. The clause also determines which symbol shall be used in a picture character string to specify the presence of this currency string. This symbol is referred to as the currency symbol. *literal-7* represents the value of the currency string.

    If the CURRENCY SIGN clause is specified with the PICTURE SYMBOL phrase, *literal-8* is the currency symbol; if the clause is specified without the PICTURE SYMBOL phrase, *literal-7* is the currency symbol, and it must be one character in length.

    If the currency symbol is a lowercase letter, it is treated as its uppercase equivalent.

34. If the PICTURE SYMBOL phrase is not specified, *literal-7* must consist of a single character that is not one of the following:

    - 0 through 9
    - A, B, C, D, E, N, P, R, S, V, X, Z, or the lowercase equivalents; or the space
    - Asterisk (*), plus sign (+), minus sign (-), comma (,), period (.), semicolon (;), quotation mark ("), equal sign (=), slash (/), left parenthesis (( ), or right parenthesis ( ))

35. If the PICTURE SYMBOL phrase is specified, *literal-7* can have any length and:

    - Must contain at least one nonspace character, and
    - Can consist of any characters from the computer's character set except for the digits 0 through 9 and the characters asterisk (*), plus sign (+), minus sign (-), comma (,), and period (.)

36. *literal-8* can be any character from the computer's character set except for the following:

    - 0 through 9
    - A, B, C, D, E, N, P, R, S, V, X, Z, or the lowercase equivalents; or the space
    - Asterisk (*), plus sign (+), minus sign (-), comma (,), period (.), semicolon (;), quotation mark ("), equal sign (=), slash (/), left parenthesis (( ), or right parenthesis ( ))

DECIMAL-POINT IS COMMA Clause

37. 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.

CURSOR IS Clause (Alpha, I64)

38. The CURSOR IS clause specifies the initial position of the cursor at the start of an ACCEPT (Format 5) statement. If *cursor-position* is within an input or update field on the screen, then the initial cursor position is at the start of that field. If the CURSOR IS clause is not specified, or if *cursor-position* is not within an input or update field on the screen, the cursor's initial position is at the start of the first input or update field of the screen. The *cursor-position* is updated upon completion of the ACCEPT statement to contain the position of the cursor when the ACCEPT terminated.

39. In the CURSOR IS clause, if *cursor-position* is four characters in length, the first two characters represent the line number, and the second two the column number. If *cursor-position* is six characters in length, the first three characters represent the line number, and the second three the column number.

CRT STATUS IS Clause (Alpha, I64)

40. If the CRT STATUS IS clause is specified, *crt-status-code* is updated after every ACCEPT (Format 5) statement. The first two characters are a termination code that indicates the cause of the termination of the ACCEPT operation. (The third character is currently not defined, and is reserved for future use.) The termination codes are explained in Table 4.1.

Command Line Arguments (Alpha, I64)

1. The ARGUMENT-NUMBER and ARGUMENT-VALUE clauses are used to process command line arguments. The DISPLAY statement is used to select and modify the values, and the ACCEPT statement is used to retrieve the values.

Environment Variables and System Logicals (Alpha, I64)

1. The ENVIRONMENT-NAME and ENVIRONMENT-VALUE clauses are used to process environment variables and system logicals. The DISPLAY statement is used to select and modify the values, and the ACCEPT statement is used to retrieve the values.

Table 4.1. CRT STATUS Termination Codes (Alpha, I64)

First Character	Second Character	Meaning
'0'	'0'	Terminator key pressed by the operator; normal completion
'0'	'1'	Auto-skip out of the last field; normal completion
'1'	x'00' – x'1A'	User-defined function key number for F1–F20 and the Find through Next keys[1]
'9'	x'00'	No items falling within the screen [1]

[1]The second character contains a hexadecimal value. An example of how to examine this value is given in the Examples section.

Examples

1. device-name clause:

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

On UNIX, this example allows ACCEPT and DISPLAY statements to use THE-CARDS to refer to the environment variable COBOL_CARDREADER and LOCAL-USER to refer to the environment variable COBOL_CONSOLE.

On OpenVMS, this example allows ACCEPT and DISPLAY statements to use THE-CARDS to refer to the logical name COB$CARDREADER and LOCAL-USER to refer to the logical name COB $CONSOLE.

2. Top-of-page-name clause:

```
C01 IS STARTING-NEW-FORM
```

The following WRITE statement causes the line to appear on the first line of a new page:

```
WRITE REPORT-REC AFTER STARTING-NEW-FORM.
```

3. SWITCH clause:

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

(Procedure Division statements can use the condition-names defined in the SWITCH clause. The SET statement can change the status of a switch.)

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

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

4. ALPHABET clause:

```
ALPHABET EB-CONV IS EBCDIC
```

If a file's SELECT clause contains a CODE-SET IS EB-CONV clause, this ALPHABET clause causes translation from EBCDIC to the native character set when the program reads data from the file.

5. User-defined collating sequence:

```
ALPHABET ALPH-B IS
"A" THRU "Z"
"9" THRU "0"
" " ALSO "/" ALSO "\"
","
```

This ALPHABET clause defines a collating sequence in which uppercase letters are lower than numeric characters. The space, slash (/), and backslash (\) characters have the same position in the collating sequence. The comma is the next higher character. It is implicitly followed by the rest of the character set.

The following Procedure Division conditional statements show the effect of this ALPHABET clause when the OBJECT-COMPUTER paragraph contains the PROGRAM COLLATING SEQUENCE IS ALPH-B clause:

Statements	Truth Value
MOVE “A” TO ITEMA.	
MOVE “9” TO ITEMB.	
IF ITEMA < ITEMB	true
MOVE “ ” TO ITEMA.	
MOVE “ \” TO ITEMB.	
IF ITEMA = ITEMB AND ITEMB > “Z”	true
MOVE “1” TO ITEMA.	
MOVE “9” TO ITEMB.	
IF ITEMA < ITEMB	false

6. User-defined collating sequence with numeric literals:

```
ALPHABET ALPH-C IS 128 THRU 1
```

This clause inverts the positions of the ASCII characters.

The following Procedure Division statements assume that the OBJECT-COMPUTER paragraph contains the SEQUENCE IS ALPH-C clause:

Statements	Truth Value
MOVE “A” TO ITEMA.	
MOVE “B” TO ITEMB.	

Statements	Truth Value
IF ITEMA < ITEMB	false
MOVE “9” TO ITEMA.	
IF ITEMA < “2”	true
MOVE “HELLO” TO ITEMA.	
IF ITEMA > SPACES	false

7. SYMBOLIC CHARACTERS clause:

```
SYMBOLIC CHARACTERS ESCAPE POUND DOUB-L ARE 28 36 55.
```

The following DISPLAY statement displays the literal “Enter value” in double width on an ANSI terminal.

```
DISPLAY "Enter value" ESCAPE POUND DOUB-L.
```

8. CURRENCY SIGN clause:

a. The following example applies to any system, and (if on Alpha or I64) regardless of whether /RESERVED_WORDS=200X is specified when the program is compiled:

```
CURRENCY SIGN "G" .
.
. 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,.
```

The following MOVE statements show the effect of the CURRENCY SIGN clause (the character *s* represents a space):

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

b. The following example applies only on Alpha and I64 and only if /RESERVED_WORDS=200X is specified when the program is compiled:

```
CURRENCY SIGN IS "G"
CURRENCY SIGN IS "USD" WITH PICTURE SYMBOL "U"
CURRENCY SIGN IS "DM" WITH PICTURE SYMBOL "D"
CURRENCY SIGN IS "M". .
.
. 01 ITEMA PIC GG,GG9.99.
01 ITEMB PIC U,UUU,UU9.99.
01 ITEMC PIC DD,DD9.99.
01 ITEMD PIC MMM,MM9.99.
```

Statement	Result
MOVE 12.34 TO ITEMA	ITEMA = sssG12.34

Statement	Result
MOVE 1000 TO ITEMB	ITEMB = USD1,000.00
MOVE 12.34 TO ITEMC	ITEMC = ssDM12.34
MOVE 1000 TO ITEMD	ITEMD = sM1,000.00

9. DECIMAL-POINT IS COMMA clause:

```
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,.
```

The following MOVE statements show the effect of the DECIMAL-POINT IS COMMA clause (the character *s* represents a space):

Statement	ITEMD 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

10. CURSOR IS clause (Alpha, I64):

```
SPECIAL-NAMES.
CURSOR IS CURSOR-POSITION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 CURSOR-POSITION.
02 CURSOR-LINE PIC 99.
02 CURSOR-COL PIC 99.
```

In this example, the cursor's position is defined by data items containing a two-digit line number (CURSOR-LINE) and a two-digit column number (CURSOR-COL).

11. CRT STATUS IS clause (Alpha, I64):

```
SPECIAL-NAMES.
SYMBOLIC CHARACTERS
FKEY-10-VAL
ARE 11
CRT STATUS IS CRT-STATUS.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 CRT-STATUS.
03 KEY1 PIC 9.
03 KEY2 PIC X.
88 FKEY-10 VALUE FKEY-10-VAL.
03 FILLER PIC X.
.
.
.
ACCEPT MENU-SCREEN.
IF KEY1 EQUAL "0"
```

```
PERFORM OPTION_CHOSEN
ELSE IF KEY1 EQUAL "1" AND FKEY-10
DISPLAY "You pressed the F10 key; exiting..." LINE 22.
```

The first two characters (KEY1 and KEY2) constitute the code that shows the cause of termination of an ACCEPT operation. (See Table 4.1.) Note that the SPECIAL-NAMES paragraph provides for the capturing of the F10 function key.

INPUT-OUTPUT

The the section called “INPUT-OUTPUT” contains two paragraphs:

- FILE-CONTROL paragraph (see FILE-CONTROL)
- I-O-CONTROL paragraph (see I-O-CONTROL)

The FILE-CONTROL paragraph can contain the following clauses:

- ACCESS MODE clause
- ASSIGN clause
- BLOCK CONTAINS clause
- CODE-SET clause
- LOCK MODE clause (Alpha, I64)
- ORGANIZATION clause
- PADDING CHARACTER clause
- RECORD DELIMITER clause
- RESERVE clause

The I-O-CONTROL paragraph can contain the following clauses:

- APPLY clause
- SAME AREA clause
- RERUN clause
- MULTIPLE FILE clause

This section first describes the FILE-CONTROL paragraph and its clauses, then it describes the I-O-CONTROL paragraph.

Additional References

- OBJECT-COMPUTER Paragraph
- CODE-SET Clause

- Section 6.2.6
- Section 6.5.4
- ACCEPT Statement
- DISPLAY Statement
- SET Statement
- Appendix B

FILE-CONTROL

FILE-CONTROL — The *FILE-CONTROL* paragraph declares the program's data files.

Format

FILE-CONTROL.

General Format

```
[ SELECT [ OPTIONAL ] file-name
      { ASSIGN TO [ EXTERNAL (Alpha, I64) ] file-spec (OpenVMS)
                  [ DYNAMIC (Alpha, I64) ]
      { ASSIGN TO [ EXTERNAL ] { data-name }
                  [ DYNAMIC ]  { literal   }
                               { DISK      }
                               { PRINTER   }
      { ASSIGN TO [ MULTIPLE ] { REEL } [ FILE ] (Tru64 UNIX)
                               { UNIT }
  [ RESERVE reserve-num [ AREA ] ]
                        [ AREAS ]
  [ [ ORGANIZATION IS ] SEQUENTIAL ]
* [ BLOCK CONTAINS [ smallest-block TO ] blocksize { RECORDS }
                                                   { CHARACTERS } ]
* [ CODE-SET IS alpha-name ]
  [ PADDING CHARACTER IS pad-char ]
  [ RECORD DELIMITER IS STANDARD-1 ]
* [ ACCESS MODE IS SEQUENTIAL ]
  [ LOCK MODE IS { AUTOMATIC [ WITH LOCK ON RECORD ] } (Alpha, I64) ]
                 { EXCLUSIVE                         }
* [ FILE STATUS IS file-stat ] . ]
```

Format 1

```
SELECT [ OPTIONAL ] file-name

    { ASSIGN TO [ EXTERNAL (Alpha, I64) ] file-spec (OpenVMS)
                [ DYNAMIC (Alpha, I64) ]
    { ASSIGN TO [ EXTERNAL ] { data-name }
                [ DYNAMIC ]  { literal   }
                             { DISK      }
                             { PRINTER   }

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

  [ ORGANIZATION IS ] LINE SEQUENTIAL

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

* [ CODE-SET IS alpha-name ]

  [ PADDING CHARACTER IS pad-char ]

  [ RECORD DELIMITER IS STANDARD-1 ]

* [ ACCESS MODE IS SEQUENTIAL ]

  [ LOCK MODE IS { AUTOMATIC [ WITH LOCK ON RECORD ] } (Alpha, I64) ]
                 { EXCLUSIVE                         }

* [ FILE STATUS IS file-stat ] . ◆
```

Format 2

```
SELECT [ OPTIONAL ] file-name

    { ASSIGN TO [ EXTERNAL (Alpha, I64) ] file-spec (OpenVMS)
                [ DYNAMIC (Alpha, I64) ]
    { ASSIGN TO [ EXTERNAL ] { data-name }
                [ DYNAMIC ]  { literal   }
                             { DISK      } (Alpha, I64)
                             { PRINTER   }

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

  [ ORGANIZATION IS ] RELATIVE

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

  [ RECORD DELIMITER IS STANDARD-1 ]

* [ ACCESS MODE IS { SEQUENTIAL [ RELATIVE KEY IS rel-key ] }
                   { { RANDOM  } RELATIVE KEY IS rel-key   }
                   { { DYNAMIC }                           }

  [ LOCK MODE IS { MANUAL WITH LOCK ON MULTIPLE RECORDS }
                 { AUTOMATIC [ WITH LOCK ON RECORD ]    } (Alpha, I64) ]
                 { EXCLUSIVE                            }

* [ FILE STATUS IS file-stat ]
```

Format 3

```
SELECT [ OPTIONAL ] file-name

    { ASSIGN TO [ EXTERNAL (Alpha, I64) ] file-spec (OpenVMS)
                [ DYNAMIC (Alpha, I64) ]
    { ASSIGN TO [ EXTERNAL ] { data-name } (Alpha, I64)
                [ DYNAMIC ]  { literal
                             { DISK
                             { PRINTER

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

  [ ORGANIZATION IS ] INDEXED

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

  [ RECORD DELIMITER IS STANDARD-1 ]

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

* [ RECORD KEY IS { rec-key          } [ WITH DUPLICATES ] [ ASCENDING ] ]
                  { seg-key = {seg} . . . }                [ DESCENDING ]

* [ ALTERNATE RECORD KEY IS { alt-key          } [ WITH DUPLICATES ] [ ASCENDING ] ] . . .
                            { seg-key = {seg} . . . }                [ DESCENDING ]

  [ LOCK MODE IS { MANUAL WITH LOCK ON MULTIPLE RECORDS } ] (Alpha, I64)
                 { AUTOMATIC [ WITH LOCK ON RECORD ]    }
                 { EXCLUSIVE                            }

* [ FILE STATUS IS file-stat ] .
```

Format 4

```
SELECT file-name

    { ASSIGN TO [ EXTERNAL (Alpha, I64) ] file-spec (OpenVMS)
                [ DYNAMIC (Alpha, I64) ]
    { ASSIGN TO [ EXTERNAL ] { data-name } (Alpha, I64)
                [ DYNAMIC ]  { literal
                             { DISK
                             { PRINTER
                                                              .
```

Format 5

```
SELECT file-name
    { ASSIGN TO [ EXTERNAL (Alpha, I64) ] file-spec (OpenVMS)
                [ DYNAMIC (Alpha, I64) ]
    { ASSIGN TO [ EXTERNAL ] { data-name } (Alpha, I64)
                [ DYNAMIC ]  { literal   }
                             { DISK      }
                             { PRINTER   }
    [ RESERVE reserve-num [ AREA ] ]
                          [ AREAS ]
    [ [ ORGANIZATION IS ] SEQUENTIAL ]
  * [ BLOCK CONTAINS [ smallest-block TO ] blocksize { RECORDS } ]
                                                     { CHARACTERS }
  * [ CODE-SET IS alpha-name ]
    [ PADDING CHARACTER IS pad-char ]
    [ RECORD DELIMITER IS STANDARD-1 ]
  * [ ACCESS MODE IS SEQUENTIAL ]
  * [ FILE STATUS IS file-stat ] .
```

Note

*Clauses marked with an asterisk (*) can be in either the SELECT clause of the Environment Division or the file description entry of the Data Division. They cannot be in both places for the same file.

`file-name`

is the internal name of a file connector. 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. SELECT is optional in the FILE-CONTROL paragraph.
2. If SELECT is used in the FILE-CONTROL paragraph, it must be the first clause. Other clauses may follow it in any order.
3. Each file described in the Data Division must be specified only once in the FILE-CONTROL paragraph.
4. On OpenVMS for every format, the first form of ASSIGN TO (marked "OpenVMS ONLY") is available only on the OpenVMS Alpha and OpenVMS I64 operating systems and only if the default /STANDARD=NOXOPEN qualifier is in effect.

   The second form of ASSIGN TO is available on the OpenVMS Alpha and OpenVMS I64 systems if the /STANDARD=XOPEN qualifier is in effect.

Format 6—Report Files

5. Each SELECT clause specifying a Report File must have a file description entry containing a REPORT clause in the Data Division of the same program.

General Rules

Formats 1, 2, 3, and 4—Sequential, Line Sequential, Relative, or Indexed Files

1. You must specify an OPTIONAL phrase for files opened in INPUT, I-O, or EXTEND mode that need not be present when the program runs.

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

3. If the file connector referenced by *file-name* is an external file connector, all file control entries in the run unit that reference this file connector must have the following characteristics:

   - The same specification for the OPTIONAL phrase
   - A consistent *full-file-name*
   - The same values for *reserve-num*, *smallest-block*, and *blocksize*
   - The same organization

Format 6—Report Files

4. If the file connector referenced by *file-name* is an external file connector, all file control entries in the run unit that reference this file connector must have the following characteristics:

   - A consistent *full-file-name*
   - The same value for *reserve-num*
   - Sequential organization
   - The same CODE-SET clause

Examples

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

1. Sequential file:

   ```
   SELECT FILE-A
   ASSIGN TO "INFILE".
   ```

   This example refers to a file with sequential organization. The word INFILE is equivalent to the nonnumeric literal “INFILE”. If there is no VALUE OF ID clause, the program accesses a file named INFILE.DAT on OpenVMS Alpha and I64 systems, or a file named INFILE on UNIX systems.

2. Indexed file:

   ```
   SELECT OPTIONAL FILE-A
   ASSIGN TO "INFILE"
   ORGANIZATION INDEXED.
   ```

In this example, the SELECT clause specifies that the indexed file need not be present when the program opens it for INPUT, I-O, or EXTEND.

3. Sort or merge file:

```
SELECT SORT-FILE
ASSIGN TO "SDFILE".
```

4. Report file:

```
SELECT SUMMARY-REPORT
ASSIGN TO "OUTFIL"
FILE STATUS IS REPORT-ERRORS.
```

Additional References

- OPEN statement in Chapter 6
- BLOCK CONTAINS and CODE-SET clauses in this chapter
- FILE STATUS, ACCESS MODE, RECORD KEY, and ALTERNATE RECORD KEY clauses in Chapter 5

ASSIGN

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

General Format

Format 1 (OpenVMS)

ASSIGN TO [EXTERNAL | DYNAMIC] file-spec

Format 2 (Alpha, I64)

ASSIGN TO {data-name | literal | DISK | PRINTER}

Format 3 (UNIX)

ASSIGN TO | MULTIPLE {REEL | UNIT} | FILE

file-spec

on OpenVMS is either a nonnumeric literal or a COBOL word formed according to the rules for user-defined names. It represents a partial or complete file specification. It must conform to the rules for file specifications as defined by RMS.

data-name

is the name of a COBOL data item that contains a partial or complete file specification.

literal

is a nonnumeric literal containing a partial or complete file specification.

DISK (Alpha, I64)

uses the file specification declared in the optional VALUE OF ID clause as the file name. If the VALUE OF ID clause is not present, file-name-1 is used as the file name in the current directory.

PRINTER (Alpha, I64)

creates a print file as if the PRINT-CONTROL phrase of the APPLY clause were specified in the I-O CONTROL paragraph. A print file should contain only printable characters and line and page advancing information written using the ADVANCING clause of the WRITE verb.

REEL or UNIT (UNIX)

creates the file on a magnetic tape using the ANSI standard format as defined by American National Standard X3.27-1978 (Level 3), Magnetic Tape Labels and File Structure for Information Interchange.

Syntax Rules

1. *data-name* cannot be DISK or PRINTER.
2. EXTERNAL and DYNAMIC are allowed for syntax compatibility with other COBOL vendors. They are treated as documentation only.
3. Format 1 is available only on the OpenVMS operating system and only if the default /STANDARD=NOXOPEN qualifier is in effect.

   Format 2 is available on Alpha and I64 if the /STANDARD=XOPEN qualifier is in effect.

   On UNIX, format 2 is the default.

General Rules

1. If there is no VALUE OF ID clause in the file description entry, or that clause contains no file specification, the file specification in the ASSIGN clause is the file specification.
2. If there is a file specification in an associated VALUE OF ID clause, the ASSIGN clause contains the default file specification. File specification components in the VALUE OF ID clause override those in the ASSIGN clause.
3. On OpenVMS, if *file-spec* is not a literal, the compiler:
   - Translates hyphens in the COBOL word to underline characters
   - Treats the word as if it were enclosed in quotation marks
4. *file-spec* may contain a logical name.
5. If you specify ASSIGN TO unquoted string, you need not specify this name in the WORKING-STORAGE section. For example:

   ```
   ASSIGN TO TEST1
   ```

   This assignment would use "TEST1.DAT" on OpenVMS Alpha and I64.

   On UNIX systems, you would specify:

   ```
   ASSIGN TO "TEST1.DAT"
   ```

   or:

```
ASSIGN TO TEST1
...
WORKING-STORAGE SECTION.
01 TEST1 PIC X(9) VALUE IS "TEST1.DAT".
```

6. The file specification derived from one or both of the ASSIGN and VALUE OF ID clauses might refer to an environment variable.

7. On UNIX systems, "" is not a valid file specification.

8. On all platforms, *file-spec* must conform to the rules of the operating system where the run-time I-O occurs.

   For indexed files, *file-spec* must conform to the rules of the ISAM package being used. Some older versions of ISAM on UNIX may have a 10-character maximum for *file-spec* length.

Format 3

For files assigned to magnetic tape using ASSIGN TO REEL clause:

9. If the length of the file name exceeds 17 characters, it is truncated. Any lowercase characters in a file name are uppercased and others outside the ANSI-"a" character set are converted to ’Z’.

   An "a" character is one of the set of the digits 0,1..9, the uppercase letters A,B..Z, and the following special characters:

   SP
   !
   "
   %
   &
   '
   ( )
   *
   +
   ,
   -
   .
   /
   :
   ;
   <
   =
   >
   ?

10. Magnetic tape files must be ORGANIZATION SEQUENTIAL and either fixed or variable length record format.

Technical Notes

- On all platforms, leading and trailing spaces and tabs are removed from file specifications before the OPEN statement executes.

- When a COBOL OPEN statement executes on an OpenVMS system, the RMS facility:
  - Removes spaces and tab characters from the file specification
  - Translates lowercase letters in the file specification to uppercase
  - Performs logical name translation
- .DAT is the default file type if one is not specified on an OpenVMS system.
- On UNIX, the suffixes added to indexed file names on a UNIX system are .idx and .dat.
- On UNIX, file specifications are case sensitive.
- Embedded spaces are allowed in file specifications on UNIX systems. Thus "file name a" and "Monthly Report" are valid file specifications.
- When a COBOL OPEN statement executes on a UNIX system, VSI COBOL for OpenVMS attempts to match the file specification against an environment variable with the same spelling declared in the current login environment. If an exact match is found, the value of the matching environment variable becomes the file specification. Otherwise, the file specification remains unchanged.

Additional Reference

See VALUE OF ID clause in Chapter 5. For information on defining a file connector, refer to the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/].

BLOCK CONTAINS

BLOCK CONTAINS — On OpenVMS systems, the BLOCK CONTAINS clause specifies the size of a physical record. On UNIX systems, block size for INDEXED organization is for documentation purposes only.

General format

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

`smallest-block`

is an integer literal. It specifies the minimum physical record size.

`blocksize`

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

Syntax Rule

The BLOCK CONTAINS clause can be in the file's Data Division file description entry. However, it cannot be in both the SELECT clause and the file description entry for the same file.

General Rules

1. The BLOCK CONTAINS clause specifies physical record size.

2. The compiler ignores *smallest-block*.

3. The RECORDS phrase specifies physical record size in terms of logical records.

   - For a fixed-length record magnetic tape file, each physical record except the last contains *blocksize* records.

   - For a variable-length record magnetic tape file, the compiler computes the physical record size. It equals the size of the largest logical record, plus any overhead bytes, multiplied by *blocksize*.

4. The CHARACTERS phrase specifies physical record size in terms of characters.

   The physical record size is the maximum of: (1) *blocksize* bytes, and (2) the size of the largest logical record; plus any overhead bytes for variable-length records.

5. If there is no BLOCK CONTAINS clause, physical record size assumes a default value.

   The physical record size is the size of the largest record plus any overhead bytes.

6. The size of physical records (in characters) must be a multiple of four. Otherwise, the I/O system rounds up the physical record size to the next multiple of four.

CODE-SET

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

General format

`CODE-SET IS alpha-name`

`alpha-name`

is the name of a character set defined in the SPECIAL-NAMES paragraph. It cannot be described with literals in the ALPHABET clause.

Syntax Rules

1. The CODE-SET clause can be in the file's Data Division file description entry. However, it cannot be in both the SELECT clause and the file description entry for the same file.

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

General Rules

1. The CODE-SET clause identifies *alpha-name* as the character set used to represent the file data externally.

2. *alpha-name* specifies how to convert character codes in the file to and from native character codes.

3. Code conversion occurs during execution of an input or output operation. Conversion occurs as if the data were USAGE DISPLAY.

4. Successful OPEN statement execution establishes the character set for code conversion. The set used is the one specified by *alpha-name* in the FILE-CONTROL paragraph implied by the OPEN statement.

5. If there is no CODE-SET clause, no character conversion occurs during input-output operations. The native character set is the default.

Example

In this example, the CODE-SET clause specifies that the data in INFILE is coded in the EBCDIC character code set as specified by an alphabet named EB. The SPECIAL-NAMES paragraph defines EB as the EBCDIC character set.

```
SPECIAL-NAMES.
  ALPHABET EB IS EBCDIC.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT INFILE ASSIGN TO INFILE
    CODE-SET IS EB.
```

Additional Reference

See the SPECIAL-NAMES paragraph.

LOCK MODE (Alpha, I64)

LOCK MODE (Alpha, I64) — The LOCK MODE clause specifies a locking technique to use for a file. LOCK MODE is part of the X/Open COBOL standard.

General format

```
LOCK MODE IS { MANUAL WITH LOCK ON MULTIPLE RECORDS
             { AUTOMATIC [ WITH { LOCK ON RECORD } ] }
             { EXCLUSIVE        { ROLLBACK       } }
```

Syntax Rules

1. X/Open standard and VSI standard syntax cannot both be specified for the same file connector. Hence, if LOCK MODE is specified, the ALLOWING, APPLY LOCK-HOLDING, and REGARDLESS phrases cannot be specified for that file.

2. The WITH LOCK ON RECORD clause is for documentation purposes only.

3. The LOCK MODE IS MANUAL clause is not available for sequential or line sequential files.

General Rules

1. When you specify LOCK MODE IS AUTOMATIC or LOCK MODE IS MANUAL, an OPEN statement (without the WITH LOCK phrase) opens the file in shareable mode. The LOCK MODE clause can be overridden by the WITH LOCK phrase of the OPEN statement.

2. When you specify LOCK MODE IS EXCLUSIVE, a successful OPEN statement opens the file in exclusive mode. The OPEN statement cannot override LOCK MODE IS EXCLUSIVE.

3. If you omit the LOCK MODE clause, opening the file causes it to become exclusive, unless you open it for INPUT, in which case the file becomes shareable.

4. When you specify LOCK MODE IS AUTOMATIC for a file, a record lock is acquired by the successful execution of the READ statement and released on the successful execution of a subsequent I/O statement.

   If you specify LOCK MODE IS MANUAL, a record lock is acquired by the READ WITH LOCK statement.

5. On UNIX, the ROLLBACK phrase is used by ACMSxp applications to specify a recoverable file. You must compile with the `-tps` option to specify a recoverable file.

6. A file that is opened in exclusive mode cannot be opened by any other access stream.

7. A file that is in shareable mode can be opened by any number of access streams that do not require exclusive use.

8. A file that does not reside on a mass storage device cannot be opened in shareable mode.

ORGANIZATION

ORGANIZATION — The ORGANIZATION clause specifies a file's logical structure. On Alpha and I64 systems, the ORGANIZATION IS LINE SEQUENTIAL clause specifies a variant of sequential files compatible with the system text editor.

General Rules

1. File organization is established when the file is created. It cannot be subsequently changed.

2. If there is no ORGANIZATION clause, the default is SEQUENTIAL.

3. On Alpha and Itanium systems, when LINE SEQUENTIAL organization is specified, the file is treated as consisting of variable length records, with each record containing one line of data. A line is a sequence of characters ending with a record terminator (`n` or x'0A'). The terminator is not counted in the length of the record.

4. On Alpha and Itanium systems, a file with LINE SEQUENTIAL organization should only contain printable characters and the record terminator.

5. On Alpha and Itanium systems, a file with LINE SEQUENTIAL organization may not be opened in I-O mode.

6. All programs that open an existing file must specify the same organization with which the file was created.

Note

On UNIX, a third-party product is required for INDEXED run-time support. Refer to the Release Notes for the latest details on how to obtain the INDEXED run-time support.

PADDING CHARACTER

PADDING CHARACTER — The PADDING CHARACTER clause specifies the character to be used to pad blocks in sequential files.

Format

PADDING CHARACTER IS pad-char

pad-char

is a one-character nonnumeric literal or the data-name of a one-character data item. The data-name can be qualified.

General Rule

The PADDING CHARACTER clause is for documentation only.

RECORD DELIMITER (OpenVMS)

RECORD DELIMITER (OpenVMS) — The RECORD DELIMITER clause indicates the method of determining the length of a variable record on the external medium. It is for documentation only.

General Format

RECORD DELIMITER IS STANDARD-1

General Rule

On OpenVMS, STANDARD-1 is the I/O system (OpenVMS Record Management System [RMS]) default for tape files. It is the method used for determining the length of a variable-length record. This method is specified in the *American National Standard X3.27-1978,* “Magnetic Tape Labels and File Structure for Information Interchange, ” and *International Standard 1001 1979*, “Magnetic Tape Labels and File Structure for Information Interchange.”

Additional Reference

For OpenVMS systems, refer to the *VSI OpenVMS Record Management Services Reference Manual* for more information.

RESERVE

RESERVE — The RECORD DELIMITER clause indicates the method of determining the length of a variable record on the external medium. It is for documentation only.

General Format

RESERVE reserve-num [AREA | AREAS]

reserve-num

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

General Rule

On OpenVMS systems, if there is no RESERVE clause, the number of input-output areas equals the I/O system default.

Technical Note

For OpenVMS systems, two DCL commands change and display the defaults for block count: SET RMS DEFAULT and SHOW RMS DEFAULT. The number of areas is stored in the MBF field of the RAB.

Additional References

Refer to the RMS documentation for field RAB$B_MBF.

I-O-CONTROL

I-O-CONTROL — The *I-O-CONTROL* paragraph specifies the input-output techniques to use for a file. On UNIX systems, a number of the elements in the I-O-CONTROL paragraph are for documentation only. See the Technical Notes for more information.

General Format

```
I-O-CONTROL. [
  [ APPLY
      { DEFERRED-WRITE
        EXTENSION extend-amt
        FILL-SIZE
        LOCK-HOLDING
        MASS-INSERT
        [ CONTIGUOUS
          CONTIGUOUS-BEST-TRY ]
          PREALLOCATION preall-amt
        PRINT-CONTROL
        WINDOW window-ptrs
      } ON { file-name } . . . ] . . .
  [ SAME
      [ RECORD
        SORT
        SORT-MERGE ] AREA FOR {same-area-file} {same-area-file} . . . ] . . .
  [ RERUN
      [ ON file-name ] EVERY
      { { [ END OF ] { REEL
                       UNIT } } OF file-name
        { rec-count RECORDS
        clock-count CLOCK-UNITS
        condition-name } ] . . .
  [ MULTIPLE FILE TAPE CONTAINS
      { multiple-file [ POSITION pos-integer ] } . . . ] . . . . ]
```

`extend-amt`

is an integer from 0 to 65,535. It specifies the number of blocks in each extension of a disk file.

`file-name`

is the internal name of a file connector. 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.

`preall-amt`

is an integer from 0 to 4,294,967,295. It specifies the number of blocks to initially allocate when the program creates a disk file.

`window-ptrs`

is an integer from 0 to 127. Its value can also be 255. It specifies the number of retrieval pointers in the window that maps the disk file.

`same-area-file`

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

`rec-count`

is an integer specifying the number of records to process before writing the rerun information.

`clock-count`

is an integer specifying an interval of time to elapse before writing the rerun information.

`condition-name`

names a switch status which, when set, causes the rerun information to be written. The switch is defined in the SPECIAL-NAMES paragraph of Section 4.1.

`multiple-file`

is a file described in a Data Division file description. It specifies that the file shares storage on a reel/unit device with other files. No more than 255 files can be specified.

`pos-integer`

is an integer from 1 to 255. It specifies the relative location of a file on a tape that contains multiple files.

Syntax Rules

1. The I-O-CONTROL clauses can appear in any order.

2. As the following table shows, each phrase of the APPLY clause can refer only to some file types.

Phrase	File Type
EXTENSION	Disk file
FILL-SIZE	Indexed organization
LOCK-HOLDING	Disk file
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. You can specify the LOCK-HOLDING phrase only if you specify the ALLOWING option of the OPEN statement.

6. The RERUN and MULTIPLE FILE clauses cannot refer to a sort or merge file.

7. In the SAME AREA clause, SORT and SORT-MERGE are equivalent.

8. If *same-area-file* refers to a sort or merge file, you must use the SORT, SORT-MERGE, or RECORD phrase.

9. 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 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.

     same-area-files cannot have the global or the external attribute if the program specifies the SAME RECORD AREA phrase.

10. Files specified in a MULTIPLE FILE TAPE clause must be sequential.

11. A file cannot be specified in more than one MULTIPLE FILE TAPE clause.

General Rules

APPLY Clause

1. An APPLY clause remains active for a *file-name* until the image terminates.

2. If the file connector referenced by *file-name* is an external file connector, all file control entries in the run unit that reference this file must have the same APPLY clause.

3. 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*.

4. The EXTENSION phrase specifies the number of disk blocks to be added each time a file is extended. The I/O system extends a file when it needs more file space to add a record.

   If *extend-amt* equals zero, the I/O system extends the file by its default value.

5. The FILL-SIZE phrase causes the I/O system to use the fill size specified when an indexed file is created to fill the file's buckets. If there is no FILL-SIZE phrase, the I/O system fills buckets completely. The FILL-SIZE phrase applies only to indexed files.

6. The LOCK-HOLDING phrase declares the VSI standard manual record-locking attribute for a sequential, relative, or indexed file in a file-sharing environment on disk.

   Once a record is manually locked (see the READ, REWRITE, START, and WRITE statements in Chapter 6), it remains locked until one of the following occurs:

   - An UNLOCK statement executes.

   - A CLOSE statement executes for the subject file.

- The image terminates.

Usage of the APPLY LOCK-HOLDING option requires additional syntax for the OPEN, READ, REWRITE, START, and WRITE verbs. Table 4.2 summarizes the additional syntax required for Procedure Division I/O statements accessing a file possessing the manual record-locking attribute.

7. X/Open standard and VSI standard syntax cannot both be specified for the same file connector. Hence, APPLY LOCK-HOLDING cannot be specified if LOCK MODE was specified for that file in the SELECT statement.

Table 4.2. Required Manual Record-Locking Phrases (VSI Standard)

	Procedure Division Options Required by the Manual Record-Locking Facility (VSI Standard)	
I/O Operation	ALLOWING …	*REGARDLESS OF LOCK
OPEN	X	N/A
READ	X	X
REWRITE	X	N/A
START	X	X
WRITE	X	N/A
Legend: **X—Required** **N/A—Not Applicable** ***—If the ALLOWING option is not specified**		

8. The MASS-INSERT phrase is for documentation only. It has no effect on program execution.

9. On OpenVMS the PREALLOCATION phrase causes the I/O system to allocate *preall-amt* disk blocks when it creates the file.

   - The CONTIGUOUS phrase specifies that the preallocated disk blocks must be contiguous. If the I/O system cannot find *preall-amt* contiguous disk blocks, the OPEN operation fails.

   - The CONTIGUOUS-BEST-TRY phrase causes the I/O system to try to preallocate disk blocks contiguously. If the I/O system cannot find *preall-amt* contiguous disk blocks, it preallocates disk blocks in the largest possible contiguous areas.

10. The PRINT-CONTROL phrase specifies that the file has print file format. Additionally, the PRINT-CONTROL phrase applies only to sequentially organized files.

    The PRINT-CONTROL phrase is redundant if:

    - The file description entry contains a LINAGE clause

    - The program contains a WRITE statement with the ADVANCING phrase for the file

    - The Report Writer Control System is in effect

11. The WINDOW phrase causes the I/O system to use *window-ptrs* number of retrieval pointers in mapping the files. *window-ptrs* must fall in the range of 0 to 127 inclusive or be equal to 255. If *window-ptrs* is 255, then the I/O system attempts to map the entire file.

SAME AREA Clause

12. The SAME AREA clause is for documentation only.

SAME RECORD AREA Clause

13. 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.

14. 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.

15. 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

16. The SAME SORT (SORT-MERGE) AREA clause is for documentation only.

RERUN Clause

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

MULTIPLE FILE TAPE Clause

18. The MULTIPLE FILE TAPE clause specifies the location of a file or files on a reel/unit device. The location of the file or files can be specified as a relative location by providing a *multiple-file* series. The specific file location can be specified by the POSITION phrase.

19. The MULTIPLE FILE TAPE clause specifies the location of a file or files when more than one file shares the same physical reel of tape. If the files in the *multiple-file* sequence are listed in consecutive order, the POSITION phrase is not required. If the files in the *multiple-file* sequence are not listed in consecutive order, the position of the file or files (relative to the beginning of the tape) must be specified in the POSITION phrase.

20. Only those *multiple-files* referenced by the program need to be specified in a MULTIPLE FILE TAPE clause.

21. If a file is specified with a POSITION phrase of a MULTIPLE FILE TAPE clause, subsequent files listed in that MULTIPLE FILE TAPE clause which are not specified with a POSITION phrase are assumed to be in the next higher position.

22. Only one file listed in a MULTIPLE FILE TAPE clause sequence can be open at any one time.

23. If, at run-time, the run-time system determines that the files referenced are not located on a reel device, the MULTIPLE FILE TAPE clause is ignored.

Technical Notes

- On UNIX systems, many elements of the I-O-CONTROL paragraph are for documentation only. They are accepted and ignored by the compiler. These elements are as follows:

  DEFERRED-WRITE
  EXTENSION
  FILL-SIZE
  CONTIGUOUS
  CONTIGUOUS-BEST-TRY
  PREALLOCATION
  PRINT-CONTROL
  WINDOW

- On OpenVMS, the following notes describe the effects of APPLY clause phrases on parameters in the RMS file access block (FAB) and RMS record access block (RAB) associated with *file-name* on OpenVMS Alpha and I64 systems. The FAB and RAB fields are described in the *VSI OpenVMS Record Management Services Reference Manual*.

  - The DEFERRED-WRITE phrase sets the DFW bit in the FOP field of the FAB.
  - The EXTENSION phrase stores *extend-amt* in the DEQ field of the FAB.
  - The FILL-SIZE phrase sets the LOA bit in the ROP field of the RAB.
  - The LOCK-HOLDING phrase sets the ULK bit in the ROP field of the RAB.
  - 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.
  - The CONTIGUOUS-BEST-TRY phrase sets the CBT bit in the FOP field of the FAB.
  - The PRINT-CONTROL phrase sets bits in two FAB fields:
    - The PRN bit in the RAT field
    - The VFC bit in the RFM field

    The WINDOW phrase stores *window-ptrs* in the RTV field of the FAB.

Additional References

- RESERVE clause
- Technical Notes for the DELETE statement in Chapter 6
- OPEN statement in Chapter 6
- READ statement in Chapter 6
- REWRITE statement in Chapter 6
- START statement in Chapter 6
- UNLOCK statement in Chapter 6

- WRITE statement in Chapter 6

Additionally, refer to the *VSI COBOL User Manual [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/]* for more information.

Chapter 5. Data Division

This chapter describes the logical and physical concepts that apply to the Data Division. In addition, this chapter presents the general formats for all Data Division entries and clauses, describes their basic elements, and lists rules of use.

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, a database, Oracle CDD/Repository, or is developed only for local use in your program.

The File and Report Sections of your program define data contained in files. A file description, sort-merge file description, or report file description entry creates a logical structure, or file connector, that refers to the physical 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. A **record description entry** is a set of one or more data description entries, organized in a hierarchical structure which logically defines a set of related data within the file. The **data description entries** 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) within the record description 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.

A report description entry must be associated with a report group description, which specifies both the logical hierarchy of data in the report and the data's physical attributes.

A **screen description entry** describes a video form or a portion of a video form.

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. Additionally, the following sections describe how record descriptions impose logical structures on data, and how the physical attributes of data affect the way data is stored and manipulated.

5.1. Logical Concepts of Data Storage

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

When you refer to a data-name in a Procedure Division 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 the I/O system. On OpenVMS systems, the I/O system is called OpenVMS Record Management Services (RMS). A logical record may correspond to one physical record, either alone or grouped with other logical records. Or, on disk, a logical record may 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 the I/O system requires.

During program execution, data transfer between the program and a physical record can involve translation if the SELECT clause contains a CODE-SET clause.

5.1.1. Record Description Entries

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 (except for COMP-1 or COMP-2). 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 5.1 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 Date, and Termination Date. The third group item, Payroll Data, also directly contains two elementary items: Current Salary and Previous Salary.

Figure 5.1. Hierarchical Record Structure

```
Personnel Record                (record level group item)
    Identification Data             (group item)
        Employee Number                 (group item)
            Department Code                 (elementary item)
            Badge Number                    (elementary item)
        Name                            (elementary item)
        Address                         (group item)
            Street                          (elementary item)
            City                            (elementary item)
            State                           (elementary item)
            ZIP Code                        (elementary item)
        Job Title                       (elementary item)
    History                         (group item)
        Hire Date                       (elementary item)
        Last Promotion Date             (elementary item)
        Termination Date                (elementary item)
    Payroll Data                    (group item)
        Current Salary                  (elementary item)
        Previous Salary                 (elementary item)
                                                VM-0583A-AI
```

5.1.2. 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 to 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 level-number that is less than or equal to that group's level number.

Figure 5.2 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 5.2. 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)
                                                   VM-0584A-AI
```

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 the RENAMES clause for more information.
- 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 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 Chapter 1: *Overview of the COBOL Language*, for more information on condition-names.

5.1.3. Multiple Record Description Entries for the Same Data

Example 5.1 shows a sample file description entry (FD) that contains three record description entries. The three record description entries define three logical templates the program can impose on a record to access data from it.

Example 5.1. 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.
```

The three record description entries in Example 5.1, T1, T2, and RECORD-TYPE, each define a fixed-length record of 30 characters. Once the program reads a record, it can use the last two characters (REC-TYPE) to determine which record description entry to use.

5.2. Physical Concepts of Data Storage

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

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

The size of a physical record and the way it is recorded 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 multiple units of a fixed number of bytes, which is determined by the hardware and operating system involved.

On OpenVMS 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 the I/O system software 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 on the following:

- The block size specified by the BLOCK CONTAINS clause (tape files only)
- The number of logical records contained in a physical record

The maximum physical record size depends on file organization and device. On OpenVMS systems, the maximum physical record sizes for sequential files on tape devices and for sequential, indexed, and relative files on disk are shown in terms of number of bytes in Table 5.1.

Table 5.1. Maximum Physical Record Size for Tape and Disk Devices

Type of File	Magnetic Tape Devices	Disk
Sequential	65,535 bytes	65,024 bytes
Indexed	N/A	32,234 bytes
Relative	N/A	32,255 bytes

Note

A compile-time informational diagnostic appears if the physical record size exceeds 65,024 bytes for a sequential file. However, VSI COBOL for OpenVMS programs are device-independent. Therefore, a fatal run-time error can also occur if the file is assigned to disk when the program runs.

5.2.1. Categories and Classes of Data

The size and storage format of an elementary data item depend 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 the PICTURE and USAGE clauses for more information.

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 (see Section 5.2.2) specify the relationship between category and positioning.

Depending on the symbols contained in its PICTURE clause, every elementary item belongs to one of the classes and categories of data items shown in Table 5.2. COMP-1, COMP-2, index data items, and index-names do not have PICTURE clauses; the format of these elementary items is specified by the compiler and they belong to the numeric category.

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

Table 5.2. Classes and Categories of Data Items

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

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

5.2.2. COBOL Standard Alignment Rules

The COBOL Standard Alignment Rules specify how characters are positioned in an elementary data 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 then aligned by decimal point as described in the preceding list item.
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 receiving item, the rules for the JUSTIFIED clause override rule 3. See the JUSTIFIED clause for more information.

5.2.3. Additional Alignment Rules for Record Allocation

As stated in Section 5.2.2, the COBOL Standard Alignment Rules specify data positioning only within elementary data items. VSI defines additional alignment rules that affect the positioning of:

- Records on the file media
- Group items within a record
- Elementary items within a group item

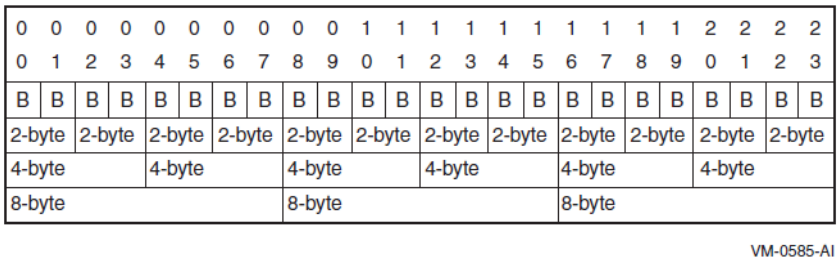
On Alpha and I64 systems, VSI COBOL for OpenVMS offers the option of allocating subordinate record items along performance-optimal boundaries through the use of the `alignment` compiler option or directives (or the SYNCHRONIZED clause). If you select one of these options, subordinate data items will be aligned automatically along optimal boundaries for their data type. The compiler may have to skip one or more bytes before assigning a location to the next data item. These skipped bytes, called **fill bytes**, are spaces between one data item and the next. Refer to the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/] for information on using `alignment` compiler options and directives.

If you do not select one of these Alpha- and I64-only alignment options, the VSI COBOL for OpenVMS compiler will locate the data item at the next unassigned byte location.

The presence of fill bytes can make a record's structure different from what you might expect. In particular, if a record contains many items requiring alignment, its size can increase significantly. If, unaware of the fill bytes, you tried to move a group item containing fill bytes to a single data item, right-end truncation would occur. You would not have this problem, however, if you moved the record into another identically defined group item. The method the compiler uses to allocate storage ensures that identically described group items have the same structure, even when their subordinate items are aligned on their required boundaries.

Figure 5.3 shows alignment boundaries for a record. The boundary is the leftmost location of the 1-, 2-, 4-, or 8-byte area. All boundaries are relative to the beginning of the record as byte number 0.

Figure 5.3. Record Alignment Boundaries



The VSI COBOL for OpenVMS compiler allocates storage for data items within records according to the rules of the **major-minor equivalence technique**. The major-minor equivalence technique ensures that identically defined group items have the same structure, even when their subordinate items are aligned. Therefore, group moves always produce predictable results. This technique is based on the following two rules:

- Location Equivalence—The leftmost location of a group item is the same as the leftmost location of its first subordinate item.
- Boundary Equivalence—The VSI COBOL for OpenVMS compiler aligns a group item on a boundary that is as large as the largest boundary for *any* aligned data item within its scope.

Location Equivalence

Location equivalence forces a group (major) item to the same storage location as its first subordinate (minor) item. This forced positioning occurs regardless of the boundary alignment of either the group or subordinate item.

Refer to the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/] chapter on aligning binary data for information on how location equivalence allocates storage.

The following example results in the major-minor location format:

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

```
    03  ITEM-F PIC X.
```

The following example (omitting SYNCHRONIZED) results in the left-right location format:

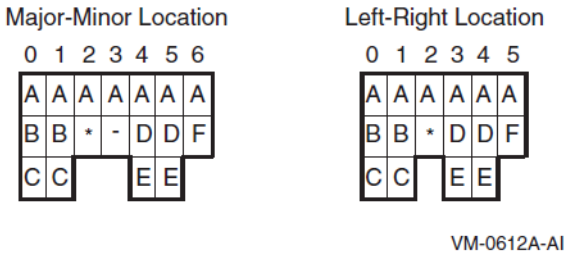
```
01  ITEM-A.
    03  ITEM-B.
        05  ITEM-C PIC 9(4) COMP.
    03  FILLER PIC X.
    03  ITEM-D.
        05  ITEM-E PIC 9(4) COMP.
    03  ITEM-F PIC X.
```

Table 5.3 compares the major-minor technique of storage allocation with the left-to-right technique that assigns locations to a group item before its subsidiary items. Note that major-minor storage allocation adds a fill byte before ITEM-D. This forces location equivalence with ITEM-E, which is explicitly aligned by the SYNCHRONIZED clause.

Table 5.3. Comparison of Major-Minor and Left-Right Locations

Data Item	Major-Minor Location	Left-Right Location
ITEM-A	00	00
ITEM-B	00	00
ITEM-C	00	00
FILLER	02	02
ITEM-D	04	03
ITEM-E	04	03
ITEM-F	06	05

The following diagram also shows the storage allocation for the record ITEM-A in Table 5.3 using both techniques. A hyphen (-) represents fill bytes caused by explicit alignment; an asterisk (*) represents the FILLER data item.



Regardless of the record allocation technique, an elementary move always produces the expected result. For example:

```
MOVE ITEM-C TO ITEM-E
```

Effect on Group Moves

A group move may produce an unexpected result, as in the following two situations:

- If ITEM-A of the major-minor location format is moved to ITEM-A of the left-right location format, the fill byte of the major-minor location format overlays the first byte of ITEM-E in the left-right location format; then the first byte of ITEM-E in the major-minor location format overlays the second byte of ITEM-E in the left-right location format, and the second byte of ITEM-E in the

major-minor location format overlays ITEM-F in the left-right location format. Finally, ITEM-F in the major-minor location format is truncated.

- A different set of unexpected results occurs if a group move is done in the reverse direction. If ITEM-A of the left-right location format is moved to ITEM-A of the major-minor location format, the first byte of ITEM-D of the left-right location format is moved to the fill byte of the major-minor location format. Then the second byte of ITEM-E in the left-right location format is moved to the first byte of ITEM-E in the major-minor location format, and ITEM-F of the left-right location format is moved to the second byte of ITEM-E in the major-minor location format. Finally, ITEM-F is filled with a space because of the padding rule.

Boundary Equivalence

Boundary equivalence forces a group item to a boundary determined by the alignment of its subordinate items.

Within a record, a group item aligns on a boundary as large as the forced alignment boundary of any data item that:

- Is subordinate to the group
- Redefines the group
- Is subordinate to a data item that redefines the group

Refer to the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/] chapter on alignment for more information about boundary equivalence.

Figure 5.4 shows how the compiler determines the boundary where each item begins when you specify the no-alignment compiler option.

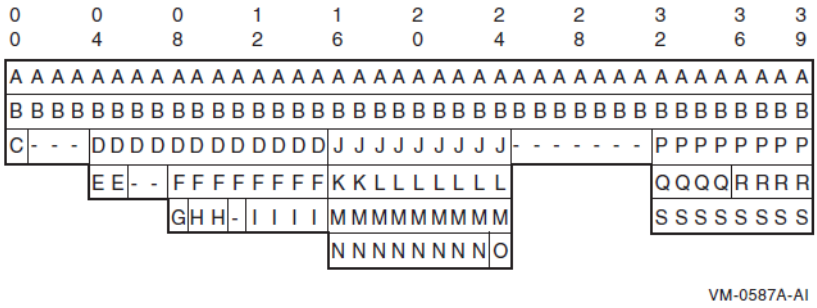
Figure 5.4. Effect of Boundary and Location Equivalence Rules on Sample Record

```
                                                  Boundary   Reason
01  ITEM-A.                                          -       01-level item
    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) COMP.         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.         byte      Default alignment
          07 ITEM-R         PIC 9(7) COMP.         byte      Default alignment
       05 ITEM-S REDEFINES ITEM-P                  8-byte    Explicit SYNC clause
             PIC 9(15) COMP SYNC.
```

VM-0586A-AI

Figure 5.5 graphically represents Figure 5.4. It shows the result of location and boundary equivalence applied to the description of record ITEM-A. A hyphen (-) indicates fill bytes.

Figure 5.5. Storage Allocation for Sample Record



Note the location of ITEM-D. Location equivalence requires only that it have the same location as ITEM-E, its first subordinate item. ITEM-E requires only 2-byte boundary alignment. However, another of ITEM-D's subordinate items, ITEM-F, contains ITEM-I, which must be aligned on a 4-byte boundary. Therefore, boundary equivalence forces ITEM-D to a 4-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:

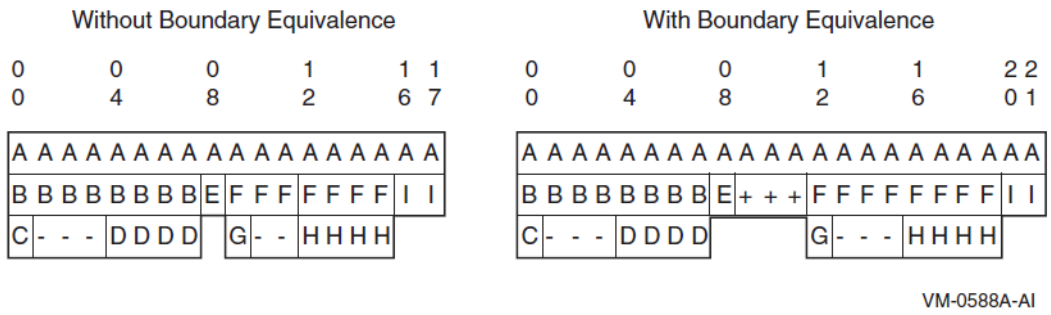
```
01    ITEM-A.
   03    ITEM-B.
       05    ITEM-C    PIC X.
       05    ITEM-D    PIC 9(8) COMP SYNC.
   03    ITEM-E        PIC X.
   03    ITEM-F.
       05    ITEM-G    PIC X.
       05    ITEM-H    PIC 9(8) COMP SYNC.
   03    ITEM-I        PIC XX.
```

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 5.6 shows how storage for the record would be allocated without and with boundary equivalence. A hyphen (-) indicates fill bytes caused by the SYNCHRONIZED clause. A plus sign (+) represents fill bytes resulting from boundary equivalence.

Figure 5.6. Storage Allocation Without and With Boundary Equivalence



Without boundary equivalence, ITEM-B occupies 8 bytes, and ITEM-F occupies 7 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.

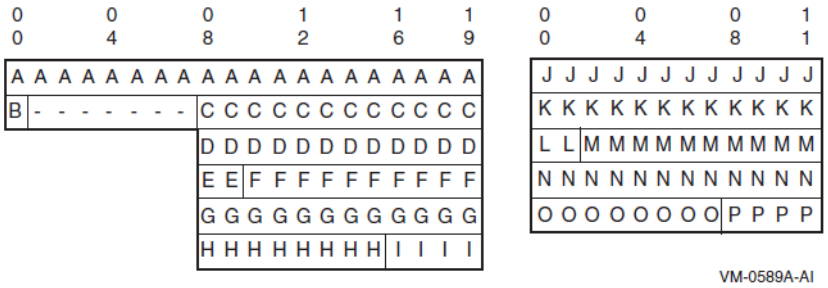
In contrast, boundary equivalence eliminates this unforeseen result. The elementary items occupy the same relative positions in each group. Therefore, the structures of ITEM-B and ITEM-F are the same, and the results of both group and elementary moves are predictable.

Examples

The following series of examples show major-minor storage allocation. The notes after each example indicate its significant features. A hyphen (-) 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.
```



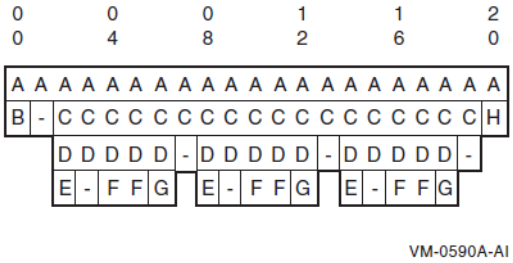
In this example:

- The relative locations of records (01-level items) in the Working-Storage and Linkage Sections are neither defined nor predictable.
- The structures of ITEM-J (a record) and ITEM-C (a group item within a record) are identical.

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.
```

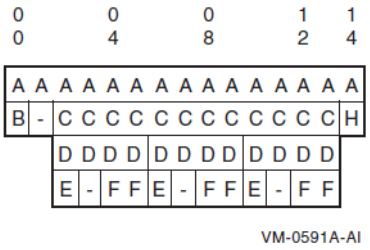


In this example:

- A fill byte is added after each occurrence of ITEM-D to maintain 2-byte boundary alignment of the next occurrence.
- ITEM-D is 5 bytes long. The fill byte following ITEM-D is not included in its length.
- ITEM-C is 18 bytes long. Its length includes the fill bytes associated with its subordinate items.
- The record ITEM-A is 21 bytes long.

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.
```



In this example:

- ITEM-G is omitted.
- ITEM-D is 4 bytes long. No fill bytes are added, since the next occurrence is already aligned on a 2-byte boundary.
- ITEM-C is 12 bytes long.

- The record ITEM-A is 15 bytes long.

5.2.4. Alpha and I64 Alignment and Padding

The VSI OpenVMS Calling Standard for the UNIX, OpenVMS Alpha, and OpenVMS I64 systems specify Alpha natural data alignment and padding. You invoke this alignment by adding the `alignment padding` compiler option to the compile command line, or by using `pad align` directives in your source code. (Refer to the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/] for additional information on the command.)

The Alpha natural alignments and field sizes that apply to elementary COBOL data fields are shown in Table 5.4.

Table 5.4. Alpha Alignment and Padding

Data Types	Alignment Starting Position	Pictures	Usages
8-bit character string	Byte boundary	A, X, 9, Edited	Display
16-bit integer	Word boundary, multiple of 2	9(4)	COMP
32-bit integer	Longword boundary, multiple of 4	9(8)	COMP
Single-precision real			COMP-1
64-bit integer	Quadword boundary, multiple of 8	9(16)	COMP
Double-precision real			COMP-2

These alignments and field sizes apply to elementary data items. However, they are extended to group data items at all level numbers by requiring that the alignment of the group data item conforms to the alignment of the largest natural alignment of any elementary data item that is subordinate to the group-item. Every intermediate group data item in VSI COBOL for OpenVMS is a candidate for Alpha natural alignment and padding. Every higher-level data item is padded to be the smallest multiple of the largest natural alignment of any of its subordinate elementary data items that contains the data structure. The alignment and padding can be determined in all cases by following the tree structure through as many levels as required until the elementary data item with the largest natural alignment is found. All elementary data items are aligned and sized within their data structures according to Table 5.4.

5.3. 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 } ... ] ...
  [ report-file-description-entry ] ...
  [ sort-merge-file-description-entry { record-description-entry } ... ] ... ]

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

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

[ REPORT SECTION. [ report-description-entry { report-group-description-entry } ... ] ]

[ SCREEN SECTION. [ screen-description-entry ] ... ] (Alpha, I64)
```

Syntax Rules

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

General Rules

1. The Data Division is subdivided into sections. These sections must be in the following order:

   FILE SECTION.
   WORKING-STORAGE SECTION.
   LINKAGE SECTION.
   REPORT SECTION.
   SCREEN SECTION. (Alpha, I64)

File Section

2. The File Section defines the structure of data files. It begins with the File Section header containing the reserved words FILE SECTION, followed by a period (.) separator character.
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.
5. The FD clause specifies the following:
   - How the file records data

- The sizes of logical and physical records
- The names of data records (except for report files)
- The number of lines on a logical printer page

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. No record description entries may follow the report file description entry.
8. The sort-merge file description entry consists of a level indicator (SD), a file-name, and a series of independent clauses.
9. An SD clause specifies the following:
   - How the file records data
   - The sizes of logical and physical records
   - The names of data records
10. An SD entry specifies the sizes and names of data records for a sort-merge file referred to by SORT and MERGE statements.
11. 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

12. 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.
13. The Working-Storage Section also describes data items assigned values by the source program.
14. The Working-Storage Section consists of a section header, followed by record description entries.
15. The section header consists of the reserved words WORKING-STORAGE SECTION, followed by a period (.) separator character.
16. 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.
17. Record description clauses can be used in the File Section, the Working-Storage Section, or the Linkage Section.
18. 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).
19. If the VALUE IS clause does not specify an initial value, the default initial value for an item depends on the type of data item:

Data Item Type	**Default Initial Value**
Numeric	Zero

Data Item Type	Default Initial Value
Index-name	Occurrence number one
Index data item	Undefined
Tables	Undefined
All others	Spaces

Linkage Section

20. The Linkage Section is used only in a called program.
21. The Linkage Section describes data available through the calling program; both the calling and called programs can access this data.
22. To access calling program data items through the Linkage Section, the called program must have a Procedure Division header USING phrase.
23. 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 period (.) separator character.

Report Section

24. The Report Section defines report files. It begins with the Report Section header: the reserved words REPORT SECTION, followed by a period (.) separator character.
25. Report description entries follow the Report Section header.
26. The report description entry consists of a level indicator (RD), a report name, and a series of independent clauses.
27. An RD clause specifies the following:
    - Identifying characters to be prefixed to each print line in a report
    - The physical structure and organization of a report
    - The name of the report
28. An RD entry, followed by one or more report group description entries, defines a report. Report group description entries must immediately follow the associated report description entry.
29. A report group description entry defines a report group. It specifies the characteristics of a report group and the individual items within a report group.

Screen Section (Alpha, I64)

30. The Screen Section describes a video form and specifies the attributes, behavior, size, and location of each screen item within the video form. The Screen Section is for use with ACCEPT and DISPLAY statements.

Additional References

- VALUE IS clause
- CALL statement in Chapter 6

- User-defined words in Section 1.2.1: COBOL Words
- REPORT clause
- Refer to the chapter describing alignment in the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/].

FD (File Description)

FD (File Description)—Sequential, Line Sequential (Alpha, I64), Relative, Indexed, and Report File Descriptions — A file description entry describes the physical structure, identification, record names, and names for sequential, line sequential (Alpha, I64), relative, indexed, and report files. It also specifies the internal or external attributes of a file connector and the local or global attributes of a file-name.

General Format

Format 1—Sequential or Line Sequential (Alpha, I64) Files

```
FD file-name

    [ IS EXTERNAL ]
    [ IS GLOBAL ]
*   [ 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 / RECORD IS } { STANDARD / OMITTED } ]
    [ VALUE OF ID IS { quoted-literal-string / data-name } ]
    [ DATA { RECORDS ARE / RECORD IS } { rec-name } . . . ]
    [ LINAGE IS page-size LINES
          [ WITH FOOTING AT footing-line ]
          [ LINES AT TOP top-lines ]
          [ LINES AT BOTTOM bottom-lines ] ]
*   [ CODE-SET IS alpha-name ]
*   [ [ ACCESS MODE IS ] SEQUENTIAL ]
*   [ FILE STATUS IS file-stat ].
```

Format 2—Relative Files

```
FD file-name

  [ IS EXTERNAL ]
  [ IS GLOBAL ]

*  [ 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 / RECORD IS } { STANDARD / OMITTED } ]

   [ VALUE OF ID IS { quoted-literal-string / data-name } ]

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

*  [ [ ACCESS MODE IS ] { SEQUENTIAL [ RELATIVE KEY IS rel-key ]
                          { RANDOM / DYNAMIC } RELATIVE KEY IS rel-key } ]

*  [ FILE STATUS IS file-stat ].
```

Format 3—Indexed Files

```
FD file-name

    [ IS EXTERNAL ]
    [ IS GLOBAL ]
*   [ 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 | RECORD IS } { STANDARD | OMITTED } ]
    [ VALUE OF ID IS { quoted-literal-string | data-name } ]
    [ DATA { RECORDS ARE | RECORD IS } { rec-name } . . . ]
*   [ [ ACCESS MODE IS ] { SEQUENTIAL | RANDOM | DYNAMIC } ]
*   [ RECORD KEY IS { rec-key | seg-key = {seg} . . . } [ WITH DUPLICATES ] [ ASCENDING | DESCENDING ] ]
*   [ ALTERNATE RECORD KEY IS { alt-key | seg-key = {seg} . . . } [ WITH DUPLICATES ] [ ASCENDING | DESCENDING ] ] . . .
*   [ FILE STATUS IS file-stat ].
```

Format 4—Report Files

```
FD file-name

    [ IS EXTERNAL ]
    [ IS GLOBAL ]
*   [ 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 / RECORD IS } { STANDARD / OMITTED } ]
    [ VALUE OF ID IS { quoted-literal-string / data-name } ]
*   [ [ ACCESS MODE IS ] SEQUENTIAL ]
    { REPORT IS / REPORTS ARE } { report-name } . . .
*   [ CODE-SET IS alpha-name ]
*   [ FILE STATUS IS file-stat ] .
```

*Clauses marked with an asterisk (*) can be in either the SELECT clause of the Environment Division or the file description entry of the Data Division. They cannot be in both places for the same file.

Syntax Rules

Formats 1, 2, 3, and 4—All Files

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 period (.) separator character must terminate a file description entry.
4. On OpenVMS, the file name written to disk (see the ASSIGN clause in Chapter 4) has the file type .DAT added if no other file type is specified in the corresponding file specification.

Format 1—Sequential or Line Sequential (Alpha, I64) Files

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

Format 2—Relative Files

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

Format 3—Indexed Files

10. *file-name* can refer only to an indexed file.
11. On UNIX, for information on *file-names* for indexed files, see the ASSIGN clause in Chapter 4.
12. *alt-key* cannot have the same leftmost character position as that of *rec-key* or any other *alt-key* for the same file.
13. One or more record description entries must follow the file description entry.

Format 4—Report Files

14. *file-name* can refer only to a report file.
15. No record description entries may follow the file description entry for a report file.
16. Only the CLOSE statement and the OPEN statement with the OUTPUT or EXTEND phrase may reference this file description entry.

General Rules

Formats 1, 2, 3, and 4—All Files

1. A file description entry associates *file-name* with a file connector.
2. On OpenVMS, if the file description entry contains the EXTERNAL clause, the RMS special registers RMS-STS, RMS-STV, and RMS-FILENAME are external registers.
3. If the file description entry contains the GLOBAL clause, the RMS special registers RMS-STS, RMS-STV, and RMS-FILENAME are global registers.

Format 1—Sequential and Line Sequential (Alpha, I64) Files

1. If the file description entry contains the LINAGE clause and the EXTERNAL clause, the LINAGE-COUNTER special register is an external data item.
2. If the file description entry contains the LINAGE clause and the GLOBAL clause, the special register LINAGE-COUNTER is a global name.

Format 3—Indexed Files

1. If the file description entry contains the EXTERNAL clause, the segmented key *seg-key* has the external attribute.

2. If the file description entry contains the GLOBAL clause, the segmented key *seg-key* is a global name.

Examples

Refer to the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/] for examples of the file description entry formats.

SD (Sort-Merge File Description)

SD (Sort-Merge File Description) — 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 period (.) separator character 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

Refer to the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/] for examples of the sort-merge file description entry.

RD (Report Description)

RD (Report Description) — The Report Description names a report, specifies any identifying characters to be prefixed to each print line in the report, and describes the physical structure and organization of that report. It also determines whether a *report-name* is a local name or global name.

General Format

```
RD report-name

[ IS GLOBAL ]

[ CODE report-code ]

[ { CONTROL IS   } { { control-name } . . .          }
  { CONTROLS ARE } {   FINAL [ control-name ] . . .  } ]

[ PAGE [ LIMIT IS   ] page-size [ LINE  ]
       [ LIMITS ARE ]           [ LINES ]

       [ HEADING heading-line ]

       [ FIRST DETAIL first-detail-line ]

       [ LAST DETAIL last-detail-line ]

       [ FOOTING footing-line ] ]
.
```

Syntax Rules

1. *report-name* must appear in one and only one REPORT clause.
2. The clauses following *report-name* may appear in any order. *report-name* is the highest permissible qualifier that can be specified for LINE-COUNTER, PAGE-COUNTER, and all data-names in the Report Section.

General Rules

1. If the Report Description entry contains the GLOBAL clause, *report-name* and the special registers LINE-COUNTER and PAGE-COUNTER are global names.
2. The reserved word PAGE-COUNTER references a special register that the compiler creates for each report in the Report Section.
3. In the Report Section, a reference to PAGE-COUNTER can appear only in a SOURCE clause. In the Procedure Division, PAGE-COUNTER can be used anywhere an integer data item can appear.
4. If more than one PAGE-COUNTER exists in a program, PAGE-COUNTER must be qualified by a *report-name* wherever it is referenced in the Procedure Division.

   In the Report Section, an unqualified reference to PAGE-COUNTER is qualified implicitly by the name of the report containing the reference. Whenever the PAGE-COUNTER of a different report is referenced, PAGE-COUNTER must be explicitly qualified by the other report's *report-name*.
5. The INITIATE statement causes the Report Writer Control System to set the PAGE-COUNTER of the referenced report to one.
6. PAGE-COUNTER is automatically incremented by one each time the Report Writer Control System executes a page advance.
7. Procedure Division statements may alter the contents of PAGE-COUNTER.
8. The reserved word LINE-COUNTER references a special register that the compiler creates for each report in the Report Section.

9. In the Report Section, a reference to LINE-COUNTER can appear only in a SOURCE clause. In the Procedure Division, LINE-COUNTER can be used anywhere a data item with an integer value can appear. However, only the Report Writer Control System can change the contents of LINE-COUNTER.

10. If there is more than one LINE-COUNTER in a program, Procedure Division references to LINE-COUNTER must be qualified by a *report-name*.

    In the Report Section, an unqualified reference to LINE-COUNTER is qualified implicitly by the name of the report containing the reference. Whenever the LINE-COUNTER of a different report is referenced, LINE-COUNTER must be explicitly qualified by the other report's *report-name*.

11. The INITIATE statement causes the Report Writer Control System to set the LINE-COUNTER of the referenced report to zero. The Report Writer Control System also automatically resets LINE-COUNTER to zero each time it executes a page advance.

12. The execution of SUPPRESS statements and the processing of nonprintable report groups do not change the value of LINE-COUNTER.

13. At the time each print line is presented, the value of LINE-COUNTER represents the line number on which the print line is presented. After the presentation of the report group, the value of LINE-COUNTER is governed by the Report Writer Presentation Rules and Tables.

Example

The following is an example of a global Report Description entry:

```
FILE SECTION.
FD   WEEKLY-REPORTS...
     REPORTS ARE PAYROLL-REPORT
                 PAYROLL-IRS.
REPORT SECTION.

RD   PAYROLL-REPORT
     IS GLOBAL
     CODE "AA"
     CONTROL GRAND-TOT
             SITE-TOT
             DEPT-TOT
             GROUP-TOT
     PAGE LIMITS ARE 60 LINES
             HEADING       2
             FIRST DETAIL  9
             LAST DETAIL  55
             FOOTING      58.

RD  PAYROLL-IRS
    CODE "BB"...
```

The previous example uses the CODE clause to flag PAYROLL-REPORT records from other records (see PAYROLL-IRS) included in the same file (WEEKLY-REPORTS). The entry defines four control totals. GRAND-TOT is the most major control total; it will be printed only at the end of the report. SITE-TOT, DEPT-TOT, and GROUP-TOT are major, intermediate, and minor control totals, respectively. These totals are printed whenever the Report Writer Control System (RWCS) processes a control break. The entry also defines a report page with 60 lines. On each page the RWCS is to

print PAYROLL-REPORT headings beginning on line 2, detail lines from lines 9 to 55, and footings beginning on line 58.

Additional References

- LINE NUMBER (Alpha, I64) clause
- REPORT clause
- FD (File Description)
- Appendix D: *Report Writer Presentation Rules and Tables*

Data Description

Data Description — A data description entry specifies the characteristics of a data item.

General Formats

```
level-number [ data-name ]
             [ FILLER   ]
[ REDEFINES other-data-item ]
[ IS EXTERNAL ]
[ IS GLOBAL ]
[ { PICTURE } IS character-string ]
  { PIC     }

[ [USAGE IS] { BINARY                                          }
             { BINARY-CHAR (Alpha, I64) { SIGNED   }           }
             {                          { UNSIGNED }           }
             { BINARY-SHORT (Alpha, I64) { SIGNED   }          }
             {                           { UNSIGNED }          }
             { BINARY-LONG (Alpha, I64) { SIGNED   }           }
             {                          { UNSIGNED }           }
             { BINARY-DOUBLE (Alpha, I64) { SIGNED   }         }
             {                            { UNSIGNED }         }
             { COMPUTATIONAL                                   }
             { COMP                                            }
             { COMPUTATIONAL-1                                 }
             { COMP-1                                          }
             { COMPUTATIONAL-2                                 }
             { COMP-2                                          }
             { COMPUTATIONAL-3                                 }
             { COMP-3                                          }
             { COMPUTATIONAL-5 (Alpha, I64)                    }
             { COMP-5 (Alpha, I64)                             }
             { COMPUTATIONAL-X (Alpha, I64)                    }
             { COMP-X (Alpha, I64)                             }
             { DISPLAY                                         }
             { FLOAT-SHORT (Alpha, I64)                        }
             { FLOAT-LONG (Alpha, I64)                         }
             { FLOAT-EXTENDED (Alpha, I64)                     }
             { INDEX                                           }
             { PACKED-DECIMAL                                  }
             { POINTER                                         }
             { POINTER-64 (Alpha, I64)                         } ]
```

```
[ [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 } ... ] ]

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

[ { JUSTIFIED } RIGHT ]
  { JUST      }

[ BLANK WHEN ZERO ]

[ VALUE IS { lit                  } ]
           { EXTERNAL external-name }
           { REFERENCE data-name    }
.
```

Format 2

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

Format 3

88 condition-name { VALUE IS | VALUES ARE }

```
{ { EXTERNAL external-name }
  { REFERENCE data-name    }
  { low-val                }
      [ { THRU    } { EXTERNAL external-name } ]
      [ { THROUGH } { REFERENCE data-name    } ]
      [             { high-val               } ]
} ....
```

Syntax Rules

1. *level-number* in Format 1 can be any number from 01 to 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. The EXTERNAL clause can appear in a level 01 or 77 data description entry in the Working-Storage Section.

4. The GLOBAL clause can appear in a level 01 or 77 data description entry in the Working-Storage Section, or in a level 01 data description entry in the File Section.

5. The EXTERNAL and REDEFINES clauses cannot be in the same data description entry.

6. *data-name* must appear in any Format 1 entry that contains the EXTERNAL clause or GLOBAL clause, or in the record descriptions of a file description entry that contains the EXTERNAL or GLOBAL clause.

7. There must be a PICTURE clause for all elementary items except the following:

   - An index data item
   - A COMP-1 or COMP-2 data item
   - The subject of a RENAMES clause
   - A POINTER data item

   In these cases, there must be no PICTURE clause.

8. The words THRU and THROUGH are equivalent.

9. The SYNCHRONIZED, PICTURE, JUSTIFIED, and BLANK WHEN ZERO clauses can appear only in Data Description entries for elementary items.

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.

   Any *condition-name* associated with a global conditional variable is global.

   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.

Report Group Description

Report Group Description — The report group description entry specifies the characteristics of a report group and of the individual items within a report group.

General Formats

Format 1

```
01 [ group-data-name ]

[ LINE NUMBER IS { line-num [ ON NEXT PAGE ] }
                 { PLUS line-num-plus       }  ]

[ NEXT GROUP IS { next-group-line-num          }
                { PLUS next-group-line-num-plus }
                { NEXT PAGE                    }  ]

                { { REPORT HEADING } }
                { { RH             } }
                { { PAGE HEADING   } }
                { { PH             } }
                { { CONTROL HEADING } { control-head-name } }
                { { CH             } { FINAL             } }
TYPE IS         { { DETAIL         } }
                { { DE             } }
                { { CONTROL FOOTING } { control-foot-name } }
                { { CF             } { FINAL             } }
                { { PAGE FOOTING   } }
                { { PF             } }
                { { REPORT FOOTING } }
                { { RF             } }

[ [ USAGE IS ] DISPLAY ] .
```

Format 2

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

[ LINE NUMBER IS { line-num [ ON NEXT PAGE ] }
                 { PLUS line-num-plus        }  ]

[ [ USAGE IS ] DISPLAY ] .
```

Format 3

```
level number [ group-data-name ]
[ BLANK WHEN ZERO ]
[ COLUMN NUMBER IS column-num ]
[ GROUP INDICATE ]
[ { JUSTIFIED } RIGHT ]
  { JUST      }
[ LINE NUMBER IS { line-num [ ON NEXT PAGE ] } ]
                 { PLUS line-num-plus        }
  { PICTURE } IS character-string
  { PIC     }
[ [ SIGN IS ] { LEADING  } SEPARATE CHARACTER ]
              { TRAILING }
{ SOURCE IS source-name                                                  }
{ VALUE IS lit                                                           }
{ { SUM { sum-name } ... [ UPON { detail-report-group-name } ... ] } ... }
{       [ RESET ON { control-foot-name } ]                                }
{                  { FINAL             }                                 }
[ [ USAGE IS ] DISPLAY ] .
```

Syntax Rules

All Formats

1. The report group description entry can appear only in the Report Section.
2. Except for the *group-data-name* clause, which when present must immediately follow *level-number*, the clauses may be in any sequence.
3. The description of a report group may consist of one, two, or three hierarchical levels:
   a. The first entry that describes a report group must be a Format 1 entry.
   b. Both Format 2 and Format 3 entries may be immediately subordinate to a Format 1 entry.
   c. At least one Format 3 entry must be immediately subordinate to a Format 2 entry.
   d. Format 3 entries must define elementary data items.
4. In the Report Section, the USAGE clause is used only to declare the usage of printable items.

a. If the USAGE clause appears in a Format 1 or Format 2 entry, at least one subordinate entry must define a printable item.

b. In Format 3, the USAGE clause must define a printable item.

5. An entry containing a LINE NUMBER clause must not have a subordinate entry that also contains a LINE NUMBER clause.

Format 1

6. *group-data-name* is required only when:

   a. A GENERATE statement references a DETAIL report group.

   b. An UPON phrase of a SUM clause references a DETAIL report group.

   c. A USE BEFORE REPORTING sentence references a DETAIL report group.

   d. The name of a CONTROL FOOTING report group qualifies a reference to a sum counter.

   If specified, *group-data-name* can be used as a sum counter qualifier and can be referenced only by:

   - GENERATE statements
   - UPON phrases of the SUM clause
   - USE BEFORE REPORTING declaratives

Format 2

7. *level-number* can be any integer from 02 to 48 inclusive.

8. A Format 2 entry must contain at least one optional clause.

9. In a Format 2 entry, *group-data-name* is optional. It can only qualify a sum counter reference.

Format 3

10. *level-number* can be any integer from 02 to 49 inclusive.

11. A GROUP INDICATE clause can appear only in a DETAIL report group.

12. A SUM clause can appear only in a CONTROL FOOTING report group.

13. An entry containing a COLUMN NUMBER clause but no LINE NUMBER clause must be subordinate to an entry containing a LINE NUMBER clause.

14. *group-data-name* is optional but can be specified in any entry. *group-data-name* can be referenced only if the entry defines a sum counter.

15. A LINE NUMBER clause must not be the only clause specified. Refer to Syntax Rule 3d.

16. An entry containing a VALUE clause must also have a COLUMN NUMBER clause.

17. A printable item is a data item whose size and content are specified by an elementary report entry.

18. An elementary report entry contains a COLUMN NUMBER clause, a PICTURE clause, and a SOURCE, SUM, or VALUE clause.

19. Figure 5.7 shows all permissible clause combinations for Format 3. You read the table from left to right along the selected row.

Figure 5.7. Format 3 Clause Combinations

PIC	COLUMN	SOURCE	SUM	VALUE	JUST	BLANK WHEN ZERO	GROUP INDICATE	USAGE	SIGN	LINE
M	-	-	M	-	-	-	-	-	P	P
M	M	-	M	-	-	P	-	P	P	P
M	P	M	-	-	P	-	P	P	P	P
M	P	M	-	-	-	P	P	P	P	P
M	M	-	-	M	P	-	P	P	P	P

Legend:

M Mandatory

P Permitted but not required

- Not permitted

VM-0592A-AI

General Rules

1. Format 1 is the Report Group entry. The report group is defined by the contents of this entry and all of its subordinate entries.

2. The BLANK WHEN ZERO clause, the JUSTIFIED clause, and the PICTURE clause for Report Writer are the same as those in the Data Description Section.

Examples

The *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/] contains examples of each report group description entry format.

Screen Description (Alpha, I64)

Screen Description (Alpha, I64) — A screen description entry describes a video form or a portion of a video form and specifies the attributes, behavior, size, and location of screen items within the video form. The screen description entry is referenced in the Procedure Division by the ACCEPT and DISPLAY statements.

General Formats

Format 1 (Group Screen Item)

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

[ BLANK SCREEN ]

[ FOREGROUND-COLOR IS color-num-1 ]

[ BACKGROUND-COLOR IS color-num-2 ]

[ AUTO ]

[ SECURE ]

[ REQUIRED ]

[ [ USAGE IS ] DISPLAY ]

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

[ FULL ] .
```

Format 2 (Elementary Screen Item)

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

[ BLANK { LINE   } ]
        { SCREEN }

[ BELL ]

[ BLINK ]

[ ERASE { EOL } ]
        { EOS }

[ HIGHLIGHT ]
[ LOWLIGHT  ]

[ REVERSE-VIDEO ]

[ UNDERLINE ]

[ LINE NUMBER IS [ PLUS ] { identifier-1 } ]
                          { integer-1    }

[ COLUMN NUMBER IS [ PLUS ] { identifier-2 } ]
                            { integer-2    }

[ FOREGROUND-COLOR IS color-num-1 ]

[ BACKGROUND-COLOR IS color-num-2 ]

VALUE IS literal-1 .
```

Format 3 (Elementary Screen Item)

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

[ BLANK { LINE   } ]
        { SCREEN }

[ BELL ]

[ BLINK ]

[ ERASE { EOL } ]
        { EOS }

[ HIGHLIGHT ]
[ LOWLIGHT  ]

[ REVERSE-VIDEO ]

[ UNDERLINE ]

[ LINE NUMBER IS [ PLUS ] { identifier-1 } ]
                          { integer-1    }

[ COLUMN NUMBER IS [ PLUS ] { identifier-2 } ]
                            { integer-2    }

[ FOREGROUND-COLOR IS color-num-1 ]

[ BACKGROUND-COLOR IS color-num-2 ]

{ PICTURE } IS picture-string-1
{ PIC     }

      { USING identifier-3                                  }
      {       { FROM { identifier-4 } |                     }
      {       {      { literal-1    }                       }
      {       { TO identifier-5      |                     }

[ [ USAGE IS ] DISPLAY ]

[ BLANK WHEN ZERO ]

[ { JUSTIFIED } RIGHT ]
  { JUST      }

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

[ AUTO ]

[ SECURE ]

[ REQUIRED ]

[ FULL ] .
```

Syntax Rules

All Formats

1. *level-number* can be any number from 01 to 49.

2. Each elementary screen description entry must contain at least one of the following clauses:

   BELL
   BLANK
   COLUMN
   LINE
   PICTURE
   VALUE

3. If the FOREGROUND-COLOR, BACKGROUND-COLOR, or SIGN clauses are specified in both the group screen description entry and the subordinate description entry for a screen item, then the subordinate screen description entry's clauses will take effect.

4. *screen-name* assigns a name to the screen item described in the screen description entry and must conform to the rules for user-defined names. If either the optional *screen-name* or the key word FILLER is specified, it must be the first word following the level number in each screen description entry.

5. If both *screen-name* and FILLER are omitted, the screen item being described is treated as though FILLER had been specified, and cannot be referenced in an ACCEPT or DISPLAY statement.

6. Each level 01 item must have a screen name.

7. A screen item can be referenced only in an ACCEPT or DISPLAY statement.

8. *color-num-1* and *color-num-2* are integers in the range 0–7. *color-num-1* and *color-num-2* represent specific colors as described in Table 5.5:

   Table 5.5. Color Table

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

9. The USING phrase is equivalent to the combination of FROM and TO phrases, each specifying the same identifier.

10. *identifier-3*, *identifier-4* and *identifier-5* must be defined in the File, Working-Storage, or Linkage Section.

11. *identifier-1* and *identifier-2* must be described as elementary unsigned numeric integer data items.

12. *literal-1* must be a nonnumeric literal.

13. For a description of *picture-string-1*, see the PICTURE Clause section of this chapter.

General Rules

All Formats

1. An input screen item is one whose description contains a TO clause.
2. An output screen item is one whose description contains a FROM clause.
3. A literal screen item is one whose description contains a VALUE clause.
4. An update screen item is one whose description contains a USING clause.
5. An input-output screen item is one whose description contains both a FROM phrase and a TO phrase that may or may not reference the same identifier. The rules for update screen items also apply to input-output screen items.
6. The LINE and COLUMN clauses should not be specified within a screen description entry in such a way that fields overlap on the screen or fall beyond the screen boundaries.

Format 1

7. Format 1 is used for group screen items.
8. All clauses within a group screen description entry are inherited by subordinate screen description entries with the exception of the BLANK SCREEN clause.
9. If the SECURE clause is specified, it applies to each subordinate input screen item.
10. If the AUTO, FULL, or REQUIRED clauses are specified, they apply to each subordinate input and update screen item.
11. If the BACKGROUND-COLOR, FOREGROUND-COLOR, or SIGN clauses are specified, they apply to each subordinate input, output, and update screen item.

Format 2

12. Format 2 is used to describe a literal screen item.

Format 3

13. Format 3 is used to describe input, output, or update screen items.

ACCESS MODE

ACCESS MODE — 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

[ACCESS MODE IS] { SEQUENTIAL | RELATIVE KEY IS rel-key { 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). It can be qualified but cannot be in a record description entry for the same *file-name*.
2. The ACCESS MODE clause can be in the file's SELECT clause. However, it cannot be in both the SELECT clause and file description entry for the same file.
3. If the USING or GIVING phrases of a SORT or MERGE statement contain the name of the file, the ACCESS MODE RANDOM clause cannot be used for the file.
4. If *rel-key* is associated with an external file connector, *rel-key* must reference the same data item in every program in the run unit.
5. If a START statement references a relative file, the program must specify the RELATIVE KEY phrase for that file.

General Rules

All Formats

1. If there is no ACCESS MODE clause, the access mode is sequential.
2. For sequential access, the sequence in which the program accesses the records depends on the organization of the file, as follows:
   - 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 sort order (ascending or descending) of record key values in the established Key of Reference.

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 consecutively higher relative record numbers.

6. The Relative Key data item associated with the execution of an Input/Output statement is *rel-key* in the file description entry (or SELECT clause) associated with the statement.

ALTERNATE RECORD KEY

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

General Format

```
ALTERNATE RECORD KEY IS { alt-key          }[ WITH DUPLICATES ]
                        { seg-key = {seg} ... }

[ ASCENDING  ]
[ DESCENDING ]
```

`alt-key`

is the Record Key for the file. It is the data-name of a data item in a record description entry for the file. It can be qualified, but it cannot be a group item that contains a variable-occurrence data item. The data item must be described as one of the following:

- Alphanumeric item
- Alphabetic item
- Group item
- Unsigned numeric display item
- COMP-3 integer
- COMP integer

`seg-key`

is a segmented-key name that represents the concatenation of one or more (up to eight) occurrences of *seg*.

`seg`

is the data-name of a data item in a record description entry for the file. It can be qualified, but it cannot be a group item that contains a variable-occurrence data item. The data item must be described as one of the following:

- Alphanumeric item
- Alphabetic item
- Group item

- Unsigned numeric display item

Syntax Rules

1. The ALTERNATE RECORD KEY clause can be in the file's SELECT clause. However, for the same file, it cannot be in both the SELECT clause and file description entry.

2. *alt-key* or the segments of *seg-key* cannot have the same leftmost character position as that of the Prime Record Key data item or any other *alt-key* or segment of *seg-key* for the same file.

General Rules

1. *seg-key* is the concatenation of all specified key segments in the order specified.

2. *seg-key* can be referenced only in a READ (Format 3) or START statement.

3. 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* or the segments of *seg-key*.

   - Define the same relative location in the record as *alt-key* or the segments of *seg-key*.

   - Specify the same number (or less) of ALTERNATE RECORD KEY clauses.

     On UNIX systems, you can specify a different number of keys than was specified when the file was created, if the relaxed key check option (`-relax_key_checking`) is used.

   - Maintain the same order of ALTERNATE RECORD KEY clauses.

   - Specify the same order of keys (ASCENDING or DESCENDING) in each ALTERNATE RECORD KEY clause as the order used when the file was created.

4. The DUPLICATES phrase specifies that two or more records in the file can have duplicate values in the same *alt-key* or the segments of *seg-key*. If there is no DUPLICATES phrase, two records cannot have the same value in corresponding Alternate Record Keys.

   On OpenVMS, if the program was compiled with the /CHECK=DUPLICATE_KEYS qualifier on the command line, and the duplicate key specification on a file's FD (in other words, specified in the WITH DUPLICATES phrase) does not match that of the actual file, a run-time diagnostic will be issued when an attempt is made to open the file with an OPEN statement.

   The /CHECK=DUPLICATE_KEYS qualifier is not supported for remotely accessed files. Duplicate keys, key length, and number of keys are not checked for remote files, that is, files accessed over the network.

   On UNIX systems, DUPLICATES must match the specification for DUPLICATES when the file was created, unless the relaxed key check option is used.

5. If a file has more than one record description entry, only one of these record description entries must describe *alt-key* or the segments of *seg-key*. The character positions referenced by *alt-key* or the segments of *seg-key* in that record description are implicitly referenced in all other record description entries for the file.

6. A file can have up to 254 Alternate Record Keys.

7. If the associated file connector is an external file connector, all File Description entries in the run unit that are associated with the file connector must define the same data description entry for *alt-key* or the segments of *seg-key*, with the same relative location within the record.

8. Each key can be specified as ASCENDING or DESCENDING (ASCENDING is the default). In an ASCENDING key, lower key values occur toward the beginning of the sorted file. In a DESCENDING key, higher key values occur toward the beginning of the sorted file.

Additional Reference

- RECORD KEY clause
- SORT statement in Chapter 6
- MERGE statement in Chapter 6

AUTO

AUTO — In the context of ACCEPT, the AUTO clause moves the cursor to the next field when the last character of an input or update field that was defined with the AUTO clause is entered.

General Format

AUTO

Syntax Rule

The AUTO clause cannot be specified in the description of a literal screen item.

General Rules

1. If the AUTO clause is specified at group level, it applies to each input and update screen item in that group.

2. The AUTO clause is significant in the context of an ACCEPT.

3. The AUTO clause is ignored in the description of an output screen item.

4. If there is only one field to input, or if the field is the last one of the screen, the ACCEPT statement is completed when the last character of the field is entered.

Additional Reference

ACCEPT statement in Chapter 6

BACKGROUND-COLOR (Alpha, I64)

BACKGROUND-COLOR (Alpha, I64) — The BACKGROUND-COLOR clause specifies the background color for the screen item.

General Format

BACKGROUND-COLOR IS color-num

color-num

is an integer in the range 0–7 specifying a color as follows:

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

Syntax Rule

The BACKGROUND-COLOR clause can be specified in any screen description entry.

General Rules

1. The BACKGROUND-COLOR clause is effective only with color screens.
2. If the BACKGROUND-COLOR clause is omitted, the initial default background color is black.
3. If the clause is specified at group level, it applies to all subordinate screen items.
4. If the BLANK SCREEN clause is specified and the BACKGROUND-COLOR clause is specified or inherited, then when a DISPLAY statement displays the screen item, the specified color becomes the default background color. It remains the default background color until another screen item with this combination of options is displayed (whether in the same DISPLAY statement or in another).

Technical Note

The colors in the list above are supported only on terminals and workstations that support the ANSI Standard color sequences.

Additional References

- ACCEPT statement in Chapter 6
- DISPLAY statement in Chapter 6

BELL

BELL — The BELL clause sounds the workstation or terminal audio tone.

General Format

BELL

Syntax Rule

The BELL clause can be specified only for elementary screen description entries.

General Rule

The audio tone sounds when a DISPLAY statement displays a screen item whose description contains a BELL clause.

Additional Reference

DISPLAY statement in Chapter 6

BLANK

BLANK — The BLANK clause clears a screen line or clears the whole screen before displaying the screen item.

General Format

BLANK { LINE | SCREEN }

Syntax Rules

1. The BLANK SCREEN clause can be specified for any screen description entry.
2. The BLANK LINE clause can be specified only for elementary screen description entries.

General Rules

1. The BLANK SCREEN clause executes before a screen item displays, no matter where it appears in the screen item's description. When the BLANK SCREEN clause is specified, the screen is cleared and the cursor is placed at line 1, column 1.
2. When BLANK LINE is specified in an elementary screen item's description, blanking begins at column 1 of the specified line and continues through to the end of the line.
3. If neither the BLANK clause nor the ERASE clause (Alpha, I64) is specified, only the particular character positions corresponding to the screen item are modified when the item is displayed. The remainder of the screen content is not changed.
4. The BLANK SCREEN clause returns the screen to the initial defaults for background and foreground color if the BACKGROUND-COLOR and FOREGROUND-COLOR clauses are not specified, respectively.
5. The BLANK clause is ignored in an ACCEPT statement.

Additional Reference

DISPLAY statement in Chapter 6

BLANK WHEN ZERO

BLANK WHEN ZERO — The BLANK WHEN ZERO clause replaces zeros with spaces when a data item's value is zero. In the context of the Screen Section, it displays spaces when the value of a screen item to be displayed on the screen is zero.

General Format

BLANK WHEN { ZERO | ZEROES | ZEROS }

Syntax Rules

1. The BLANK WHEN ZERO clause can be used only for a numeric or numeric edited elementary item.
2. A data item or screen item containing the BLANK WHEN ZERO clause must be implicitly or explicitly described with DISPLAY usage.
3. The syntax for a data item allows the spelling ZERO or ZEROES or ZEROS. The syntax for a screen item allows the spelling ZERO only.

General Rules

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

Additional Reference

DISPLAY statement in Chapter 6

BLINK (Alpha, I64)

BLINK (Alpha, I64) — The BLINK clause displays characters on the screen with the *blink on* character attribute.

General Format

`BLINK`

Syntax Rule

The BLINK clause can be specified only in an elementary screen description entry.

General Rule

Blinking is only detectable when any of the following conditions are true:

- Nonspace characters are displayed.
- The underline and/or reverse-video attributes are specified.
- The terminal screen is set to light background.

Additional References

- DISPLAY statement in Chapter 6
- ACCEPT statement in Chapter 6

CODE

CODE — The CODE clause specifies a two-character literal that identifies each print line as belonging to a specific report.

Function

CODE [report-code]

report-code

must be a two-character nonnumeric literal.

Syntax Rule

If the CODE clause is specified for any report in a file, it must be specified for all reports in that file.

General Rules

1. When the CODE clause is specified, *report-code* is automatically placed in the first two character positions of each Report Writer logical record.
2. The positions occupied by *report-code* are not included in the description of the print line, but are included in the logical record size.

Example

The following file contains three reports:

```
FILE SECTION.
FD  REPORT-FILE
    LABEL RECORDS ARE STANDARD
    REPORTS ARE REPORT1
                REPORT2
                REPORT3.
REPORT SECTION.
RD  REPORT1 ...
    CODE "AA".

RD  REPORT2...
    CODE "BB".

RD  REPORT3...
    CODE "CC".
```

Additional Reference

RD (Report Description)

COLUMN NUMBER

COLUMN NUMBER — In a report group description, the COLUMN NUMBER clause identifies a printable item and specifies the position of the item on a print line. In a screen description, the COLUMN NUMBER clause specifies the horizontal screen coordinate for a screen item.

General Formats

Format 1 (Report Description)

COLUMN NUMBER IS column-num

Format 2 (Screen Description)

COLUMN NUMBER IS [PLUS] { identifier-1 | integer-1 }

column-num

is a positive integer greater than zero.

identifier-1

is an elementary unsigned numeric integer data item. It cannot be subscripted.

integer-1

is an unsigned integer value.

Syntax Rules (Report Description)

1. The COLUMN NUMBER clause can be specified only at the elementary level within a report group. The COLUMN NUMBER clause, if present, must appear in a Format 3 Report Group Description entry, or be subordinate to an entry that contains a LINE NUMBER clause in a Format 2 Report Group Description entry.

2. A printable item is a data item whose size and content is specified by an elementary report entry.

3. An elementary report entry contains a COLUMN NUMBER clause, a PICTURE clause, and a SOURCE, SUM, or VALUE clause.

4. Each printable item within a given print line must be defined in ascending column number order such that each printable item occupies a unique sequence of contiguous character positions.

Syntax Rules (Screen Description)

1. The COLUMN clause can be specified only in an elementary screen description entry.

2. *identifier-1* cannot be subscripted.

General Rules (Report Description)

1. The presence of a COLUMN NUMBER clause indicates that these items, if present, are to be presented on the print line:

   - The object of a SOURCE clause
   - The object of a VALUE clause
   - The sum counter in a SUM clause

The absence of a COLUMN NUMBER clause indicates that the entry is not printable.

2. Column number 1 is the leftmost position of the print line.

3. *column-num* specifies the column number of the leftmost character position of the printable item.

4. The Report Writer Control System supplies space characters for all positions of a print line not occupied by printable items.

General Rules (Screen Description)

1. The COLUMN clause, in conjunction with the LINE clause, establishes the starting position for a screen item. This position is an offset from the starting screen coordinates specified in the ACCEPT or DISPLAY statement. The COLUMN clause specifies the horizontal coordinate.

2. The COLUMN clause without the PLUS phrase specifies the absolute column position of the screen item.

3. The COLUMN clause with the PLUS phrase specifies a column number relative to that at which the preceding item ends, regardless of whether or not the ACCEPT or DISPLAY statement displays the preceding item on the screen.

4. A setting of COLUMN 1 is assumed in screen description entries that specify the LINE clause but omit the COLUMN clause.

5. If both the LINE clause and the COLUMN clause are omitted, the following apply:

   - If no previous elementary screen item is defined, LINE 1 COLUMN 1 is assumed.

   - If a previous screen item is defined, the ending line of that previous item and COLUMN PLUS 1 is assumed. The screen item then starts immediately following the preceding screen item.

Additional References

- Report Group Description

- LINE NUMBER (Alpha, I64) clause

- ACCEPT statement in Chapter 6

- DISPLAY statement in Chapter 6

Examples (Report Description)

1. The following is an example of the COLUMN NUMBER clause in a LINE NUMBER clause:

```
  02     LINE 10 COLUMN 1 PIC X(11) VALUE "TOTAL ITEMS".

                       1         2         3         4
column:     12345678901234567890123456789012345678 90

            TOTAL ITEMS
```

2. The following is an example of the COLUMN NUMBER clause subordinate to a LINE NUMBER clause:

```
  02     LINE 5 ON NEXT PAGE.
    03      COLUMN 1  PIC X(10)        VALUE "(Id Number".
    03      COLUMN 12 PIC 9999         VALUE 1234.
    03      COLUMN 16 PIC X            VALUE ")".
    03      COLUMN 18 PIC X(11)        VALUE "TOTAL SALES".
    03 TSAL COLUMN 30 PIC $$$$,$$$.99- VALUE 123456.78.

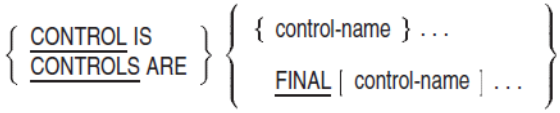
                        1         2         3         4
column:     123456789012345678901234567890123456789012345

            (Id Number 1234) TOTAL SALES $123,456.78
```

CONTROL

CONTROL — The CONTROL clause establishes the levels of the control hierarchy for the report.

General Format



control-name

is any *data-name* in the Subschema, File, Working-Storage, or Linkage Section.

Syntax Rules

1. *control-name* can be qualified.
2. Each occurrence of *control-name* must identify a different data item.
3. *control-name* must not have a variable-occurrence data item subordinate to it.
4. If the associated report file connector is an external file connector, *control-name* must reference the same external data item in all programs in the run unit.

General Rules

1. The word FINAL specifies the most major control item. From here, the hierarchy descends to *control-name*, which is the major control; to the next recurrence of *control-name*, which is an intermediate control; and so forth to the last recurrence of *control-name*, which is the minor control.
2. A control break is a change in the value of a *control-name*.
3. FINAL is used when the most inclusive control group in the report is not associated with a *control-name*.
4. The first time a GENERATE statement is executed, the Report Writer Control System (RWCS) saves the values of all control data items associated with that report. After that, every time a GENERATE statement is executed, the RWCS tests those control data items to see if their values have changed. If so, a control break occurs. This control break is associated with the highest level control item whose value has changed.

5. Control breaks cause the RWCS to present appropriate CONTROL HEADER and CONTROL FOOTING report groups for printing. Figure 5.8 shows the report groups the RWCS processes (X) when you define FINAL, major, intermediate, or minor *control-name* in a CONTROL HEADING or CONTROL FOOTING phrase in a Report Group Description entry. For example, if the value in a major *control-name* changes, the RWCS processes all major, intermediate, and minor control groups specified in CONTROL HEADING and CONTROL FOOTING report groups.

Figure 5.8. Control Break Levels and Their Printed Report Groups

Control Groups to Process / Control Break Level	CONTROL HEADING CONTROL FOOTING			
	FINAL	Major	Intermediate	Minor
FINAL	X	X	X	X
Major		X	X	X
Intermediate			X	X
Minor				X

VM-0593A-AI

6. The RWCS tests for a control break by comparing the contents of each control data item with the prior contents of each control data item that were saved when the previous GENERATE statement for the same report was executed. The RWCS applies the inequality relation test as follows:

   - If the control data item is a numeric data item, the relation test is for the comparison of two numeric operands.
   - If the control data item is an index data item, the relation test is for the comparison of two index data items.
   - If the control data item is other than as described in Figure 5.8, the relation test is for the comparison of two nonnumeric operands.

Examples

1. This example prints a total record count from TOTAL-LINE at the end of the report because control is FINAL. It is a major control break and prints only once.

```
WORKING-STORAGE SECTION.
01      RECORD-COUNT    PIC 9(9) VALUE 0.

REPORT SECTION.
RD      MASTER-REPORT...
        CONTROL IS FINAL.

01      DETAIL-LINE TYPE IS DETAIL...

01      TOTAL-LINE  TYPE IS CONTROL FOOTING FINAL.
   02   COLUMN 20 PIC X(17) VALUE "TOTAL RECORDS: ".
   02   COLUMN 40 PIC ZZZ,ZZZ,ZZ9 SOURCE RECORD-COUNT.
PROCEDURE DIVISION.
BEGIN.
    OPEN INPUT...
    OPEN OUTPUT...
    INITIATE MASTER-REPORT.
```

```
010-READ-FILE.
    READ... AT END GO TO 999-EOJ.
    GENERATE DETAIL-LINE.
    ADD 1 TO RECORD-COUNT.
    GO TO 010-READ-FILE.
999-EOJ.
    TERMINATE MASTER-REPORT.
    CLOSE...
    STOP RUN.
```

2. In the following example, a report defines four control totals in the control clause. The source of these control totals is in an input file—INPUT-FILE. The file is presorted in ascending sequence by MAJOR-CONTROL, INTERMEDIATE-CONTROL, and MINOR-CONTROL. The RWCS will monitor these fields in the input file for any changes. If a new record contains data different from the previous record read, the RWCS triggers a control break.

   In this example, if the value in MINOR-CONTROL changes, a break occurs and the RWCS processes the minor control report group CONTROL FOOTING MINOR-CONTROL. If the value in INTERMEDIATE-CONTROL changes, a break occurs and the RWCS processes the intermediate and minor control report groups CONTROL FOOTING INTERMEDIATE-CONTROL and CONTROL FOOTING MINOR-CONTROL. If the value in MAJOR-CONTROL changes, a break occurs and the RWCS processes the major, intermediate, and minor control report groups CONTROL FOOTING MAJOR-CONTROL, CONTROL FOOTING INTERMEDIATE-CONTROL, and CONTROL FOOTING MINOR-CONTROL.

```
FILE SECTION.

FD      INPUT-FILE...
01      INPUT-RECORD.
    02    MAJOR-CONTROL           PIC...
    02    ...
    02    MINOR-CONTROL           PIC...
    02    ...
    02    INTERMEDIATE-CONTROL    PIC...
    02    ...

FD      REPORT-FILE...
        REPORT IS SUMMARY-REPORT.
REPORT SECTION.
RD      SUMMARY-REPORT...
        CONTROLS ARE FINAL
                     MAJOR-CONTROL
                     INTERMEDIATE-CONTROL
                     MINOR-CONTROL.

01  DETAIL-LINE TYPE IS DETAIL...

01  TYPE IS CONTROL FOOTING FINAL ...
01  TYPE IS CONTROL FOOTING MINOR-CONTROL...
01  TYPE IS CONTROL FOOTING MAJOR-CONTROL...
01  TYPE IS CONTROL FOOTING INTERMEDIATE-CONTROL...
```

Additional References

- Report Group Description
- Section 6.5.1: Relation Conditions

Data-Name

Data-Name — *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 Rules

1. 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.

2. In the Report Section, *data-name* need not appear in a report group description entry and the key word FILLER must not be used.

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*.

4. In the Report Section, *data-name* must be used when:

   a. *data-name* represents a report group to be referred to by a GENERATE or a USE statement in the Procedure Division.

   b. Reference will be made to the sum counter in the Procedure Division or Report Section.

   c. The UPON phrase of the SUM clause references a DETAIL report group.

   d. *data-name* provides sum counter qualification.

5. If this clause is omitted, the Report Writer Control System does not allow explicit references to the data item.

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.
```

3. Report Writer items:

   In this Report Writer example, the program can refer to DL-NAME and DETAIL-LINE, but not to the data beginning in LINE 10, COLUMN 25. Note that FILLER cannot be used in place of a Report Writer *data-name*.

```
01  DETAIL-LINE TYPE IS DETAIL.
    02  LINE 10.
        03  DL-NAME     COLUMN 1   SOURCE INPUT-NAME.
        03              COLUMN 25  SOURCE INPUT-ADDRESS.
```

DATA RECORDS

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

General Format

DATA { RECORD IS | RECORDS ARE } {ec-name} ...

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.

ERASE (Alpha, I64)

ERASE (Alpha, I64) — The ERASE clause clears from the starting cursor position to the end of either the line or the screen.

General Format

ERASE { EOL | EOS }

Syntax Rule

The ERASE clause can be specified only for elementary screen description entries.

General Rules

1. Blanking begins at the starting position of the screen item in whose description the ERASE EOL clause is included, and continues to the end of the line.
2. Blanking begins at the starting position of the screen item in whose description the ERASE EOS clause is included, and continues through to the end of the screen.
3. If you specify neither the BLANK nor the ERASE clause, only the particular character positions corresponding to the screen item are modified when the element is displayed. The rest of the screen content remains the same.
4. The ERASE clause is ignored in an ACCEPT statement.

Additional References

DISPLAY statement in Chapter 6

EXTERNAL

EXTERNAL — The EXTERNAL clause specifies that a data item or a file connector in a defining program is common to other programs in the run unit if the program defines it identically. The group and elementary data items of an external data record and files associated with an external file connector are available to every program in the image that describes them.

General Format

IS EXTERNAL

Syntax Rules

1. The EXTERNAL clause can appear only in file description entries or in record description entries in the Working-Storage Section.
2. In a record description entry, only level numbers 01 and 77 can specify the EXTERNAL clause.
3. A program and any program it contains, cannot define identical *data-names* if their data description entries or file description entries have EXTERNAL clauses.
4. The VALUE clause or the REDEFINES clause cannot be in a data description entry that contains or is subordinate to, an entry that contains the EXTERNAL clause.
5. When using the SAME RECORD AREA clause for several files, the Record Description entries or the file description entries for these files must not include the EXTERNAL clause.
6. Entries that contain the EXTERNAL clause must be named.

General Rules

1. Data in a record either subordinate to an external FD, or named by the subject of the EXTERNAL clause, is external. Any program in the image that describes and optionally redefines this data may access and process this data subject to the following general rules.

2. If two or more programs in the image describe the same external data record, the associated data description entries (except the GLOBAL clause) must be identical. All subordinate *data-names*, data items, and redefinitions must also be identical.

3. A program that describes an external data record can contain a Data Description entry that redefines the complete external record. This redefinition need not be the same in other programs in the image.

4. Use of the EXTERNAL clause does not imply that the associated *file-name* or *data-name* is a global name.

5. The file connector associated with a file description entry is an external file connector.

6. If two or more programs in an image describe the same external file connector, the clauses associated with the description of that file must be functionally identical and any data items referenced by those clauses must be external.

Technical Notes

- Each external sequential file becomes a print format file.

- Each external data record becomes a PSECT (on OpenVMS systems) or a global (on UNIX systems), whose name is the 01-level record.

  Each file connector for external files becomes a PSECT (on OpenVMS systems) or a global (on UNIX systems), whose name is the name of the file. The records associated with this file become external data records.

  External record items and files are implemented as overlayable shared PSECTs (on OpenVMS systems) or globals (on UNIX systems). Therefore, the same storage area is shared among all separately compiled programs for that named external record or file. The PSECT or global is created for compatibility with BASIC COMMON/MAP and FORTRAN labeled COMMON.

  On OpenVMS, for more information on overlayable PSECTs, refer to the LINK documentation in the OpenVMS documentation set.

  On UNIX, for more information on globals, refer to the `ld` documentation in the UNIX documentation set.

- On UNIX systems, an external data item is case-sensitive. By default, an external data item is converted to lowercase for all separately compiled program units. Other programs (VSI COBOL for OpenVMS as well as other languages) must specify the data item in lowercase.

  However, if the `names` option is set to `uppercase` on the command line, other programs must specify the data item in uppercase. If the `names` option is set to `as_is`, the effect on an external data item is as if `uppercase` were specified. (The `as_is` setting is used for calling non-COBOL programs with mixed case.)

Examples

In the following Working-Storage entries, the data items in RECORD-A are available to any program in the run unit that also describes RECORD-A and its data items. RECORD-B and the data items in it are not available to any other program.

```
01 RECORD-A EXTERNAL.
    03  ITEMA  PIC X.
    03  ITEMB  PIC X(22).
```

```
    03  ITEMC  PIC 999.
01 RECORD-B.
    03  ITEMA  PIC X(12).
    03  ITEMD  PIC X.
    03  ITEME  PIC 9(18).
```

Additional References

- GLOBAL clause
- REDEFINES clause

FILE STATUS

FILE STATUS — The FILE STATUS clause specifies a data item to contain the status of an input/output operation.

General Format

FILE STATUS IS file-stat

file-stat

is the data-name of a two-character alphanumeric Working-Storage Section, or Linkage Section data item. *file-stat* is the file's FILE STATUS data item.

Syntax Rules

1. *file-stat* can be qualified.
2. The FILE STATUS clause can be in the file's SELECT clause or in its file description entry. However, it cannot be in both the SELECT clause and the file description entry for the same file.
3. If the FILE STATUS clause is associated with an external file connector, *file-stat* must reference the same data item in all programs in the run unit.

General Rule

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

Additional References

- Appendix C: *File Status Values*
- Section 6.6.8: I-O Status

FOREGROUND-COLOR (Alpha, I64)

FOREGROUND-COLOR (Alpha, I64) — The FOREGROUND-COLOR clause specifies the foreground color for a screen item.

General Format

FOREGROUND-COLOR IS color-num

`color-num`

is an integer in the range 0–7 specifying a color as follows:

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

Syntax Rule

The FOREGROUND-COLOR clause can be specified in any screen description entry.

General Rules

1. The FOREGROUND-COLOR clause is effective only with color screens.
2. If the FOREGROUND-COLOR clause is omitted, the initial default foreground color is white.
3. If the clause is specified at group level, it applies to all subordinate screen items.
4. If the BLANK SCREEN clause is specified and the FOREGROUND-COLOR clause is specified or inherited, then when a DISPLAY statement displays the screen item, the specified color becomes the default foreground color. It remains the default foreground color until another screen item with this combination of options is displayed (whether in the same DISPLAY statement or in another).
5. If the HIGHLIGHT clause is also specified, foreground and background colors are brightened and lightened; for example, black may become grey and brown may become yellow.

Technical Note

The colors in the list above are supported only on terminals and workstations that support the ANSI Standard color sequences.

Additional References

- ACCEPT in Chapter 6
- DISPLAY in Chapter 6

FULL (Alpha, I64)

FULL (Alpha, I64) — The FULL clause specifies that a screen item must be left either completely empty or it must be entirely filled with data.

General Format

`FULL`

Syntax Rules

1. If the FULL clause is specified in a screen description entry, the JUSTIFIED clause cannot be specified.

2. The FULL clause is valid only in the description of an input or update screen item.

General Rules

1. If the FULL clause is specified at group level, it applies to all subordinate input or update screen items.
2. The FULL clause is effective during the execution of any ACCEPT statement when the cursor enters the screen item. Until this clause is satisfied, the operator cannot leave the field and normal terminator keystrokes are rejected.
3. To satisfy the FULL clause for an alphanumeric screen item, either the field must contain all spaces, or both the first and last character positions must contain nonspace characters.
4. To satisfy the FULL clause for a numeric or numeric edited screen item, either the value must be zero or there must be no digit position in which zero suppression has taken effect.
5. For update fields, the FULL clause can be satisfied by the contents of the identifier or literal referenced in the FROM or USING phrase of the PICTURE clause, as well as by operator-keyed data.
6. The FULL clause is not effective if a function key terminates the accept operation.
7. Specifying the FULL and REQUIRED clauses together requires that the user must always entirely fill the field.
8. The FULL clause is ignored for an elementary output field.

Additional Reference

ACCEPT statement in Chapter 6

GLOBAL

GLOBAL — The GLOBAL clause specifies that *data-name*, *file-name*, or *report-name* is available to every program contained within the program that declares it.

General Format

IS GLOBAL

Syntax Rules

1. The GLOBAL clause can appear only in file description entries, Report Description entries, a data description entry whose level number is 01, in the File or Working-Storage Section, or a data description entry whose level number is 77, in the Working-Storage Section.
2. In the same Data Division, the GLOBAL clause must not appear in Data Description entries that contain identical *data-names*.
3. If you use the SAME RECORD AREA clause for several files, the Record Description entries or the file description entries for these files must not include the GLOBAL clause.
4. Entries that contain the GLOBAL clause must be named.

General Rules

1. Any *data-name*, *file-name*, or *report-name* specifying the GLOBAL clause is a global name. All data items subordinate to a global *data-name* or *file-name* are global names. All *condition-names* associated with a global name are global names.

2. A statement in a program contained directly or indirectly within a program that describes a global name may reference the name without describing it again.

3. If the GLOBAL clause is used in a data description entry that contains the REDEFINES clause, the global attribute applies only to the subject of the REDEFINES clause.

Technical Note

Each global sequential file becomes a print format file.

Additional Reference

Section 6.2.6: Scope of Names

GROUP INDICATE

GROUP INDICATE — The GROUP INDICATE clause specifies that the associated printable item is presented only on the first occurrence of its DETAIL report group after a control break or page advance.

General Format

```
GROUP INDICATE
```

Syntax Rule

The GROUP INDICATE clause must be specified only in a DETAIL report group entry that defines a printable item.

General Rules

1. If the program contains a GROUP INDICATE clause, the compiler suppresses printing of the printable item and supplies spaces, except:

   a. On the first presentation of the DETAIL report group

   b. On the first presentation of the DETAIL report group after every page advance

   c. On the first presentation of the DETAIL report group after every control break

2. If the program specifies neither the PAGE clause nor the CONTROL clause in a Report Description entry, then the first time a DETAIL report group is presented a GROUP INDICATE printable item is also presented. Thereafter, spaces are supplied for indicated items with SOURCE or VALUE clauses.

Additional Reference

Appendix D: *Report Writer Presentation Rules and Tables*

Example

The following example shows the effect of the GROUP INDICATE clause on a printable item (SOURCE I-NAME).

Sample Program

Without the GROUP INDICATE Clause	With the GROUP INDICATE Clause
01 DETAIL-LINE TYPE IS DETAIL	01 DETAIL-LINE TYPE IS DETAIL
LINE IS PLUS 1.	LINE IS PLUS 1.
02 COLUMN 1 PIC X(15)	02 COLUMN 1 PIC X(15)
	**********
	GROUP INDICATE
	**********
SOURCE I-NAME.	SOURCE I-NAME.
02 COLUMN 20 PIC 9(6)	02 COLUMN 20 PIC 9(6)
SOURCE I-INV-NO.	SOURCE I-INV-NO.

Sample Report

Without the GROUP INDICATE Clause	With the GROUP INDICATE Clause
1 2 3	1 2 3
12345678901234567890123456789 0	12345678901234567890123456789 0
Name Invoice Number	Name Invoice Number
Hendrexon B. 123456	Hendrexon B. 123456
Hendrexon B. 123456	123456
Hendrexon B. 123456	123456
Blare R. 123456	Blare R. 123456
Blare R. 123456	123456
Blare R. 123456	123456
Blare R. 123456	123456
Provinchet R. 123456	Provinchet R. 123456
Provinchet R. 123456	123456
Provinchet R. 123456	123456
Provinchet R. 123456	123456
Provinchet R. 123456	123456

ZK-6150-GE

HIGHLIGHT (Alpha, I64)

HIGHLIGHT (Alpha, I64) — The HIGHLIGHT clause specifies that the field is to appear on the screen with the highest intensity.

General Format

HIGHLIGHT

Syntax Rule

The HIGHLIGHT clause can be specified only for an elementary screen description entry.

Additional References

- ACCEPT in Chapter 6
- DISPLAY in Chapter 6

JUSTIFIED

JUSTIFIED — The JUSTIFIED clause specifies nonstandard data positioning in a screen item or another receiving item.

General Format

{ JUSTIFIED | JUST } RIGHT

Syntax Rules

1. The JUSTIFIED clause can be used only for elementary items and alphanumeric data items. It cannot be used for index data items, numeric data items, or edited data items.
2. JUST is the abbreviated form of JUSTIFIED.

General Rules

1. If a COBOL statement transfers data to a receiving item whose data description contains the JUSTIFIED clause, the Run-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).

Examples

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

Additional References

- MOVE statement in Chapter 6
- Section 5.2.2: COBOL Standard Alignment Rules

LABEL RECORDS

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

General Format

LABEL { RECORDS ARE | RECORD IS } { STANDARD | OMITTED }

General Rule

The LABEL RECORDS clause is for documentation only.

Level-Number

Level-Number — The *level-number* shows the position of a data item or screen item within the hierarchical structure of a logical record or a report group or a screen description. 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 or a screen description entry.
2. Data description entries that are subordinate to a file description (FD) entry have *level-numbers* 01 to 49, 66, or 88.
3. Data description entries in the Working-Storage and Linkage Sections have *level-numbers* 01 to 49, 66, 77, or 88.
4. Report group description entries in the Report Section have *level-numbers* 01 to 49 only. See the Report Group Description entry for additional rules for Report Writer *level-numbers*.
5. Screen description entries in the Screen Section have *level-numbers* 01 to 49 only. See the Screen Description (Alpha, I64) entry for additional rules for Screen Section *level-numbers*.

General Rules

1. The *level-number* 01 identifies the first entry in a record description, report group description, or screen description entry.
2. Multiple level 01 entries subordinate to a file description entry represent implicit redefinitions of the same area.
3. Multiple level 01 entries subordinate to a report description entry do not represent implicit redefinitions of the same area.
4. *Level-number* 66 identifies a RENAMES entry. It can be used only in a Format 2 data description entry.
5. *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.

6. *Level-number* 88 defines a *condition-name* associated with a conditional variable. It can be used only in a Format 3 data description entry.

7. *Level-numbers* 66, 77, and 88 do not imply a hierarchical position.

Additional References

- RD (Report Description) entry
- Data Description entry
- Report Group Description entry
- RENAMES clause
- Section 1.2.1.1 in Section 1.2.1: COBOL Words
- Section 5.1.1: Record Description Entries
- Screen Description (Alpha, I64) entry

LINAGE

LINAGE — 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 is 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. Other open modes (Input, I-O, and Extend) are not permitted and have unpredictable results.

10. LINAGE-COUNTER is global if a file description entry specifies the GLOBAL clause and the LINAGE clause.

11. LINAGE-COUNTER is a 9-digit numeric special register. Procedure Division statements can refer to LINAGE-COUNTER but cannot change its value.

12. If the program has more than one LINAGE-COUNTER, all Procedure Division references to it must be qualified by *file-name*.

13. 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.

14. Execution of an OPEN statement for the file sets its LINAGE-COUNTER to one.
15. Each logical page follows the preceding logical page with no spacing between them.
16. If the file connector associated with this file description entry is an external file connector, all file description entries in the run unit associated with this file connector must have the following features:
    - A LINAGE clause, if any file description entry has a LINAGE clause
    - The same corresponding integer values for *page-lines*, *footing-lines*, *top-lines*, and *bottom-lines*
    - The same corresponding external data items referenced by *page-lines*, *footing-lines*, *top-lines*, and *bottom-lines*

Technical Notes

- On OpenVMS, 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 DCL 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.

  The /NOFEED compiler option to the PRINT command suppresses the insertion of form-feed characters and prints LINAGE files correctly. For example:

  ```
  $ PRINT/NOFEED full-file-name
  ```

- The /NOVFC compiler option can be used on OpenVMS Alpha and I64 to produce a Stream_LF record-formatted print file. The default (/VFC) behavior is to produce a VFC record-formatted file.
- VSI COBOL for OpenVMS on UNIX systems writes LINAGE files with blank lines to simulate WRITE ADVANCING behavior. These blank lines would not be produced on an OpenVMS Alpha or I64 system. When you input a LINAGE file, you must compensate for the difference. For example, use an extra initial READ statement (on UNIX systems) to skip over the leading blank line in the LINAGE file.

Example

The following example specifies a logical page whose size is 26 lines:

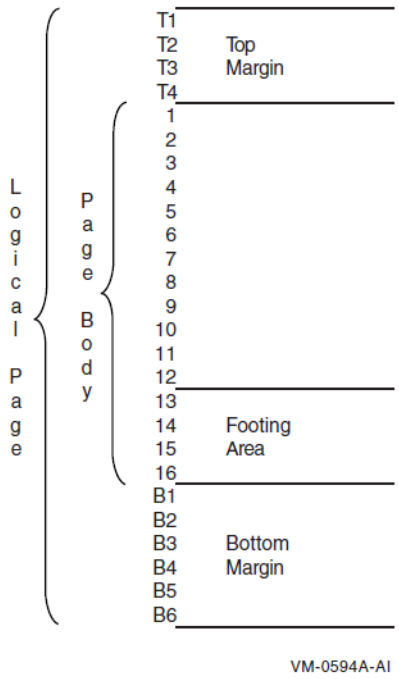
```
FD  PRINT-FILE
    VALUE OF ID IS "REPORT1.LIS"
```

```
LINAGE IS 16 LINES WITH FOOTING AT 13
  LINES AT TOP 4 LINES AT BOTTOM 6.
```

In this example, 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 to 16. The page overflow condition occurs when a WRITE statement causes the LINAGE-COUNTER value to exceed 16.

Figure 5.9 shows the logical page areas resulting from the example.

Figure 5.9. Logical Page Areas Resulting from a LINAGE Clause



Additional References

- GLOBAL clause
- WRITE statement in Chapter 6

LINE NUMBER (Alpha, I64)

LINE NUMBER (Alpha, I64) — The LINE NUMBER clause specifies vertical positioning information for a report group, or specifies the vertical screen coordinate for a screen item.

General Format

Format 1 (Report Description)

LINE NUMBER IS { line-num | [ON NEXT PAGE] | PLUS line-num-plus }

Format 2 (Screen Description)

LINE NUMBER IS [PLUS] { identifier-1 | integer-1 }

`line-num`

is a nonnegative integer. *line-num* represents an absolute line number on a logical page and establishes a print line for a Report Writer report group.

`line-num-plus`

is a positive integer. *line-num-plus* represents a relative line number on a logical page and establishes a print line for a Report Writer report group.

`identifier-1`

is an elementary unsigned numeric integer data item. It cannot be subscripted.

`integer-1`

is an unsigned integer value.

Syntax Rules (Report Description)

1. Neither *line-num* nor *line-num-plus* can exceed three significant digits.

   The PAGE clause defines the length of a logical page and the vertical subdivisions within which each report group is presented. Neither *line-num* nor *line-num-plus* may specify a line outside of the PAGE clause limits. See PAGE clause for more information.

2. Within a given Report Group Description, an entry containing a LINE NUMBER clause must not contain a subordinate entry that also contains a LINE NUMBER clause.

3. Within a given Report Group Description, all absolute LINE NUMBER clauses must precede all relative LINE NUMBER clauses.

4. Within a given Report Group Description, successive absolute LINE NUMBER clauses must specify integers in ascending order. The integers need not be consecutive.

5. If a given Report Description (RD) does not contain a PAGE clause, the program may specify only relative LINE NUMBER clauses in any Report Group Description within that report.

6. Within a given Report Group Description, a NEXT PAGE phrase may appear only once. If present, it must be the first LINE NUMBER clause in that Report Group Description.

7. A LINE NUMBER clause with the NEXT PAGE phrase may appear only in the description of a CONTROL HEADING, DETAIL, CONTROL FOOTING, or REPORT FOOTING report group.

8. Every entry defining a printable item must either contain a LINE NUMBER clause or be subordinate to an entry that contains a LINE NUMBER clause. See the COLUMN NUMBER clause for more information.

9. The first LINE NUMBER clause in a PAGE FOOTING report group must define an absolute *line-num* value.

10. *line-num-plus* may be zero. If *line-num-plus* is zero, the line will be printed on the same line as the previous print line (overprint); however, *line-num-plus* cannot be zero for the first print line of a report group.

Syntax Rules (Screen Description)

1. The LINE clause can be specified only in an elementary screen description entry.
2. *identifier-1* cannot be subscripted.

General Rules (Report Description)

1. To establish each print line for a report group, a program must specify the LINE NUMBER clause.
2. Before presenting the print line, the Report Writer Control System (RWCS) causes line positioning as specified by a LINE NUMBER clause.
3. The NEXT PAGE phrase defines the line number of a new page on which to present the report group.
4. For a complete specification on how to determine the first print line for a report group, see Appendix D: *Report Writer Presentation Rules and Tables*. A partial summary of these rules follows:

   If a relative clause is not the first LINE NUMBER clause in a report group, then the line number on which its print line is presented is determined by the sum of the following:

   - The line number from the previous print line of the report group
   - *line-num-plus* of the relative LINE NUMBER clause

   If the first LINE NUMBER clause in the Report Group Description entry is relative and a PAGE clause is specified, the first print line for the report group is determined as follows. See the PAGE clause for the definitions of *page-size*, *heading-line*, *first-detail-line*, *last-detail-line*, and *footing-line*.

   a. REPORT HEADING

      The RWCS presents this group on a line number whose value is the sum of *line-num* of the first LINE NUMBER clause and *heading-line* minus 1.

   b. PAGE HEADING

      If a REPORT HEADING report group has been presented on the page on which this report group is to appear, the RWCS presents the PAGE HEADING relative to the final LINE-COUNTER setting of the REPORT HEADING.

      If no REPORT HEADING has been presented on the page, the RWCS presents this report group on the line number whose value is the sum of *line-num* of the first LINE NUMBER clause and *heading-line* minus 1.

   c. DETAIL, CONTROL HEADING, or CONTROL FOOTING

      If the value in LINE-COUNTER is less than first-detail-line, the RWCS presents the report group on first-detail-line.

      If the value in LINE-COUNTER is greater than or equal to first-detail-line and if this is the first body group to print on the page, the RWCS presents the report group on the line corresponding to the value in LINE-COUNTER.

      If the value in LINE-COUNTER is greater than or equal to first-detail-line and if this is not the first body group to print on the page, the RWCS presents the report group on the line whose

value is the sum of LINE-COUNTER and *line-num* of the first LINE NUMBER clause of the current CONTROL HEADING, DETAIL, or CONTROL FOOTING report group.

d. PAGE FOOTING

Not applicable. The first LINE NUMBER clause of a PAGE FOOTING report group must contain an absolute line number reference.

e. REPORT FOOTING

If a PAGE FOOTING report group has been presented on the current page, the RWCS presents the REPORT FOOTING report group on the line whose value is the sum of the current value in LINE-COUNTER and *line-num* of the first LINE NUMBER clause of the REPORT FOOTING report group.

If no PAGE FOOTING report group has been presented on the current page, the RWCS presents the REPORT FOOTING report group on the line whose value is the sum of *footing-line* and *line-num* of the first LINE NUMBER clause of the REPORT FOOTING report group.

General Rules (Screen Description)

1. The LINE clause, in conjunction with the COLUMN clause, establishes the starting position for a screen item. This position is an offset from the starting screen coordinates specified in the ACCEPT or DISPLAY statement. The LINE clause specifies the vertical coordinate.
2. The LINE clause without the PLUS phrase specifies the absolute line number.
3. The LINE clause with the PLUS phrase specifies a line number relative to that at which the preceding item ends, regardless of whether or not the ACCEPT or DISPLAY statement displays the preceding item on the screen.
4. If the LINE clause is omitted, the following apply:
   - If no previous screen item is defined, LINE 1 is assumed.
   - If a previous screen item is defined, the ending line of that previous item is assumed.

Additional References

- COLUMN NUMBER clause
- PAGE clause
- Appendix D: *Report Writer Presentation Rules and Tables*
- ACCEPT statement in Chapter 6
- DISPLAY statement in Chapter 6

LOWLIGHT (Alpha, I64)

LOWLIGHT (Alpha, I64) — The LOWLIGHT clause specifies that the field is to appear on the screen with the lowest intensity. When only two levels of intensity are available, normal intensity and LOWLIGHT will be the same.

General Format

LOWLIGHT

Syntax Rule

The LOWLIGHT clause can be specified only for an elementary screen description entry.

Additional Reference

- ACCEPT in Chapter 6
- DISPLAY in Chapter 6

NEXT GROUP

NEXT GROUP — The NEXT GROUP clause specifies information for the vertical positioning of the next report group on a logical page following the presentation of the last line of a report group.

General Format

NEXT GROUP IS { next-group-line-num | PLUS next-group-line-num-plus | NEXT PAGE }

next-group-line-num

is a positive, 1- to 3-digit integer value greater than zero. It represents an absolute line number on a logical page and establishes a print line for the next Report Writer report group.

next-group-line-num-plus

is a positive, 1- to 3-digit integer value. It represents a relative line number on a logical page and establishes a print line for the next Report Writer report group.

Syntax Rules

1. A Report Group entry must not contain a NEXT GROUP clause unless the description of that Report Group contains at least one LINE NUMBER clause.
2. *next-group-line-num* and *next-group-line-num-plus* must not exceed three significant digits.
3. If a Report Description entry omits the PAGE clause, all Report Group Description entries within that report can specify a relative NEXT GROUP clause only.
4. A PAGE FOOTING Report Group must not specify the NEXT PAGE phrase of the NEXT GROUP clause.
5. A PAGE HEADING and REPORT FOOTING Report Group must not specify the NEXT GROUP clause.

General Rules

1. Page positioning occurs after the presentation of the Report Group in which the NEXT GROUP clause appears.
2. To determine a new value for LINE-COUNTER, the Report Writer Control System (RWCS) uses the vertical positioning information from the NEXT GROUP clause along with information from

the TYPE and PAGE clauses, and the value in LINE-COUNTER. See Appendix D: *Report Writer Presentation Rules and Tables*.

3. The RWCS ignores the NEXT GROUP clause on a CONTROL FOOTING Report group when it detects a control break at a level other than the highest level.

4. The NEXT GROUP clause of a CONTROL HEADING, DETAIL, and CONTROL FOOTING report group refers to the next CONTROL HEADING, DETAIL, and CONTROL FOOTING to be presented, and therefore can affect the location at which the next CONTROL HEADING, DETAIL, and CONTROL FOOTING report group is presented. See Appendix D.

5. The NEXT GROUP clause of a REPORT HEADING report group can affect the location at which the PAGE HEADING report group is presented. See Appendix D.

6. The NEXT GROUP clause of a PAGE FOOTING report group can affect the location at which the REPORT FOOTING report group is presented. See Appendix D.

Additional References

- General Rules (Report Description) (General Rule 4) of the LINE NUMBER (Alpha, I64) clause
- Appendix D: *Report Writer Presentation Rules and Tables*

OCCURS

OCCURS — 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 zero.

`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. The subject of a Format 2 OCCURS clause can be followed, in the same record description, only by data description entries subordinate to it.
8. The OCCURS clause cannot be used in a data description entry that has the following:
   - A level-number of 01, 66, 77, or 88
   - A subordinate variable occurrence data item (Format 2 OCCURS clause)
9. 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

10. Each *ind-name* must be a unique word in the program.

11. If the OCCURS clause is in a record description entry containing the GLOBAL clause, *depending-item* must refer to a global item described in the same Data Division.

12. If the OCCURS clause is in a record description entry containing the EXTERNAL clause, *depending-item* must refer to an external item described in the same Data Division.

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.

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* to *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 part of the table used is determined by the location of *depending-item*. If *depending-item* is included in the group, then the operation uses the maximum length of the group. If *depending-item* is not included in the group, then the operation uses only the part of the table area specified by *depending-item*.

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

If the subject of the OCCURS clause (or any of its subordinates) has the SYNCHRONIZED clause, the length of the subject of the OCCURS clause, or the group containing it, could increase.

SYNCHRONIZED clause alignment can add fill bytes to the group containing the subject of the OCCURS clause and to the subject itself.

Examples

1. One-dimensional table:

   This record description entry describes a 20-character record. The record contains 10 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 8 occurrences of ITEMB, a 40-character data item. ITEMB contains 10 occurrences of ITEMC, a 4-character data item. Each ITEMC contains 2 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 10 occurrences of ITEMC. Similarly, ITEMB (5) refers to the fifth group of 10 occurrences of ITEMC.

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

   ```
   ITEMB (1)    DEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEE
   ITEMB (2)    DEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEE
   ITEMB (3)    DEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEE
   ITEMB (4)    DEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEE
   ITEMB (5)    DEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEE
   ITEMB (6)    DEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEE
   ITEMB (7)    DEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEE
   ITEMB (8)    DEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEEDEEE
                                                ZK-1422A-GE
   ```

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.

   There are 10 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.
```

Additional References

- SEARCH statement in Chapter 6
- Section 5.2.3: Additional Alignment Rules for Record Allocation
- Section 6.5.1.1: Comparison of Numeric Operands section in Chapter 6
- Section 6.5.1.2: Comparison of Nonnumeric Operands section in Chapter 6

PAGE

PAGE — The PAGE clause defines the length of a logical page and the vertical subdivisions within which report groups are presented.

General Format

```
PAGE [ LIMIT IS / LIMITS ARE ] page-size [ LINE / LINES ]
     [ HEADING heading-line ]
     [ FIRST DETAIL first-detail-line ]
     [ LAST DETAIL last-detail-line ]
     [ FOOTING footing-line ]
```

`page-size`

is a 1- to 3-digit integer. It defines the number of lines available on a logical page.

`heading-line`

is a 1- to 3-digit integer. It defines the first line number for a REPORT HEADING or PAGE HEADING report group on the logical page.

`first-detail-line`

is a 1- to 3-digit integer. It defines the first line number for a CONTROL HEADING, DETAIL, and CONTROL FOOTING report group on the logical page.

`last-detail-line`

is a 1- to 3-digit integer. It defines the last line number for a CONTROL HEADING or DETAIL report group on the logical page.

`footing-line`

is a 1- to 3-digit integer. It defines the last line number for a CONTROL FOOTING report group and the first line number for the PAGE FOOTING report group on the logical page.

Syntax Rules

1. The HEADING, FIRST DETAIL, LAST DETAIL, and FOOTING phrases may be written in any order.
2. *page-size* must not exceed three significant digits and must be greater than or equal to *footing-line*.

3. *heading-line* must be greater than or equal to one.
4. *first-detail-line* must be greater than or equal to *heading-line*.
5. *last-detail-line* must be greater than or equal to *first-detail-line*.
6. *footing-line* must be greater than or equal to *last-detail-line*.
7. The rules in Table 5.6 summarize the rules presented in Appendix D: *Report Writer Presentation Rules and Tables*. They indicate the vertical subdivision of the page in which each type of report group may appear when the PAGE clause is specified.

   a. To present a REPORT HEADING report group on a page by itself (NEXT GROUP NEXT PAGE), define the REPORT HEADING clause to be in the vertical subdivision of the page extending from *heading-line* to *page-size*, inclusive.

      To present a REPORT HEADING report group on a page with other report groups, define the REPORT HEADING clause to be in the vertical subdivision of the page extending from *heading-line* to *first-detail-line* minus one, inclusive.

   b. A PAGE HEADING clause must be defined in the vertical subdivision of the page extending from *heading-line* to *first-detail-line* minus one, inclusive.

   c. A CONTROL HEADING or DETAIL clause must be defined in the vertical subdivision of the page extending from *first-detail-line* to *last-detail-line*, inclusive.

   d. A CONTROL FOOTING clause must be defined in the vertical subdivision of the page extending from *first-detail-line* to *footing-line*, inclusive.

   e. A PAGE FOOTING clause must be defined in the vertical subdivision of the page extending from *footing-line* plus one to *page-size*, inclusive.

   f. To present a REPORT FOOTING report group on a page by itself, define the REPORT FOOTING clause in the vertical subdivision of the page extending from *heading-line* to *page-size*, inclusive.

      To present a REPORT FOOTING report group on a page with other report groups, define the REPORT FOOTING clause in the vertical subdivision of the page extending from *footing-line* plus one to *page-size*.

8. All report groups must be defined such that they can be presented on one logical page. The Report Writer Control System (RWCS) never splits a multiline report group across logical page boundaries.

General Rules

1. REPORT HEADING and PAGE HEADING report groups may not be presented on or beyond the *first-detail-line*.
2. PAGE FOOTING and REPORT FOOTING report groups must follow the *footing-line*.
3. If the PAGE clause is specified, the following implicit default values are assumed for any omitted phrases:

   a. If the HEADING phrase is omitted, *heading-line* equals one.

   b. If the FIRST DETAIL phrase is omitted, *first-detail-line* equals *heading-line*.

c. If the LAST DETAIL and FOOTING phrases are both omitted, *last-detail-line* and *footing-line* equal *page-size*.

d. If the FOOTING phrase is specified and the LAST DETAIL phrase is omitted, *last-detail-line* equals *footing-line*.

e. If the LAST DETAIL phrase is specified and the FOOTING phrase is omitted, *footing-line* equals *last-detail-line*.

4. If the PAGE clause is omitted, the report consists of a single page of infinite length with relative line numbering.

5. If a REPORT HEADER report group is to appear on a page with other report groups, the first line following the heading report groups (REPORT HEADER and PAGE HEADER) must be blank.

6. If a REPORT FOOTING report group is to appear on a page with other report groups, the first line preceding the footing report groups (PAGE FOOTING and REPORT FOOTING) must be blank.

Additional References

- General Rules (Report Description) (General Rule 4) of the LINE NUMBER (Alpha, I64) clause
- Appendix D: *Report Writer Presentation Rules and Tables*

Table 5.6 shows the page regions established by the PAGE clause.

Table 5.6. Page Regions Established by the PAGE Clause

	Region Boundaries		
Report Groups that Can Be Presented in a Region	**First Line Number of the Region**	**Last Line Number of the Region**	**Line Positioning for the First Report Group Within the Region**
Report Heading Described with NEXT GROUP NEXT PAGE Report Footing Described with LINE *line-num* NEXT PAGE	*heading-line*	*page-size*	LINE-NUMBER plus *heading-line* minus 1
Page Heading			
Report Heading Not Described with NEXT GROUP NEXT PAGE	*heading-line*	*first-detail-line* minus 1	LINE-NUMBER plus *heading-line* minus 1
Control Heading	*first-detail-line*	*last-detail-line*	If LINE-COUNTER is greater than or equal to *first-detail-line,* position on LINE-COUNTER plus 1
Detail			If LINE-COUNTER is less than *first-detail-line,* position on *first-detail-line*
Control Footing	*first-detail-line*	*footing-line*	Same as preceding
Page Footing	*footing line*	*page-size*	

	Region Boundaries		
Report Groups that Can Be Presented in a Region	**First Line Number of the Region**	**Last Line Number of the Region**	**Line Positioning for the First Report Group Within the Region**
Report Footing Not Described with LINE *line-num* NEXT PAGE	plus 1		LINE-NUMBER plus footing-line

PICTURE

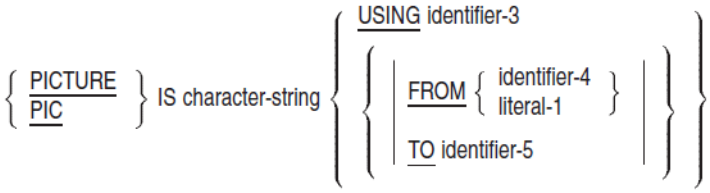
PICTURE — The PICTURE clause specifies the general characteristics and editing requirements of an elementary item, including an elementary screen item.

General Format

Format 1

{ PICTURE | PIC } IS character-string

Format 2 (Screen Section)



Syntax Rules (Both Formats)

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 from 1 to 255 symbols.
4. PIC is an abbreviation for PICTURE.
5. The asterisk (*), when used as a zero suppression symbol, and the BLANK WHEN ZERO clause cannot be used in the same entry.

Syntax Rule (Format 1)

6. The PICTURE clause is required for every elementary item except an item specified by the USAGE IS BINARY-CHAR, BINARY-SHORT, BINARY-LONG, BINARY-DOUBLE, COMP-1, COMP-2, FLOAT-SHORT, FLOAT-LONG, FLOAT-EXTENDED, POINTER, or INDEX clause and the subject of a RENAMES clause. Data description entries for these items cannot contain a PICTURE clause.

Syntax Rules (Format 2)

7. The PICTURE clause for a numeric screen item must either define a numeric edited item or must contain only “9”s and an optional “S”.

8. Each PICTURE clause in a screen description entry must contain a FROM or a TO phrase, or both, or a USING phrase.

9. In a screen description entry, if the PICTURE clause is specified, the VALUE clause cannot be specified.

10. *identifier-3*, *identifier-4*, and *identifier-5* must be defined in the File, Working-Storage, or Linkage Section.

General Rules (Both Formats)

1. The PICTURE clause categorizes a data item or screen item and determines what the item can contain. In the case of a PICTURE clause containing all Xs, the USAGE clause determines whether the item is alphanumeric or numeric. Table 5.7 shows the valid contents of both *character-string* and the item itself for each category. The general rules following this table supplement the information it contains.

Table 5.7. 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. May contain P's, one S, and one V. If USAGE IS COMP-5 or USAGE IS COMP-X, may contain all Xs.	One or more numeric characters.	S9(4)V99 9PPP SPP9
Alphanumeric	Must 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)
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 31 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 **.**

Note

COMP-1 and COMP-2 data items are numeric. However, their data description entries cannot have a PICTURE clause.

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. An item's size is determined by adding up all the symbols that represent a character position. For example, a numeric item with a PICTURE of 999V99 has a size of five characters. The symbol 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 (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, a period (.), CR, and DB.

5. The PICTURE clause symbols and their functions appear in Table 5.8.

Table 5.8. PICTURE Clause Symbols

Picture Clause Symbol	Function
A	Represents a character position that can contain only an alphabetic character. An alphabetic character belongs to the set of characters: A to Z, a to z, and the space.
	Can occur more than once.
	Counts toward the size of the item.
B	Represents a character position into which a space is inserted.
	Can occur more than once.
	Counts toward the size of the item.
N	For USAGE IS DISPLAY, represents a character position that can contain any 2-byte character from the national character set. This is available only if /NATIONALITY=JAPAN or `-nationality japan` is specified.
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 the size of the item. However, each P counts toward the maximum number of digit positions (31) 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 an item with P characters in *character-string*, the compiler treats each P position as if it contained the value zero. For example, an item with PICTURE 99PPP can have 100 unique values that range

Picture Clause Symbol	Function
	from 0 to 99,000 (0, 1000, 2000, …, 99,000). An item with PICTURE PP9 can have 10 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 numeric and its PICTURE *character-string* contains the symbol P ● A MOVE statement where the sending operand is numeric edited and its PICTURE *character-string* contains the symbol P, and the receiving operand is numeric or numeric edited ● A comparison operation where both operands are numeric
	In all other operations, the 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 the size of the item unless the data or screen 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 the size of the item.
	Cannot be used if an explicit decimal point (.) appears in the PICTURE.
X	For USAGE IS DISPLAY, represents a character position that can contain any character from the computer character set. For USAGE IS COMP-5 or USAGE IS COMP-X, represents a byte of computer storage.
	Can occur more than once.
	Counts toward the size of the item.
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 the size of the item.
	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 to 9.
	Can occur more than once.
	Counts toward the size of the item.
0	Represents a character position into which 0 is inserted.
	Can occur more than once.
	Counts toward the size of the item.

Picture Clause Symbol	Function
/	Represents a character position into which a slash (/) is inserted.
	Can occur more than once.
	Counts toward the size of the item.
,	Represents a character position into which a comma (,) is inserted. [DAG]
	Can occur more than once.
	Counts toward the size of the item.
.	Represents a character position into which a decimal point (.) is inserted. It also represents the decimal point for alignment purposes. [DAG]
	Can occur only once.
	Counts toward the size of the item.
	Cannot be used if V or P appears in *character-string*.
+ −	Represents the editing sign control symbols, the plus sign (+) and minus sign (-).
	Each can occur more than once.
	Each counts as one character toward the size of the item.
	character-string can contain either a plus sign (+) or minus (-), but not both. Also, the use of either character excludes the use of both CR and DB.
CR DB	Represents 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 the size of the item.
	character-string can contain either CR or DB, but not both. Also, the use of either excludes the use of the plus sign (+) and minus sign (-) as fixed insertion characters.
*	Represents a leading digit position that is replaced by an asterisk (*) when its value and the values of all digit positions to its left are zero.
	Can occur more than once.
	Counts toward the size of the item.
	Use of an asterisk (*) 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 ($) or the character specified in the CURRENCY SIGN clause of the SPECIAL-NAMES paragraph or (on OpenVMS) the character specified at DCL command level in the definition of the SYS$CURRENCY logical name.
	Can occur more than once.
	Counts as one character toward the size of the item.

[DAG] 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.

General Rules (Format 2)

6. The USING, FROM, and TO phrases have meaning only when the screen item's name or a screen name in its hierarchy, is specified in an ACCEPT or DISPLAY statement.

7. When data is to be transferred to the screen from one data item, possibly edited, and stored in a different data item, both the FROM and TO phrases must be used in the PICTURE clause of the screen item.

8. When data is to be transferred to the screen, possibly modified, and stored in the same data item (as when reading, modifying, and rewriting records of a file), the USING phrase must be used in the PICTURE clause of the screen item.

9. *identifier-3*, *identifier-4*, *identifier-5*, and *literal-1* need not be the same length as the screen item containing the PICTURE clause.

10. Transfers between *identifier-3*, *identifier-4*, *identifier-5*, and *literal-1*, on the one hand, and the screen item are made in accordance with the rules of the MOVE statement. (See the MOVE Statement in Chapter 6.)

11. When the FROM phrase is specified:

    a. On DISPLAY statement execution, data is transferred from *identifier-4* or *literal-1*, after being edited in accordance with *character-string*, and displayed on the screen. The display begins at the screen position defined either implicitly or explicitly by the LINE and COLUMN clauses and the starting screen coordinates specified in the DISPLAY statement.

    b. The FROM phrase has no meaning in the execution of an ACCEPT statement.

12. When the TO phrase is specified:

    a. At ACCEPT statement completion, the data entered into the field on the screen is transferred to *identifier-5*, after being edited in accordance with the picture string specified for *identifier-5*.

    b. The TO phrase has no meaning in the execution of a DISPLAY statement.

13. When the USING phrase, or the FROM and TO phrases are specified:

    a. On DISPLAY statement execution, data is transferred from *identifier-3*, *identifier-4*, or *literal-1* as described in rule 11a above.

    b. On ACCEPT statement execution, data is transferred from *identifier-3*, *identifier-4*, or *literal-1* as described in rule 11a above. At ACCEPT statement completion, the data entered into the screen item is transferred to *identifier-3* or *identifier-5* as described in rule 12a above.

Editing Rules

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

Editing Method	Variations in Each Method
Insertion	Simple insertion editing, special insertion editing, fixed insertion editing, or floating insertion editing

Editing Method	Variations in Each Method
Suppression and Replacement	Zero suppression and replacement with spaces, or zero suppression and replacement with asterisks

2. The types of editing that a program can perform on an item depend 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 space (Z) and asterisk (*) symbols cannot appear in the same PICTURE clause.

Simple Insertion Editing

1. A comma (,) space (B), zero (0), and slash (/) are symbols you can use in simple insertion editing. They indicate an item position to contain the character they represent. These symbols count toward the size of the item.

   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, a comma followed by a period (,.) 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.

Special Insertion Editing

2. The period (.) is the only symbol used in special insertion editing. It represents the 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 the size of the item.

   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, a comma followed by period (,.) are the last two characters of the data description entry.

Fixed Insertion Editing

3. The currency symbol (cs) 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 plus sign (+) and minus sign (-) must be either the leftmost or rightmost character position that counts toward the size of the item.

The currency symbol (cs) must be the leftmost character position that counts toward the size of the item; however, a plus sign (+) or minus sign (-) can precede it.

Fixed insertion editing causes the insertion symbol to occupy the same position in the edited item as in *character-string*. Table 5.9 shows that the results of using editing sign control symbols depend on the item's value.

Table 5.9. Using Sign Control Symbols in Fixed Insertion Editing

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

Floating Insertion Editing

4. The currency symbol (cs), the plus sign (+), and the minus sign (-) 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 appear in results only when the value of the 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 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 the following:

   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 the first nonzero digit in the item or the decimal point, whichever appears leftmost in the data. For example, an item whose PICTURE is $$$.99 and whose value is zero appears 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 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 item has a zero value, run-time results show neither a floating insertion character nor the decimal point. For example, a item whose PICTURE is $$$.$$ and whose value is zero appears as spaces.

If the floating insertion symbol is a plus sign (+) or minus sign (-), the actual character inserted depends on the value of the item. Table 5.10 shows the possible results of using editing sign control symbols in floating insertion editing.

Table 5.10. Using Sign Control Symbols in Floating Insertion Editing

	Result	
Editing Symbol in PICTURE Character-String	**Item Positive or Zero**	**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

Zero Suppression and Replacement Editing

5. One or more occurrences of the space symbol (Z) or the asterisk (*) define a floating suppression string, which can suppress leading zeros in numeric character positions. The space symbol (Z) causes spaces to replace the zeros; an asterisk (*) causes asterisks to replace the zeros.

   The suppression symbols are mutually exclusive. That is, *character-string* can contain either the space symbol (Z) or the asterisk (*), but not both.

   Each suppression symbol counts toward the size of the item.

   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 appear in results only when the value of the 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 the first nonzero digit in the data represented by the suppression string or at 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 a Z, all character positions in the edited item (including any editing characters) contain spaces.
- If the value is zero and the suppression symbol is an asterisk (*), all character positions in the edited item (including any insertion editing characters other than the decimal point) contain asterisks. The decimal point appears in the item.

6. The plus sign (+), minus sign (-), asterisk (*), space (Z), and currency symbol (cs) 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 asterisk (*)
   - At least two of the symbols plus sign (+), minus sign (-), or currency symbol (cs)
2. Figure 5.10 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 *Y* at an intersection means the symbols at the top of the column *(First Symbol)* can precede the symbols at the left of the row (*Second Symbol*).
   - Braces ({ }) enclose symbols that are mutually exclusive.
   - The currency symbol appears as cs.
   - 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 as follows:
     - Fixed insertion symbols (+ and –)
     - Floating symbols Z, asterisk (*), plus sign (+), minus sign (-), and currency symbol (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.

Figure 5.10. PICTURE Symbol Precedence Rules

Second Symbol \ First Symbol		Nonfloating Insertion Symbols								Floating Insertion Symbols						Other Symbols					
		B	O	/	,	.	{+} { } {-}	{+} { } {-}	{CR} { } {DB}	CS	{Z} { } {*}	{Z} { } {*}	{+} { } {-}	{+} { } {-}	CS	CS	9	A X	S	V	P	P
Non-floating Insertion Symbols	B	Y	Y	Y	Y	Y	Y			Y	Y	Y	Y	Y	Y	Y	Y	Y		Y		Y
	O	Y	Y	Y	Y	Y	Y			Y	Y	Y	Y	Y	Y	Y	Y	Y		Y		Y
	/	Y	Y	Y	Y	Y	Y			Y	Y	Y	Y	Y	Y	Y	Y	Y		Y		Y
	,	Y	Y	Y	Y	Y	Y			Y	Y	Y	Y	Y	Y	Y	Y			Y		Y
	.	Y	Y	Y	Y		Y			Y	Y		Y		Y		Y					
	{+ -}																					
	{+ -}	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 Insertion Symbols	{Z *}	Y	Y	Y	Y		Y			Y	Y											
	{Z *}	Y	Y	Y	Y	Y	Y			Y	Y	Y								Y		Y
	{+ -}	Y	Y	Y	Y					Y			Y									
	{+ -}	Y	Y	Y	Y	Y				Y			Y	Y						Y		Y
	CS	Y	Y	Y	Y		Y								Y							
	CS	Y	Y	Y	Y	Y	Y								Y	Y				Y		Y
Other Symbols	9	Y	Y	Y	Y	Y	Y			Y	Y		Y		Y		Y	Y	Y	Y		Y
	A X	Y	Y	Y													Y	Y				
	S																					
	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

VM-0595A-AI

Examples

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

Additional References

- SIGN clause
- MOVE statement in Chapter 6
- ACCEPT statement in Chapter 6
- DISPLAY statement in Chapter 6

RECORD

RECORD — The RECORD clause specifies the number of character positions in either a fixed- or variable-length record. If the number of character positions does not vary, the RECORD clause specifies 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. Its value must be greater than or equal to zero.

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 the following:
   - 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.
4. If the DEPENDING ON phrase is present and if the associated file connector is an external file connector, *depending-item* must have the external attribute and must specify the same data-name in all file description entries associated with the external file connector.

General Rules

Both Formats

1. The absence of a RECORD clause is the same as a Format 1 RECORD clause with no *shortest-rec* phrase and with *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 of the following:
   - 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.

3. If the associated file connector is an external file connector, all file description entries in the run unit associated with that file connector must define the same values for *shortest-rec* and *longest-rec*. If the RECORD clause is not specified, all record description entries associated with this file connector must be the same length.

Format 1

4. 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.

5. If there is a *shortest-rec* phrase, Format 1 specifies variable-length records, the same as Format 2 without the DEPENDING phrase.

6. 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

7. Format 2 specifies variable-length records.

8. 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.

9. 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.

10. 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.

11. 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.

12. The *depending-item* value is not changed by executions of:

    - DELETE and START statements

    - Unsuccessful READ and RETURN statements

13. 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

item (an item described by the OCCURS clause or one 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 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 item, the number of character positions is the sum of the character positions in the fixed part of the record and the table elements specified by the OCCURS clause *depending-item* when the output statement executes.

Additional References

- EXTERNAL clause
- SYNCHRONIZED clause
- USAGE clause
- Data Description

RECORD KEY

RECORD KEY — The RECORD KEY clause specifies the Prime Record Key access path to indexed file records.

General Format

RECORD KEY IS { rec-key | seg-key = {seg} . . . } [WITH DUPLICATES] [ASCENDING | DESCENDING]

rec-key

is the Record Key for the file. It is the data-name of a data item in a record description entry for the file. It can be qualified, but it cannot be a group item that contains a variable-occurrence data item. The data item must be described as one of the following:

- Alphanumeric item
- Alphabetic item
- Group item
- Unsigned numeric display item
- COMP-3 integer
- COMP integer

seg-key

is a segmented-key name that represents the concatenation of one or more (up to eight) occurrences of *seg*.

seg

is the data-name of a data item in a record description entry for the file. It can be qualified, but it cannot be a group item that contains a variable-occurrence data item. The data item must be described as one of the following:

- Alphanumeric item
- Alphabetic item
- Group item
- Unsigned numeric display item

Syntax Rule

The RECORD KEY clause is required for indexed files. It can be in either the file description entry or in the file's Environment Division SELECT clause. However, it cannot be in both the SELECT clause and the file description entry for the same file.

General Rules

1. *seg-key* is the concatenation of all specified key segments in the order specified.
2. *seg-key* can be referenced only in a READ (Format 3) or START statement.
3. The RECORD KEY clause specifies the Prime Record Key for a file.
4. The order of keys, whether ASCENDING or DESCENDING, must be the same as the order used when the file was created.
5. Each key can be specified as ASCENDING or DESCENDING (ASCENDING is the default). In an ASCENDING key, lower key values occur toward the beginning of the sorted file. In a DESCENDING key, higher key values occur toward the beginning of the sorted file.
6. The data description of *rec-key*, or the segments of *seg-key*, and their relative locations in the record, must be the same as those used when the file was created.
7. Only one record description entry for the file must describe *rec-key* or the segments of *seg-key*. The Prime Record Key has the same character positions in every record of the file.
8. If the associated file connector is an external file connector, all File Description entries in the run unit that are associated with that file connector must define the same data description entry for *rec-key* or the segments of *seg-key* with the same relative location within the record.
9. The DUPLICATES phrase specifies that two or more records in the file can have duplicate values in the same *rec-key* or the segments of *seg-key*. If there is no DUPLICATES phrase, two records cannot have the same value in corresponding Prime Record Key.

   On OpenVMS, if the program was compiled with the /CHECK=DUPLICATE_KEYS qualifier on the command line, and the duplicate key specification on a file's FD (in other words, specified in the WITH DUPLICATES phrase) does not match that of the actual file, a run-time diagnostic will be issued when an attempt is made to open the file with an OPEN statement.

   On UNIX systems, DUPLICATES must match the specification for DUPLICATES when the file is created, unless the relaxed key check option is used.

Additional Reference

ALTERNATE RECORD KEY

REDEFINES

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

General Format

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

`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 part of the REDEFINES clause. They are included in the general format only to clarify the relative position of the clause.

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 either an external record or anything 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, and is not an external record, 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 position using the data-name associated with any of those data description entries.

Example

This example shows the following:

- 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

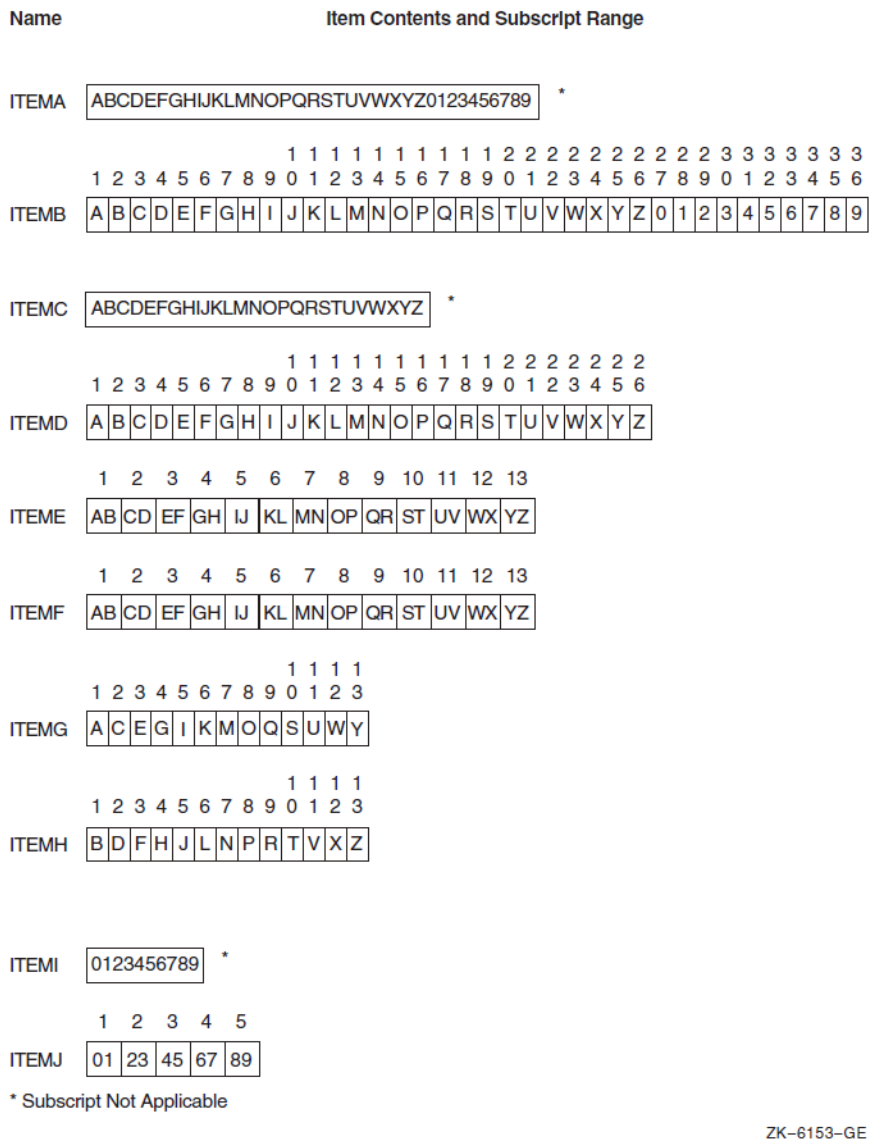
```
 1 IDENTIFICATION DIVISION.
 2 PROGRAM-ID. REDEFINES-TEST.
 3 DATA DIVISION.
 4 WORKING-STORAGE SECTION.
 5 01  ITEMA.
 6     03  FILLER PIC X(26) VALUE "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
 7     03  FILLER PIC X(10) VALUE "0123456789".
 8 01  REDEFINES ITEMA.
 9     03  ITEMB  OCCURS 36 TIMES PIC X.
10 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
```

ZK-1425A-GE

REDEFINES Clause



Additional References

- Data Description entry
- Section 5.2.3: Additional Alignment Rules for Record Allocation

RENAMES

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

General Format

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

`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 part of the RENAMES clause. They are in the general format only to clarify the relationship.

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 *rename-start*, if *rename-start* is an elementary item or the first elementary item in *rename-start*, or if *rename-start* is a group item
   - Ending with *rename-end*, if *rename-end* is an elementary item or the last elementary item in *rename-end*, or 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.

Example

In the following example, the box RESULTS displays the values given when using the RENAMES clause:

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

01  A.
    02  B   PIC XX   VALUE "1-".
    02  C   PIC XX   VALUE "2-".
    02  D   PIC XX   VALUE "3-".
    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

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

ZK-1424A-GE

Additional Reference

Data Description

REPORT

REPORT — The REPORT clause in a file description entry (FD) specifies the Report Description (RD) report names that comprise a report file.

General Format

{ REPORT IS | REPORTS ARE } {report-name} ...

Syntax Rules

1. Each *report-name* in the REPORT clause must be the subject of a Report Description entry (RD) in the Report Section of the same program. *report-name* can appear in only one REPORT clause.

2. *report-name*s can appear in any order.

3. The file-name in a file description entry for a Report File can be referenced only by the OPEN statement with the OUTPUT or EXTEND phrase or by the CLOSE statement.

General Rules

1. More than one *report-name* in a REPORT clause indicates that the file contains more than one report.

2. After executing an INITIATE statement and before executing a TERMINATE statement for the same report file, the report file is under the control of the Report Writer Control System (RWCS). While a report file is under control of the RWCS, no input/output statement may reference that report file.

3. If the associated file connector is an external file connector, every file description entry in the run unit associated with that file connector must describe it as a report file.

Technical Note

On OpenVMS, the DCL PRINT command inserts a form-feed character when a form is within four lines from the bottom. This positions the report to the top of the next logical page.

Report Writer files are written in print format. Line spacing positions the report to the top of the next logical page.

Therefore, use the PRINT/NOFEED command to suppress the insertion of form-feed characters and to print your Report Writer files correctly. For example:

```
$ PRINT/NOFEED full-file-name
```

Additional References

- FD (File Description)
- RD (Report Description)

REQUIRED (Alpha, I64)

REQUIRED (Alpha, I64) — The REQUIRED clause specifies that in the context of an ACCEPT statement, the user must enter at least one character in the input or update field.

General Format

REQUIRED

Syntax Rule

The REQUIRED clause cannot be specified in the description of a literal screen item.

General Rules

1. If the REQUIRED clause is specified at group level, it applies to each input and update screen item in that group.

2. The REQUIRED clause takes effect during the execution of any ACCEPT statement when the cursor enters the screen item. Until this clause is satisfied, the operator cannot leave the field and normal terminator keystrokes are rejected.

3. To satisfy this clause, alphanumeric screen items must contain at least one nonspace character, and numeric screen items must have a nonzero value.

4. For update fields, the REQUIRED clause can be satisfied by the contents of the identifier or literal referenced in the FROM or USING phrase of the PICTURE clause, as well as by operator-keyed data.

5. The REQUIRED clause is not effective if a function key is used to terminate the accept operation.
6. The specification of the FULL and REQUIRED clauses together requires that the field must always be filled entirely by the user.
7. The REQUIRED clause is ignored for an output field.

Additional Reference

ACCEPT statement in Chapter 6

REVERSE-VIDEO (Alpha, I64)

REVERSE-VIDEO (Alpha, I64) — The REVERSE-VIDEO clause specifies that the field is displayed with the default or specified foreground and background colors exchanged.

General Format

REVERSE-VIDEO

Syntax Rule

The REVERSE-VIDEO clause can be specified only for elementary screen items.

Additional Reference

- ACCEPT statement in Chapter 6
- DISPLAY statement in Chapter 6

SECURE (Alpha, I64)

SECURE (Alpha, I64) — The SECURE clause suppresses the display of input characters on the screen.

General Format

SECURE

Syntax Rule

The SECURE clause can only be specified for an input screen item.

General Rules

1. If the SECURE clause is specified at group level, it applies to each input screen item in that group.
2. When the SECURE clause is used, characters introduced for the input field do not appear on the screen, yet the cursor moves as usual.

Additional Reference

ACCEPT statement in Chapter 6

SIGN

SIGN — The SIGN clause specifies the operational sign's position and type of representation. For screen description entries, the SIGN clause specifies the position of the sign character in the field. The sign character always occupies a separate position in the field, regardless of whether or not you specify SEPARATE.

General Format

Format 1 (Data Description and Screen Description Entries)

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

Format 2 (Report Group Description Entries)

[SIGN IS] { LEADING | TRAILING } SEPARATE CHARACTER

Syntax Rules

Format 1

1. The SIGN clause can be used only in a numeric data description entry or screen description entry whose PICTURE contains the S symbol, or 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 description entry must contain the SIGN IS SEPARATE clause.

General Rules

Both Formats

1. The SIGN clause specifies the operational sign's position and type of representation. It applies to a numeric data description entry or screen description entry or to each numeric data description entry or screen description entry subordinate to a group.

2. The SIGN clause applies only to numeric data description entries or screen 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 or screen description entry to which no optional SIGN clause applies, but whose PICTURE contains an S symbol, has an operational sign.

- The numeric data description entry is equivalent to an entry that contains the SIGN IS TRAILING clause without the SEPARATE CHARACTER phrase.
- The screen description entry is equivalent to an entry that contains the SIGN IS TRAILING with the SEPARATE CHARACTER phrase.

6. If you specify the SEPARATE CHARACTER phrase (or it is implied):
   - The operational sign is the leading (or trailing) character 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 or screen item size. That is, it represents a character position.
   - The operational sign for positive is the plus sign (+).
   - The operational sign for negative is the minus sign (-).
7. 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.

Format 1 (Data Description)

8. 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 the size of the item. 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 5.11 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 5.11. Positive and Negative Signs for All Numeric Digits

Digit Values	Positive Sign	Negative Sign
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

SOURCE

SOURCE — The SOURCE clause identifies a data item to be sent to an associated printable item defined within a Report Group Description entry.

General Format

SOURCE IS source-id

source-id

names an elementary item in the Data Division.

Syntax Rules

1. If *source-id* is a Report Section item it must be either:
   - A PAGE-COUNTER
   - A LINE-COUNTER
   - A sum counter that is part of the report within which the SOURCE clause appears
2. The Report Writer Control System (RWCS) moves the contents of *source-id* to the printable item. *source-id* definitions must conform to the rules for sending items in the MOVE statement.

General Rule

The RWCS executes implicit MOVE statements specified by the SOURCE clauses when it formats the print lines (just before it presents them).

Additional References

- COLUMN NUMBER clause (printable item)
- TYPE clause
- MOVE statement in Chapter 6

SUM

SUM — The SUM clause establishes a Report Writer sum counter and names the data items to be summed.

General Format

```
{ SUM { sum-name } ... [ UPON { detail-report-group-name } ... ] } ...
    [ RESET ON { control-foot-name } ]
                { FINAL          }
```

`sum-name`

names a numeric data item with an optional sign in the Subschema, File, Working-Storage, or Linkage Sections, or another sum counter in the Report Section.

`detail-report-group-name`

names a DETAIL report group.

`control-foot-name`

must reference a *control-name* in the report's CONTROL clause.

Syntax Rules

1. A SUM clause can appear only in the description of a CONTROL FOOTING report group.

2. If there is no UPON phrase, any *sum-name* in the SUM clause that is itself a sum counter must be defined either in the same report group that contains this SUM clause or in a report group at a lower level in the control hierarchy of this report.

   If there is an UPON phrase, *sum-name* must not reference a sum counter.

3. If the associated report file connector is an external file connector and if *sum-name* references a numeric data item in the Subschema, File, Working-Storage, or Linkage Sections, then *sum-name* must reference the same external data item in all programs in the run unit.

4. *detail-report-group-name* must be a *control-name* in a CONTROL clause and must be the name of a DETAIL report group described in the same report as the CONTROL FOOTING report group in which the SUM clause appears.

5. *detail-report-group-name* may be qualified by a report-name.

6. *control-foot-name* must not be at a lower control level than the associated control level for the report group in which the RESET phrase appears.

   If FINAL appears in the RESET phrase, FINAL must also appear in the CONTROL clause for this report.

7. The highest permissible qualifier for *sum-name* is the report-name.

General Rules

1. The SUM clause establishes a sum counter. At run time, the Report Writer Control System (RWCS) adds the value in each *sum-name* to the sum counter. This addition is consistent with the rules for arithmetic statements.

2. The UPON phrase provides for selective subtotalling. Subtotalling occurs each time the RWCS processes the DETAIL report group referenced by *detail-report-group-name*.

3. If there is a RESET phrase, the RWCS will set the sum counter to zero when the RWCS is processing the designated level of control hierarchy. If there is no RESET phrase, the RWCS will set the sum counter in the CONTROL FOOTING report group to zero when the RWCS processes that report group.

The RWCS initially sets sum counters to zero during the execution of the INITIATE statement for the report containing the sum counter.

4. The size of the sum counter is equal to the number of receiving character positions defined in the PICTURE clause that accompanies the SUM clause in the description of the elementary item.

5. Only one sum counter exists for an elementary report entry, regardless of the number of SUM clauses specified in the elementary report entry.

6. If the elementary report entry for a printable item contains a SUM clause, the sum counter serves as a source data item. On a control break, the RWCS moves the data from the sum counter to the printable item for presentation according to the rules of the MOVE statement.

7. If a data-name appears as the subject of an elementary report entry that contains a SUM clause, the data-name is the name of the sum counter; the data-name is not the name of a printable item that the entry may also define.

8. Procedure Division statements can alter the contents of sum counters.

9. During the execution of GENERATE and TERMINATE statements, the RWCS adds the values in *sum-name* to a sum counter.

10. The RWCS adds each individual *sum-name* into the sum counter when it processes the CONTROL FOOTING report group defining the sum counter.

Technical Notes

- The three categories of sum counter accumulation are as follows:

  - Subtotalling
  - Crossfooting
  - Rolling forward

  Subtotalling occurs only during execution of GENERATE statements and after any control break processing but before processing of the DETAIL report group. Crossfooting and rolling forward occur during the processing of CONTROL FOOTING report groups.

- Subtotalling accumulates numeric data fields (*sum-names*) into a sum counter. *sum-name* must not reference a sum counter when subtotalling. If the SUM clause contains the UPON phrase, *sum-names* are subtotalled when a GENERATE statement executes for a DETAIL report group. If there is no UPON phrase, *sum-names* are subtotalled when any GENERATE *data-name* statement is executed for the report in which the SUM clause appears.

- Crossfooting accumulates sum counters (*sum-name*) from the same CONTROL FOOTING report group into another sum counter. It is a horizontal sum of sums.

  Crossfooting occurs when a control break takes place and when the CONTROL FOOTING report group is processed.

  Crossfooting is performed according to the sequence in which sum counters are defined within the CONTROL FOOTING report group. That is, all crossfooting into the first sum counter defined in the CONTROL FOOTING report group is completed, and then all crossfooting into the second sum

counter defined in the CONTROL FOOTING report group is completed. This procedure repeats until all crossfooting operations are completed.

When one of the *sum-names* is the sum counter defined by the Data Description entry in which that sum clause appears, the initial value of that sum counter is used in the summing operation.

- Rolling forward accumulates sum counters (*sum-name*) defined in lower level CONTROL FOOTING report groups into another sum counter. It is a vertical sum of sums. A sum counter in a lower level CONTROL FOOTING report group is rolled forward when a control break occurs and at the time the lower level CONTROL FOOTING report group is processed.
- If two or more *sum-names* specify the same sum counter, then the sum counter is added as many times as the sum counter is referenced in the SUM clause. It is permissible for two or more of the *sum-names* to specify the same DETAIL report group. When a GENERATE data-name statement for such a DETAIL report group is given, the incrementing occurs repeatedly, as many times as the *sum-name* appears in the UPON phrase.

Additional References

- GENERATE statement in Chapter 6
- TYPE clause
- Section 6.6.1: Arithmetic Operations
- Section 6.6.7: Overlapping Operands and Incompatible Data

SYNCHRONIZED

SYNCHRONIZED — 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

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

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. 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 items:

   - To which the elementary item belongs

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

  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.

3. The size of a SYNCHRONIZED data item equals the number of character positions between its natural boundaries. Therefore, the LEFT and RIGHT phrases have the same effect; they are equivalent to each other, and to the SYNCHRONIZED clause with neither the LEFT nor RIGHT phrases.
4. The SYNCHRONIZED clause does not change the size or operational sign position of the data item it specifies.
5. Each occurrence of the data item is synchronized if the clause applies to a data item whose data description entry also has an OCCURS clause, or to a data item subordinate to another data item whose data description entry has an OCCURS clause.

Technical Notes

- The SYNCHRONIZED clause does not affect the alignment of DISPLAY data items.
- The SYNCHRONIZED clause explicitly aligns COMP, COMP-1, COMP-2, POINTER, and INDEX data items on boundaries that are related to the size of the item.

  One word COMP items are aligned on 2-byte boundaries, longword items on 4-byte boundaries, and quadword items on 8-byte boundaries. All boundaries are relative to the beginning of the record containing the data item.

- The following table shows the alignment for each data type that the SYNCHRONIZED clause affects:

Data Type	**Boundary**
COMP (1 to 4 digits)	2-byte
COMP (5 to 9 digits)	4-byte
COMP (10 to 18 digits)	8-byte
COMP (19 to 31 digits)	16-byte
COMP-1	4-byte
COMP-2	8-byte
INDEX	4-byte
POINTER	4-byte (OpenVMS)
POINTER	8-byte (UNIX)

Additional Reference

Section 5.2.3: Additional Alignment Rules for Record Allocation

TYPE

TYPE — The TYPE clause identifies the report group type and indicates when the Report Writer Control System (RWCS) is to process it.

General Format

```
TYPE IS { { REPORT HEADING } }
        { { RH             } }
        { { PAGE HEADING   } }
        { { PH             } }
        { { CONTROL HEADING } { control-head-name } }
        { { CH             } { FINAL              } }
        { { DETAIL         } }
        { { DE             } }
        { { CONTROL FOOTING } { control-foot-name } }
        { { CF             } { FINAL              } }
        { { PAGE FOOTING   } }
        { { PF             } }
        { { REPORT FOOTING } }
        { { RF             } }
```

`control-head-name`

names a control-name in the CONTROL clause.

`control-foot-name`

names a control-name in the CONTROL clause.

Syntax Rules

1. RH is an abbreviation for REPORT HEADING.

   PH is an abbreviation for PAGE HEADING.

   CH is an abbreviation for CONTROL HEADING.

   DE is an abbreviation for DETAIL.

   CF is an abbreviation for CONTROL FOOTING.

   PF is an abbreviation for PAGE FOOTING.

   RF is an abbreviation for REPORT FOOTING.

2. These report groups may appear no more than once in the description of a report:

   - REPORT HEADING
   - PAGE HEADING
   - CONTROL HEADING FINAL
   - CONTROL FOOTING FINAL
   - PAGE FOOTING
   - REPORT FOOTING

3. The TYPE DETAIL report group may appear more than once in the description of a report.

4. If the TYPE clause specifies a CONTROL HEADING or CONTROL FOOTING report group, the *control-head-name*, *control-foot-name*, or FINAL entries must be specified in the CONTROL

clause of the corresponding Report Description entry. For each control-name or FINAL phrase in the CONTROL clause of a Report Description entry, you can specify one CONTROL HEADING report group and one CONTROL FOOTING report group. However, the RWCS does not require either a CONTROL HEADING report group or a CONTROL FOOTING report group for each control-name or FINAL phrase in the CONTROL clause of a Report Description.

5. PAGE HEADING and PAGE FOOTING report groups may appear only if the corresponding Report Description entry specifies a PAGE clause.

6. In CONTROL FOOTING, PAGE HEADING, PAGE FOOTING, and REPORT FOOTING report groups, SOURCE clauses and USE statements must not reference any of the following:

   - Formats 2 and 3 Report Group Description data items containing a control data item
   - Data items subordinate to a control data item
   - A redefinition or renaming of any part of a control data item

7. In PAGE HEADING and PAGE FOOTING report groups, SOURCE clauses and USE statements must not reference either *control-head-name* or *control-foot-name*.

8. When the Procedure Division specifies a GENERATE *report-name* statement, the corresponding Report Description entry must define no more than one DETAIL report group. If there are no GENERATE *group-data-name* statements in the Procedure Division, the RWCS does not require a DETAIL report group. If there are multiple TYPE DETAIL report groups in the report, the GENERATE *group-data-name* statement must be used.

General Rules

1. The Report Writer Control System (RWCS) processes DETAIL report groups as a direct result of the GENERATE statement. If a report group specified in the GENERATE statement is not a TYPE DETAIL report group, a *summary report* is produced. If a report group specified in the GENERATE statement is a TYPE DETAIL report group, a *detailed report* is produced.

2. The RWCS executes the following procedures (a to f) when it processes a DETAIL report group in response to a GENERATE statement.

   When the description of a report includes exactly one DETAIL report group, the detail-related processing that the RWCS executes in response to a GENERATE report-name statement is described in procedures a to d. The RWCS performs these procedures as though a GENERATE group-data-name statement were being executed.

   When the description of a report includes no DETAIL report groups, the detail-related processing that the RWCS executes in response to a GENERATE report-name statement is described in procedures a and b. These procedures are performed as though the description of the report included exactly one DETAIL report group, and a GENERATE detail-report-group statement were being executed.

   a. The RWCS performs any control break processing.

   b. The RWCS performs any subtotalling that has been designated for the DETAIL report group.

   c. If there is a USE BEFORE REPORTING procedure referring to the data-name of the report group, the RWCS executes the USE procedure.

d. If a SUPPRESS statement has been executed, or if the report group is not printable, no further processing is done for the report group.

e. If the RWCS processes a DETAIL report group as a consequence of the GENERATE report-name statement, no further processing is done for the report group.

f. If neither procedure d nor procedure e applies, the RWCS formats the print lines and presents the DETAIL report group.

3. To detect and trigger control breaks for a specific report, the RWCS:

   a. Establishes the initial values of control data items as the prior values when the INITIATE statement executes.

   b. Compares the prior values to the current values of control data items when a GENERATE statement executes. If the current values do not compare to the prior values, a control break occurs. If a control break occurs, the current values are saved as prior values and steps c, d, and e are performed.

   c. Presents the CONTROL FOOTING and CONTROL HEADING report groups associated with the control break. The CONTROL FOOTING report groups presented are at a less major level than the level at which the control break occurred. The CONTROL HEADING report groups presented are in the order of major level to break level.

   d. Processes any PAGE HEADING and PAGE FOOTING report groups when it must start a new page to present a CONTROL HEADING, DETAIL, or CONTROL FOOTING.

   e. Repeats steps b, c, and d until the last control break is processed.

4. The prior values (refer to General Rule 3) may be referenced by the program:

   - During the control break processing of a CONTROL FOOTING report group. Any references to control data items in a USE procedure or SOURCE clause associated with that CONTROL FOOTING report group are supplied with prior values.

   - When a TERMINATE statement executes. The RWCS makes the prior values available to the SOURCE clause or the USE procedure references in CONTROL FOOTING report groups as though the control break had been detected in the highest control data-name.

   - At the time the RWCS processes the report group. All other data item references within report groups and their USE procedures access the current values contained within the data items.

5. The RWCS presents the REPORT HEADING report group only once for each report, as the first report group of that report. It is processed when the first GENERATE statement is executed.

6. The RWCS presents the PAGE HEADING report group as the first report group on each page of the report, except for the following conditions:

   - A page containing only a REPORT HEADING report group.

   - A page containing only a REPORT FOOTING report group.

   - A page containing a REPORT HEADING report group that is not the only report group on the page. In this case, the PAGE HEADING report group is the second report group on the page.

7. The RWCS processes the CONTROL HEADING report group at the end of a control break for a specific *control-head-name*.

   The CONTROL HEADING FINAL report group is presented only once for each report, as the first body group (CONTROL HEADING, DETAIL, and CONTROL FOOTING) of that report. Other CONTROL HEADING report groups are presented when the RWCS detects a control break on the *control-head-name* during the execution of GENERATE statements. Control break processing for any CONTROL HEADING report group occurs with the highest control level of the break and includes all lower levels.

8. The RWCS presents CONTROL FOOTING report group at the beginning of a control break for a specific *control-foot-name*.

   The CONTROL FOOTING FINAL report group is presented only once for each report, as the last body group (CONTROL HEADING, DETAIL, and CONTROL FOOTING) of that report. If, during the execution of a GENERATE statement, the RWCS detects a control break, control break processing for any CONTROL FOOTING report group occurs with the highest control level of the break and includes all lower levels. Upon execution of the TERMINATE statement, the RWCS processes all CONTROL FOOTING report groups if the GENERATE statement has executed at least once.

9. The RWCS processes the PAGE FOOTING report group as the last report group on each page of the report, except for the following conditions:

   - A page containing only a REPORT HEADING report group.
   - A page containing only a REPORT FOOTING report group.
   - A page containing a REPORT FOOTING report group that is not to be the only report group on the page. In this case, the PAGE FOOTING report group is the second to the last report group on the page.

10. The RWCS processes the REPORT FOOTING report group, if defined, only once per report and as the last report group of that report. During the execution of a TERMINATE statement, the RWCS processes the corresponding REPORT FOOTING report group if at least one GENERATE statement is executed for the report.

11. The RWCS checks for these three conditions before it processes a REPORT HEADING, PAGE HEADING, CONTROL HEADING, PAGE FOOTING, or a REPORT FOOTING report group:

    - If there is a USE BEFORE REPORTING procedure referencing the data-name of the report group, the USE procedure executes.
    - If a SUPPRESS statement has been executed, or if the report group is not printable, there is no further processing for the report group.
    - If a SUPPRESS statement has not been executed and the report group is printable, the RWCS formats the print lines and presents the report group according to the presentation rules for that type of report group.

12. The RWCS executes the following procedures when it processes a CONTROL FOOTING report group.

    Control breaks occur during the processing of a GENERATE statement. The GENERATE rules specify that the RWCS produces the CONTROL FOOTING report groups beginning at the minor

level, and proceeding upwards, through and including the highest control level. Although no CONTROL FOOTING report group has been defined for a given control data-name, the RWCS will still have to execute procedure 12f if a RESET phrase within the report description specifies that control data-name.

a. Sum counters are crossfooted. All sum counters defined in this report group that are operands of SUM clauses in the same report group are added to their sum counters.

b. Sum counters are rolled forward. All sum counters defined in the report group that are operands of SUM clauses in higher level CONTROL FOOTING report groups are added to the higher level sum counters.

c. If there is a USE BEFORE REPORTING *group-data-name* declarative procedure, the RWCS executes the USE procedure.

d. If a SUPPRESS statement has been executed, or if the report group is not printable, the RWCS executes procedure 12f.

e. If a suppress statement has not been executed and the report group is printable, the RWCS formats the print lines and presents the report group according to the presentation rules for CONTROL FOOTING report groups.

f. The RWCS resets those sum counters that are to be reset when the RWCS processes this level in the control hierarchy.

Additional References

- CONTROL clause
- Data-Name
- LINE NUMBER (Alpha, I64) clause (General Rule 4)
- SUM clause
- TERMINATE statement in Chapter 6
- Appendix D: *Report Writer Presentation Rules and Tables*

UNDERLINE

UNDERLINE — The UNDERLINE clause specifies that each character of the field is underlined when it is displayed on the screen.

General Format

UNDERLINE

Syntax Rule

The UNDERLINE clause may be specified only for elementary screen items.

USAGE

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

General Format

```
[ USAGE IS ] { BINARY
               BINARY-CHAR (Alpha, I64) { SIGNED | UNSIGNED }
               BINARY-SHORT (Alpha, I64) { SIGNED | UNSIGNED }
               BINARY-LONG (Alpha, I64) { SIGNED | UNSIGNED }
               BINARY-DOUBLE (Alpha, I64) { SIGNED | UNSIGNED }
               COMPUTATIONAL
               COMP
               COMPUTATIONAL-1
               COMP-1
               COMPUTATIONAL-2
               COMP-2
               COMPUTATIONAL-3
               COMP-3
               COMPUTATIONAL-5 (Alpha, I64)
               COMP-5 (Alpha, I64)
               COMPUTATIONAL-X (Alpha, I64)
               COMP-X (Alpha, I64)
               DISPLAY
               FLOAT-SHORT (Alpha, I64)
               FLOAT-LONG (Alpha, I64)
               FLOAT-EXTENDED (Alpha, I64)
               INDEX
               PACKED-DECIMAL
               POINTER
               POINTER-64 (Alpha, I64) }
```

Syntax Rules

1. BINARY is a synonym for COMPUTATIONAL and COMP.

   On Alpha and I64 systems, except for restrictions on the PICTURE clause, COMPUTATIONAL-5 and COMPUTATIONAL-X are synonyms for COMPUTATIONAL and COMP.

2. COMP is an abbreviation for COMPUTATIONAL.

3. COMP-1 is an abbreviation for COMPUTATIONAL-1.

4. COMP-2 is an abbreviation for COMPUTATIONAL-2.

5. COMP-3 is an abbreviation for COMPUTATIONAL-3.

6. PACKED-DECIMAL is a synonym for COMPUTATIONAL-3 and COMP-3.

7. On Alpha and I64 systems, COMP-5 is an abbreviation for COMPUTATIONAL-5.

8. On Alpha and I64 systems, COMP-X is an abbreviation for COMPUTATIONAL-X.

9. On Alpha and I64 systems, FLOAT-SHORT is a synonym for COMPUTATIONAL-1.

10. On Alpha and I64 systems, FLOAT-LONG and FLOAT-EXTENDED are synonyms for COMPUTATIONAL-2.

11. You can use the USAGE clause in any data description entry with a level-number other than 66 or 88.

12. 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.

13. The PICTURE character-string of a COMP or COMP-3 item can contain only the following symbols:

    - 9
    - S
    - V
    - P

14. On Alpha and I64 systems, the PICTURE character-string of a COMP-5 or COMP-X item can contain only the following symbols:

    - 9
    - S
    - X (but not in combination with 9 or S)

15. 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

16. A report description entry or a screen description entry can only specify USAGE IS DISPLAY.

17. 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

18. 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.

19. The data description entry of a BINARY-CHAR, BINARY-SHORT, BINARY-LONG, BINARY-DOUBLE, COMP-1, COMP-2, POINTER, or POINTER-64 item cannot have a PICTURE clause. However, they are numeric and signed.

20. The subject of a data description entry containing the USAGE IS POINTER clause must not include any of the following clauses:

    - BLANK WHEN ZERO
    - JUSTIFIED
    - PICTURE

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 for more information.

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 BINARY-CHAR, BINARY-SHORT, BINARY-LONG, BINARY-DOUBLE, COMP, COMP-1, COMP-2, COMP-3, COMP-5, COMP-X, FLOAT-SHORT, FLOAT-LONG, or FLOAT-EXTENDED item can represent a value used in computations. The PICTURE clauses for COMP and COMP-3 items must be numeric. The PICTURE clauses for COMP-5 and COMP-X items may be numeric or X.

4. A POINTER data item can represent an address value used in computations. The compiler internally treats this item as a binary integer. References to a POINTER item are allowed in the same context as references to a COMP integer.

5. If the data description entry for a group item specifies BINARY-CHAR, BINARY-SHORT, BINARY-LONG, BINARY-DOUBLE, COMP, COMP-1, COMP-2, COMP-3, COMP-5, COMP-X, FLOAT-SHORT, FLOAT-LONG, FLOAT-EXTENDED, POINTER, or POINTER-64 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.

6. The USAGE IS DISPLAY clause specifies that the data item is in Standard Data Format.

7. If no USAGE clause applies to an elementary item, its usage is DISPLAY.

8. 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.

9. 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.

10. 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. Conversion will occur when the CONVERSION option is specified on ACCEPT or DISPLAY.

11. The USAGE IS POINTER clause can be used only in a File, Working-Storage, or Linkage Section data description entry.

12. On OpenVMS Alpha and I64 systems, the USAGE IS POINTER-64 clause is provided for limited use in interfacing with applications in languages requiring a 64-bit pointer. See Technical Notes.

Technical Notes

1. The way a data item is represented in the Data Division of a COBOL program determines whether it will be stored as an integer, floating-point, packed decimal, display numeric, or character string (text) data type. Tables 5.12 and 5.13 show the following:

   - COBOL data description entries and their corresponding data types
   - The allocated storage in bytes for each entry; allocated storage is the same on UNIX and OpenVMS Alpha and I64 (except for POINTER)

   Table 5.12 gives the corresponding data types for unscaled data items, and Table 5.13 gives the data types for scaled data items.

   For example, a data item described as PIC S9(4) USAGE IS DISPLAY SIGN IS TRAILING is stored in 4 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 5.12. Unscaled Data Items, Allocated Storage, and Corresponding Data Types

PICTURE Clause	USAGE Clause	SIGN Clause	Allocated Storage in Bytes	Standard Data Type
PIC S9(n) [n <= 31 (Alpha, I64)]	DISPLAY		n	Right (trailing) overpunch
PIC S9(n) [n <= 31 (Alpha, I64)]	DISPLAY	TRAILING	n	Right (trailing) overpunch
PIC S9(n) [n <= 31 (Alpha, I64)]	DISPLAY	LEADING	n	Left (leading) overpunch
PIC S9(n) [n <= 31 (Alpha, I64)]	DISPLAY	TRAILING SEPARATE	n+1	Right (trailing) separate
PIC S9(n) [n <= 31 (Alpha, I64)]	DISPLAY	LEADING SEPARATE	n+1	Left (leading) separate
PIC 9(n) [n <= 31 (Alpha, I64)]	DISPLAY		n	Unsigned numeric
PIC 9(n) [n <= 4]	COMP COMP-5 (Alpha, I64) COMP-X (Alpha, I64)		2	Word integer[1]

PICTURE Clause	USAGE Clause	SIGN Clause	Allocated Storage in Bytes	Standard Data Type
PIC 9(n) [5 <= n <= 9]	COMP COMP-5 (Alpha, I64) COMP-X (Alpha, I64)		4	Longword integer[1]
PIC 9(n) [10 <= n <= 18]	COMP COMP-5 (Alpha, I64) COMP-X (Alpha, I64)		8	Quadword integer[1]
PIC 9(n) [19 <= n <= 31] (Alpha, I64)	COMP COMP-5 COMP-X		16	Octaword integer[1] (Alpha, I64)
PIC S9(n) [n <= 4]	COMP COMP-5 (Alpha, I64) COMP-X (Alpha, I64)		2	Word integer
PIC S9(n) [5 <= n <= 9]	COMP COMP-5 (Alpha, I64) COMP-X (Alpha, I64)		4	Longword integer
PIC S9(n) [10 <= n <= 18]	COMP COMP-5 (Alpha, I64) COMP-X (Alpha, I64)		8	Quadword integer
PIC S9(n) [19 <= n <= 31] (Alpha, I64)	COMP COMP-5 COMP-X		16	Octaword integer (Alpha, I64)
PIC X(n) [n <= 2] (Alpha, I64)	COMP-5 COMP-X		2	Word integer[1] (Alpha, I64)
PIC X(n) [3 <= n <= 4] (Alpha, I64)	COMP-5 COMP-X		4	Longword integer[1] (Alpha, I64)
PIC X(n) [5 <= n <= 8] (Alpha, I64)	COMP-5 COMP-X		8	Quadword integer[1] (Alpha, I64)
Not applicable	INDEX		4	Longword integer
Not applicable	POINTER		4	Longword integer (OpenVMS)

PICTURE Clause	USAGE Clause	SIGN Clause	Allocated Storage in Bytes	Standard Data Type
	POINTER		8	Quadword integer (UNIX)
	POINTER-64 (Alpha, I64)		8	Quadword integer (OpenVMS Alpha, I64)
Not applicable	BINARY-CHAR UNSIGNED (Alpha, I64)		2	Word integer[1] (Alpha, I64)
Not applicable	BINARY-SHORT UNSIGNED (Alpha, I64)		2	Word integer[1] (Alpha, I64)
Not applicable	BINARY-LONG UNSIGNED (Alpha, I64)		4	Longword integer[1] (Alpha, I64)
Not applicable	BINARY-DOUBLE UNSIGNED (Alpha, I64)		8	Quadword integer[1] (Alpha, I64)
Not applicable	BINARY-CHAR SIGNED (Alpha, I64) BINARY-CHAR (Alpha, I64)		2	Word integer (Alpha, I64)
Not applicable	BINARY-SHORT SIGNED (Alpha, I64) BINARY-SHORT (Alpha, I64)		2	Word integer (Alpha, I64)
Not applicable	BINARY-LONG SIGNED (Alpha, I64) BINARY-LONG (Alpha, I64)		4	Longword integer (Alpha, I64)
Not applicable	BINARY-DOUBLE SIGNED (Alpha, I64) BINARY-DOUBLE (Alpha, I64)		8	Quadword integer (Alpha, I64)
Not applicable	COMP-1		4	F_floating S_format[2] (Alpha, I64)
Not applicable	COMP-2		8	D_floating G_floating (Alpha, I64) T_format[2] (Alpha, I64)

PICTURE Clause	USAGE Clause	SIGN Clause	Allocated Storage in Bytes	Standard Data Type
PIC 9(n) [n <= 31 (Alpha, I64)]	COMP-3		(n+1)/2 rounded up	Packed decimal[1]
PIC S9(n) [n <= 31 (Alpha, I64)]	COMP-3		(n+1)/2 rounded up	Packed decimal
PIC X(n) [n <= 268,435,455]	DISPLAY		n	ASCII text
PIC A(n) [n <= 268,435,455]	DISPLAY		n	ASCII text

[1]The generated code treats this data type as a positive value in all contexts except when it is a receiving-field operand. In this case, the compiler stores the absolute value of the source data item.

[2]On OpenVMS Alpha and I64 systems, the data type depends on the /FLOAT qualifier. On UNIX systems, it is always S format for COMP-1, and always T format for COMP-2. Refer to the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/] Appendix B, describing compatibility with VSI COBOL for information on the /FLOAT=IEEE qualifier.

Table 5.13. Scaled Data Items, Allocated Storage, and Data Types

PICTURE Clause	USAGE Clause	SIGN Clause	Allocated Storage in Bytes	Standard Data Type
PIC S9(n)V9(s) [(n+s) <= 31 (Alpha, I64)	DISPLAY		n+s	Right (trailing) overpunch
PIC S9(n)V9(s) [(n+s) <= 31 (Alpha, I64)]	DISPLAY	TRAILING	n+s	Right (trailing) overpunch
PIC S9(n)V9(s) [(n+s) <= 31 (Alpha, I64)]	DISPLAY	LEADING	n+s	Left (leading) overpunch
PIC S9(n)V9(s) [(n+s) <= 31 (Alpha, I64)	DISPLAY	TRAILING SEPARATE	n+s+1	Right (trailing) separate
PIC S9(n)V9(s) [(n+s) <= 31 (Alpha, I64)	DISPLAY	LEADING SEPARATE	n+s+1	Left (leading) separate
PIC 9(n)V9(s) [(n+s) <= 31 (Alpha, I64)	DISPLAY		n+s	Unsigned numeric
PIC 9(n)V9(s) [(n+s) <= 4]	COMP		2	Word integer[1]
PIC 9(n)V9(s) [5 <= (n+s) <= 9]	COMP		4	Longword integer[1]
PIC 9(n)V9(s) [10 <= (n+s) <= 18]	COMP		8	Quadword integer[1]

PICTURE Clause	USAGE Clause	SIGN Clause	Allocated Storage in Bytes	Standard Data Type
PIC 9(n)V9(s) [19 <= (n+s) <= 31] (Alpha, I64)	COMP		16	Octaword integer[1] (Alpha, I64)
PIC S9(n)V9(s) [(n+s) <= 4]	COMP		2	Word integer
PIC S9(n)V9(s) [5 <= (n+s) <= 9]	COMP		4	Longword integer
PIC S9(n)V9(s) [10 <= (n+s) <= 18]	COMP		8	Quadword integer
PIC S9(n)V9(s) [19 <= (n+s) <= 31]	COMP		16	Octaword integer (Alpha, I64)
PIC 9(n)V9(s) [(n+s) <= 31 (Alpha, I64)]	COMP-3		(n+s+1)/2 rounded up	Packed decimal[1]
PIC S9(n)V9(s) [(n+s) <= 31 (Alpha, I64)]	COMP-3		(n+s+1)/2 rounded up	Packed decimal

[1]The generated code treats this data type as a positive 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.

2. The OpenVMS Alpha operating system (as of Version 7.0) and OpenVMS I64 can dynamically allocate data in 64-bit address space. VSI COBOL for OpenVMS does not support data in 64-bit address space, but limited use of it may be made with the USAGE IS POINTER-64 clause on OpenVMS Alpha and I64. You might need to describe a data item as USAGE IS POINTER-64 if your program interfaces with an application in another language that requires a 64-bit pointer. Then you would use the SET statement or a VALUE clause to assign the address of a static COBOL variable to the pointer variable. The pointer variable can be passed to a routine whose interface definition requires a 64-bit pointer.

   You can also use an appropriate system service or the Run-Time Library routine LIB$GET_VM_64 to allocate data and store the address in the pointer variable. Because COBOL does not support dynamic allocation, there is no way to dereference the pointer and access the allocated data. However, you can pass the pointer to other languages that require a 64-bit pointer in 64-bit address space.

Additional References

- PICTURE clause
- VALUE IS clause (Format 3) *Alpha Architecture Reference Manual*, available from Digital Press.

VALUE IS

VALUE IS — The VALUE IS clause defines the values associated with condition-names, the initial value of Working-Storage Section data items, the value of Report Section printable items, the compile-time initialization of variables to the address of data, external constants, and the constant values of literal screen items.

General Format

Format 1

VALUE IS lit

Format 2

88 condition-name

{ VALUE IS / VALUES ARE }

{ { EXTERNAL external-name / REFERENCE data-name / low-val } [{ THRU / THROUGH } { EXTERNAL external-name / REFERENCE data-name / high-val }] } ...

Format 3

VALUE IS { REFERENCE data-name | numeric-integer-lit }

Format 4

VALUE IS EXTERNAL external-name

lit

is a numeric or nonnumeric literal. In a screen description entry, it is a nonnumeric literal.

external-name

names a COBOL link-time bound constant. It must define a word or longword integer value. See Technical Notes for more information.

data-name

names a data item in the File or Working-Storage or Subschema Section. *data-name* may be qualified.

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.

numeric-integer-lit

is a positive numeric integer literal.

Syntax Rules

1. The words THRU and THROUGH are equivalent.
2. You must associate a signed numeric literal with either of the following:
   a. A data item that has a signed numeric PICTURE character-string
   b. A COMP-1 or COMP-2 data item
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.
5. The Format 1, 3, and 4 VALUE IS clause cannot be used in any entry that is part of the description or redefinition of an external data record.
6. The Format 3 VALUE IS clause is allowed only for an item containing the USAGE IS POINTER phrase.
7. The subject of the associated data description entry in a Format 4 VALUE IS clause must define a word or longword data item.
8. In a screen description entry, the VALUE clause can be specified only at the elementary level.

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.
   - 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 clause does not directly affect initialization. However, the BLANK WHEN ZERO clause can change the category of the data item. If the category of the data item changes, the rules that apply change accordingly.
   - The JUSTIFIED clause does not affect initialization.
2. In the File Section, the VALUE IS clause can apply only to *condition-name* entries. That is, you can use the clause only for level 88 data items. In the Linkage Section, VALUE IS produces a warning for the other 88 data items.
3. Format 2 applies only to *condition-name* entries.

4. If a VALUE IS clause is specified in a data description entry that contains an OCCURS clause with a DEPENDING ON phrase, every occurrence of the associated data item is set to the maximum value.

   A data item is associated with a variable occurrence data item in any of the following cases:

   - It is a group data item that contains a variable occurrence data item.
   - It is a variable occurrence data item.
   - It is subordinate to a variable occurrence data item.

   If a VALUE IS clause is associated with the data item referenced by a DEPENDING ON phrase, that value is considered to be placed in the data item after the variable occurrence data item is initialized.

5. If a VALUE IS clause is specified in a data description entry that contains an OCCURS clause, or in an entry that is subordinate to an OCCURS clause, every occurrence of the associated data item is assigned the specified value. (This applies to General Formats 1, 3, and 4.)

Condition-Name Rules for Format 2

6. 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.
7. The characteristics of a *condition-name* are implicitly the same as those of its conditional variable.
8. When using the EXTERNAL option, the associated conditional variable must be a word or longword COMP data item.
9. When using the REFERENCE option, the associated conditional variable must be POINTER usage.
10. If the THRU phrase is used, each *low-val*, *external-name*, and *data-name* must be less than the corresponding *high-val*, *external-name*, and *data-name*.

Rules for Other Data Description Entries

11. A Working-Storage Section VALUE IS clause takes effect only when the program enters its initial state.
12. The VALUE IS clause initializes the data item to the value of *lit*.
13. If a data item's data description entry does not have a VALUE IS clause, the initial contents of the data item are the following:

    - Zero, for numeric items
    - Undefined, for index data items, and data items whose descriptions include or are subordinate to an OCCURS clause
    - Spaces, for all other items

14. In the Report Section, if an elementary report entry contains a VALUE IS clause but does not contain a GROUP INDICATE clause, the printable item assumes the specified value each time the Report Writer Control System (RWCS) prints the Report Group. However, if the entry contains the GROUP INDICATE clause, the RWCS presents the specified value only when certain run-time conditions exist. See the description of the GROUP INDICATE clause for more information.

15. The VALUE IS clause cannot be used in a data description entry that has a REDEFINES clause or is subordinate to a data description entry with a REDEFINES clause.

16. 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.

17. The VALUE IS clause cannot be used in the data description entry for a group that contains subordinate items with any of the following clauses:

    - JUSTIFIED
    - SYNCHRONIZED
    - USAGE (other than USAGE IS DISPLAY)

18. The VALUE IS clause cannot be used in the report group description entry for a group that contains subordinate items with a JUSTIFIED clause.

19. The Format 3 VALUE IS clause results in the compile-time initialization of its data description entry to the address of *data-name* or to *numeric-integer-lit*. Use this clause to pass arguments to non-COBOL procedures requiring an address rather than a user-defined word.

20. In Format 4, *external-name* must be the name of an external symbol (a symbol in another program unit) that is known to the linker when the program is linked.

21. The Format 4 VALUE IS clause results in the linker storing the value of *external-name* at the storage location defined by the data description entry containing the VALUE IS EXTERNAL clause.

Technical Notes

- *external-name* is a COBOL word formed according to the rules for user-defined names. The compiler translates hyphens in the COBOL word to underscore characters.
- *external-name* names a constant whose value is unknown at compile time but known at link time.
- Although the VALUE IS clause is not valid in the LINKAGE SECTION, the compiler allows such a specification, as a VSI extension. The clause, when specified in the LINKAGE SECTION, has no effect on program execution, and is flagged with an informational VSI extension diagnostic.

Examples

1. The following is an example of initializing alphanumeric data items:

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

2. The following is an example of initializing numeric data items:

```
01  ITEMX  PIC S9999 VALUE IS -39.
```

```
01  ITEMZ  PIC 9     VALUE ZERO.
```

3. The following is an example of assigning condition-name values:

```
01  ITEMC  PIC 99.
       88  VAL1      VALUE IS 4.
       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. The VALUE IS EXTERNAL clause allows a COBOL program to equate a mnemonic system constant to a value representing a return status code rather than the numeric equivalent. The following are some examples of this clause:

   On OpenVMS

```
WORKING-STORAGE SECTION.
*
* System Services
*
01  BADHEADER         PIC S9(9) COMP
                      VALUE IS EXTERNAL SS$_BADFILHDR.
01  BADNAME           PIC S9(9) COMP
                      VALUE IS EXTERNAL SS$_BADFILENAME.
01  NORMAL            PIC S9(9) COMP
                      VALUE IS EXTERNAL SS$_NORMAL.
*
* Record Management Services
*
01  RMSDEV            PIC S9(9) COMP
                      VALUE IS EXTERNAL RMS$_DEV.
*
* Database
*
01  DBMDBBUSY         PIC S9(9) COMP
                      VALUE IS EXTERNAL DBM$_DBBUSY.
01  DBMEND            PIC S9(9) COMP
                      VALUE IS EXTERNAL DBM$_END.
*
* Run-Time Library
*
01  LIBINVARG         PIC S9(9) COMP
                      VALUE IS EXTERNAL LIB$_INVARG.
01  LIBINVSCRPOS      PIC S9(9) COMP
                      VALUE IS EXTERNAL LIB$_INVSCRPOS.

PROCEDURE DIVISION.

    OPEN...
    IF RMS-STS = BADHEADER PERFORM...
    IF RMS-STS = BADNAME   PERFORM 100-FIX-NAME.
```

5. The following example shows the VALUE IS REFERENCE clause:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  ITEM-LIST.
```

```
    02  ITEM-PROCESS-NAME.
        03  PIC S9(4) COMP VALUE 15.
        03  PIC S9(4) COMP VALUE EXTERNAL JPI$_PRCNAM.
        03  POINTER VALUE REFERENCE PROCESS-NAME.
        03  POINTER VALUE REFERENCE PROCESS-NAME-LENGTH.
    02  ITEM-USER-NAME.
        03  PIC S9(4) COMP VALUE 12.
        03  PIC S9(4) COMP VALUE EXTERNAL JPI$_USERNAME.
        03  POINTER VALUE REFERENCE USER-NAME.
        03  PIC S9(9) COMP VALUE 0.
    02  ITEM-CPU-TIME.
        03  PIC S9(4) COMP VALUE 4.
        03  PIC S9(4) COMP VALUE EXTERNAL JPI$_CPUTIM.
        03  POINTER VALUE REFERENCE CPU-TIME.
        03  PIC S9(9) COMP VALUE 0.
    02  ITEM-TURMINAL.
        03  PIC S9(4) COMP VALUE 7.
        03  PIC S9(4) COMP VALUE EXTERNAL JPI$_TERMINAL.
        03  POINTER VALUE REFERENCE TURMINAL.
        03  POINTER VALUE REFERENCE TURMINAL-LENGTH.
    02  TERMINATOR-ENTRY  PIC S9(9) COMP VALUE 0.

01  PROCESS-NAME          PIC X(15) VALUE SPACES.
01  PROCESS-NAME-LENGTH   PIC S9(4) COMP VALUE 0.
01  USER-NAME             PIC X(12) VALUE SPACES.
01  CPU-TIME              PIC S9(9) COMP VALUE 0.
01  TURMINAL              PIC X(7)  VALUE SPACES.
01  TURMINAL-LENGTH       PIC S9(4) COMP VALUE 0.
```

Additional References

- PROGRAM-ID paragraph in Chapter 3
- PICTURE clause
- USAGE clause
- Section 1.2.1.1 in Section 1.2.1
- Section 5.2.2: COBOL Standard Alignment Rules

VALUE OF ID

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

General Format

VALUE OF { ID | FILE-ID } IS { file-name | data-name }

file-name

is a nonnumeric literal. It contains the full or partial file specification.

data-name

is the data-name of an alphanumeric Working-Storage Section data item. It contains the full or partial file specification.

General Rules

1. Each file specification field in *file-name* augments the specification in the ASSIGN clause of the SELECT statement.

2. A file specification field in the VALUE OF ID clause overrides the corresponding field in the SELECT statement. If a file specification field is either in the SELECT statement or in the VALUE OF ID clause (but not in both), it becomes part of the file specification.

3. On UNIX systems, if you specify a VALUE OF ID clause with which you specified an OpenVMS logical, you must use an environment variable, as follows:

   ```
   VALUE OF ID "DISK1"
   ```

   Define the environment variable using one of the following:

   ```
   % setenv DISK1
   % setenv DISK1 /usr/data/
   % setenv DISK1 /usr/data/test1.dat
   ```

4. The number of bytes in the string making up *file-name* or *data-name* must not exceed 255.

Technical Notes

- *file-name* is a complete or partial file specification. The resultant file specification must adhere to the rules for file specifications as defined by the file system.

- If the associated file connector is an external file connector, all file description entries in the run unit that are associated with that file connector must define the same file specification. For a *data-name* it must be external and reference the same data item in all programs defining the file.

Additional References

- ASSIGN

- *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/], on exception condition handling

- On OpenVMS, *VSI OpenVMS Record Management Services Reference Manual* in the OpenVMS documentation set

Chapter 6. Procedure Division

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

6.1. 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 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, can define conditional action when you 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 phrases you use in the statement.

There are four types of COBOL statements:

- **Compiler-directing statements** specify an action taken by the compiler during compilation. See Section 6.1.1 for more information.
- **Imperative statements** specify an unconditional action taken by the object program at run time. See Section 6.1.2 for more information.
- **Conditional statements** specify a conditional action taken by the object program at run time. See Section 6.1.3 for more information.
- **Delimited-scope statements** specify their explicit scope terminator. See Section 6.1.4 for more information.

Table 6.1 shows the four 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 6.1. Types and Categories of COBOL Statements

Type	Category	Verb
Compiler-Directing	Compiler-Directing	COPY REPLACE USE
Legend: **(1) Without the optional [NOT] ON SIZE ERROR phrase** **(2) Without the optional [NOT] ON EXCEPTION or [NOT] ON OVERFLOW phrase** **(3) Without the optional [NOT] INVALID KEY phrase** **(4) Without the optional [NOT] AT END or [NOT] INVALID KEY phrase** **(5) Without the optional [NOT] ON OVERFLOW phrase** **(6) Without the optional [NOT] INVALID KEY or [NOT] END-OF-PAGE phrase**		

Type	Category	Verb
		RECORD
Conditional	Conditional	ACCEPT ([NOT] AT END or [NOT] ON EXCEPTION) ADD ([NOT] ON SIZE ERROR) CALL ([NOT] ON EXCEPTION or [NOT] ON OVERFLOW) COMPUTE ([NOT] ON SIZE ERROR) DELETE ([NOT] INVALID KEY) DISPLAY ([NOT] ON EXCEPTION) DIVIDE ([NOT] ON SIZE ERROR) EVALUATE IF MULTIPLY ([NOT] ON SIZE ERROR) READ ([NOT] AT END or [NOT] INVALID KEY) RETURN([NOT] AT END) REWRITE ([NOT] INVALID KEY) SEARCH(AT END) START ([NOT] INVALID KEY) STRING ([NOT] ON OVERFLOW) SUBTRACT ([NOT] ON SIZE ERROR) UNSTRING ([NOT] ON OVERFLOW) WRITE ([NOT] INVALID KEY or [NOT] END-OF-PAGE)

Legend:

(1) Without the optional [NOT] ON SIZE ERROR phrase
(2) Without the optional [NOT] ON EXCEPTION or [NOT] ON OVERFLOW phrase
(3) Without the optional [NOT] INVALID KEY phrase
(4) Without the optional [NOT] AT END or [NOT] INVALID KEY phrase
(5) Without the optional [NOT] ON OVERFLOW phrase
(6) Without the optional [NOT] INVALID KEY or [NOT] END-OF-PAGE phrase

Type	Category	Verb
Imperative	Arithmetic	ADD (1) COMPUTE (1) DIVIDE (1) INSPECT (TALLYING) MULTIPLY (1) SUBTRACT (1)
	Data-Movement	ACCEPT (DATE, DAY, DAY-OF-WEEK or TIME) INITIALIZE INSPECT (REPLACING or CONVERTING) MOVE SET (TO TRUE) STRING (5) UNSTRING (5)
	Ending	STOP
Imperative	Input-Output	ACCEPT (identifier or CONTROL KEY IN identifier) CLOSE DELETE (3) DISPLAY OPEN READ (4) REWRITE (3) SET (TO ON or TO OFF) START (3)

Legend:

(1) Without the optional [NOT] ON SIZE ERROR phrase
(2) Without the optional [NOT] ON EXCEPTION or [NOT] ON OVERFLOW phrase
(3) Without the optional [NOT] INVALID KEY phrase
(4) Without the optional [NOT] AT END or [NOT] INVALID KEY phrase
(5) Without the optional [NOT] ON OVERFLOW phrase
(6) Without the optional [NOT] INVALID KEY or [NOT] END-OF-PAGE phrase

Type	Category	Verb
		STOP (literal) UNLOCK WRITE (6)
	Inter-Program Communications	CALL (2) CANCEL
	Procedure-Branching	ALTER CALL CONTINUE EXIT GO TO PERFORM
	Table-Handling	SEARCH SET (TO, UP BY, or DOWN BY) SORT
	Ordering	MERGE RELEASE RETURN SORT
	Report Writing	GENERATE INITIATE SUPPRESS TERMINATE
Delimited-Scope	Delimited-Scope	ACCEPT (END-ACCEPT) ADD (END-ADD) CALL (END-CALL)

Legend:

(1) Without the optional [NOT] ON SIZE ERROR phrase
(2) Without the optional [NOT] ON EXCEPTION or [NOT] ON OVERFLOW phrase
(3) Without the optional [NOT] INVALID KEY phrase
(4) Without the optional [NOT] AT END or [NOT] INVALID KEY phrase
(5) Without the optional [NOT] ON OVERFLOW phrase
(6) Without the optional [NOT] INVALID KEY or [NOT] END-OF-PAGE phrase

Type	Category	Verb
		COMPUTE (END-COMPUTE) DELETE (END-DELETE) DIVIDE (END-DIVIDE) EVALUATE (END-EVALUATE) IF (END-IF) MULTIPLY (END-MULTIPLY) PERFORM (END-PERFORM) READ (END-READ) RETURN (END-RETURN) REWRITE (END-REWRITE) SEARCH (END-SEARCH) START (END-START) STRING (END-STRING) SUBTRACT (END-SUBTRACT) UNSTRING (END-UNSTRING) WRITE (END-WRITE)
Legend: **(1) Without the optional [NOT] ON SIZE ERROR phrase** **(2) Without the optional [NOT] ON EXCEPTION or [NOT] ON OVERFLOW phrase** **(3) Without the optional [NOT] INVALID KEY phrase** **(4) Without the optional [NOT] AT END or [NOT] INVALID KEY phrase** **(5) Without the optional [NOT] ON OVERFLOW phrase** **(6) Without the optional [NOT] INVALID KEY or [NOT] END-OF-PAGE phrase**		

Like statements, COBOL sentences also can be compiler-directing, imperative, or conditional. Sentence type depends upon the types of statements the sentence contains. Table 6.2 summarizes the contents of the three types of COBOL sentences. The remaining text in this section describes each type of statement and sentence in greater detail.

Table 6.2. Contents of COBOL Sentences

Type	Contents of Sentence
Imperative	One or more consecutive imperative statements ending with a period
Conditional	One or more conditional statements, optionally preceded by an imperative statement, terminated by the separator period
Compiler-Directing	Only one compiler-directing statement ending with a period

6.1.1. Compiler-Directing Statements and Sentences

A compiler-directing statement causes the compiler to take an action during compilation. The verbs COPY, REPLACE, RECORD, and USE define compiler-directing statements. A compiler-directing sentence can contain other statements but it must contain only one compiler-directing statement. The compiler-directing statement must be the last statement in the sentence and must be followed immediately by a period.

6.1.2. 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, an 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."     END-READ.
```

An imperative sentence contains only imperative statements and ends with a separator period.

6.1.3. Conditional Statements and Sentences

A conditional statement determines a condition's **truth value**. (A truth value is either a yes or no answer to the question, “Is the condition true?”.) The statement uses the truth value generated by the program to determine subsequent program action.

Conditional statements are as follows:

- An EVALUATE, IF, RETURN, or SEARCH statement
- An ACCEPT statement with the [NOT] AT END or [NOT] ON EXCEPTION phrase
- A DISPLAY statement with the [NOT] ON EXCEPTION phrase
- A READ statement with the [NOT] AT END or [NOT] INVALID KEY phrase
- A WRITE statement with the [NOT] INVALID KEY or [NOT] END-OF-PAGE phrase
- A DELETE, REWRITE, or START statement with the [NOT] INVALID KEY phrase
- An arithmetic statement (ADD, COMPUTE, DIVIDE, MULTIPLY, SUBTRACT) with the [NOT] ON SIZE ERROR phrase

- A STRING or UNSTRING statement with the [NOT] ON OVERFLOW phrase
- A CALL statement with the [NOT] ON EXCEPTION or [NOT] ON OVERFLOW phrase

A conditional sentence is a conditional statement that ends with a separator period. It can include an optional preceding 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.”

6.1.4. Scope of Statements

Scope terminators delimit the scope of some Procedure Division statements.

The scope of statements contained (nested) in other statements can also terminate implicitly. When statements are contained in other statements, a separator period (that terminates the sentence) terminates all nested statements as well.

In the following example, the separator period terminates the IF, MOVE, and PERFORM statements:

```
IF ITEMA = ITEMB
   MOVE ITEMC TO ITEMB
   PERFORM PROCA.
```

In the following example, the ELSE phrase of the IF statement terminates the scope of the READ and the first MOVE statements:

```
IF ITEMA = ITEMB
   READ FILEA
        AT END MOVE ITEMC TO ITEMB
ELSE
   MOVE ITEMD TO ITEME.
```

A **delimited-scope statement** is a special category of statement used in structured programming. A delimited-scope statement is any statement that includes its explicit scope terminator. See Section 6.3.4 for a list of explicit scope terminators.

A delimited-scope statement can be nested in another delimited-scope statement with the same verb. Then, each explicit scope terminator terminates the statement that begins with the closest unpaired preceding occurrence of the verb.

In the following example, the END-IF after the ADD statements (line 8) terminates the IF on line 5. The END-IF after the SUBTRACT (line 10) terminates the IF on line 3. The scope of the first IF statement (line 1) is terminated by the separator period on line 11.

```
1. IF ITEMA = ITEMB

2.     MULTIPLY ITEMH BY ITEMI

3.     IF ITEMI > 18

4.         MOVE ITEMC TO ITEMD
```

```
5.          IF ITEMD > ITEME
6.              ADD ITEME TO ITEMF
7.              ADD ITEMG TO ITEMH
8.          END-IF
9.          SUBTRACT 6 FROM ITEMH
10.     END-IF
11.     PERFORM PROCA.
```

6.2. Uniqueness of Reference

Every user-defined name in a COBOL program names a resource. (See the section on User-Defined Words in Section 1.2.1.) To use a resource, however, a statement in a COBOL program must contain a reference that uniquely identifies that resource. Qualification, reference modification, and subscripting or indexing allow unique and unambiguous references to that resource. Qualified procedure-names allow uniqueness of reference to procedures, and qualified condition-names allow uniqueness of reference to condition-names.

When you assign the same name in separate (contained) programs to two or more occurrences of a resource, certain conventions apply that limit the scope of names. **Name scoping** and **scope of names** are COBOL language terms that describe the methods for resolving references to user-defined words in a contained program environment. (See Section 6.2.6: Scope of Names.)

Some user-defined words can be made available to every program in the run unit. (See the EXTERNAL clause in Chapter 5.) These words are called **external** data. Other user-defined words can be made available to programs contained within the program that defines that resource. (See the GLOBAL clause in Chapter 5.) These words are called **global data**.

6.2.1. Qualification

A reference to a user-defined word is unique if one or more of the following conditions exists:

- No other name has the same spelling, including hyphenation.
- It is part of a REDEFINES clause. (The reference following the word REDEFINES is unique because of the placement of the REDEFINES clause.) See the Syntax Rules for the REDEFINES clause.
- Scoping rules make it unique. (See Section 6.2.6: Scope of Names.)

A nonunique name within a hierarchy of names can be used 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 those necessary to eliminate 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.

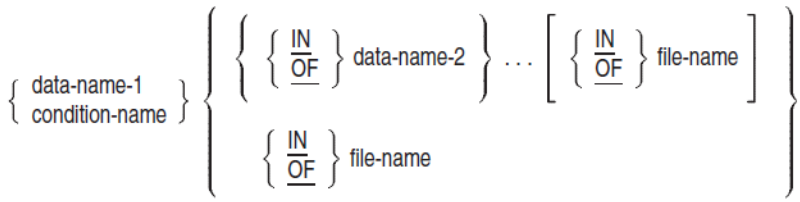
Regardless of the preceding, you cannot use the same data-name as:

- The name of an external record and as the name of any other external data item in any program contained within or containing the program describing the external data record
- The name of an item possessing the global attribute and as the name of any other data item in the program describing the global data item

When a program is contained within a program or contains another program, specific conventions apply. (See Section 6.2.6.)

The general formats for qualification are as follows:

Format 1



Format 2

paragraph-name { IN | OF } section-name

Format 3

text-name { IN | OF } library-name

Format 4

LINAGE-COUNTER { IN | OF } file-name

Format 5

{ PAGE-COUNTER | LINE-COUNTER } { IN | OF } report-name

Format 6

data-name-3 { { IN | OF } data-name-4 | [{ IN | OF } report-name] | { IN | OF } report-name }

Format 7

screen-name-1 {{ IN | OF } | screen-name-2 } ...

Format 8

{ RMS-STS | RMS-STV | RMS-FILENAME } { IN | OF } file-name

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 more than one set of qualifiers ensures uniqueness, any set can be used.

3. IN and OF are equivalent.

4. In Format 1, each qualifier must be either the name associated with a level indicator, the name of a group to which the item being qualified is subordinate, or the name of a condition variable with which the condition-name being qualified is associated. (See Section 6.2.5.) Qualifiers must be ordered from least- to most-inclusive levels in the hierarchy.

5. In Format 7, each qualifier must be 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.

6. In Format 1, *data-name-2* can be a record-name.

7. 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. You cannot reference a *paragraph-name* or *section-name* from any other program.

8. On OpenVMS, in Format 3, a COPY statement that accesses an OpenVMS Librarian *library-record* must qualify *text-name* with *library-name*.

9. In Format 3, on UNIX systems, the *library-name* for the COPY statement will direct COPY to access the *text-name* file from the *library-name* subdirectory.

   See Chapter 8 for information on the COPY statement.

10. If the program has more than one file description entry with a LINAGE clause, every reference to LINAGE-COUNTER must be qualified.

11. If the program has more than one report description entry, every Procedure Division reference to LINE-COUNTER must be qualified.

12. In the Report Section, an unqualified reference to LINE-COUNTER is qualified implicitly by the name of the report in whose report description entry the reference is made. Whenever the LINE-

COUNTER of a different report is referenced, LINE-COUNTER must be qualified explicitly by the report name associated with the different report.

13. If the program has more than one report description entry, every Procedure Division reference to PAGE-COUNTER must be qualified.

14. In the Report Section, an unqualified reference to PAGE-COUNTER is qualified implicitly by the name of the report in whose report description entry the reference is made. Whenever the PAGE-COUNTER of a different report is referenced, PAGE-COUNTER must be qualified explicitly by the report name associated with the different report.

15. On OpenVMS, if the program has more than one file description entry, every reference to RMS-STS, RMS-STV, and RMS-FILENAME must be qualified.

6.2.2. 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 operation; indexing is a special form of subscripting.

Unless otherwise specified by the rules for a statement, subscripts and indexes are evaluated once, at the beginning of a statement. If a statement contains rules describing the evaluation of subscripts, those rules also apply to the evaluation of indexes.

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 the OCCURS clause in Chapter 5.)

The general format for subscripting is as follows:

Format 1

{ data-name | condition-name } ({arithmetic-expression}...)

Format 2

argument ({ <u>ALL</u> | integer-1 | data-name [{ + | - }integer-2] | index-name [{ + | - }integer-3] }…)

All restrictions in the rules for subscripting also apply to indexing (See the following subsection describing the section called “Indexing”.) The following rules apply to subscripting:

1. A subscript can be represented by any arithmetic expression.

2. In Format 2, *argument* is an intrinsic function argument that is allowed to be repeated a variable number of times. Note that Format 1 also applies to intrinsic function arguments, but not with ALL subscripts. When ALL is specified as a subscript, the effect is as if each table element associated with that subscript position were specified. (For a list of the intrinsic functions that permit arguments with ALL subscripts, and for more information, see Chapter 7.) Also in Format 2, *data-name* is the data-name of a numeric integer elementary item.

3. Identifiers in subscript arithmetic expressions must refer to elementary numeric data items.

4. 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.

5. The highest valid subscript value is the maximum number of occurrences of the item specified in the OCCURS clause.

6. The subscript or set of subscripts that identifies the table element is delimited by a balanced pair of left and right parentheses.

7. Each table element reference must include subscripting. However, the reference cannot include subscripting when it is one of the following:

   - The subject of a SEARCH statement
   - In a REDEFINES clause
   - In the KEY IS phrase of an OCCURS clause

8. The subscript or set of subscripts follows the table element's *data-name*. The *data-name* is then called a subscripted *data-name* or an identifier.

9. The number of subscripts following a table element reference 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 the table element and one for the table element itself.

10. A *data-name* can have up to 48 subscripts.

11. Subscripts must appear in the order of successively less inclusive dimensions of the table.

12. An arithmetic expression in a subscript cannot begin with a left parenthesis if the preceding arithmetic expression ends with a *data-name*.

Note

Use the `check` compiler option with the `bounds` keyword for run-time upper- and lower-bound subscript range verification. The default action is not to check. For more information, refer to the COBOL online help file for your particular platform.

In the following examples, references to ITEME require two subscripts. The first subscript refers to the occurrence number of the most inclusive dimension, ITEMD (that contains ITEME).

Example 6.1. Subscripting Example

```
WORKING-STORAGE SECTION.
01  ITEMA PIC 99 COMP VALUE IS 3.
01  ITEMB PIC 99 COMP VALUE IS 5.
01  ITEMC VALUE IS "ABCDEFGHIJKLMNOPQRSTUVWX".
    03  ITEMD OCCURS 4 TIMES.
        05  ITEME OCCURS 6 TIMES PIC X.
```

IDENTIFIER	VALUE
ITEME (4,3)	U
ITEME (ITEMA,ITEMB)	Q

IDENTIFIER	VALUE
ITEME (ITEMA * 2 - 4, ITEMB - 2)	I
ITEME (ITEMA * ITEMB / 15, (ITEMA + ITEMB) / 4)	B

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.

The general format for indexing follows:

{ data-name | condition-name } ({ , index-name [{ + | - } literal-2] | , literal-1 }...)

All the restrictions in the rules for subscripting apply to indexing. (See the section called “Subscripting”.) The following rules apply only to indexing:

1. You must give *index-name* an initial value before using it. You can do this in:
   - A SET statement
   - A SEARCH statement with the ALL phrase
   - A PERFORM statement with the VARYING phrase

   Furthermore, only the statements in the previous list can change the value of *index-name*.
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

Use the `check` compiler option with the `bounds` keyword for run-time upper- and lower-bound index range verification. The default is not to check. For more information, refer to the COBOL online help file for your particular platform.

Example 6.2 is similar to Example 6.1 that illustrates subscripting. However, this example shows the use of index-names in references to the table, initializing indexes with the SET statement, and storing index-name values in index data items.

Example 6.2. Indexing 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).          ❶
DISPLAY ITEME (DX, EX).      ❷
DISPLAY ITEME (DX - 3, EX)   ❸
SET ITEMA TO DX.         SET ITEMB TO EX.
```

This example produces the following results:

❶ :STUVWX
❷ :S
❸ :A

6.2.3. Reference Modification

Reference modification defines a subset of a data item by specifying its leftmost character and length.

General Format

{ data-name | FUNCTION function-name [({argument} ...)] } (leftmost-character-position : [length])

data-name must refer to a data item whose usage is DISPLAY.

function-name must refer to an alphanumeric function.

The specifications for *leftmost-character-position* and *length* must be arithmetic expressions.

Each character of a data item has an ordinal number corresponding to its position. The leftmost position is number 1; successive positions to the right are numbered 2, 3, 4, and so on. If the *data-name's* data description entry has a SIGN IS SEPARATE clause, the sign position is assigned an ordinal number in the data item.

For a data item defined as numeric, numeric edited, alphanumeric, alphabetic, or alphanumeric edited, reference modification operates as if the data item were redefined as an alphanumeric data item the same size as that referred to by *data-name*.

Unless otherwise specified by the rules for a statement, reference modification is evaluated only once, at the beginning of a statement. Reference modification is evaluated immediately after subscripting or indexing evaluation. Rules that describe the evaluation of subscripting for the various statements also apply to the evaluation of reference modification.

The components of reference modification define the data item as follows:

- The evaluation of *leftmost-character-position* specifies the ordinal position of the data item's leftmost character. This position is relative to the leftmost character of the data item referred to by *data-name*. Evaluation of *leftmost-character-position* must result in an integer that is not less than 1, or greater than the number of characters in the data item referred to by *data-name*.
- The evaluation of *length* specifies the size of the unique data item. The evaluation must result in a positive integer. The sum of *leftmost-character-position* and *length* minus the value 1 must not exceed the number of characters in the data item referred to by *data-name*.
- If there is no *length*, the data item extends:
  - From and including the character identified by *leftmost-character-position* of the data item referred to by *data-name*

- To and including the rightmost character of the data item referred to by *data-name*

The resulting unique data item is treated as an elementary item without the JUSTIFIED clause. It has the same class and category as the data item referred to by *data-name*. However, the categories numeric, numeric edited, and alphanumeric edited are considered category alphanumeric.

Reference modification is valid anywhere an alphanumeric identifier is allowed unless specific rules for a general format prohibit it.

Note

Use the `check` compiler option with the `bounds` keyword for run-time upper- and lower-bound reference modification range verification. The default is not to check. For more information, refer to the COBOL online help file for your particular platform.

Examples

```
WORKING-STORAGE SECTION.
01  ITEMA PIC X(15) VALUE IS "ABCDEFGHIJKLMNO".
01  ITEMB PIC 99 VALUE IS 10.
```

IDENTIFIER	VALUE
ITEMA (2:3)	BCD
ITEMA (ITEMB:2)	JK
ITEMA (ITEMB / 2:ITEMB - 6)	EFGH
ITEMA (ITEMB:)	JKLMNO

6.2.4. Identifiers

In Procedure Division rules, the term *identifier* means the complete specification of a data item. The term refers to all words required to make your reference to the item unique.

To reference a data item that is a function, a function-identifier is used. For information on functions, see Chapter 7.

The general formats for identifiers are as follows:

Format 1

data-name [qualification] [subscripting] [reference modification]

Format 2

data-name [qualification] [indexing] [reference modification]

Format 3

FUNCTION function-name [({argument}…)] [reference modification]

For more information on the methods of uniquely specifying data items, see the following:

- Section 6.2.1: Qualification
- Section 6.2.2: Subscripts and Indexes
- Section 6.2.3: Reference Modification
- Section 6.2.6: Scope of Names

6.2.5. 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. Your *condition-name* also is unique when the scope of names conventions by themselves ensure this as described in Section 6.2.6: Scope of Names.

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 used for an identifier, except *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.

6.2.6. Scope of Names

A contained COBOL program can refer to a user-defined word in its containing program if the user-defined word has the global attribute. (See Section 1.2.1.1 in Section 1.2.1.) Some user-defined words always have the global attribute, some never have the attribute (that is, they are local), and some might or might not, depending on the use of the GLOBAL clause. The following rules explain how to use different kinds of user-defined words and what kinds of local and global name scoping to expect.

1. The following types of user-defined words are always local and can be referenced only by statements and entries in the program declaring them:
   - Paragraph-name
   - Section-name
2. These user-defined words are always local when you define them in the Report Section. Only those statements and entries in the program containing the entries can reference them.
   - Condition-name
   - Data-name
   - Record-name
3. The following user-defined word is always local when you define it in the Screen Section. Only those statements and entries in the program containing the entries can reference it.
   - Screen-name

4. Because you cannot specify a Configuration Section for a program contained within another program, the following types of user-defined words are always global when declared in the Configuration Section. You can reference them only by statements and entries either in the program that contains the Configuration Section or in any program contained within that program.
   - Alphabet-name
   - Condition-name (declared in the Special Names paragraph)
   - Mnemonic-name
   - Symbolic-character-name
   - Switch-name
   - Class-name
5. The following user-defined words are global if you specify the GLOBAL clause:
   - Condition-name (declared in the Data Division)
   - Data-name
   - File-name
   - Index-name
   - Record-name
   - Report-name
   - Segmented-key-name (if you specify the GLOBAL clause on the corresponding file-name)

Specific conventions for declarations and references apply to these types of user-defined words whenever the previous conditions do not apply.

Whenever duplicate names exist, a program always references the resource in its own program. If the resource is not in the referencing program, the following two conventions are used:

- Conventions for resolving program-name references
- Conventions for resolving other references

The next two sections describe these conventions.

6.2.6.1. Conventions for Resolving Program-Name References

The PROGRAM-ID paragraph of the Identification Division declares the *program-name*; a user-defined word to identify the program. Only the CALL and CANCEL statements and the END PROGRAM header can reference a *program-name*.

A run unit can contain multiple programs with duplicated *program-names*. However, when two programs have duplicate *program-names*, one of the two programs must directly or indirectly be contained within a separately compiled program that does not contain the program with the duplicated *program-name*.

The following rules regulate the scope of *program-name*:

1. Within a run unit, any separately compiled program can reference any other separately compiled program.

2. If a *program-name* does not have the COMMON attribute and it is contained directly within another program, the contained program can be referenced only by statements included in the directly containing program.

   For example, in the run unit consisting of the three separately compiled programs illustrated in Example 6.3, Example 6.4, and Example 6.5:

   - MAIN-PROGRAM (See ❶ in Example 6.3) directly contains program PROG-NAME-A ❸ and indirectly contains PROG-NAME-B ❺, PROG-NAME-C ❻, PROG-NAME-D ❼, and PROG-NAME-F ❾.

   - PROG-NAME-B (See ❶ in Example 6.4.)

   - PROG-NAME-E (See ❶ in Example 6.5.)

     The CALL “PROG-NAME-B” statement in PROG-NAME-A (See ❹ in Example 6.3.) references PROG-NAME-B ❺ in the same separately compiled program (MAIN-PROGRAM) because PROG-NAME-B ❺ is directly contained in PROG-NAME-A. All other CALL “PROG-NAME-B” statements (❷ and ❽ and ❿ in Example 6.3 and ❷ in Example 6.5) all reference PROG-NAME-B ❶ in Example 6.4, the second separately compiled program.

Example 6.3. Separately Compiled Program 1

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN-PROGRAM. ❶
.
.
.
     CALL "PROG-NAME-B".  ❷
.
.
.
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-A.  ❸
.
.
.
     CALL "PROG-NAME-B".  ❹
.
.
.
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-B.  ❺
.
.
.
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-C.  ❻
.
.
.
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-D.  ❼
.
```

```
.
.
     CALL "PROG-NAME-B".  ❽
.
.
.
END PROGRAM PROG-NAME-D.
END PROGRAM PROG-NAME-C.
END PROGRAM PROG-NAME-B.
.
.
.
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-F.  ❾
.
.
.
     CALL "PROG-NAME-B".  ❿
.
.
.
END PROGRAM PROG-NAME-F.
END PROGRAM PROG-NAME-A.
END PROGRAM MAIN-PROGRAM.
```

Example 6.4. Separately Compiled Program 2

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-B.  ❶
.
.
.
```

Example 6.5. Separately Compiled Program 3

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-E.  ❶
.
.
.
     CALL "PROG-NAME-B".  ❷
.
.
.
```

a. If a *program-name* has the COMMON attribute and it is contained directly within another program, the contained program can be referenced only by the following:

- Statements in the directly containing program
- Statements in any programs, directly or indirectly contained within the directly containing program, except statements in the program with the COMMON attribute and in any program it directly or indirectly contains

For example, in the run unit consisting of the three separately compiled programs illustrated in Example 6.6, Example 6.7, and Example 6.8:

- MAIN-PROGRAM (see ❶ in Example 6.6) directly contains PROG-NAME-A ❸, and indirectly contains PROG-NAME-B (IS COMMON) ❺, PROG-NAME-C ❻, PROG-NAME-D ❼, PROG-NAME-F ❾, and PROG-NAME-G ❿.
- PROG-NAME-B (See ❶ in Example 6.7.)
- PROG-NAME-E (See ❶ in Example 6.8.)

  The CALL “PROG-NAME-B” statement in PROG-NAME-A (See ❶ in Example 6.6) references PROG-NAME-B IS COMMON ❺ because it is directly contained in PROG-NAME-A. The CALL “PROG-NAME-B” statement in PROG-NAME-F (See ❿ in Example 6.6) references PROG-NAME-B IS COMMON ❺ because PROG-NAME-F is directly contained in PROG-NAME-A. The CALL “PROG-NAME-B” statement in PROG-NAME-G (See ⓬ in Example 6.6) references PROG-NAME-B IS COMMON ❺ because PROG-NAME-G is indirectly contained in PROG-NAME-A. The remaining CALL “PROG-NAME-B” statements (❷ and ❽ in MAIN-PROGRAM and ❷ in PROG-NAME-E) all reference the separately compiled program, PROG-NAME-B ❶.

Example 6.6. Separately Compiled Program 1

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN-PROGRAM. ❶
.
.
.
     CALL "PROG-NAME-B".  ❷
.
.
.
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-A.  ❸
.
.
.
     CALL "PROG-NAME-B".  ❹
.
.
.
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-B IS COMMON. ❺
.
.
.
IDENTIFICATION DIVISION.
```

```
PROGRAM-ID. PROG-NAME-C.  ❻
.
.
.
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-D.  ❼
.
.
.
     CALL "PROG-NAME-B".  ❽
.
.
.
END PROGRAM PROG-NAME-D.
END PROGRAM PROG-NAME-C.
END PROGRAM PROG-NAME-B.
.
.
.
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-F.  ❾
.
.
.
     CALL "PROG-NAME-B".  ❿
.
.
.
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-G.  ⓫
.
.
.
     CALL "PROG-NAME-B".  ⓬
.
.
.
END PROGRAM PROG-NAME-G.
END PROGRAM PROG-NAME-F.
END PROGRAM PROG-NAME-A.
END PROGRAM MAIN-PROGRAM.
```

Example 6.7. Separately Compiled Program 2

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-B.  ❶
.
.
.
```

Example 6.8. Separately Compiled Program 3

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-E.  ❶
.
.
.
     CALL "PROG-NAME-B".  ❷
```

.
.
.

6.2.6.2. Conventions for Resolving Other References

When a source program declares *condition-names*, *data-names*, *file-names*, *record-names*, *report-names*, and *segmented-key-names*, only the declaring source program can reference these names. The only exception is when names have the GLOBAL attribute and the program contains other programs.

For example, when a program such as PROG-NAME-A (See ❶ in Example 6.9) contains other programs (PROG-NAME-B ❿ and PROG-NAME-C ㉕), each program can define the same user-defined word. When such duplicated names are referenced, the rules for qualification of names (see Section 6.2.1) apply; and, if necessary, the following three hierarchical rules resolve any ambiguity:

1. References in a program to names defined in that program are resolved within the program. For example:
   - The following names: ❷, ❸, ❹, and ❺ are both defined and referenced within PROG-NAME-A.
   - The following names: ⓫, ⓬, ⓭, ⓮, and ⓯ are both defined and referenced within PROG-NAME-B.
   - The following names: ㉖ and ㉗ are both defined and referenced within PROG-NAME-C.
2. A program cannot reference any *condition-name*, *data-name*, *file-name*, *record-name*, or *report-name* defined in a program it contains. For example, statements in PROG-NAME-A (See ❻, ❼, ❽, and ❾) cannot reference items in either PROG-NAME-B or PROG-NAME-C. Statements in PROG-NAME-B (see ⓰ through ㉔) cannot reference items in PROG-NAME-C.
3. If a program contains another program, any GLOBAL names defined in the containing program can be referenced by the following:
   - Statements in a directly contained program, provided that the directly containing program does not declare the same user-defined word, in which case, rule 1 applies. For example, compare the Procedure Division statement MOVE EXAMPLE1 … ㉑ with MOVE EXAMPLE2 … ㉒ in the same contained program.
   - Statements in an indirectly contained program, provided that neither the indirectly containing program nor any program in between declare the same name as a GLOBAL name. For example, compare the Procedure Division statement MOVE EXAMPLE3 … ㉙ with MOVE EXAMPLE2 … ㉘ in the same contained program.

Example 6.9. Resolving References to Miscellaneous Names

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-A.      ❶
DATA DIVISION.
FILE SECTION.
FD    FILE-NAME …            ❷
01    RECORD-NAME …          ❸
FD    GLOBAL-FILE-NAME... IS GLOBAL...
 WORKING-STORAGE SECTION.
01    EXAMPLE1 ... IS GLOBAL...
01    EXAMPLE2 ... IS GLOBAL...
01    EXAMPLE3 ... IS GLOBAL...
```

```
     01    SWITCH-STATUS.          ❹
           88 ON   VALUE IS "1".
           88 OFF  VALUE IS "0".
     01    DATA-NAME …             ❺
     PROCEDURE DIVISION.
          MOVE DATA-NAME TO …     ❻
          IF SWITCH-STATUS IS ON … ❼
          MOVE RECORD-NAME TO ...❽
          OPEN INPUT FILE-NAME ...❾


     IDENTIFICATION DIVISION.
     PROGRAM-ID. PROG-NAME-B.       ❿
     DATA DIVISION.
     FILE SECTION.
     FD    FILE-NAME ...⓫
     01    RECORD-NAME ...⓬
     WORKING-STORAGE SECTION.
     01    SWITCH-STATUS.           ⓭
           88 ON   VALUE IS "1".
           88 OFF  VALUE IS "0".
     01    DATA-NAME ...⓮
     01    EXAMPLE2 ...⓯
     01    EXAMPLE3 ... IS GLOBAL...
     01    EXAMPLE4 ... IS GLOBAL...
     PROCEDURE DIVISION.
          MOVE DATA-NAME TO ...⓰
          IF SWITCH-STATUS IS ON ...⓱
          MOVE RECORD-NAME TO ...⓲
          OPEN INPUT FILE-NAME ...⓳
          OPEN OUTPUT GLOBAL-FILE-NAME.       ⓴
          MOVE EXAMPLE1 ...㉑
          MOVE EXAMPLE2 ...㉒
          MOVE EXAMPLE3 ...㉓
          MOVE EXAMPLE4 ...㉔


     IDENTIFICATION DIVISION.
     PROGRAM-ID. PROG-NAME-C.    ㉕
     WORKING-STORAGE SECTION.
     01    EXAMPLE2 ...㉖
     01    EXAMPLE4 ...㉗
     PROCEDURE DIVISION.
          OPEN OUTPUT GLOBAL-FILE-NAME.
          MOVE EXAMPLE1 ...
          MOVE EXAMPLE2 ...㉘
          MOVE EXAMPLE3 ...㉙
          MOVE EXAMPLE4 ...

END PROGRAM PROG-NAME-C.
END PROGRAM PROG-NAME-B.
END PROGRAM PROG-NAME-A.
```

If a data item possesses either or both the EXTERNAL or GLOBAL attributes and includes a table defining an *index-name*, that *index-name* also possesses either or both attributes.

If the file associated with a segmented key possesses either or both the EXTERNAL or GLOBAL attributes, that segmented key also possesses either or both attributes.

6.2.7. External and Internal Data

External data is associated with a run unit. Any program in the run unit describing the external data can reference that data. (See the EXTERNAL clause in Chapter 5.) There is only one representation of an external data object.

Internal data is associated with a specific program.

External and internal data can have global names. (See the GLOBAL clause in Chapter 5.)

6.3. Explicit and Implicit Specifications

The four types of explicit and implicit specifications follow:

- Procedure Division references
- Control transfers
- Attributes
- Scope terminators

6.3.1. Explicit and Implicit Procedure Division References

A source program can refer to data items explicitly or implicitly in Procedure Division statements.

An explicit reference occurs when the name of the item is in a Procedure Division statement or copied into the Procedure Division by a COPY statement.

An implicit reference occurs under the following conditions:

- When a Procedure Division statement refers to an item whose name does not appear in the statement.
- During PERFORM statement execution. The PERFORM statement's control mechanism can initialize, change, and evaluate an index or data item referred to in the VARYING, AFTER, and UNTIL phrases. These implicit references occur only if the PERFORM statement execution involves the data item.

6.3.2. Explicit and Implicit Control Transfers

The mechanism that controls program flow implicitly transfers control from one statement to another in the order in which the statements appear in the source program. The transfer occurs in this sequence unless an explicit control transfer overrides it, or there is no next executable statement.

A program can contain both explicit and implicit changes to the control transfer mechanism.

Implicit control transfer can also occur when normal program flow changes without executing a procedure-branching statement. For example:

- A paragraph can execute under the control of another COBOL statement (such as PERFORM, USE, SORT, and MERGE). If the paragraph is the last in the controlling statement's range, an implied control transfer occurs from the last statement in the paragraph to the controlling statement's control mechanism.

An implicit control transfer occurs between the control mechanism of a PERFORM statement that causes iterative execution and the first paragraph in its range. The transfer occurs for each iterative execution of the paragraph.

- When a SORT or MERGE statement executes, an implicit control transfer occurs to associated input or output procedures.
- When the execution of a statement causes the execution of a Declaratives Section, the control transfer is implicit. Another implicit control transfer occurs after execution of the Declaratives Section.
- If the Procedure Division does not have any Declaratives Sections, the program's first executable statement is the first executable statement in the Procedure Division. Otherwise, the program's first executable statement is the first executable statement after the declaratives part of the Procedure Division.

An explicit control transfer is a change to the implicit control transfer mechanism caused only by execution of either:

- A procedure-branching statement
- A conditional statement

The EXIT procedure-branching statement causes an explicit control transfer only when it has the PROGRAM phrase.

The Procedure Branching statement ALTER does not cause an explicit control transfer. However, it affects the explicit control transfer of the associated GO TO statement.

The term **next executable statement** refers to the next COBOL statement to which control transfers according to these rules and those associated with each language element.

There is no next executable statement when the program has no Procedure Division. This is also the case after:

- The last statement in a Declaratives Section, when the paragraph in which it appears is not executing under the control of another COBOL statement
- The last statement in a program, when the paragraph in which it appears is not executing under the control of another COBOL statement
- A STOP RUN or EXIT PROGRAM statement, when execution control transfers outside of the COBOL program containing the statement
- An END PROGRAM header

When there is no next executable statement and control does not transfer out of the program, program control flow is undefined. However, an EXIT PROGRAM statement implicitly executes when the program is under the control of a CALL statement.

6.3.3. Explicit and Implicit Attributes

An explicit attribute is an attribute the program explicitly specifies. If the program does not explicitly specify an attribute, the attribute assumes a default; it is then an implicit attribute.

For example, a program need not specify USAGE for a data item. If it does not, the data item's implicit usage is DISPLAY.

6.3.4. Explicit and Implicit Scope Terminators

Scope terminators delimit the scope of some Procedure Division statements as described in Section 6.1.4.

The following are explicit scope terminators:

END-ACCEPT	END-ADD	END-CALL
END-COMPUTE	END-DELETE	END-DIVIDE
END-EVALUATE	END-IF	END-MULTIPLY
END-PERFORM	END-READ	END-RETURN
END-REWRITE	END-SEARCH	END-START
END-STRING	END-SUBTRACT	END-UNSTRING
END-WRITE		

The following are implicit scope terminators:

- At the end of a sentence the separator period terminates the scope of all previously unterminated statements.
- In a statement containing another statement the next phrase of the containing statement after the end of the contained statement terminates the scope of all unterminated contained statements. Examples are ELSE and WHEN.

6.4. 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
- A figurative constant ZERO (ZEROS, ZEROES)
- Two or more of the above separated by arithmetic operators
- Two or more arithmetic expressions separated by arithmetic operators
- 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 of the following:

- Numeric elementary items
- Numeric literals on which arithmetic can be performed

Evaluation rules for arithmetic expressions depend on whether the mode of arithmetic in effect is native or standard.

6.4.1. Arithmetic Operators

Arithmetic expressions can use five binary and two unary **arithmetic operators**. A space must precede each operator and follow each binary operator.

The operators are as follows:

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

6.4.2. Formation and Evaluation of Arithmetic Expressions

The following rules apply regardless of the mode of arithmetic that is in effect.

Parentheses can be used to specify 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, the compiler 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 within a single set of parentheses.

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 eliminate ambiguities in logic when there are consecutive operations at the same hierarchical level, or change the normal hierarchical sequence of evaluation.

Consider the following expression:

```
(3 * ITEMA - 2) / ((4 + ITEMB) * -ITEMA - ITEMC ** 2)
```

The order of evaluation is as follows:

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)

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 in an arithmetic expression is a unary operator, a left parenthesis (() must immediately precede it when the arithmetic expression immediately follows an identifier or another arithmetic expression. For example:

```
CALL "OTHERPROG" USING ITEMA (-ITEMB) ITEMC.
```

The following rules apply to the evaluation of exponentiation:

1. If the value of an expression to be raised to a power is zero, the exponent value must be greater than zero. Otherwise, the size error condition exists. (See Section 6.6.4.)
2. If the evaluation yields both a positive and negative real number, the positive number is the result.
3. If the evaluation yields no real number, the size error condition exists.

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 6.6.1.)

6.4.3. Standard Arithmetic (Alpha, I64)

When a floating-point data item is an operand in an arithmetic expression or an arithmetic statement, the rules for evaluation are described in Section 6.4.4.1.

When standard arithmetic is in effect, the following rules apply:

1. Any operand of an arithmetic expression that is not already contained in a standard intermediate data item is converted into a standard intermediate data item.
2. The size error condition is raised if the value is too large or too small to be contained in a standard intermediate data item.

A standard intermediate data item is of the class numeric and the category numeric. It is the unique value zero or an abstract, signed, normalized decimal floating-point temporary data item.

A standard intermediate data item has the unique value of zero or a value whose magnitude is in the range 10**-100 through 10**99 - 10**67, that is, (.100 000 000 000 000 000 000 000 000 000 00E-99) through (.999 999 999 999 999 999 999 999 999 999 99E+99) inclusive, with a precision of 32 decimal digits.

When the value of a standard intermediate data item is not zero, the fraction contains no digits to the left of the decimal point and contains a digit other than zero to the immediate right of the decimal point.

A standard intermediate data item is rounded to 31 digits in the situations listed below.

1. When a standard intermediate data item is compared.
2. When a standard intermediate data argument is the argument of a function and there is no equivalent arithmetic expression defined for the rules of the function, unless otherwise specified in the rules for a function or unless situation 1, above, applies.
3. When a standard intermediate data item is being moved to a resultant-identifier for which the ROUNDED phrase has not been specified. Rounding of a standard intermediate data item may cause the size error condition to be raised.

When a standard intermediate data item is being moved to a resultant-identifier for which the ROUNDED phrase is specified, the number of digits to which rounding occurs is as specified in the ROUNDED phrase.

When arithmetic expressions using addition, subtraction, multiplication, division, exponentiation, unary plus, and unary minus are evaluated, the exact result is truncated to 32 significant digits, normalized, and stored in a standard intermediate data item.

6.4.4. Native Arithmetic (Alpha, I64)

When a floating-point data item is an operand in an arithmetic expression or an arithmetic statement, the rules for evaluation are those described in Section 6.4.4.1.

When native arithmetic is in effect, the following rules apply:

1. If the result of an arithmetic expression can be represented without loss of significance in 38 decimal digits or less, then decimal or computational operations are used to evaluate the expression.
2. When it is possible for an expression to produce more than 38 decimal digits, an intermediate data item is selected based on the MATH_INTERMEDIATE qualifier.

The compiler assumes that all possible digit positions of a variable are significant.

6.4.4.1. FLOAT Arithmetic (Alpha, I64)

A double-precision binary floating-point intermediate data item is selected when /MATH_INTERMEDIATE=FLOAT is specified. On OpenVMS Alpha and I64 this is a G_floating or T_floating data item; on UNIX, this is a T_floating data item. Refer to the *Alpha Architecture Reference Manual* for more information on floating-point data types and operations.

A G_floating data item has a sign bit, an 11-bit binary exponent, and a normalized 53-bit fraction with the redundant most significant fraction bit not represented. The magnitude of a G_floating data item is in the approximate range 0.56 * 10**-308 through 0.9 * 10**308. The precision of a G_floating data item is approximately one part in 2**52, typically 15 decimal digits.

A T_floating data item has a sign bit, an 11-bit binary exponent, and a 52-bit fraction. VSI COBOL for OpenVMS generates code that uses the finite, normalized, floating-point range capabilities of T_floating. The magnitude of a T_floating data item is in the approximate range 2.2 * 10**-308 through 1.8 * 10**308. The precision of a T_floating data item is approximately one part in 2**52, typically 15 decimal digits.

When the destination of an arithmetic statement is a floating-point data item, normal rounding takes place.

When an arithmetic expression references a floating-point operand, floating-point operations are used to evaluate the expression, and the result is represented in a floating-point intermediate data item. Floating-point operations use normal rounding; implicit conversions to integer are chopped. VSI COBOL for OpenVMS provides support for finite (normalized) floating-point values only.

When arithmetic expressions using addition, subtraction, multiplication, division, exponentiation, unary plus, and unary minus are evaluated, the exact result is truncated to 53 significant bits, normalized, and stored in a floating-point intermediate data item.

6.4.4.2. CIT3 Arithmetic (Alpha, I64)

A decimal floating-point intermediate data item is selected when the qualifier /MATH_INTERMEDIATE=CIT3 is specified.

A CIT3 intermediate data item has the unique value of zero or a value whose magnitude is in the range 10**-100 through 10**99 - 10**81, that is, (.100 000 000 000 000 000E-99) through (.999 999 999 999 999 999E+99) inclusive, with a precision of 18 decimal digits.

When a CIT3 intermediate data item is being moved to a resultant-identifier for which the ROUNDED phrase is specified, the number of digits to which rounding occurs is as specified in the ROUNDED phrase; when the ROUNDED phrase is not present, no rounding takes place.

When arithmetic expressions addition, subtraction, multiplication, division, exponentiation, unary plus, and unary minus are evaluated, the exact result is truncated to 18 significant digits, normalized, and stored in a CIT3 intermediate data item.

6.4.4.3. CIT4 Arithmetic (Alpha, I64)

A decimal floating-point intermediate data item is selected when /MATH_INTERMEDIATE=CIT4 is specified.

A CIT4 intermediate data item has the unique value of zero or a value whose magnitude is in the range 10**-100 through 10**99 - 10**67, that is, (.100 000 000 000 000 000 000 000 000 000 00E-99) through (.999 999 999 999 999 999 999 999 999 999 99E+99) [4] inclusive, with a precision of 32 decimal digits.

Rounding rules for CIT4 arithmetic are the same as those described in Section 6.4.3.

[4]The blanks are added for readability.

When arithmetic expressions using addition, subtraction, multiplication, division, exponentiation, unary plus, and unary minus are evaluated, the exact result is truncated to 32 significant digits, normalized, and stored in a CIT4 intermediate data item.

6.5. 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, EVALUATE, PERFORM UNTIL, PERFORM VARYING, and SEARCH statements use conditional expressions. Any statement that can contain another imperative statement can contain a conditional expression.

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, sign, and success/failure 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.

6.5.1. 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 an identifier, a literal, or 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 as follows:

```
Subject                  Relational Operator                    Object
                         IS [ NOT ] GREATER THAN
                         IS [ NOT ] >
                         IS [ NOT ] LESS THAN
                         IS [ NOT ] <
{ identifier-1           IS [ NOT ] EQUAL TO                    { identifier-2
  literal-1              IS [ NOT ] =                             literal-2
  arithmetic-expression-1 } IS GREATER THAN OR EQUAL TO          arithmetic-expression-2 }
                         IS >=
                         IS LESS THAN OR EQUAL TO
                         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.

The following relational operators are equivalent:

- IS NOT GREATER THAN is equivalent to IS LESS THAN OR EQUAL TO
- IS NOT LESS THAN is equivalent to IS GREATER THAN OR EQUAL TO

Table 6.3 specifies valid true conditions that correspond to each relational operator.

Table 6.3. 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 NOT = TO	Subject is either greater than or less than object.
IS GREATER THAN OR EQUAL TO IS >=	Subject is greater than or equal to object.
IS LESS THAN OR EQUAL TO IS <=	Subject is less than or equal to object.

The following two sections specify the rules that apply to numeric and nonnumeric comparisons in relation conditions.

6.5.1.1. 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.

6.5.1.2. 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 Chapter 4.)

If one of the operands is numeric, it must be either an integer literal or a data item described as an integer. The data item must be implicitly or explicitly described with USAGE DISPLAY. The treatment of the numeric data item is further affected by the following:

- If the nonnumeric operand is an elementary data item or a nonnumeric literal, the numeric data item is treated as though it were moved to an elementary alphanumeric data item of the same size. The content of this alphanumeric data item is then compared to the nonnumeric operand.
- If the nonnumeric operand is a group item, the numeric operand is treated as though it were moved to a group item of the same size. The content of this group item is then compared to the nonnumeric operand.
- 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 (unless the program's collating sequence dictates otherwise).

Comparisons of Index-Names or Index Data Items

A program can compare the following:

- 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

6.5.2. Class Condition

The **class condition** tests whether the contents of an operand are numeric or alphabetic. It also determines if an alphabetic operand contains only uppercase characters, only lowercase characters, or if an operand is in conformance with *class-name*. The general format is as follows:

```
                    { NUMERIC          }
                    { ALPHABETIC       }
identifier IS [ NOT ] { ALPHABETIC-LOWER }
                    { ALPHABETIC-UPPER }
                    { class-name       }
```

The identifier must reference a data item whose usage is explicitly or implicitly DISPLAY or COMP-3. If the identifier is a function-identifier, it must reference an alphanumeric function.

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 to Z, a to z, and the space; otherwise, it is false.

2. You cannot use the ALPHABETIC test for an operand described as numeric.

The ALPHABETIC-LOWERCASE test is true when the operand contains only the characters a to z, and the space; otherwise, it is false.

The ALPHABETIC-UPPERCASE test is true when the operand contains only the characters A to Z, and the space; otherwise, it is false.

The *class-name* test is true when the operand consists entirely of the characters listed in the definition of *class-name* in the SPECIAL-NAMES paragraph. The *class-name* test must not be used with an item whose data description describes the item 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.

6.5.3. 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 the Value of the Conditional Variable 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 6.5.1.)

6.5.4. Switch-Status Condition

The **switch-status condition** tests the on or off setting of an external logical program switch. Its general format is as follows:

condition-name

You use the SWITCH clause of the SPECIAL-NAMES paragraph to associate *condition-name* with a logical switch setting. (See Chapter 4.) The condition is true if the switch setting in effect during program execution is the same one assigned to *condition-name*.

Note

The translated value of the OpenVMS Alpha or I64 logical name COB$SWITCHES or the UNIX environment variable COBOL_SWITCHES specifies logical program switch settings. (Refer to the description of program switches in the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/].)

6.5.5. 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 as follows:

arithmetic-expression IS [NOT] { POSITIVE | NEGATIVE | ZERO }

An operand is defined as:

- 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 nonzero condition.

6.5.6. Success/Failure Condition

The **success/failure condition** tests the return status codes of COBOL and non-COBOL procedures for success or failure conditions.

status-code-id IS { SUCCESS | FAILURE }

status-code-id

must be a COMP integer represented by PIC 9(1 to 9) COMP or PIC S9(1 to 9) COMP.

You can use the SET statement to initialize or alter the status of *status-code-id*.

The SUCCESS class condition is true if you specify *status-code-id* IS SUCCESS and *status-code-id* is in a SUCCESS state. Otherwise, the SUCCESS class condition is false.

The FAILURE class condition is true if you specify *status-code-id* IS FAILURE and *status-code-id* is in a FAILURE state. Otherwise, the FAILURE class condition is false.

status-code-id is in the SUCCESS state when the low-order bit of *status-code-id* is 1. It is in the FAILURE state when its low-order bit is 0.

Examples

1. On OpenVMS, calling a non-COBOL procedure:

```
WORKING-STORAGE SECTION.
01  RMS-EOF       PIC S9(9) COMP VALUE EXTERNAL RMS$_EOF.
01  RETURN-STATUS PIC S9(9) COMP.
PROCEDURE DIVISION.
A000-BEGIN.
    .
    .
    .
    CALL "LIB$GET_SCREEN"
              USING BY DESCRIPTOR INPUT-TEXT, PROMPT,
                    BY REFERENCE  OUT-LEN,
              GIVING RETURN-STATUS.
    IF RETURN-STATUS = RMS-EOF PERFORM CTRL-Z-TRAP-ROUTINE.
    IF RETURN-STATUS IS FAILURE PERFORM FAILURE-ROUTINE.
    .
    .
    .
```

2. Calling a COBOL procedure:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN-PROGRAM.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  RETURN-STATUS      PIC S9(9) COMP.
PROCEDURE DIVISION.
    .
    .
    .
    CALL "SUB" GIVING RETURN-STATUS.
    IF RETURN-STATUS IS FAILURE PERFORM FAILURE-ROUTINE.
```

```
    .
    .
    .
IDENTIFICATION DIVISION.
PROGRAM-ID. SUB.
    .
    .
    .
WORKING-STORAGE SECTION.
01  RETURN-STATUS      PIC S9(9) COMP.
PROCEDURE DIVISION GIVING RETURN-STATUS.
    .
    .
    .
    IF A = B
           SET RETURN-STATUS TO SUCCESS
       ELSE
           SET RETURN-STATUS TO FAILURE.
    .
    .
    .
    EXIT PROGRAM.
END PROGRAM SUB.
END PROGRAM MAIN-PROGRAM.
```

6.5.7. 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 the following:

- The truth value of each condition it contains
- The effect of the logical operators

Table 6.4 shows the effect of each logical operator in complex conditions.

Table 6.4. How Logical Operators Affect Evaluation of Conditions

Logical Operator	Meaning and Effect
AND	Logical conjunction. The truth value is true if both connected conditions are true. It is false if one or both connected conditions are false.
OR	Logical inclusive OR. The truth value is true if one or both connected conditions are true. It is false if both conditions are false.
NOT	Logical negation or reversal of truth value. The truth value is true if the original condition is false. It is false if the original condition is true.

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 as follows:

NOT simple-condition

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 as follows:

condition {{ AND | OR } | condition } ...

In the general format, *condition* can be one of the following:

- 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 6.5)

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.

When the relevant parentheses are missing from a complex condition, the evaluation order of the logical operators determines the conditions to which the specified logical operators apply and implies the equivalent parentheses. The evaluation order is NOT, AND, OR. Thus, specifying:

```
a OR NOT b AND c
```

implies and is equivalent to specifying:

```
a OR ((NOT b) AND c)
```

(See also Section 6.5.9.)

Table 6.5 shows the permissible combinations of conditions, logical operators, and parentheses.

Table 6.5. 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,)

	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**
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 6.5 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

6.5.8. 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
- Ensuring that a relation condition in the consecutive sequence contains a subject (or subject and relational operator) that is common with the preceding relation condition
- Ensuring that there are no parentheses in the consecutive sequence

The general format for **abbreviated combined relation conditions** is as follows:

relation-condition { { AND | OR } | [NOT] | [relational-operator] | object } ...

The evaluation of a sequence of combined relation conditions proceeds as if the last preceding subject appears in place of the omitted subject and the last preceding relational operator appears in place of the omitted relational operator. The result of these substitutions must form a valid condition. (See Table 6.5.)

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.

Table 6.6 shows abbreviated combined (and negated combined) relation conditions and their expanded equivalents:

Table 6.6. Expanded Equivalents for Abbreviated Combined Relation Conditions

Abbreviated Combined Relation Condition	Expanded Equivalent
a > b AND NOT < c OR d	((a > b) AND (a NOT < c)) OR (a NOT < d)

Abbreviated Combined Relation Condition	Expanded Equivalent
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))))

6.5.9. 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:
   a. Relation
   b. Class
   c. Condition-name
   d. Switch-status
   e. Sign
   f. Success/failure
3. Truth values for negated simple conditions
4. Truth values for combined conditions, in this order:
   a. AND logical operators
   b. 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.

6.6. Common Rules and Options for Data Handling

This section describes the rules and options that apply when statements handle data. Data handling includes the following:

- Arithmetic operations
- Multiple receiving fields in arithmetic statements
- The ROUNDED phrase
- The ON SIZE ERROR phrase
- The CORRESPONDING phrase
- The ON EXCEPTION phrase
- Overlapping operands and incompatible data
- I/O status
- The INVALID KEY phrase
- The AT END phrase
- The FROM phrase
- The INTO phrase

6.6.1. Arithmetic Operations

The **arithmetic statements** begin with the verbs ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT. When an operand in these statements is a data item, its PICTURE must be numeric and specify no more than 31 digit positions on Alpha or I64. 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 receive the results of the operation. These data items are called **resultant identifiers**. However, the evaluation of each arithmetic statement can also use an **intermediate data item**. An intermediate data item is a compiler-supplied signed numeric data item that the program cannot access. It stores the results of intermediate steps in the arithmetic operation before moving the final value to the resultant identifiers.

When the final value of an arithmetic operation is moved to the resultant identifiers, it is transferred according to MOVE statement rules. Rounding and size error condition checking occur just before this final move. (See the MOVE statement, Section 6.6.4: ON SIZE ERROR Phrase, and Section 6.6.3: ROUNDED Phrase.)

6.6.2. 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
```

6.6.3. ROUNDED Phrase

The ROUNDED phrase 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 low-order digits in the arithmetic result are truncated when the result is moved to the resultant identifiers.

6.6.4. ON SIZE ERROR Phrase

The ON SIZE ERROR phrase allows you to specify an action for your program to take when a size error condition exists.

The NOT ON SIZE ERROR phrase allows you to specify an action for your program to take when a size error condition does not exist.

The format is as follows:

[NOT] ON SIZE ERROR stment

stment is an imperative statement.

Size error checking occurs after decimal point alignment. Rounding occurs before size error checking. Also, truncation of rightmost digits occurs before size error checking.

A size error condition is caused by the following:

- Division by zero or invalid evaluation of exponentiation (see Section 6.4.2). Both actions terminate the arithmetic operation.
- The absolute value of an arithmetic operation's result exceeds the value that is specified by the PICTURE clause of one or more of the resultant identifiers.
- Evaluation of an arithmetic expression would cause the new value to be outside the allowed range for the intermediate data item.

In the second case above, the size error condition affects the contents of only those resultant identifiers for which the size error exists.

When a size error condition occurs and the statement contains an ON SIZE ERROR phrase:

1. When standard arithmetic is in effect, the values of those resultant identifiers for which the size error exists are the same as before the operation began; when native arithmetic is in effect, those values are undefined.
2. The values of those resultant identifiers for which no size error exists are the same as they would have been if the size error condition had not occurred for any of the resultant identifiers.

3. The imperative statement in the ON SIZE ERROR phrase executes.
4. The NOT ON SIZE ERROR phrase, if specified, is ignored.
5. Control is transferred to the end of the arithmetic statement unless control has been transferred by executing the imperative statement of the ON SIZE ERROR phrase.
6. 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.

When a size error condition occurs and the statement does not contain an ON SIZE ERROR phrase:

1. The values of those resultant identifiers for which the size error exists are undefined.
2. The NOT ON SIZE ERROR phrase, if specified, is ignored.
3. Control is transferred to the end of the arithmetic statement.

When a size error condition does not occur:

1. The ON SIZE ERROR phrase, if specified, is ignored.
2. The imperative-statement in the NOT ON SIZE ERROR phrase, if specified, is executed.
3. Control is transferred to the end of the arithmetic statement unless control has been transferred by executing the imperative statement of the NOT ON SIZE ERROR phrase.

If you use the ADD or SUBTRACT statements with the CORRESPONDING phrase, any individual operation can cause a size error condition. In this instance, the imperative statement in the ON SIZE ERROR phrase executes after all the individual additions or subtractions are complete.

6.6.5. CORRESPONDING Phrase

The CORRESPONDING phrase allows you to specify group items as operands in order to use their corresponding subordinate items in an operation. See the ADD, SUBTRACT, and MOVE statements.

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, 77 or 88, or the USAGE IS INDEX clause.
5. Identifiers cannot be reference-modified.

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 in which results of the operation are stored.

1. Data items subordinate to both *identifier-1* and *identifier-2* must have the same *data-name*.
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 *identifier-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, but the resulting move must be legal according to the MOVE statement rules. (See the MOVE statement.)
5. The CORRESPONDING phrase disallows data items with the following:
   - 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 disallowed.
6. FILLER data items and their subordinates are disallowed.
7. Neither identifier-1 nor identifier-2 can be reference modified.

6.6.6. ON EXCEPTION Phrase

The ON EXCEPTION phrase allows execution of an imperative statement when an exception (or error) condition occurs.

The NOT ON EXCEPTION phrase allows execution of an imperative statement when an exception condition (or any other error condition) does not occur.

The format is as follows:

[NOT] ON EXCEPTION stment

stment is an imperative statement.

The ON EXCEPTION phrase of the CALL statement prevents control transfer of the CALL and triggers the execution of the imperative statement related to the CALL.

The ON EXCEPTION phrase of the ACCEPT statement (Formats 3 and 4) allows you to handle data entry errors when data is accepted into a numeric data field using ACCEPT WITH CONVERSION. For additional information, see the ACCEPT statement.

The ON EXCEPTION phrase allows execution of an imperative statement when an ACCEPT statement (Format 5) terminates unsuccessfully. When there is an applicable CRT STATUS clause, unsuccessful termination is indicated by a value of '1' or '9' in the first character of the CRT STATUS data item (see the SPECIAL-NAMES section of Chapter 4).

When an exception condition occurs and the statement contains an ON EXCEPTION phrase:

1. The imperative statement associated with the ON EXCEPTION phrase is executed.
2. The NOT ON EXCEPTION phrase, if specified, is ignored.
3. Control is transferred to the end of the statement unless control has been transferred by executing the imperative statement of the ON EXCEPTION phrase.

When an exception condition occurs and the statement does not contain an ON EXCEPTION phrase:

1. The NOT ON EXCEPTION phrase, if specified, is ignored.
2. The program terminates abnormally.

When an exception condition does not occur:

1. The imperative statement associated with the NOT ON EXCEPTION phrase, if specified, is executed.
2. The ON EXCEPTION phrase, if specified, is ignored.
3. Control is transferred to the end of the statement unless control has been transferred by executing the imperative statement of the NOT ON EXCEPTION phrase.

6.6.7. 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 INITIALIZE, 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, or (if the item is a function) its function definition. Conditional statements containing the class condition allow you to do the following:

- Determine whether or not an item contains numeric or alphabetic data.
- Specify corrective action when it does not.

See Section 6.5.2 for more information on class condition.

6.6.8. 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 the following I/O statements:

- CLOSE
- DELETE
- OPEN
- READ
- REWRITE
- START
- UNLOCK
- WRITE

The two characters from the FILE STATUS data item combine to form a **file status value**. The first character (Status Key 1), which occupies the leftmost character position in the item, represents a specific

class of I/O operation (**0**–success, **1**–at end, **2**–invalid key, **3**–permanent error, **4**–logic error, or **9**–VSI defined). The second character (Status Key 2), which occupies the rightmost position, provides additional information about the result of an I/O operation. In combination, Status Key 1 and Status Key 2 indicate the status of an I/O operation. For example, if you are interested in duplicate keys, you check for File Status 02.

When Status Key 1 contains **1**, the AT END phrase executes. When Status Key 1 contains **2**, the INVALID KEY phrase executes. When Status Key 1 contains **3**, **4**, or **9** the Declarative USE procedure executes. Any applicable USE AFTER EXCEPTION procedure executes after the FILE STATUS value is set.

Figure 6.1 shows the possible combinations of Status Keys 1 and 2. In the figure, X indicates a valid combination of keys.

Figure 6.1. Possible Combinations of Status Keys 1 and 2

Status Key 2 / 1	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)	Open Error (9)
Successful Completion (0)	X		X		X	X		X		
At End (1)	X				X					
Invalid Key (2)		X	X	X	X					
Permanent Error (3)	X				X	X		X	X	X
Logic Error (4)		X	X	X	X		X	X	X	X
HP- Defined (9)	X	X	X			X			X	

VM-0596A-AI

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 of the following:

- 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 on a relative or indexed file
- 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.

4

Logic Error. The input-output statement was unsuccessfully executed as a result of an improper sequence of input-output operations that were performed on the file or as a result of violating a limit set by the user.

9

VSI defined. The input-output statement executed unsuccessfully because of a condition defined by VSI.

Status Key 2

Status Key 2 further describes the result of the input-output operation as follows:

- 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**, **4**, **5**, or **7**.

  2

  Applies to a REWRITE, WRITE, or READ statement.

  - For REWRITE and WRITE statements it means that the record just written created a duplicate key value for at least one alternate record key for which duplicates are allowed.
  - For READ statements it means that the record just read has duplicate key values for the current key of reference.

  4

  Applies to a READ statement. It means the size of the record read does not agree with the size defined in the program.

  5

  Applies to the OPEN statement. It means that the optional file was not present when the OPEN statement executed. If the open mode is I-O or EXTEND, the file has been created.

7

Applies to the CLOSE and OPEN statements. It means one of the following:

- The program tried to execute a CLOSE REEL/UNIT, a CLOSE NO REWIND, or a CLOSE FOR REMOVAL statement for a file on a nonreel/unit medium.
- The program tried to execute an OPEN NO REWIND statement for a file on a nonreel/unit medium.

- When Status Key 1 contains **1** (indicating an at end condition), Status Key 2 describes the condition's cause:

0

Indicates that the file has no next logical record or it indicates that a file you specified as optional is not present.

4

Means that the relative record number of the record read was too big for the relative key data item.

- When Status Key 1 contains **2** (indicating an invalid key condition), Status Key 2 describes the condition's cause as follows:

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 sort order sequence requirements for successive record key values. (See the 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, or the file you specified as optional is not present.

4

Indicates a boundary violation. The program tried to write beyond the boundaries defined for the file by the I/O system OpenVMS Record Management Services (RMS) on OpenVMS Alpha and I64 systems), or the program attempted a sequential WRITE statement and the number of significant digits in the relative record number is larger than the size of the relative key data item.

- When Status Key 1 contains **3** (indicating a permanent error condition), Status Key 2 describes the condition causes as follows:

0

Indicates that no more information is available. This value results from any input-output error that cannot be described by any other combination of values in Status Keys 1 and 2. For example, "filename too long" is indicated this way.

4

Indicates a boundary violation on a sequential file. This means that the program tried to write to a disk that was full.

5

Indicates that the program tried to open a file that does not exist.

7

Indicates that the program tried to create a file on a device that is not appropriate for the OPEN statement mode.

8

Indicates that the program tried to open a file that is closed with a lock.

9

Indicates a conflict of file attributes. The attributes of the file that the program attempted to open do not match the attributes of the file described in the program. The attributes that are checked are BLOCK SIZE, ORGANIZATION, INDEX KEYS, and MAXIMUM RECORD SIZE. (Refer to the UNIX reference page or COBOL online help for information on the effect of the relaxed key checking option.)

- When Status Key 1 contains **4** (indicating an error in the program's logic), Status Key 2 describes the condition's cause:

1

Indicates that the program tried to open a file that is already open.

2

Indicates that the program tried to close a file that either: (a) is already closed, or (b) has not been opened during the program's execution.

3

Indicates that the program tried to execute either: (a) a DELETE or REWRITE statement without first successfully executing a READ or START statement, or (b) an UNLOCK RECORD statement without first establishing a current record.

4

Indicates that the program attempted to REWRITE a record that is not the same size as the record being replaced.

6

Indicates that the program did not establish a valid next record.

The values 10 and 46 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, 10, 46, 46, … , 46

7

Indicates the program tried a READ or START operation on a file that: (a) has not been opened, or (b) has been opened in a mode that is incompatible with the operation.

8

Indicates the program attempted a WRITE operation on a file that: (a) has not been opened, or (b) has been opened in a mode that is incompatible with the operation.

9

Indicates the program tried a DELETE or REWRITE on a file that: (a) has not been opened, or (b) has been opened in a mode that is incompatible with the operation.

- When Status Key 1 contains **9** (indicating a VSI defined condition), Status Key 2 further describes the condition, as follows:

0

Means that the record your program is reading has been locked by another access stream. Because the record is available in the record area, the input operation is successful. This condition results from using the REGARDLESS option. Without the REGARDLESS option, the same scenario causes a Status Key 2 value of **2**.

1

Indicates that a file is locked. The access stream 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 access stream.

In this case, the record is not available in the record area, so the input operation is unsuccessful.

5

Means that the program tried to open a file when there was not enough space on the device.

8

Indicates that an unspecified error occurred when the program attempted to close a file.

Appendix C lists all the possible file status values that can appear in the FILE STATUS data item, along with the I-O status condition corresponding to each value.

6.6.9. AT END Phrase

The AT END phrase specifies the action your program takes when an at end condition occurs (when Status Key 1 contains **1**).

The NOT AT END phrase specifies the action your program takes if an at end (or any other error condition) does not occur.

The format is as follows:

[NOT] AT END stment

stment is one or more imperative statements.

When a program detects the end of a file, the condition is called the **at end condition**. The at end condition might occur as a result of ACCEPT, READ, RETURN, or SEARCH statement execution. (For additional information, see the previously mentioned statements.)

When an at end condition occurs and the statement contains an AT END phrase:

1. The imperative statement associated with the AT END phrase, if specified, executes.
2. The NOT AT END phrase, if specified, is ignored.
3. Control is transferred to the end of the ACCEPT, READ, RETURN, or SEARCH statement unless control has been transferred by executing the imperative statement of the AT END phrase.

When an at end condition occurs and the statement does not contain an AT END phrase:

1. If the at end condition is associated with a READ statement, the applicable USE AFTER EXCEPTION procedure, if specified, executes.
2. If the at end condition is associated with an ACCEPT, RETURN, or SEARCH statement, any USE procedure associated with that file is not applicable.
3. The NOT AT END phrase, if specified, is ignored.

When an at end condition does not occur, and no other exception condition exists:

1. The AT END phrase, if specified, is ignored.
2. The imperative statement associated with the NOT AT END phrase, if specified, is executed. Otherwise, control is transferred to the end of the ACCEPT, READ, RETURN, or SEARCH statement.

When an at end condition does not occur, and another exception condition does exist:

- The applicable USE AFTER EXCEPTION procedure, if specified, executes and control is then transferred according to the rules of the USE statement.
- If there is no applicable USE AFTER EXCEPTION procedure, the imperative statement associated with the NOT AT END phrase, if specified, is executed, unless the exception condition causes the run unit to terminate abnormally.
- If there is neither a USE AFTER EXCEPTION procedure nor a NOT AT END phrase, then control is transferred to the end of the statement, unless the exception condition causes the run unit to terminate abnormally.

6.6.10. INVALID KEY Phrase

The INVALID KEY phrase specifies the action your program takes when an invalid key condition is detected (when Status Key 1 contains **2**) for the file being processed.

The NOT INVALID KEY phrase specifies the action your program takes when an invalid key condition (or any other error condition) is not detected for the file being processed.

The format is as follows:

[NOT] INVALID KEY stment

stment is one or more imperative statements.

The invalid key condition occurs when the I/O system cannot complete a COBOL DELETE, READ, REWRITE, START, or WRITE statement because of one of the following conditions:

- Sequence error
- Duplicate key when the COBOL program did not specify this condition
- No record found
- Boundary violation on a relative or indexed file
- Optional file not present

(For more information on these conditions, refer to Section 6.6.8.) When the invalid key condition occurs, execution of the statement that produced the condition is unsuccessful, and the file is unaffected. (For additional information, see the previously mentioned statements.)

When the invalid key condition occurs:

1. A value that indicates the invalid key condition is placed in the FILE STATUS data item for the file.
2. If the statement that caused the condition has the INVALID KEY phrase:
   a. Any USE AFTER EXCEPTION procedure is not executed.
   b. The imperative statement associated with the INVALID KEY phrase executes.
   c. The NOT INVALID KEY phrase, if specified, is ignored.

   d. Control is transferred to the end of the I-O statement unless control has been transferred by executing the imperative statement of the INVALID KEY phrase.

3. If the statement that caused the condition does not have an INVALID KEY phrase:

   a. The NOT INVALID KEY phrase, if specified, is ignored.

   b. Control is transferred to the applicable USE AFTER EXCEPTION procedure for the file.

When an invalid key condition does not occur, and no other exception condition exists:

1. The INVALID KEY phrase, if specified, is ignored.

2. The imperative statement associated with the NOT INVALID KEY phrase, if specified, is executed. Otherwise, control is transferred to the end of the I/O statement.

When an invalid key condition does not occur, and another exception condition does exist:

- The applicable USE AFTER EXCEPTION procedure, if specified, executes and control is then transferred according to the rules of the USE statement.

- If there is no applicable USE AFTER EXCEPTION procedure, the imperative statement associated with the NOT INVALID KEY phrase, if specified, is executed, unless the exception condition causes the run unit to terminate abnormally.

- If there is neither a USE AFTER EXCEPTION procedure nor a NOT INVALID KEY phrase, then control is transferred to the end of the statement, unless the exception condition causes the run unit to terminate abnormally.

6.6.11. FROM Phrase

Format 1

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 I-O-CONTROL, the REWRITE statement, and the WRITE statement.)

6.6.12. INTO Phrase

The INTO phrase implicitly moves a current record from the record storage area into an identifier.

The format is as follows:

file-name INTO identifier

A READ or RETURN statement can have the INTO phrase if either of the following conditions is true:

- Only one record description is subordinate to the file description entry.
- 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 the identifier.

6.7. Segmentation

VSI COBOL for OpenVMS programs execute in a virtual memory environment. Therefore, programs need not manage physical memory by overlaying Procedure Division code. VSI COBOL for OpenVMS provides support for segmentation only for compatibility with existing applications developed on older hardware such as the PDP-11. You should not use segmentation in newly written COBOL programs since segmentation results in the generation of extra code which might impact performance.

Segmentation controls the assignment of Procedure Division sections to fixed or independent segments. The optional *segment-number* in the section header determines the type of segment.

section-name SECTION [segment-number]

`section-name`

names a Procedure Division section.

`segment-number`

must be an integer in the range 0 to 99. If there is no *segment-number* in a section header, the implied *segment-number* is 0.

segment-number classifies a segment into **fixed segments** or **independent segments**. Sections with *segment-numbers* from 0 to 49 are in fixed segments. Those with *segment-numbers* from 50 to 99 are in independent segments.

Sections in the Declaratives part of the Procedure Division must have *segment-numbers* less than 50.

A segment consists of all sections that have the same *segment-number*.

Both fixed and independent segments are in their initial state the first time entered. A fixed segment appears to reside in memory at all times and is, therefore, in its last used state each time it is entered.

The state of an independent segment depends on how and when it receives control. On subsequent control transfers, VSI COBOL for OpenVMS resets the segment's ALTERed GO TO statements to their initial states whenever an independent segment is entered in one of the following ways:

1. Explicitly, by means of a GO TO statement with a target within the section.

2. Explicitly, by means of an out-of-line PERFORM statement in another segment whose range is within the section.
3. Implicitly, when a SORT or MERGE statement in another segment specifies an input or output procedure within the section.
4. Implicitly, by transfer of control between consecutive statements from a segment with a different segment-number.

6.8. General Formats and Rules for Statements

General Format

Format 1

```
PROCEDURE DIVISION [ USING { data-name } ... ] [ GIVING identifier ] .
  [ 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 } ... ] [ GIVING identifier ] .
  [ paragraph-name . [ sentence ] ... ] ...
```

The Procedure Division contains the routines that process the files and data described in the Environment and Data Divisions.

Syntax Rules

1. The Procedure Division follows the Data Division.
2. The Procedure Division must begin with the Procedure Division header.
3. The end of the Procedure Division is indicated by one of the following:
   - The Identification Division header of another source program
   - The END PROGRAM header
   - The physical position in the Procedure Division after which no further processing occurs

4. A procedure consists of either:
   - One or more successive sections
   - One or more successive paragraphs
5. If one paragraph is in a section, all paragraphs must be in sections.
6. 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*.
7. 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. See the section called “Declaratives” for more information on declaratives.
8. A paragraph consists of a *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 DECLARATIVE. See the section called “Declaratives” for more information on declaratives.
9. *sentence* contains one or more statements terminated by a separator period.
10. A statement is a syntactically valid combination of words and symbols that begins with a COBOL verb.
11. *identifier* is the word or words necessary to refer uniquely to a data item.

Procedure Division Header

1. The Procedure Division header identifies and begins the Procedure Division. It consists of the reserved words PROCEDURE DIVISION and optional USING and GIVING phrases followed by a separator period.
2. The USING phrase is required only if the program is invoked by a CALL statement with a USING phrase.
3. 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.
4. Each *data-name* in the USING phrase must be defined in the Linkage Section with a level-01 or level-77 entry.
5. Each *data-name* cannot appear more than once in the USING phrase.
6. In the USING phrase, *data-name* cannot have the external attribute.
7. In the USING phrase, the data description for *data-name* cannot contain a REDEFINES clause. However, the data description can be the object of a REDEFINES clause.
8. The Procedure Division header GIVING phrase specifies a function result of the program. The *identifier* must refer to an elementary integer numeric data item with COMP, COMP-1, or COMP-2 usage and no scaling positions. The *identifier* cannot be subscripted, but it can be qualified.

Procedure Division Body

1. The Procedure Division body consists of all Procedure Division text following the Procedure Division header.

General Rules

1. References to USING phrase *data-name*s 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 *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 conditions.
   - It is a condition-name or index-name associated with a data item that satisfies any of the previous conditions.

3. On Alpha and I64 systems, when a called program returns control to the calling program, the return value is made available to the calling program in the data item specified in its CALL statement GIVING phrase. The value is moved to that data item according to the rules for the MOVE statement. If the calling program does not specify a GIVING phrase, then the return value is made available in the calling program's RETURN-CODE special register. Note that the value in the called program's RETURN-CODE is not returned to the caller.

4. On Alpha and I64 systems, if no GIVING phrase is specified in the called program, the value in the RETURN-CODE special register is made available to the calling program in the data item specified in the calling program's CALL statement GIVING phrase. The value is moved according to the rules for the MOVE statement. If the calling program does not specify a CALL GIVING phrase, the value in the called program's RETURN-CODE special register is made available to the calling program in the calling program's RETURN-CODE special register.

   Table 6.7 shows the relationship between the GIVING phrase and RETURN-CODE.

Table 6.7. Relation of GIVING Phrase to RETURN-CODE Special Register (Alpha, I64)

Calling program has CALL GIVING X	Called program has PROCEDURE DIVISION GIVING Y	Called program puts result in	Calling program gets result in
YES	YES	Y (also RETURN-CODE)	X (moved from Y)
YES	NO	RETURN-CODE	X (moved from called program's RETURN-CODE)
NO	YES	Y (also RETURN-CODE)	RETURN-CODE (moved from Y)

Calling program has CALL GIVING X	Called program has PROCEDURE DIVISION GIVING Y	Called program puts result in	Calling program gets result in
NO	NO	RETURN-CODE	RETURN-CODE (moved from called program's RETURN-CODE)

Technical Notes

1. On Alpha and I64 systems, because the reserved word RETURN-CODE is one of the X/Open reserved words, you cannot use the compilation flag `-rsv noxopen` (for UNIX systems) or the corresponding qualifier /RESERVED_WORDS = NOXOPEN (for OpenVMS systems) if your program uses the RETURN-CODE special register.

2. On Alpha and I64 systems, VSI COBOL for OpenVMS supports passing status to the operating system for RETURN-CODE and PROCEDURE DIVISION GIVING when EXIT PROGRAM or STOP RUN is executed.

   Four of the data types supported by PROCEDURE DIVISION GIVING can be used to communicate status to the operating system. Following is a summary of what is supported for both RETURN CODE and PROCEDURE DIVISION GIVING:

```
RETURN-CODE (Alpha and I-64 only*)
 EXIT PROGRAM /STA=V3    yes
 EXIT PROGRAM /STA=85    yes
 STOP RUN                yes
PROCEDURE DIVISION GIVING
 EXIT PROGRAM /STA=V3    yes
 EXIT PROGRAM /STA=85    yes
 STOP RUN                yes*
 Data Types
  COMP-1,COMP-2          no
  PIC  9(04) COMP        no
  PIC S9(04) COMP        no
  PIC  9(09) COMP        yes
  PIC S9(09) COMP        yes
  PIC  9(18) COMP        yes
  PIC S9(18) COMP        yes
  PIC  9(31) COMP        no
  PIC S9(31) COMP        no
```

   This support is subject to the limitations on status handling imposed by the operating system. If PIC S9(18) COMP or PIC 9(18) COMP is used, the high-order 32 bits are truncated before the status is passed on to the operating system.

   To display the operating system status information, do the following:

```
[UNIX]              echo $status
[OpenVMS]           show symbol $status
```

Additional References

- CALL statement
- USE statement

- Section 6.7: Segmentation

Example

The following is an example of a Procedure Division header:

```
WORKING-STORAGE SECTION.
01      RETURN-RESULT PIC 9(8) COMP.
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 GIVING RETURN-RESULT.
        .
        .
        .
        MOVE 17 TO RETURN-RESULT.
        EXIT PROGRAM.
```

ACCEPT

ACCEPT — The ACCEPT statement makes low-volume data available to the program. The VSI extensions to the ACCEPT statement (Formats 3, 4, and 5) are COBOL language additions that facilitate video forms design and data handling. The WITH CONVERSION phrase and some other options in Format 1 are also VSI extensions. Format 6 retrieves the number of arguments on the program run command line, Format 7 reads those command line arguments into designated program variables, and Format 8 reads environment variables and logicals into designated program variables.

General Format

Format 1

```
ACCEPT dest-item [ FROM input-source ] [ WITH CONVERSION ]
    [ AT END stment ]
    [ NOT AT END stment2 ]
    [ END-ACCEPT ]
```

Format 2

```
                    { DATE [ YYYYMMDD ]
                    { DAY [ YYYYDDD ]
ACCEPT dest-item FROM { DAY-OF-WEEK
                    { 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 { prot-size-lit
                         prot-size-item }
                  WITH AUTOTERMINATE
                  WITH NO BLANK
                  WITH EDITING
                  WITH FILLER prot-fill-lit } } ]
  WITH CONVERSION
  REVERSED
  WITH NO ECHO
  DEFAULT IS { def-src-lit
               def-src-item
               CURRENT VALUE }
  CONTROL KEY IN key-dest-item }

[ { [ ON EXCEPTION stment ] [ NOT ON EXCEPTION stment2 ]
    [ AT END stment ] [ NOT AT END stment2 ] } ]
  [ END-ACCEPT ]
```

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 ] [ NOT ON EXCEPTION stment2 ] } ]
  { [ AT END stment ] [ NOT AT END stment2 ]               }

[ END-ACCEPT ]
```

Format 5 (Alpha, I64)

```
ACCEPT screen-name

[ AT { | LINE NUMBER { line-id   }                    | } ]
     { |             { line-num  }                    |
     { | COLUMN NUMBER { column-id }                  |
     { |               { column-num }                 | }

[ ON EXCEPTION stment ]
[ NOT ON EXCEPTION stment2 ]
[ END-ACCEPT ]
```

Format 6 (Alpha, I64)

```
ACCEPT dest-item FROM arg-count [ END-ACCEPT ]
```

Format 7 (Alpha, I64)

```
ACCEPT dest-item FROM arg-value

[ ON EXCEPTION stment3 ]
[ NOT ON EXCEPTION stment5 ]
[ END-ACCEPT ]
```

Format 8 (Alpha, I64)

```
ACCEPT dest-item [ FROM envlog-value ]
    [ ON EXCEPTION stment4 ]
    [ NOT ON EXCEPTION stment5 ]
    [ END-ACCEPT ]
```

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.

stment

is an imperative statement executed when the relevant condition (at end or on exception) occurs.

stment2

is an imperative statement executed when the relevant condition (not at end or not on exception) occurs.

stment3

is an imperative statement executed when an attempt is made to read beyond the last argument on the command line, or if the command line argument does not exist.

stment4

is an imperative statement executed if the name of a referenced environment variable or logical has not been set, or if the referenced environment variable or logical does not exist.

stment5

is an imperative statement executed if the exception condition does not exist.

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.

line-id

is the identifier of a data item that provides a line position on the terminal screen. It must be a positive integer; it cannot be zero.

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.

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.

`column-id`

is the identifier of a data item that provides a column position on the terminal screen. It must be a positive integer; it cannot be zero.

`prot-size-lit`

is a numeric literal that specifies the maximum length of the video screen field into which data can be typed. *prot-size-lit* must be a positive integer; it cannot be zero.

`prot-size-item`

is the identifier of a numeric integer data item that specifies the maximum length of the video screen field into which data can be typed. *prot-size-item* must be a positive integer; it cannot be zero.

`prot-fill-lit`

is a single character alphanumeric literal that is used to initialize each character position of a protected video screen field into which data can be typed.

`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.

`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.

`screen-name`

is the name of a screen item defined in the SCREEN SECTION of the program.

`arg-count`

is a mnemonic name associated with ARGUMENT-NUMBER in the SPECIAL-NAMES paragraph in the Environment Division. It represents the number of arguments present on the run command line.

is a mnemonic name associated with ARGUMENT-VALUE in the SPECIAL-NAMES paragraph in the Environment Division. It contains the value of the argument on the run command line specivied by the current argument position indicator.

`envlog-value`

is a mnemonic name associated with ENVIRONMENT-VALUE in the SPECIAL-NAMES paragraph in the Environment Division. It contains the value of a selected environment variable or system logical.

Syntax Rules

Format 3

1. You cannot specify a phrase more than once for any *dest-item*.
2. When you use the DEFAULT phrase and the PROTECTED phrase without the SIZE option, the size of *def-src-item* or *def-src-lit* must be less than or equal to the size of *dest-item*.
3. When you use the DEFAULT phrase and the PROTECTED phrase with the SIZE option, the size of *def-src-item* and *def-src-lit* must be less than or equal to *prot-size-lit*. If *prot-size-item* is specified and the specified size at run time is less than the length of *def-src-item* or *def-src-lit*, reprompting occurs.
4. The FILLER phrase cannot be used with the EDITING phrase. If both are present, the FILLER phrase is ignored.

Format 4

5. You cannot specify a phrase more than once for any *key-dest-item*.

Format 6 (Alpha, I64)

6. *dest-item* must reference a data item described as an unsigned integer.

Formats 7 and 8 (Alpha, I64)

7. *dest-item* must reference an alphanumeric data 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, unless the WITH CONVERSION phrase is specified. 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, unless the WITH CONVERSION phrase is specified.
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, DAY-OF-WEEK, and TIME are not actual data items. Therefore, the source program must not describe them.

8. DATE has three elements. From left to right, they are as follows:

   - Year of century (four digits if you specify YYYYMMDD, two digits if you do not)
   - Month of year (two digits)
   - Day of month (two digits)

     The ACCEPT statement operates as if DATE were described in the program as an eight-digit or six-digit, unsigned, elementary, numeric integer data item (PIC 9(8) or PIC 9(6)).

     For example, June 3, 1997 is expressed as 19970603 or 970603.

   - DAY has two elements. From left to right, they are as follows:

     Year of century (four digits if you specify YYYYDDD, two digits if you do not)

   - Day of year (three digits)

   The ACCEPT statement operates as if DAY were described in the program as a seven-digit or five-digit, unsigned, elementary, numeric integer data item (PIC 9(7) or (PIC 9(5)).

   For example, the fifteenth day of 1998 is expressed as 1998015 or 98015.[5]

   The YYYYMMDD and YYYYDDD options are VSI extensions.

9. DAY-OF-WEEK is a one-digit item that represents the day of the week.

   The ACCEPT statement operates as if DAY-OF-WEEK were described in the program as a one-digit, unsigned, elementary numeric integer data item.

   The values of DAY-OF-WEEK range from 1 (for Monday) to 7 (for Sunday).

10. TIME represents elapsed time after midnight, as shown on a 24-hour clock. It has four, two-digit elements. From left to right, they are as follows:

    - 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 PM is expressed as 18130000. The minimum and maximum values of TIME are 00000000 and 23595999.

Formats 3 and 4

12. 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). Format 3 can also update *key-dest-item*.

[5] VSI COBOL also supports four-digit years using the CURRENT-DATE intrinsic function (see Chapter 7.) VSI recommends the use of four-digit years.

13. The presence of either the LINE NUMBER phrase or the COLUMN NUMBER phrase implies NO ADVANCING. The cursor remains on the character position immediately following the position of the last input character or in the position immediately following the rightmost position in the protected area. (For example, ACCEPT … PROTECTED SIZE 10 LINE 1 COLUMN 1, leaves the cursor at line 1, column 11, no matter what is typed in.) This is the default starting position of the next data item the program will 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.

Formats 3, 4, and 5

15. The execution of certain extended ACCEPTs when the input source is assigned to a file (for example, in batch mode on OpenVMS systems), is a restriction. Syntax and actions that result in outputs from the ACCEPT operation (positioning, erasing, setting character attributes, reprompting, and protecting) to a nonvideo terminal are not supported and are ignored.

LINE NUMBER Phrase (Formats 3 and 4)

16. The LINE NUMBER phrase positions the cursor on a specific line of the video screen for data input.

17. 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.

18. 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.) Scrolling results if relative positioning is attempted past the bottom of the screen.

19. If you use *line-id* without its PLUS option, the line position is the value of *line-id*.

20. If you use *line-id* with its PLUS option, the line position is the sum of *plus-num* and the value of *line-id*.

21. 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.

22. If you use the PLUS option, but you do not specify *plus-num*, then PLUS 1 is implied.

23. Data input results are undefined if your program generates a value for *line-id* that is one of the following:

    - Zero
    - Negative
    - Greater than the bottommost line position of the current screen

COLUMN NUMBER Phrase (Formats 3 and 4)

24. The COLUMN NUMBER phrase positions the cursor on a specific column of the video screen.

25. If the COLUMN NUMBER phrase does not appear, but the LINE NUMBER phrase does, then data is accepted from column 1 of the specified line position.

26. If you use *column-id* without its PLUS option, the column position is the value of *column-id*.

27. If you use *column-id* with its PLUS option, the column position is the sum of *plus-num* and the value of *column-id*.

28. 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.

29. If you use the PLUS option, but do not specify *plus-num*, PLUS 1 is implied.

30. 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

LINE NUMBER and COLUMN NUMBER Phrases (Format 5) (Alpha, I64)

31. The LINE NUMBER and COLUMN NUMBER phrases together give the starting screen coordinates.

32. The position of each screen item within the referenced *screen-name* is offset from the LINE and COLUMN positions.

33. If either LINE or COLUMN is not specified, the default value is 1.

ERASE Phrase (Formats 3 and 4)

34. The ERASE phrase erases all, or part, of a line (or screen) before accepting data. You must specify SCREEN or LINE with the ERASE phrase.

35. If you use the 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).

36. If you do not use the TO END option, the ERASE phrase erases the entire line (or screen).

BELL Phrase (Formats 3 and 4)

37. The BELL phrase rings the terminal bell before accepting data.

CONTROL KEY Phrase (Formats 3 and 4)

38. 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 Notes.)

39. *key-dest-item* stores the terminator key code; unused character positions, if any, are filled with spaces. (See Technical Notes.)

ON EXCEPTION Phrase (Formats 3 and 4)

40. The ON EXCEPTION phrase allows execution of an imperative statement when an exception (or error) condition occurs. ON EXCEPTION takes effect when illegal numeric data has been entered or there is an overflow on the left or right of the decimal point when CONVERSION is specified.

41. ON EXCEPTION can be used to detect numeric data entry errors only when accepting numeric data while CONVERSION is being used.

42. ON EXCEPTION can be used to detect end-of-file in any Format 3 or Format 4 ACCEPT statement.

43. ON EXCEPTION and AT END are mutually exclusive. If ON EXCEPTION is specified, the end-of-file indication is also a control key.

44. A DISPLAY statement within an ACCEPT ON EXCEPTION must be terminated (with, for example, END-DISPLAY) on Alpha and I64 systems. (If you are concerned with the different VAX behavior, refer to the appendix on compatibility in the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/].)

NOT ON EXCEPTION Phrase (Formats 3 and 4)

45. The NOT ON EXCEPTION phrase allows execution of an imperative statement when an exception (or error) condition does not occur.

ON EXCEPTION Phrase (Format 5, Alpha, I64)

46. The ON EXCEPTION phrase allows execution of an imperative statement when the ACCEPT statement terminates unsuccessfully. When there is an applicable CRT STATUS clause, unsuccessful termination is indicated by a value of '1' or '9' in the first character of the CRT STATUS data item (see the SPECIAL-NAMES section of Chapter 4).

NOT ON EXCEPTION Phrase (Format 5, Alpha, I64)

47. The NOT ON EXCEPTION phrase allows execution of an imperative statement when the ACCEPT statement terminates successfully. When there is an applicable CRT STATUS clause, successful termination is indicated by a value of '0' in the first character of the CRT STATUS data item (see the SPECIAL-NAMES section of Chapter 4).

NOT ON EXCEPTION Phrase (Formats 3 and 4; and Formats 5, 7, and 8, Alpha, I64)

48. A DISPLAY statement within an ACCEPT [NOT] ON EXCEPTION statement must be terminated (with, for example, END-DISPLAY).

AT END Phrase (Formats 3 and 4)

49. The AT END phrase allows execution of an imperative statement when an end-of-file condition occurs.

50. AT END and ON EXCEPTION are mutually exclusive.

51. If AT END is specified, the end-of-file indication is also a control key. If you do not specify AT END or ON EXCEPTION, and end-of-file is entered, an error condition occurs.

NOT AT END Phrase (Formats 3 and 4)

52. The NOT AT END phrase allows execution of an imperative statement when an end-of-file condition (or other error) does not occur.

Format 3

UNDERLINED Phrase (Format 3)

53. The UNDERLINED phrase echoes input characters to the terminal with the *underscore on* character attribute.

54. When you use the UNDERLINED phrase with the PROTECTED phrase, the input field is underlined prior to accepting data.

BOLD Phrase (Format 3)

55. The BOLD phrase echoes input characters to the terminal with the *bold on* character attribute.

56. When you use the BOLD phrase with the PROTECTED phrase, the input field is visibly bolded prior to accepting data only if: (1) the underlined or reversed attributes are also specified, or (2) the video terminal screen is set to light background.

BLINKING Phrase (Format 3)

57. The BLINKING phrase echoes input characters to the terminal with the *blink on* character attribute.

58. When you use the BLINKING phrase with the PROTECTED phrase, the input field is visibly blinked prior to accepting data only if: (1) the underlined or reversed attributes are also specified, or (2) the video terminal screen is set to light background.

REVERSED Phrase (Format 3)

59. The REVERSED phrase echoes input characters to the terminal with the *reverse video on* character attribute.

60. When you use the REVERSED phrase with the PROTECTED phrase, the input field appears in reverse video prior to accepting data.

CONVERSION Phrase (Formats 1 and 3)

61. The CONVERSION phrase allows you to accept data into a field and achieve the same results as you would with the MOVE statement for non-floating-point items, and provides conversion from display, display scaled or E-notation to floating point in the case of floating-point data items. It enables validation of the accepted data and facilitates editing and alignment of data within *dest-item*. The effect of the CONVERSION phrase on data handling depends on the category of *dest-item*. (Numeric data can be described by any USAGE clause.)

62. When *dest-item* is numeric or numeric edited (other than floating point), the CONVERSION phrase:

    - Converts input numeric data to a numeric literal (TRAILING SEPARATE SIGNED DISPLAY DECIMAL)
    - Moves the result to *dest-item* (using MOVE statement rules)

63. When *dest-item* is floating point, the CONVERSION phrase:

    - Converts input data to floating point (COMP-1 or COMP-2 as appropriate).
    - Moves the converted result to the destination as if a numeric literal equivalent to the input data was moved to the destination with the MOVE statement.

64. When *dest-item* is numeric or numeric edited (other than floating point), and you use the CONVERSION phrase, valid input characters are as follows:

    - 0 to 9
    - Period (.), unless DECIMAL POINT IS COMMA is specified
    - Comma (,), if DECIMAL POINT IS COMMA is specified
    - 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.

65. When *dest-item* is floating point, and you use the CONVERSION phrase, valid input characters are as follows:

    - <zero or more blanks>
    - < “+”, “-”, or null>
    - <zero or more decimal digits>
    - < “.” or null> if DECIMAL POINT IS COMMA then < “,” or null>

- <zero or more decimal digits>

For example:

```
2.5E2
-0.08e4
10.0E-1
-2.14158E0
```

Note that numbers can be expressed in several ways. For example, the number 257.0 can be represented in any of the following ways:

```
257e0   2.57E2   0.000257E+6   2570E-1
```

66. When you use the CONVERSION phrase and *dest-item* is numeric, data input results in a conversion error condition if the operator enters any of the following:

    - For fixed point numeric, too many characters on either side of the decimal point. (The PICTURE clause of *dest-item* determines this overflow condition.)

    - For floating-point numeric, an exponent outside the valid range:

        - For IEEE double format (T_floating format) and G_floating format, the valid range is +308 to -308.

        - For IEEE single format (S_floating format), D_floating format, and F_floating format, the valid range is +38 to -38.

    - Invalid numeric data (for both fixed and floating-point numeric).

    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. The input field is not erased if the EDITING phrase is used.

    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.

67. 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, or a reference modified 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.

68. When you use the CONVERSION phrase, and the operator types the terminator key prior to any data input:

    - ZEROES (or spaces if BLANK WHEN ZERO is specified) 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. If ON EXCEPTION is used, an exception path is followed. If ON EXCEPTION is not used, an automatic reprompt for data occurs.

69. If you do not use the CONVERSION phrase, data is transferred to *dest-item* according to Format 1 ACCEPT statement rules when CONVERSION is not specified.

PROTECTED Phrase (Format 3)

70. The PROTECTED phrase limits the number of characters that can be entered from the terminal.

71. 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.

72. When you specify the PROTECTED phrase without the AUTOTERMINATE 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 position to the right of 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.

73. If you use the PROTECTED phrase without the SIZE phrase, the maximum number of characters that the operator can enter is as follows:

    - The number of characters in a fixed point *dest-item*

    - Thirteen, if *dest-item* is COMP-1 F_floating format or IEEE S_floating format

    - Twenty two, if *dest-item* is COMP-2 D_floating or G_floating format or IEEE T_floating format

    For non-floating-point numeric items, 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*, all of the following character strings are valid input:

    ```
    2222.22
    2222
    .22
    +2222.22
    222.22-
    -02.2
    ```

    Although the sign does not represent a character position (and does not count toward item size), a space in the input field is allocated for it. A space is also allocated for the implied decimal point.

74. When you use the PROTECTED phrase without the NO BLANK or FILLER phrase, the input field is filled with spaces prior to accepting data. If you also use the UNDERLINED, REVERSED, BOLD, or BLINKING phrase, those spaces have the specified character attributes.

75. If you use the PROTECTED phrase on a field that causes the cursor to position past the last column position of the screen, the results are undefined.

76. If you do not use the PROTECTED phrase, an overflow condition is treated according to rules for the Format 1 ACCEPT statement.

SIZE Phrase (Format 3)

77. You can use the SIZE phrase only when you also specify the PROTECTED phrase.

78. 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*.

79. Data input results are undefined if the program generates a value for *prot-size-item* that is zero or negative.

AUTOTERMINATE Phrase (Format 3)

80. You can use the AUTOTERMINATE phrase only when you also specify the PROTECTED phrase.

81. The AUTOTERMINATE phrase terminates a protected ACCEPT as soon as the maximum number of characters has been entered. If the maximum number is entered, you should not enter a legal terminator. If you enter the maximum number of characters and a terminator, the terminator is retained in the terminal driver type-ahead buffer and will terminate the next ACCEPT statement. If you enter fewer than the maximum number of characters, a legal terminator is required to terminate the ACCEPT statement. The maximum number of characters is determined by the same rules that apply to the PROTECTED phrase.

82. If you do not use the AUTOTERMINATE phrase, the ACCEPT statement is terminated when you enter a legal terminator key.

NO BLANK Phrase (Format 3)

83. You can use the NO BLANK phrase only when you also specify the PROTECTED phrase.

84. The NO BLANK phrase specifies that the input field is not to be changed until you enter the first keystroke. Once you enter a keystroke, the remainder of the field is filled with spaces or filler (or the default value, if the EDITING and DEFAULT phrases are used) using any character attributes specified on the ACCEPT statement.

EDITING Phrase (Format 3)

85. You can use the EDITING phrase only when you also specify the PROTECTED phrase.

86. On OpenVMS systems, the EDITING phrase enables keys to perform field editing functions, described in Table 6.8.

Table 6.8. Field Editing Keys for OpenVMS Systems

Key	Description
Left, Ctrl/D	Move left.

Key	Description
Right, Ctrl/F	Move right.
Ctrl/H, F12	Move to beginning of line.
Ctrl/E	Move to end of line.
Ctrl/A, F14	Toggle insert and overstrike mode.
Ctrl/K	Erase to end of line.
Ctrl/U	Erase to beginning of line.
Ctrl/M, TAB, CR	Terminate input.
Ctrl/Z	End-of-file termination.

On UNIX systems, the EDITING phrase enables the keys described in Table 6.9.

Table 6.9. Field Editing Keys for UNIX Systems

Key	Description
Left, Ctrl/B	Move left
Right, Ctrl/F	Move right
Ctrl/H	Erase previous character
F12, Ctrl/A	Move to beginning of line
Ctrl/E	Move to end of line
F14, Ctrl/T	Toggle insert and overstrike mode
Ctrl/K	Erase to end of line
Ctrl/U	Erase to beginning of line
Ctrl/M, TAB, CR	Terminate input
Ctrl/D	End of file termination

87. In insert editing, each new character is entered where the cursor indicates and the cursor moves to the right. The rest of the field also moves one character position to the right. A character moved beyond the last character position of the field is lost. The delete key causes the character to the left of the cursor to be deleted. The rest of the field moves one character position to the left, and space is inserted in the last position of the field.

    In overstrike editing, each new character replaces the one where the cursor is positioned and the cursor moves to the right. The delete key causes the character to the left of the cursor to be replaced by a space and the cursor moves to the left.

88. When program execution begins, overstrike or insert editing is used, according to your terminal's current setting. The same editing mode is used for all ACCEPT statements with the EDITING phrase until the next use of the switch-mode function.

89. The EDITING phrase affects the syntax and semantics of other phrases. For more information, see the sections on NO BLANK, FILLER, CONVERSION, NO ECHO, DEFAULT, CURRENT VALUE, and CONTROL KEY.

FILLER Phrase (Format 3)

90. You can use the FILLER phrase only when you also specify the PROTECTED phrase.

91. The FILLER phrase initializes each character position of the input field with the character specified in *prot-fill-lit*. As you enter characters, the filler characters are replaced by your input. If you strike the delete key after you have entered data, the position made available by the delete operation is refilled with the character specified in *prot-fill-lit*. When you terminate the ACCEPT operation, any remaining filler characters are replaced by space characters.

92. When you use the FILLER phrase with the NO BLANK phrase, the input field is filled with the character specified in *prot-fill-lit*, after you have entered the first character.

93. The FILLER phrase is not allowed to be used with the EDITING phrase. If both are present, the FILLER phrase is ignored.

NO ECHO Phrase (Format 3)

94. The NO ECHO phrase suppresses the display of input characters on the screen.

95. When you do not use the NO ECHO phrase, valid input characters are displayed on the screen as they are typed.

96. When the EDITING phrase is used, the field editing functions still take place, but the display field is not modified.

DEFAULT Phrase (Format 3)

97. 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.)

98. When the null input condition occurs, *def-src-lit* or the value of *def-src-item* is moved to *dest-item* according to theMOVE rules. When the move occurs, the specified default value is not displayed on the terminal screen.

99. Conversion of the DEFAULT item will occur if CONVERSION is specified.

100. When the EDITING phrase is used, the default value is displayed in the input field. The value can be blank-filled on the right or truncated, depending on the relative lengths of the default value and the input field.

101. When the EDITING phrase is used and a terminator is entered, the contents of the input field are moved to *dest-item* according to the MOVE rules.

CURRENT VALUE Phrase (Format 3)

102. The CURRENT VALUE phrase can be used only when you specify the DEFAULT phrase.

103. The CURRENT VALUE phrase specifies that the default input value is the initial value of the ACCEPT destination item.

104. The value of the ACCEPT destination item is the same as it was before the execution of the ACCEPT statement if all the following conditions exist:

- You specify the CURRENT VALUE phrase.
- The EDITING phrase is not used.
- The default path is taken.

105. When you use the EDITING phrase, *dest-item* can be alphabetic, alphanumeric, or non-floating-point numeric. In this case, the input field is always updated to be what is on the screen. It cannot be numeric edited, alphanumeric edited, COMP-1, or COMP-2; if it is, the program will ignore the DEFAULT and CURRENT VALUE phrases.

CONTROL KEY Phrase (Format 3)

106. When you use the CONTROL KEY phrase in Format 3, the operator must terminate data input with a legal terminator key. Ctrl/Z is a legal terminator if ON EXCEPTION or AT END is specified.

107. When you do not use the CONTROL KEY phrase in Format 3, the operator can terminate data input only with RETURN or TAB.

108. When the EDITING phrase is used, the keys which invoke field editing functions do not terminate the ACCEPT statement and are not stored in *key-dest-item*.

Format 4

109. When any key other than a valid control key is entered: (1) the contents of *key-dest-item* do not change, and (2) the terminal bell rings. This occurs until a proper control key is entered.

Format 5

110. The end-of-file indication is considered a normal, successful termination.

111. The data in each field is converted and validated as you leave the field. The updated value is displayed in the field if the SECURE clause is not specified.

Conversion occurs in the following instances:

- If the field is not numeric and the JUSTIFIED clause is specified, the data is right-justified.
- If the field is numeric or numeric edited, the data is formatted according to the PICTURE, SIGN, and BLANK WHEN ZERO clauses. This formatting is always successful because only the following characters from the data are formatted:
  - 0–9
  - Period (.), if DECIMAL POINT IS COMMA is not specified
  - Comma (,), if DECIMAL POINT IS COMMA is specified
  - Sign (+, –, DB, db, CR, or cr)

  Note that only the first occurrence of the period, comma and sign is accepted; multiple occurrences are ignored. Also, to the left and right of the decimal point excess leftmost digits and excess rightmost digits are truncated, respectively.

Validation occurs when the FULL or REQUIRED clauses are specified.

112. The default value for each field is displayed when the operator enters the field for the first time during an ACCEPT statement. The default value is determined as follows:

- If the USING clause is specified, the default is the current value of the USING item.

- If the TO and FROM clauses are specified, the default is the current value of the FROM item.
- If only the TO clause is specified:
  - The default is ZEROES (or SPACES if BLANK WHEN ZERO is specified) for numeric and numeric edited items.
  - The default is SPACES for alphabetic, alphanumeric, and alphanumeric edited items.

113 If the operator types a terminator key prior to entering data in every field, the default value for each untouched field is moved to the field's destination item.

114 The operator is limited to entering the number of characters specified by the PICTURE clause. If the operator attempts to type beyond the rightmost position of the field: (1) the cursor remains on the position to the right of the rightmost position and (2) the last character of the field is overwritten with the new character.

115 There are special keys that allow the operator to edit data within a field and to move among the fields within a screen. Except where noted otherwise, the operator is allowed to move among the fields in the order in which the fields are defined within the Screen Description Entry.

The keys defined on OpenVMS Alpha and I64 systems are described in Table 6.10.

Table 6.10. SCREEN SECTION Keys for OpenVMS Alpha and I64 Systems

Key	Description
Left, Ctrl/B, Ctrl/D	Move left (if not at beginning of field, move left within field; if at beginning of field, move to previous field).
Right, Ctrl/F	Move right (if not at end of field, move right within field; if at end of field, move to next field).
Up	Move to the nearest field that is positioned above the current cursor position; this movement ignores the order in which fields are defined within the Screen Description Entry and is based simply on the location of items on the screen.
Down	Move to the nearest field that is positioned below the current cursor position; this movement ignores the order in which fields are defined within the Screen Description Entry and is based simply on the location of items on the screen.
Ctrl/P, Ctrl/L	Move to previous field.
TAB, Ctrl/N, Ctrl/I	Move to next field.
Ctrl/H, Ctrl/W	Move to beginning of line (if at beginning of line within a multiple-line field, move to beginning of previous line).
Ctrl/E	Move to end of text (if at end of text within a multiple-line field, move to end of text on next line).
Ctrl/A, Ctrl/T	Toggle insert and overstrike mode (if $ SET NOCONTROL=T).
Ctrl/K	Erase to end of line (always performed in insert mode).
Ctrl/U, Ctrl/X	Erase to beginning of line (always performed in insert mode).
CR, Ctrl/M, Ctrl/Z	Terminate input.

The keys defined on UNIX systems are described in Table 6.11.

Table 6.11. SCREEN SECTION Keys for UNIX Systems

Key	Description
Left, Ctrl/B, Ctrl/D	Move left (if not at beginning of field, move left within field; if at beginning of field, move to previous field).
Right, Ctrl/F	Move right (if not at end of field, move right within field; if at end of field, move to next field).
Up	Move to the nearest field that is positioned above the current cursor position; this movement ignores the order in which fields are defined within the Screen Description Entry and is based simply on the location of items on the screen.
Down	Move to the nearest field that is positioned below the current cursor position; this movement ignores the order in which fields are defined within the Screen Description Entry and is based simply on the location of items on the screen.
Ctrl/P, Ctrl/L	Move to previous field.
TAB, Ctrl/N, Ctrl/I	Move to next field.
Ctrl/A, Ctrl/H, Ctrl/W	Move to beginning of line (if at beginning of line within a multiple-line field, move to beginning of previous line).
Ctrl/E	Move to end of text (if at end of text within a multiple-line field, move to end of text on next line).
Ctrl/T	Toggle insert and overstrike mode.
Ctrl/K	Erase to end of line (always performed in insert mode).
Ctrl/U, Ctrl/X	Erase to beginning of line (always performed in insert mode).
CR, Ctrl/M	Terminate input.

116. The description of insert and overstrike editing for the EDITING Phrase (Format 3) also applies here.

Formats 6, 7, and 8

117. When a Format 6 ACCEPT statement is specified, the value of *arg-count* is moved to *dest-item*. This represents the number of arguments on the program run command line (see ARGUMENT-NUMBER in the SPECIAL-NAMES paragraph in Chapter 4).

118. When the current argument position indicator is zero, it refers to the zeroth command line argument, in other words the command that invoked the COBOL program.

119. When a Format 7 ACCEPT statement is specified, the value of the command line argument indicated by the current argument position indicator is moved to *dest-item* (see ARGUMENT-VALUE in the SPECIAL-NAMES paragraph in Chapter 4).

120. The current argument position indicator is determined by the following:

- In the absence of a Format 4 DISPLAY, the initial value of the current argument position indicator is 1.

- The current argument position indicator is incremented by 1 after execution of a Format 7 ACCEPT statement.

121When a Format 8 ACCEPT statement is specified, the value of *envlog-value* is moved to *dest-item* (refer to ENVIRONMENT-VALUE and ENVIRONMENT-NAME in the SPECIAL-NAMES paragraph in Chapter 4). This value represents the value of the environment variable or system logical named by the current ENVIRONMENT-NAME item.

122*stment3* is executed if an attempt is made to read beyond the last argument on the command line, or if the argument does not exist.

123*stment4* is executed if the name of the environment variable or logical has not been set by a Format 5 DISPLAY, or if the environment variable or logical does not exist.

124*stment5* is executed if the exception condition does not exist.

Technical Notes

All Formats

1. On OpenVMS systems, if the data transfer is from a terminal, Ctrl/Z is equivalent to an end-of-file indication.

2. On UNIX systems, if the data transfer is from a terminal, Ctrl/D is equivalent to an end-of-file indication.

Format 1

3. An ACCEPT statement without the FROM phrase takes input from the default input device (the keyboard). To take input from a file on UNIX systems, the environment variable COBOL_INPUT can be used to specify a text file containing input data. To take input from a file on OpenVMS systems, the logical COB$INPUT or SYS$INPUT can be used to specify a text file containing input data.

   Alternatively, input device redirection (<) can be used on UNIX systems to name an input file.

4. An ACCEPT statement that includes the FROM phrase transfers data from the *file-device-name* associated with the SPECIAL-NAMES paragraph description of *input-source*.

5. On OpenVMS systems, the object of a logical name is not necessarily a device. Therefore, no open mode is implied. As a result, *input-source* can be associated with any *device-name* in the SPECIAL-NAMES paragraph. For example, *input-source* can refer to PAPER-TAPE-PUNCH as well as PAPER-TAPE-READER.

6. An end-of-file indication during ACCEPT statement execution with an AT END phrase causes control to transfer to the AT END imperative statement.

7. An end-of-file indication during ACCEPT statement execution without the AT END phrase is an error. The program terminates abnormally.

8. The ACCEPT statement fills *dest-item* with spaces if the input is an empty record (for example, a carriage return only).

9. On UNIX, you can enter a maximum of 256 characters during a Format 1 ACCEPT statement.

Formats 3 and 4

10. The VSI extensions to the ACCEPT and DISPLAY statements support data input and display only on VT100 and later terminal types, including emulators of these terminal types.

11. On OpenVMS Alpha and I64 systems, control sequences from SMGTERMS.TXT are used to accomplish cursor positioning, screen erasure, and video attributes. Refer to the chapter on support for non VSI terminals of the OpenVMS RTL Screen Management (SMG$) Manual if you wish to customize SMGTERMS.TXT.

12. You should accept data only 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 an input field that positions the cursor outside screen boundaries, the result is not an error condition, but your program might produce unexpected results.

    Values for screen boundaries depend on the terminal attributes. Refer to the appropriate terminal user's guide for more information on screen boundaries.

13. 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 has 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 operation (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.

14. When you use the CONTROL KEY phrase, *key-dest-item* stores the terminator key code. The *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/] contains information on these key code values in its description of programming video forms.

15. In Formats 3 and 4, the maximum number of characters in *dest-item* or *key-dest-item* is 1024.

16. When you use the CONTROL KEY phrase with the PROTECTED WITH AUTOTERMINATE phrase, and the maximum number of characters is entered to terminate the ACCEPT statement, *key-dest-item* is filled with spaces.

17. The ALPHABET clause has no effect on the CONVERSION clause for either ACCEPT or DISPLAY.

18. Unexpected behavior can occur when an ACCEPT statement with the EDITING, PROTECTED, and DEFAULT IS CURRENT phrases and without the CONVERSION phrase is executed. The behavior occurs when a numeric data item has a negative scale factor or is signed. To avoid this behavior, it is suggested that the CONVERSION phrase be used in these circumstances.

Examples

In the following examples, the character s represents a space. The examples assume that the time is just after 2:15 *p.m.* on October 7, 1992. 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.
01      ITEMG   COMP-1.
01      ITEMH   PIC S9(5) COMP.
```

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

3. `ACCEPT ITEMB WITH CONVERSION.`

Input	ITEMB	Equivalent to
1623	1623	16.23
4	4sss	04.00
60000	6000	60.00
-1.2	-1.2	01.20
1.23	1.23	01.23

Input	ITEMB	Equivalent to
COMPUTER	COMP	Invalid data

STATEMENT	RESULT
`ACCEPT ITEME FROM DATE.`	ITEME = 921007
`ACCEPT ITEMC FROM TIME.`	ITEMC = 14150516 (OpenVMS and UNIX)
`ACCEPT ITEMD FROM DAY.`	ITEMD = 92280
`ACCEPT ITEMF FROM DAY-OF-WEEK.`	ITEMF = 3
`ACCEPT ITEMA FROM TIME.`	ITEMA = 141505
`ACCEPT ITEME FROM TIME.`	ITEME = 150516
`ACCEPT ITEMD FROM DAY-OF-WEEK.`	ITEMD = 00003
`ACCEPT ITEMG WITH CONVERSION.`	

Input	Result Equivalent to
.123E-02	0.00123
-12.3E+02	-1230
1004E-07	1.004000E-04

4. `ACCEPT ITEMH WITH CONVERSION.`

Input	Result Equivalent to
27	27
-44	-44

Additional examples containing VSI extensions to the ACCEPT statement (Formats 3, 4, and 5) are described in the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/]. Refer to the description of programming video forms.

Also, examples containing extensions to the ACCEPT statement (Formats 6, 7 and 8) that access command line arguments are described in the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/].

Additional References

- SPECIAL-NAMES section in Chapter 4
- *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/] sections on using command-line arguments and environment variables
- Section 6.1.4: Scope of Statements
- DISPLAY
- MOVE statement

ADD

ADD — The ADD statement adds two or more numeric operands and stores the sum in one or more receiving fields.

General Format

Format 1

```
ADD { num } . . . TO { rsult [ ROUNDED ] } . . .

  [ ON SIZE ERROR stment ]
  [ NOT ON SIZE ERROR stment2 ]
  [ END-ADD ]
```

Format 2

```
ADD num . . . TO { num } GIVING { rsult [ ROUNDED ] } . . .

  [ ON SIZE ERROR stment ]
  [ NOT ON SIZE ERROR stment2 ]
  [ END-ADD ]
```

Format 3

```
ADD { CORRESPONDING } grp-1 TO grp-2 [ ROUNDED ]
    { CORR        }

  [ ON SIZE ERROR stment ]
  [ NOT ON SIZE ERROR stment2 ]
  [ END-ADD ]
```

`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.

`stment`

is an imperative statement executed when a size error condition has occurred.

`stment2`

is an imperative statement executed when no on size error condition has occurred.

`grp-1`

is the identifier of numeric group item.

`grp-2`

is the identifier of numeric group item.

Syntax Rule

CORR is an abbreviation for CORRESPONDING.

General Rules

1. In Format 1, the values of the operands before the word TO are added together. This total is then added to each occurrence of *rsult*.

2. In Format 2, the values of the operands before the word GIVING are added. The sum is then stored in each *rsult*.

3. In Format 3, data items in *grp-1* are added to and stored in the corresponding data items in *grp-2*.

Examples

Each of the examples assume the following data descriptions and initial values:

INITIAL VALUES

```
03  ITEMA  PIC 99 VALUE 85.                85
03  ITEMB  PIC 99 VALUE 2.                 2
03  ITEMC  VALUE "123".
    05  ITEMD  OCCURS 3 TIMES              1 2 3
            PIC 9.
```

1. TO phrase: RESULTS

```
ADD 2 ITEMB TO ITEMA.                          ITEMA = 89
```

2. SIZE ERROR clause:

```
ADD 38 TO ITEMA ITEMB                          ITEMA = 85
                                               ITEMB = 40
  ON SIZE ERROR
    MOVE 0 TO ITEMB.                           ITEMB = 0
```

(When the SIZE ERROR condition occurs, the value of the affected resultant identifier does not change. The SIZE ERROR condition occurs on ITEMA but not on ITEMB.)

3. NOT ON SIZE ERROR clause:

```
ADD 14 TO ITEMA                                ITEMA = 99
   ON SIZE ERROR
       MOVE 0 TO ITEMB.
   NOT ON SIZE ERROR
       MOVE 1 TO ITEMB.                        ITEMB = 1
```

(If the SIZE ERROR condition had occurred, the value of ITEMA would have been 85 and ITEMB would have been 0.)

4. Multiple receiving fields:

```
ADD 1 TO ITEMB ITEMD (ITEMB).                  ITEMB = 3
                                               ITEMD (3) = 4
```

(The operations proceed from left to right. Therefore, the subscript for ITEMD is evaluated after the addition changes its value.)

5. GIVING phrase:

```
ADD ITEMB ITEMD (ITEMB) GIVING ITEMA.          ITEMA = 4
```

6. END-ADD:

```
IF ITEMB < 10
  ADD 7 ITEMB TO ITEMD (ITEMB)                    ITEMD (2) = 2
    ON SIZE ERROR
      MOVE 0 TO ITEMB                             ITEMB = 0
  END-ADD
  ADD 1 TO ITEMB.                                 ITEMB = 1
```

   (The first ADD terminates with END-ADD. If the SIZE ERROR condition had not occurred, the second ADD statement would have executed anyway; the value of ITEMB would have been 3.)

Additional References

- Section 6.1.4: Scope of Statements
- Section 6.6.1: Arithmetic Operations
- Section 6.6.2: Multiple Receiving Fields in Arithmetic Statements
- Section 6.6.3: ROUNDED Phrase
- Section 6.6.4: ON SIZE ERROR Phrase
- Section 6.6.5: CORRESPONDING Phrase
- Section 6.6.7: Overlapping Operands and Incompatible Data

ALTER

ALTER — The ALTER statement changes the destination of a GO TO statement.

General Format

ALTER {proc TO [PROCEED TO] new-proc } ...

Description

proc

is the name of a paragraph that contains one sentence: a GO TO statement without the DEPENDING phrase.

new-proc

is a procedure-name.

General Rules

1. The ALTER statement changes the destination of the GO TO statement in *proc*.
2. When the changed GO TO executes, it transfers control to *new-proc* instead of the procedure it previously referred to.

   However, when the GO TO statement is in an independent segment (segment-number 50 to 99), the GO TO statement could return to its initial state under some circumstances.

3. A GO TO statement in a section with a segment-number greater than 49 cannot be changed by an ALTER statement in a section with a different segment-number.

Examples

The examples assume the following Procedure Division code:

```
PROC-AA.
        DISPLAY "PROC-A".
PROC-A.
        GO TO PROC-BB.
PROC-BB.
        DISPLAY "PROC-B".
PROC-B.
        GO TO PROC-DD.
PROC-CC.
        DISPLAY "PROC-C".
PROC-C.
        GO TO PROC-FF.
PROC-DD.
        DISPLAY "PROC-D".
PROC-D.
        GO TO PROC-CC.
PROC-EE.
        DISPLAY "PROC-E".
PROC-E.
        GO TO.
PROC-FF.
        DISPLAY "PROC-F".
PROC-F.
        EXIT.
```

1. As written.

   Output
   PROC-A
   PROC-B
   PROC-D
   PROC-C
   PROC-F

2. `ALTER PROC-A TO PROC-EE PROC-E TO PROC-CC.`

   Output
   PROC-A
   PROC-E
   PROC-C
   PROC-F

3. `ALTER PROC-D TO PROC-EE PROC-C TO PROC-AA.`

   Output
   PROC-A
   PROC-B
   PROC-D
   PROC-E
   error at PROC-E

Additional References

- Section 6.7: Segmentation
- GO TO statement

CALL

CALL — The CALL statement transfers control to another program in the executable image.

General Format

```
CALL prog-name
  [ USING
      { { { { [ BY REFERENCE ]
              BY CONTENT
              BY DESCRIPTOR (OpenVMS)
              BY VALUE } { arg } ... } }
        OMITTED }
      [ { { BY REFERENCE
            BY CONTENT
            BY DESCRIPTOR (OpenVMS)
            BY VALUE } { arg } ... } ] ...
      OMITTED ]
  [ GIVING function-res ]
  { [ ON EXCEPTION stment ] [ NOT ON EXCEPTION stment2 ]
    [ ON OVERFLOW stment ] [ NOT ON OVERFLOW stment2 ] }
[ END-CALL ]
```

`prog-name`

is a nonnumeric literal or the identifier of an alphanumeric data item. 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, or it is a nonnumeric literal. It must not be a function-identifier.

`function-res`

is the identifier of an elementary integer numeric data item with COMP, COMP-1, or COMP-2 usage and no scaling positions. *function-res* can be subscripted, and it can be qualified. When control returns to the calling program, *function-res* can contain a function result.

`stment`

is an imperative statement executed for an on exception or an overflow condition.

stment2

is an imperative statement executed for a not on exception or a not on overflow condition.

Syntax Rules

1. *prog-name* must be from 1 to 31 characters long. It can contain the characters "A" to "Z", "a" to "z", "0" to "9", and hyphen (-), dollar sign ($), and underline (_).
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, VALUE, CONTENT, or, for 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 a USING phrase is in the Procedure Division header of the called program. Both USING phrases must have the same number of arguments.
8. If *arg* is a nonnumeric literal, only BY REFERENCE, BY CONTENT, or for OpenVMS systems, BY DESCRIPTOR can be used.
9. OMITTED, a reserved word, indicates the absence of a specific argument. OMITTED does not change the default argument-passing mechanism; it generates BY VALUE 0 for the omitted argument.
10. If the argument-passing mechanism is BY VALUE, *arg* must be either: (a) an integer numeric literal in the range -2**31 to +2**31-1, (b) a COMP-1 data item, or (c) a word or longword integer COMP data 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, the contents of *prog-name* are interpreted as follows:
   - Hyphens are treated as underline characters.
   - Lowercase letters are treated as uppercase (See the *Technical Notes* relating to case sensitivity later in this section).
   - Leading and trailing spaces and tab characters are ignored.
3. The CALL statement transfers control to the called program.
4. Two or more programs in the run unit can have the same *prog-name*. The scope of names conventions for program-names resolve the CALL statement references to duplicate *prog-names*. (See the section on Conventions for Resolving Program-Name References.)
5. If *prog-name* is an identifier, the CALL statement can transfer control only to VSI COBOL for OpenVMS programs.

6. The ON EXCEPTION phrase is interchangeable with the ON OVERFLOW phrase.

7. If *prog-name* is not in the executable image and there is an ON EXCEPTION phrase, any NOT ON EXCEPTION phrase is ignored, *stment* executes, and control is transferred to the end of the CALL statement.

8. If *prog-name* is in the executable image, and there is an ON OVERFLOW phrase or ON EXCEPTION phrase, both phrases are ignored. Control is transferred either to the end of the CALL statement or, if NOT ON EXCEPTION is specified, to *stment2*. After *stment2* executes, control is transferred to the end of the CALL statement.

9. If *prog-name* is not in the executable image and there is no ON EXCEPTION phrase, an error condition exists; the program terminates abnormally.

10. If the called program does not have the initial attribute, it, and each program directly or indirectly contained in it, is in its initial state: (a) the first time it is called in an image, and (b) the first time it is called after a CANCEL to the called program.

    On all other entries, the state of the called program is the same as when it was last exited. The program state includes internal data.

11. If the called program has the initial attribute, it, and each program directly or indirectly contained in it, is in its initial state every time it is called.

12. Files associated with a called program's internal file connectors are not in the open mode:

    - The first time the program is called
    - The first time the program is called after execution of a CANCEL statement referring to the program
    - Every time the program is called, if it has the initial attribute

    On all other entries, the status and positioning of such files in a called program are the same as when the program was last exited.

13. The process of calling a program or exiting from a called program does not alter the status or positioning of a file associated with any external file connector.

14. The arguments' order of appearance in the USING phrases of the CALL statement and the called program's Procedure Division header determine correspondence between the data-names used by the calling and called programs. Data-names correspond by position in the USING phrase, not by name.

    No correspondence 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.

15. The arguments in the CALL statement USING phrase are made available to the called program when the CALL executes.

16. Called programs can contain CALL statements. However, a called program must not execute a CALL statement that directly or indirectly calls the calling program.

17. The CALL statement can make data available to the called program by four 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.
- CONTENT—The address of a temporary data item that contains the contents of *arg* is passed to the called program.
- On OpenVMS, DESCRIPTOR—The address of (pointer to) the data item's descriptor is passed to the called program.

  The parameter-passing mechanism BY DESCRIPTOR is not supported on UNIX systems, because the UNIX calling standard does not define such a mechanism. Programs that include BY DESCRIPTOR will receive a compile-time diagnostic.
- VALUE—The value of *arg* is passed to the called program. If *arg* is a data-name, its description in the Data Division can be either:
  - COMP usage with no scaling positions; the picture can specify no more than nine digits
  - COMP-1 usage

Note that OMITTED, an VSI COBOL for OpenVMS reserved word, is equivalent to BY VALUE 0 and can be used in place of that BY VALUE argument-passing mechanism.

18. If the called program is a COBOL program, the CALL statement can pass arguments only BY REFERENCE or BY CONTENT. 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.
19. 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 are always identical with the contents of the parameter in the called program.
20. If the BY CONTENT phrase is either specified or implied for a parameter, a copy of *arg* is moved to a temporary memory location, and the address of the temporary memory location is passed to the called program. This mechanism ensures that the called program cannot change the original contents of *arg*. However, the called program can change the value of the temporary memory location.
21. The data description of each *arg* in the calling program must be identical to each *arg* in the called program. The compiler does not convert, extend, or truncate any *arg* passed to a called program.
22. On Alpha and I64 systems, if the GIVING phrase of the CALL statement is not specified, the function result is made available in the RETURN-CODE special register when control returns to the calling program.
23. If the GIVING phrase is specified, the function result is made available in *function-res* when control returns to the calling program.

Technical Notes

- On Alpha and I64 systems, because the reserved word RETURN-CODE is one of the X/Open reserved words, you cannot use the `reserved words` compiler option with the `noxopen` setting if you want to use the RETURN-CODE special register.

For more information on the relationship between the GIVING phrase and the RETURN-CODE special register, see Table 6.7 in this chapter.

- On UNIX systems, the linker is case sensitive, whereas the COBOL language is case insensitive. When *prog-name* in a CALL statement is a literal, and you are calling a program in a case-sensitive language (such as C), you might need to use a form of the `names` option when you compile.

  If you do not specify the `names` option on the command line, the default setting is `lowercase`, which causes the VSI COBOL for OpenVMS compiler to force all external names to be lowercase. Hence, there is no problem when you call a C program whose name contains no uppercase letters. If the name consists of all uppercase letters, use the `uppercase` setting of the `names` option. If it is mixed-case (for example, "Cprog") use the `as_is` setting. When you use `names as_is`, only literals in the CALL program name are affected.

- On OpenVMS, calls to the OpenVMS RTL routines LIB$ESTABLISH and LIB$REVERT, which are used to establish and cancel a condition handler, respectively, are handled specially by the VSI COBOL for OpenVMS compiler. Calls to these routines are intercepted and processed by the VSI COBOL for OpenVMS compiler at compile time (not runtime).

- CALL identifier (CALL data name) requires that all modules be specified to link the run unit. Since there are no link-time references to routines to be called with CALL identifier, the linkers on OpenVMS and UNIX do not resolve these references at link time. Instead, the references are dynamically resolved at run-time using modules which have been explicitly linked into the run unit.

Examples

1. Passing arguments by reference:

```
CALL "DATERTN" USING ITEMA ITEMB ITEMC.
```

2. On OpenVMS, mixing argument-passing mechanisms: Reference arguments are ITEMA, ITEMD, and “PAYROLL”. Descriptor arguments are ITEMB, ITEMC, ITEMD, “TOTALS”, and ITEMF. The value arguments are ITEME and “995.99”. ITEMD is passed twice—by reference and by descriptor. The content arguments are ITEMG and “SUMMARY FLAG”.

```
CALL "NEWPROG" USING ITEMA
  BY DESCRIPTOR ITEMB ITEMC "TOTALS"
  BY REFERENCE ITEMD "PAYROLL"
  BY VALUE ITEME 995.99
  BY DESCRIPTOR ITEMD ITEMF
  BY CONTENT ITEMG "SUMMARY FLAG".
```

3. Mixing argument-passing mechanisms: Reference arguments are ITEMA, ITEMD, and “PAYROLL”. The value arguments are ITEME and “995.99”. The content arguments are ITEMG and “SUMMARY FLAG”.

```
CALL "NEWPROG" USING ITEMA
  BY REFERENCE ITEMD "PAYROLL"
  BY VALUE ITEME 995.99
  BY CONTENT ITEMG "SUMMARY FLAG".
```

4. Calling a program whose name is selected at run time:

```
MOVE "PROG009" TO PROG-TO-CALL.
.
```

```
.
.
CALL PROG-TO-CALL USING ITEMA.
```

5. Receiving a function result:

```
CALL "PROG010" USING ITEMA ITEMB "XYZ"
  GIVING ITEMC.
```

Additional References

- Section 6.8: General Formats and Rules for Statements
- Chapter 3
- Section 6.1.4: Scope of Statements
- Subsection Section 6.2.6.1: Conventions for Resolving Program-Name References in Section 6.2.6: Scope of Names
- *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/] chapter on interprogram communication, including `cobcall, cobcancel,` and `cobfunc` references

Refer to the *VSI OpenVMS Calling Standard* for more information.

CANCEL

CANCEL — The CANCEL statement returns the named program to its initial state.

General Format

CANCEL {prog-name} ...

`prog-name`

is a nonnumeric literal or the identifier of an alphanumeric data item. It contains the program-name of the program to be canceled.

Syntax Rules

1. *prog-name* must be from 1 to 31 characters long. It can contain the characters A to Z, a to z, 0 to 9, dollar sign ($), hyphen (-), and underline (_).
2. *prog-name* must be the name of an VSI COBOL for OpenVMS program.

General Rules

1. Two or more programs in the run unit can have the same *prog-name*. The scope of names conventions for *prog-names* resolve the CANCEL statement references to duplicate *prog-names*. (See the section on Conventions for Resolving Program-Name References.)
2. Using the scope of names conventions (See Section 6.2.6), if *prog-name* is called again after the CANCEL statement successfully executes, *prog-name*, and all programs contained within it, are in their initial state.

3. *prog-name* must be callable by the program that contains the CANCEL statement.

4. The program named by *prog-name* must not refer directly or indirectly to any program that: (a) has been called, and (b) has not yet executed an EXIT PROGRAM statement.

5. When the CANCEL statement executes, the contents of *prog-name* are interpreted as follows:

   - Hyphens are treated as underline characters.
   - Lowercase letters are treated as uppercase.
   - Leading and trailing spaces and tab characters are ignored.

6. A called program can be canceled in three ways:

   - By being named in a CANCEL statement
   - When the executable image ends
   - When an EXIT PROGRAM statement executes if the program has the initial attribute

7. When canceling a program these items do not change: (a) the contents of its data items in external data records, and (b) the status and positioning of a file associated with any external file connector.

8. During the execution of a CANCEL statement, an implicit CLOSE statement without any optional phrases executes for each file in the open mode that is associated with an internal file connector in *prog-name*.

Examples

1. `CANCEL “PROG10”.`

2. `CANCEL THE-PROG.`

3. `CANCEL SUB-PROG-A “PROG12” SUB-PROG-B.`

Additional References

- Chapter 3
- Section 6.1.4: Scope of Statements
- Section 6.2.6: Scope of Names
- CALL statement
- *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/] `cobcall, cobcancel, cobfunc` references

CLOSE

CLOSE — The CLOSE statement ends processing of reels (or units) and files. It can also perform rewind, lock, and removal operations.

General Format

Format 1—Sequential or Line Sequential File

```
CLOSE { file-name [ { REEL } [ FOR REMOVAL ]
                    { UNIT } [ WITH NO REWIND ] ]
                  [ WITH { NO REWIND } ] ]
                         { LOCK     } } ...
```

Format 2—Relative or Indexed File (Alpha, I64)

CLOSE {file-name [WITH LOCK]}...

`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 and line sequential files.
2. The words REEL and UNIT are equivalent.

General Rules

1. A CLOSE statement can execute only for an open file.
2. Executing a CLOSE statement updates the value of the FILE STATUS data item associated with the file.
3. The TERMINATE statement must be executed before a CLOSE statement can reference a report file.
4. If an optional file is not present, standard end-of-file processing does not occur.
5. 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, except as specified in General Rule 2.
6. 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.
7. 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.
8. 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.
9. 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*.
10. In the file-sharing environment, CLOSE statement execution unlocks all locks for *file-name*.

11. If both the REEL/UNIT and WITH NO REWIND phrases are specified in the same CLOSE statement, the WITH NO REWIND phrase is ignored.

12. 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, a terminal device)
    - *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

13. For files specified with a MULTIPLE FILE TAPE clause the NO REWIND phrase, if any, is ignored.

14. Table 6.12 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 6.12. 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	C,H	C,B	C,B,A	X
CLOSE REEL	F	F,G	F,G	X
CLOSE REEL FOR REMOVAL	F	F,D,G	F,D,G	X

- 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.

- No rewind of current reel

  The position of the current reel remains the same.

- Close file

  The file is closed.

- Reel/unit removal

The current reel rewinds and is logically removed from the run unit However, the run unit can access the reel again in its proper order of reels in the file. To do this, the executable image must subsequently execute the following:

- A CLOSE statement without the REEL/UNIT phrase for the file
- An OPEN statement for the file

- File lock

  The executable image cannot open the file again in its current execution.

- 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 File Position Indicator denotes 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 File Position Indicator points to the next record in the file. If there are no records for the current volume, another reel swap occurs.

  For output files (reel/unit media), 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.

  For output files (nonreel/unit media), execution of this statement is considered successful. The file remains in the open mode and no action takes place, except as specified in General Rule 2.

- Rewind

  The current reel (or device) is positioned to its physical beginning.

- Optional phrases ignored

  The CLOSE statement is executed as if none of the optional phrases are present.

- Invalid

  This is an invalid combination of CLOSE option and file category. It results in FILE STATUS data item value 30.

Technical Note

CLOSE statement execution can result in these FILE STATUS data item values:

File Status	Meaning
00	Successful
07	CLOSE statement with NO REWIND, REEL/UNIT, or FOR REMOVAL phrase referenced a file on a nonreel/unit medium
30	Any other CLOSE error
42	File never opened, already closed, or not currently open

Additional Reference

See Section 6.6.8: I-O Status for more information.

COMPUTE

COMPUTE — The COMPUTE statement evaluates an arithmetic expression and stores the result in one or more data items.

General Format

```
COMPUTE { rsult [ ROUNDED ] } . . . { = | EQUAL } arithmetic-expression
    [ ON SIZE ERROR stment ]
    [ NOT ON SIZE ERROR stment2 ]
    [ END-COMPUTE ]
```

`rsult`

is the identifier of an elementary numeric item or elementary numeric edited item. It is the resultant identifier.

`arithmetic-expression`

is an expression as described in Section 6.4: Arithmetic Expressions.

`stment`

is an imperative statement executed when a size error condition has occurred.

`stment2`

is an imperative statement executed when no size error condition has occurred.

General Rules

1. The arithmetic expression is evaluated. Its value then replaces the current value of each occurrence of *rsult*, from left to right.

2. If the *arithmetic-expression* consists of a single identifier or literal, the COMPUTE statement behaves like a MOVE statement with the single identifier or literal acting as the source operand and each result operand acting as a destination operand.

3. For any *rsult* specification that includes the word rounded, the value of the expression is rounded before being moved to *rsult*.

Examples

Each of the examples assume these data descriptions and initial values:

INITIAL VALUES

```
03  ITEMA  PIC 999V99 VALUE 2.                      2.00
03  ITEMB  PIC 999V99 VALUE 3.                      3.00
03  ITEMC  PIC 999V99 VALUE 4.                      4.00
03  ITEMD  PIC 999V99 VALUE 5.                      5.00
```

RESULTS

1. No rounding:

```
COMPUTE ITEMC =                              ITEMC = 2.82
   (ITEMA + 6) ** (.1 * ITEMD).
```

2. With rounding:

```
COMPUTE ITEMC ROUNDED =                      ITEMC = 2.83
   (ITEMA + 6) ** (.1 * ITEMD).
```

3. The ON SIZE ERROR phrase:

```
COMPUTE ITEMB = (ITEMA * ITEMD) ** 3         ITEMB = 3.00
   ON SIZE ERROR
     MOVE 100 TO ITEMC.                      ITEMC = 100.00
```

4. The NOT ON SIZE ERROR phrase:

```
COMPUTE ITEMB = (ITEMA * ITEMD) ** 2         ITEMB = 100.00
   ON SIZE ERROR
     MOVE 100 TO ITEMC
   NOT ON SIZE ERROR
     MOVE 200 TO ITEMC.                      ITEMC = 200.00
```

Additional References

- Section 6.1.4: Scope of Statements
- Section 6.6.1: Arithmetic Operations
- Section 6.6.2: Multiple Receiving Fields in Arithmetic Statements
- Section 6.6.3: ROUNDED Phrase
- Section 6.6.4: ON SIZE ERROR Phrase
- Section 6.6.7: Overlapping Operands and Incompatible Data

CONTINUE

CONTINUE — The CONTINUE statement indicates that no executable statement is present. It causes an implicit control transfer to the next executable statement.

General Format

CONTINUE

Syntax Rule

The CONTINUE statement can be used wherever a conditional or imperative statement can be used.

General Rule

The CONTINUE statement causes an implicit control transfer to the next executable statement.

Example

```
READ FILE-A
  INVALID KEY
    CONTINUE.
MOVE ...
```

This example shows how CONTINUE can replace an INVALID KEY imperative statement. Control passes to the MOVE statement whether or not the INVALID KEY condition occurs.

DELETE

DELETE — The DELETE statement logically removes a record from a mass storage file.

General Format

```
DELETE file-name RECORD

    [ INVALID KEY stment ]
    [ NOT INVALID KEY stment2 ]
    [ END-DELETE ]
```

`file-name`

is the name of a relative or indexed file described in the Data Division. It cannot be the name of a sequential or line sequential file or a sort or merge file.

`stment`

is one or more imperative statements executed for an invalid key condition.

`stment2`

is one or more imperative statements executed for a not invalid key condition.

Syntax Rules

1. There cannot be an INVALID KEY phrase or a NOT 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. The I/O system logically removes the record that the READ statement accessed.

3. For a relative file in random or dynamic access mode, the I/O system 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 access mode, the I/O system (logically removes the record identified by the file's primary record key data item. If the file does not contain that record, an invalid key condition exists.

5. For an indexed file in dynamic access mode, the behavior depends on the DUPLICATES phrase of the RECORD KEY clause of the SELECT statement. If the primary key allows duplicates, Rule 2 applies. If the primary key does not allow duplicates, Rule 4 applies.

6. After successful DELETE statement execution, the identified record has been logically removed from the file. It is no longer accessible.

7. 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.

8. For sequential access files, DELETE statement execution does not affect the File Position Indicator.

9. For dynamic access files, the File Position Indicator can point to the record to be deleted before the DELETE statement executes. In this case, once the DELETE statement executes, the File Position Indicator:

   - 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

   In all other cases, the File Position Indicator is not affected by the execution of a DELETE statement.

10. DELETE statement execution updates the value of the FILE STATUS data item for the file.

11. 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 the first character of 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 run unit terminates abnormally.

See the rules for the INVALID KEY phrase, Section 6.6.10.

Technical Notes

- In a VSI standard file-sharing environment for relative files, records that are manually locked and then deleted retain a lock on the deleted record. This affects a subsequent WRITE from a different access stream (either from a different file connector in the same run-unit or from a file connector in a different run-unit). A WRITE statement under such conditions gets a file status 92 (record locked by another program). All other statements will treat a record that was locked and subsequently deleted in the same manner as a record that was not locked and subsequently deleted.

- DELETE statement execution can result in the following FILE STATUS data item values:

File Status	Access Method	Meaning
00	All	Successful
23	Rand, Dyn	Record not in file (invalid key)
43	Seq	No previous READ or record not locked by prior READ or START
49	All	File not open, or incompatible open mode
92	All	Record locked by another program
30	All	All other permanent errors

- If the current record to be deleted is not locked by the current stream, the delete results in a permanent error. On OpenVMS systems, RMS-STS indicates the record is not locked.

Additional References

- Section 6.1.4: Scope of Statements
- Section 6.6.8: I-O Status
- Section 6.6.10: INVALID KEY Phrase
- OPEN statement
- USE statement

DISPLAY

DISPLAY — The DISPLAY statement transfers low-volume data from the program to the default system output device or to the object of a mnemonic-name. The WITH CONVERSION phrase in Format 1 contains a VSI extension to the DISPLAY statement. The VSI extensions to Formats 2 and 3 are COBOL language additions that facilitate video forms design and data handling. Format 4 sets a program variable to the current command line argument number (to read with a Format 7 ACCEPT), Format 5 sets the name of an environment variable or system logical, and Format 6 sets the value of an environment variable or system logical.

General Format

Format 1

```
DISPLAY
{ src-item [ { | WITH CONVERSION | } ] } ... [ END-DISPLAY (Alpha, I64) ]
  [        { | UPON output-dest  |
           { | WITH NO ADVANCING |
```

Format 2

```
DISPLAY
{ src-item [ { AT LINE NUMBER
               { line-num
               { line-id [ PLUS [ plus-num ] ]
               { PLUS [ plus-num ]
             AT COLUMN NUMBER
               { column-num
               { column-id [ PLUS [ plus-num ] ]
               { PLUS [ plus-num ]
             ERASE [ TO END OF ] { SCREEN
                                 { LINE
             WITH BELL
             UNDERLINED
             BOLD
             WITH BLINKING
             REVERSED
             WITH CONVERSION
             WITH NO ADVANCING } ] } ...
[ END-DISPLAY (Alpha, I64) ]
```

Format 3 (Alpha, I64)

```
DISPLAY screen-name
    [ AT { LINE NUMBER { line-id
                       { line-num
         COLUMN NUMBER { column-id
                       { column-num } } ]
[ END-DISPLAY ]
```

Format 4 (Alpha, I64)

```
DISPLAY arg-position UPON argument-number
[ END-DISPLAY ]
```

Format 5 (Alpha, I64)

```
DISPLAY envlog-name UPON environment-name
[ END-DISPLAY ]
```

Format 6 (Alpha, I64)

```
DISPLAY envlog-value UPON environment-value [ { ON EXCEPTION stment
                                              NOT ON EXCEPTION stment2 } ]
[ END-DISPLAY ]
```

src-item

is a literal or the identifier of a data item. The literal can be any figurative constant.

arg-position

is a literal or identifier that specifies the desired argument position (on the run command line). It must be an unsigned integer.

argument-number

is a mnemonic name associated with ARGUMENT-NUMBER in the SPECIAL-NAMES paragraph in the Chapter 4, representing the current argument position.

environment-name

is a mnemonic name associated with ENVIRONMENT-NAME in the SPECIAL-NAMES paragraph in the Chapter 4, representing the name of an environment variable or system logical.

environment-value

is a mnemonic name associated with ENVIRONMENT-VALUE in the SPECIAL-NAMES paragraph in the Chapter 4, representing the contents of the variable associated with the ENVIRONMENT-NAME.

envlog-name

references an alphanumeric data item, or is a nonnumeric literal.

envlog-value

references an alphanumeric data item, or is a nonnumeric 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.

line-id

is the identifier of a data item that provides a line position on the terminal screen. It must be a positive integer; it cannot be zero.

`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.

`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.

`column-id`

is the identifier of a data item that provides a column position on the terminal screen. It must be a positive integer; it cannot be zero.

`screen-name`

is the name of a screen item defined in the SCREEN SECTION of the program.

`stment`

is an imperative statement executed if an exception condition exists; for Format 6, this means the name of the environment variable or logical has not been set by DISPLAY, or not enough space can be allocated to store the environment variable or logical.

`stment2`

is an imperative statement executed if the exception condition does not exist.

Syntax Rules

All Formats

1. In a DISPLAY statement, the number of *src-item* entries cannot exceed 254.
2. Each DISPLAY phrase can be specified only once for any *src-item*.

Formats 1 and 2

3. The WITH NO ADVANCING phrase can be specified only once per DISPLAY statement. It must be specified last (or just preceding END-DISPLAY, if used) if multiple *src-item* entries or options are specified in the statement.

Format 1

4. The UPON phrase can be specified only once per DISPLAY statement.

General Rules

Formats 1 and 2

1. The UPON and WITH NO ADVANCING phrases apply to all instances of *src-item*.

2. All phrases other than UPON and WITH NO ADVANCING apply to the immediately preceding *src-item* only.

3. If there is a WITH NO ADVANCING phrase, the DISPLAY statement does not transfer any device positioning information after the last *src-item* value.

CONVERSION Phrase (Formats 1 and 2)

4. The CONVERSION phrase allows you to display data in a field and achieve data conversion, sign, and decimal point placement. How the CONVERSION phrase affects data handling depends on the category of *src-item*. (Numeric data can be described by any USAGE clause.)

5. Numeric items do not require the CONVERSION phrase to be displayed correctly with conversion if you specify /DISPLAY_FORMATTED (on OpenVMS Alpha and I64) or `-display_formatted` (on UNIX) when you compile. (For more information on the qualifier, refer to the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/].)

6. The CONVERSION phrase or the /DISPLAY_FORMATTED (OpenVMS Alpha and I64) qualifier displays nonnumeric items and numeric edited items without change.

7. The CONVERSION phrase or the /DISPLAY_FORMATTED (OpenVMS Alpha and I64) qualifier displays non-floating-point numeric items 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.

   - If the sign is negative, a minus sign (-) is displayed. If the sign is positive, a space character is displayed. If the item is unsigned, no sign position is displayed.

   - Items with DISPLAY usage (for example, PIC 99, or PIC S99V99) are displayed after including a space, when needed, for a decimal point or sign or both.

     If you specify the SIGN TRAILING SEPARATE clause for the data item, the sign is displayed as a trailing sign. Otherwise, the sign is displayed as a leading sign.

   - Items other than DISPLAY (for example, PIC 99 COMP, or PIC S9V999 COMP SYNC) are displayed after conversion to DISPLAY usage SIGN LEADING.

   - Nonprinting numeric values are not human-readable on the terminal display unless the CONVERSION phrase or the /DISPLAY_FORMATTED qualifier (on OpenVMS Alpha and I64) or `-display_formatted` (on UNIX) is specified.

8. On OpenVMS, the CONVERSION phrase or /DISPLAY_FORMATTED (OpenVMS Alpha, I64), or `-display_formatted` (on UNIX), displays floating-point items as follows:

   - The floating-point data item is converted from internal floating point to E-notation representation. (E-notation consists of a mantissa and, optionally, an exponent.)

     The E-notation is described as follows:

Size of mantissa	COMP-1 has 7 digits (F-floating, S-floating) plus decimal point and sign

	COMP-2 has 15 digits (G-floating, T-floating) or 16 digits (D-floating) plus decimal point and sign
Sign of mantissa	Sign shown only for negative, space if positive
Form of mantissa	d.dddddd for COMP-1 d.dddddddddddddd for COMP-2 (G-floating, T-floating) d.dddddddddddddd for COMP-2 (D-floating) “.” replaced by “,” if DECIMAL POINT IS COMMA
Size of exponent	2 decimal digits for COMP-1 (F-floating, S-floating) 2 decimal digits for COMP-2 (D-floating) 3 decimal digits for COMP-2 (G-floating, T-floating)
Range of exponent	-38 to +38 (base 10) for COMP-1 (F-floating, S-floating) -38 to +38 (base 10) for COMP-2 (D-floating) -308 to +308 (base 10) for COMP-2 (G-floating, T-floating)
Form of exponent	E begins exponent sign shown for negative and positive; all digits shown (for example, E+01)

- The size of the displayed field is 13 characters for COMP-1, and 22 characters for COMP-2.

9. The CONVERSION phrase or the /DISPLAY_FORMATTED qualifier (on OpenVMS Alpha or I64) or `-display_formatted` (on UNIX) displays floating-point items as follows:

   - The floating-point data item is converted from internal floating point to E-notation representation. (E-notation consists of a mantissa and, optionally, an exponent.)
   - The size of the displayed field is 13 characters for COMP-1 and 22 characters for COMP-2.
   - On UNIX systems, the E-notation is described as follows:

Size of mantissa	COMP-1 has 7 digits (plus decimal point and sign). COMP-2 has 16 digits (plus decimal point and sign).
Sign of mantissa	Sign shown only for negative; space if positive.
Form of mantissa	d.dddddd for COMP-1. d.dddddddddddddd for COMP-2. “.” replaced by “,” if DECIMAL POINT IS COMMA.
Size of exponent	3 decimal digits for COMP-1 and COMP-2.
Range of exponent	-308 to +308 (base 10) for COMP-1 and COMP-2.
Form of exponent	E begins exponent sign shown for negative and positive; both digits shown (for example, E+01).

   On OpenVMS Alpha and I64 systems, compiled with the appropriate option on the /FLOAT qualifier, the E-notation is described as follows:

Size of mantissa	COMP-1 has 7 digits (F-floating, S-floating) plus decimal point and sign. COMP-2 has 15 digits (G-floating, T-floating) or 16 digits (D-floating) plus decimal point and sign.
Sign of mantissa	Sign shown only for negative; space if positive.
Form of mantissa	d.dddddd for COMP-1 (F-floating, S-floating) d.dddddddddddddd for COMP-2 (G-floating, T-floating.) d.ddddddddddddddd for COMP-2 (D-floating). “.” replaced by “,” if DECIMAL POINT IS COMMA.
Size of exponent	2 decimal digits for COMP-1 (F-floating, S-floating). 2 decimal digits for COMP-2 (D-floating). 3 decimal digits for COMP-2 (G-floating, T-floating).
Range of exponent	-38 to +38 (base 10) for COMP-1 (F-floating, S-floating). -38 to +38 (base 10) for COMP-2 (D-floating). -308 to +308 (base 10) for COMP-2 (G-floating, T-floating).
Form of exponent	E begins exponent sign shown for negative and positive; all digits shown (for example, E+01).

Format 1

10. The DISPLAY statement transfers data from each *src-item* (in its order of appearance in the statement) to *output-dest*.
11. No editing or conversion occurs during DISPLAY execution unless there is an applicable WITH CONVERSION phrase.
12. If *src-item* is a figurative constant, only one occurrence is displayed.
13. When there is more than one *src-item*, sending item size is the sum of the *src-item* sizes.
14. If there is no UPON phrase, the DISPLAY statement transfers data to the default system output device.
15. 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.
16. If DECIMAL POINT IS COMMA is specified, comma replacement occurs upon display.

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 WITH NO ADVANCING phrase, data is output according to Format 1 positioning rules for the DISPLAY statement. That is, a line feed and carriage return are generated automatically following data display.

LINE NUMBER Phrase (Format 2)

10. The LINE NUMBER phrase positions the cursor on a specific line of the video screen prior to displaying.
11. If the LINE NUMBER phrase does not appear but the COLUMN NUMBER phrase does, then data is displayed to the current specified column position.
12. If *line-num* or the value of *line-id* is greater than the bottommost line position of the current screen, program results are undefined.
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 your program generates a value for *line-id* that is one of the following:
    - Zero
    - Negative
    - Greater than the bottommost line position of the current screen

COLUMN NUMBER Phrase (Format 2)

18. The COLUMN NUMBER phrase positions the cursor 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 to 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*, 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 last column position on the screen

LINE NUMBER and COLUMN NUMBER Phrases (Format 3)

25. The LINE NUMBER and COLUMN NUMBER phrases together give the starting screen coordinates.

26. The position of each screen item within the referenced *screen-name* is offset from the LINE and COLUMN positions.

27. If either LINE or COLUMN is not specified, the default value is 1.

ERASE Phrase (Format 2)

28. The ERASE phrase erases all, or part, of a line (or screen) before displaying data. You must specify SCREEN or LINE.

29. 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).

30. If you do not use its TO END option, the ERASE phrase erases the entire line (or screen).

BELL Phrase (Format 2)

31. The BELL phrase rings the terminal bell before displaying data.

UNDERLINED Phrase (Format 2)

32. The UNDERLINED phrase displays characters on the screen with the *underscore on* character attribute.

BOLD Phrase (Format 2)

33. The BOLD phrase displays characters on the screen with the *bold on* character attribute. The BOLD attribute is only detectable when any of the following conditions are true:

    - Nonspace characters are displayed.
    - The underlined or reversed attributes are specified.
    - The terminal screen is set to light background.

BLINKING Phrase (Format 2)

34. The BLINKING phrase displays characters on the screen with the *blink on* character attribute. The BLINKING attribute is only detectable when any of the following conditions are true:

    - Nonspace characters are displayed.
    - The underlined or reversed attributes are specified.
    - The terminal screen is set to light background.

REVERSED Phrase (Format 2)

35. The REVERSED phrase displays characters on the screen with the *reverse video on* character attribute.

Formats 4, 5, and 6

36. When a Format 4 DISPLAY statement is specified, the value stored in *arg-position* is moved to *argument-number*. This updates the current argument position indicator for the command line (see ARGUMENT-NUMBER in the SPECIAL-NAMES paragraph in Chapter 4). This points to the selected argument to be read by a Format 7 ACCEPT statement.

37. *arg-position* must be in the range 0 to 99 and can refer to arguments, switches, and flags that appear on the run command line of the COBOL program. When the current argument position indicator is zero, it refers to the zeroth command line argument, in other words the command that invoked the COBOL program.

38. When a Format 5 DISPLAY statement is specified, the value stored in *envlog-name* is moved to *environment-name* (see ENVIRONMENT-NAME in the SPECIAL-NAMES paragraph in Chapter 4). The updated value of *environment-name* becomes the environment variable or logical to be accessed by subsequent Format 6 DISPLAY and Format 8 ACCEPT statements.

39. *environment-value*, when used with a Format 6 DISPLAY, receives the value stored in *envlog-value*. The environment variable or logical is the one named by a Format 5 DISPLAY statement (see ENVIRONMENT-VALUE in the SPECIAL-NAMES paragraph in Chapter 4).

40. *stment* is executed if the name of the environment variable or logical has not been set by a Format 5 DISPLAY, or if the environment variable or logical does not exist.

41. *stment2* is executed if the exception condition does not exist.

Technical Notes

Format 1

1. On OpenVMS, the DISPLAY statement transfers data through the I/O system (RMS), using the Variable with Fixed-Length Control (VFC) format.

2. A DISPLAY statement without the UPON phrase transfers data to the default output device (the terminal). To transfer data to a file on UNIX systems, the environment variable COBOL_OUTPUT can be used to specify a text file containing output data. To transfer data to a file on OpenVMS systems the logical COB$OUTPUT or SYS$OUTPUT can be used to specify a text file containing output data.

   Alternatively, output device redirection (>) can be used on UNIX systems to name an output file.

3. A DISPLAY statement that includes the UPON phrase transfers data to the *file-device-name* associated with the SPECIAL-NAMES paragraph description of *output-dest*.

4. Because the object of a logical name (on OpenVMS systems) is not necessarily a device, no open mode is implied. As a result, *output-dest* can be associated with any *device-name* in the SPECIAL-NAMES paragraph. For example, *output-dest* can refer to PAPER-TAPE-READER as well as PAPER-TAPE-PUNCH.

Format 2

5. Format 2 is a VSI extension to the standard COBOL use of the DISPLAY statement.

6. The VSI extensions to the ACCEPT and DISPLAY statements support data input and display only on the VT100 and later terminal types, including emulators of these terminal types.

7. The UNDERLINED, BOLD, BLINKING, and REVERSED character attributes are not available on VT100 terminals without the advanced video option.

8. You should display data only on fields that are within screen boundaries. That is, the terminal operator should see all the characters displayed. If data is displayed on fields that position the cursor 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.

9. 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.

   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 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.

10. When there is more than one *src-item*, each specific *src-item* is displayed, after application of any phrases specific to that *src-item*, in order of occurrence in the DISPLAY statement.

Formats 2 and 3

11. On OpenVMS, control sequences from SMGTERMS.TXT are used to accomplish cursor positioning, screen erasure, and video attributes. Refer to the Support for Non-HP Terminals chapter of the OpenVMS RTL Screen Management (SMG$) Manual if you wish to customize SMGTERMS.TXT.

All Formats

12. VSI COBOL for OpenVMS parses the contents of the data being displayed to determine how they affect the terminal and the cursor position. The parsing of control sequences is performed according to the DEC STD 138-0 Registry of Control Functions for Character Imaging Devices. VSI COBOL for OpenVMS does not modify the control sequences in any way; if an invalid control sequence is found, VSI COBOL for OpenVMS does not attempt to correct the sequence.

    Therefore when you display an escape or control sequence, the entire sequence must be displayed in one operation:

- If you use DISPLAY Format 1, the complete sequence must be contained in one or more *src-items* within one DISPLAY statement.
- If you use DISPLAY Format 2, the complete sequence must be contained in one *src-item*.
- If you use DISPLAY Format 3, the complete sequence must be contained within one elementary screen item.

13. A DISPLAY statement used in an ACCEPT [NOT] ON EXCEPTION statement must be terminated (with, for example END-DISPLAY) on Alpha and I64 systems. If you are concerned with the different VAX behavior, refer to the appendix on compatibility in the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/].

Examples

In the example results, the character “s” represents a space. The examples assume a maximum screen size of 24 lines. They also 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".
```

```
                                                RESULT:

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 following program uses VSI DISPLAY extensions (Format 2).

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLES.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 ITEMF        COMP-1.
01 ITEMG        COMP-2 .
01 ITEMH        PIC S9(9) COMP VALUE IS 123456789.
01 ITEMI        PIC S9(9) COMP-3.
PROCEDURE DIVISION.
01.
    MOVE 101.000000000 TO  ITEMF.
```

```
MOVE .109999999 TO ITEMG.
MOVE  123456789 TO ITEMI.
DISPLAY
    ITEMF   WITH CONVERSION LINE PLUS
    ITEMG   WITH CONVERSION LINE PLUS
    ITEMH   WITH CONVERSION LINE PLUS
    ITEMI   WITH CONVERSION LINE PLUS
    .
    .
    .
```

The *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/] contains additional examples using VSI extensions to the DISPLAY statement. Refer to the chapters that describe screen handling, command line variables, environment variables and logicals.

Additional References

- SPECIAL-NAMES section in Chapter 4
- *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/] sections on using command line arguments and environment variables

DIVIDE

DIVIDE — The DIVIDE statement divides one or more numeric data items by another and sets the value of the data items equal to the quotient, optionally storing the remainder.

General Format

Format 1

DIVIDE num INTO { rsult [ROUNDED] } . . .

[ON SIZE ERROR stment]

[NOT ON SIZE ERROR stment2]

[END-DIVIDE]

Format 2

DIVIDE num INTO num GIVING { rsult [ROUNDED] } . . .

[ON SIZE ERROR stment]

[NOT ON SIZE ERROR stment2]

[END-DIVIDE]

Format 3

DIVIDE num BY num GIVING { rsult [ROUNDED] } . . .

[ON SIZE ERROR stment]

[NOT ON SIZE ERROR stment2]

[END-DIVIDE]

Format 4

DIVIDE num INTO num GIVING rsult [ROUNDED] REMAINDER remaind

[ON SIZE ERROR stment]

[NOT ON SIZE ERROR stment2]

[END-DIVIDE]

Format 5

DIVIDE num BY num GIVING rsult [ROUNDED] REMAINDER remaind

[ON SIZE ERROR stment]

[NOT ON SIZE ERROR stment2]

[END-DIVIDE]

num

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.

stment

is an imperative statement executed when a size error condition has occurred.

stment2

is an imperative statement executed when no size error condition has occurred.

remaind

is the identifier of an elementary numeric item or an elementary numeric edited item.

General Rules

Format 1

1. The value of *num* 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

2. The value of the first *num* is divided into the value of the second. This quotient replaces the current value of each *rsult*.

Format 3

3. The value of the first *num* is divided by the value of the second. This quotient replaces the current value of each *rsult*.

Formats 4 and 5

4. 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.

5. The computation described in rule 4 determines the accuracy of *remaind*. It includes decimal point alignment and truncation (not rounding) required by the description of *remaind*.

6. 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.

Examples

The following example shows a run-time message issued for an illegal attempt to divide by zero:

```
%COB-E-DIVBY-ZER, divide by zero; execution continues
```

Each of the examples assume the following data descriptions and initial values. The initial values are listed in the right-hand column:

INITIAL VALUES

```
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
03  ITEMF  PIC 99    VALUE 47.                       47
03  ITEMG  PIC 9     VALUE 9.                        9
03  ITEMH  PIC 9     VALUE 2.                        2
03  ITEMI  PIC 99    VALUE 4.                        4
```

In each of the following examples, the right-hand column shows the results of the DIVIDE operation.

1. Without GIVING phrase or rounding:

   RESULTS

   ```
   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
```

10. The ON SIZE ERROR phrase: (IF ON SIZE ERROR occurs on an occurrence of *rsult*, the contents of that occurrence of *rsult* are unchanged.)

```
DIVIDE ITEME INTO ITEMF
   GIVING ITEMG ITEMD                            ITEMD = 15
   ON SIZE ERROR                                 ITEMG = 9
     MOVE 0 TO ITEMH.                            ITEMH = 0
```

11. The ON SIZE ERROR phrase:

    (IF ON SIZE ERROR occurs on *remaind*, the contents of *remaind* are unchanged.)

```
DIVIDE ITEMD INTO ITEMF
   GIVING ITEMI REMAINDER ITEMG                  ITEMI = 3
   ON SIZE ERROR                                 ITEMG = 9
     MOVE 0 TO ITEMH.                            ITEMH = 0
```

12. The NOT ON SIZE ERROR phrase:

```
DIVIDE ITEMD INTO ITEMF                          ITEMI = 3
   GIVING ITEMI REMAINDER ITEMC                  ITEMC = 11.00
   ON SIZE ERROR
      MOVE 0 TO ITEMH
   NOT ON SIZE ERROR
      MOVE 1 TO ITEMH.                           ITEMH = 1
```

Additional References

- Section 6.1.4: Scope of Statements
- Section 6.6.1: Arithmetic Operations
- Section 6.6.2: Multiple Receiving Fields in Arithmetic Statements
- Section 6.6.3: ROUNDED Phrase
- Section 6.6.4: ON SIZE ERROR Phrase
- Section 6.6.7: Overlapping Operands and Incompatible Data

EVALUATE

EVALUATE — The EVALUATE statement selects a program action based on the evaluation of one or more conditions.

General Format

```
EVALUATE { subj-item } [ ALSO { subj-item } ] ...
         { TRUE      }      { TRUE      }
         { FALSE     }      { FALSE     }

{ { WHEN { ANY                                          } [ ALSO { ANY                                          } ] ... } ... stment1 } ...
  {      { cond                                           {      { cond                                           }
  {      { TRUE                                           {      { TRUE                                           }
  {      { FALSE                                          {      { FALSE                                          }
  {      { [ NOT ] obj-item [ { THRU    } obj-item ] }    {      { [NOT] obj-item [ { THRU    } obj-item ] }
  {                           { THROUGH }                {                       { THROUGH }
  }                                                      }                                                      }

[ WHEN OTHER stment2 ]
[ END-EVALUATE ]
```

subj-item

is an identifier, an arithmetic or conditional expression, or a literal other than the figurative constant ZERO.

cond

is a conditional expression.

`obj-item`

is a literal, an identifier, or an arithmetic expression.

`stment1`

is an imperative statement.

`stment2`

is an imperative statement.

Syntax Rules

1. Before the first WHEN phrase: (a) *subj-item* and the words TRUE and FALSE are called subjects, and (b) all subjects comprise the subject set.
2. In a WHEN phrase: (a) ANY, TRUE, FALSE, and the operands are called objects, and (b) all objects in a single WHEN phrase comprise an object set.
3. The number of objects in the object set must equal the number of subjects in the subject set.
4. The words THROUGH and THRU are equivalent.
5. Two *obj-items* connected by a THROUGH phrase:
   - Must be of the same class
   - Combine to form one object
6. Each object in an object set must correspond to the subject by appearing in the same ordinal position as in the subject set. For each pair:
   - The *obj-item* must be a valid operand for comparison to the subject.
   - TRUE, FALSE, or cond as an object must correspond only to TRUE, FALSE, or a conditional expression as the subject.
   - ANY can correspond to any type of subject.
7. Conditional expressions can be simple or complex conditions.

General Rules

Evaluation Procedure

1. The EVALUATE statement operates as if each subject and object were evaluated and assigned one of the following:
   - A numeric or nonnumeric value
   - A range of numeric or nonnumeric values
   - A truth value

   The statement assigns values according to the following rules:

	Condition	Value Assigned
a.	An identifier for a subject, or for an object without the NOT or THROUGH phrases	Value and class of the identifier's data item.
b.	A literal for a subject, or for an object without the NOT or THROUGH phrases	Value and class of the literal.
c.	The figurative constant ZERO for an object without the NOT or THROUGH phrases	Value and class of the corresponding subject.
d.	An arithmetic expression for a subject, or for an object without the NOT or THROUGH phrases	Numeric value, according to the rules for evaluating arithmetic expressions.
e.	A conditional expression for a subject or a conditional expression for an object	Truth value, according to the rules for evaluating conditional expressions.
f.	TRUE or FALSE as a subject or object	Truth value: true for the word TRUE and false for the word FALSE.
g.	ANY for an object	No further evaluation.
h.	THROUGH phrase for an object without the NOT phrase	The range of values is all values that, when compared to the subject, are greater than or equal to the first obj-item and less than or equal to the second obj-item. If the first obj-item is greater than the second obj-item, there are no values in the range.
i.	Object with the NOT phrase	All values not equal to the value (or range of values) that would be assigned without the NOT phrase.

Comparison Procedure

2. After values have been assigned to each subject and object, comparison begins. It proceeds as if the values were compared to determine if any WHEN phrase satisfies the subject set.

3. EVALUATE compares each object in the object set of the first WHEN phrase to the subject in the same ordinal position in the subject set. The comparison is satisfied if one of the following conditions is true:

   - The items being compared are assigned numeric or nonnumeric values or a range of numeric or nonnumeric values; and the value assigned to the subject equals the value, or one of the range of values, assigned to the object, according to the rules for comparison.

   - The items being compared are assigned identical truth values.

   - The word ANY represents the object.

4. If the comparison is satisfied for every object in an object set, the WHEN phrase containing that object set is selected.

5. If the comparison is not satisfied for every object in an object set, the object set does not satisfy the subject set.

6. The comparison procedure is repeated for each object set, in order of appearance, until one of these conditions occur:
   - A WHEN phrase is selected by satisfying the subject set.
   - A WHEN OTHER phrase is selected.
   - There are no more object sets.
   - The END-EVALUATE statement is reached.
   - A separator period is reached.

Execution Procedure

7. If a WHEN phrase is selected, execution continues with *stment1*.
8. If no WHEN phrase is selected, and a WHEN OTHER phrase is present, execution continues with *stment2*.
9. EVALUATE statement execution ends when one of the following conditions occurs:
   - Execution reaches the end of the selected WHEN phrase.
   - Execution reaches the end of the WHEN OTHER phrase.
   - No WHEN phrase is selected and there is no WHEN OTHER phrase.
   - Execution reaches END-EVALUATE.
   - Execution reaches a separator period.

Examples

In these examples, the results are shown as either data item values or procedure branches. However, *stment* can be *any* imperative statement, including multiple statements.

1. One condition.

```
EVALUATE ITEMA
  WHEN "A01"            MOVE 1 TO ITEMB
  WHEN "A02" THRU "C16" MOVE 2 TO ITEMB
  WHEN "C20" THRU "L86" MOVE 3 TO ITEMB
  WHEN "R20"            ADD 1 TO R-TOT
                        GO TO  PROC-A
  WHEN OTHER            MOVE 0 TO ITEMB
  END-EVALUATE.
```

Samples:

ITEMA	Result
“A15”	ITEMB = 2
“P80”	ITEMB = 0
“F01”	ITEMB = 3
“M19”	ITEMB = 0

ITEMA	Result
“A01”	ITEMB = 1
“R20”	PROC-A

2. Multiple conditions. This example shows how EVALUATE can represent a decision table.

```
EVALUATE LOW-STOK  WEEK-USE  LOC-VNDR ON-ORDER
  WHEN   "Y",    16 THRU 999,  ANY,   "N" GO TO RUSH-ORDER
  WHEN   "Y",    16 THRU 999,  ANY,   "Y" GO TO NORMAL-ORDER
  WHEN   "Y",     8 THRU 15,   "N",   "N" GO TO RUSH-ORDER
  WHEN   "Y",     8 THRU 15,   "N",   "Y" GO TO NORMAL-ORDER
  WHEN   "Y",     8 THRU 15,   "Y",   "N" GO TO NORMAL-ORDER
  WHEN   "Y",     0 THRU 7,    ANY,   "N" GO TO NORMAL-ORDER
  WHEN   "N",       ANY,       ANY,   "Y" GO TO CANCEL-ORDER
  END-EVALUATE.
```

Samples:

LOW-STOK	WEEK-USE	LOC-VNDR	ON-ORDER	Result
“Y”	38	“N”	“Y”	NORMAL-ORDER
“N”	20	“Y”	“Y”	CANCEL-ORDER
“N”	12	“Y”	“N”	next statement
“Y”	12	“Y”	“N”	NORMAL-ORDER
“Y”	12	“Y”	“Y”	next statement
“Y”	40	“N”	“N”	RUSH-ORDER

3. Relation conditions and arithmetic expressions.

```
EVALUATE-ITEM-ROUTINE.
*
* After the imperative statement in the selected WHEN phrase
* executes (for example PERFORM PROC-A), control then
* transfers to the first statement following the end of the
* EVALUATE statement (MOVE A TO B).
*

   EVALUATE ITEMA > 6 AND < 30, 8 * ITEMB - 1
      WHEN   TRUE,              5 * ITEMC     PERFORM PROC-A
      WHEN   FALSE,             ITEMC         PERFORM PROC-B
      WHEN   ITEMC > 12,        -1            PERFORM PROC-C
      WHEN   TRUE,              NOT 7 THRU 40 PERFORM PROC-D
      WHEN   OTHER                            PERFORM PROC-E
   END-EVALUATE.
   MOVE A TO B.
```

Samples:

ITEMA	ITEMB	ITEMC	Result
12	2	3	PROC-A
25	0	14	PROC-C

ITEMA	ITEMB	ITEMC	Result
30	0	14	PROC-E
6	3	23	PROC-B
14	0	5	PROC-D
5	0	11	PROC-C

Consider how the EVALUATE statement works using the values in the previous sample:

1. The value of the first subject is a truth value (General Rule 1e). ITEMA is *not* greater than 6 and less than 30; therefore, the value of the first subject is false.

2. The value of the second subject is a numeric value (General Rule 1d):

   8 * 0-1 = -1.

3. When the first WHEN phrase is evaluated:

   - The value of the first object is a truth value (General Rule 1f): true.
   - The value of the second object is a numeric value: 55.
   - The value of the first object does not equal that of the first subject. Furthermore, the values of the second object and subject do not match. Therefore, this WHEN phrase is not selected (General Rule 5).

4. When the second WHEN phrase is evaluated:

   - The value of the first object is a truth value (General Rule 1f): false.
   - The value of the second object is a numeric value: 11.
   - The value of the first object equals that of the first subject. However, the values of the second object and subject do not match. Therefore, this WHEN phrase is not selected (General Rule 5).

5. When the third WHEN phrase is evaluated:

   - The value of the first object is a truth value (General Rule 1f). Because the value of ITEMC is *not* greater than 12, the value of this object is false.
   - The value of the second object is a numeric value: -1.
   - The value of the first object equals that of the first subject. The values of the second object and subject also match. Therefore, this WHEN phrase is selected (General Rule 4).

6. The statement following the third WHEN phrase is PERFORM PROC-C. Control transfers to that procedure, and the EVALUATE statement ends.

Additional References

- Section 6.1.4: Scope of Statements
- Section 6.4: Arithmetic Expressions
- Section 6.5: Conditional Expressions

- Section 6.5.1: Relation Conditions
- Section 6.6.1: Arithmetic Operations

EXIT

EXIT — The EXIT statement provides a common logical end point for a series of procedures.

General Format

EXIT

Syntax Rule

The EXIT statement must appear in a sentence by itself and be 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

EXIT PROGRAM — The EXIT PROGRAM statement marks the logical end of a called program.

General Format

EXIT PROGRAM

Syntax Rules

1. If the EXIT PROGRAM statement is in a consecutive sequence of imperative statements, it must be the last statement in that sequence.
2. The EXIT PROGRAM statement cannot appear in a GLOBAL USE procedure.

General Rules

1. If EXIT PROGRAM executes in a program that is not a called program, it causes execution to continue with the next executable statement. Refer to the *VSI COBOL User Manual* [https://

docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/] for information on how the `v3` setting of the `standard` compiler option affects the EXIT PROGRAM statement.

2. If the EXIT PROGRAM statement executes in a called program without the INITIAL clause in its PROGRAM-ID paragraph, 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 can 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.

3. When EXIT PROGRAM executes in a called program with the INITIAL attribute, the actions described in General Rule 2 also apply. In addition, executing the EXIT PROGRAM statement is equivalent to executing a CANCEL statement that names the called program.

4. Special handling of the EXIT PROGRAM statement is performed when you specify the `standard` compiler option with the `v3` setting on the compiler command line. Refer to the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/] for more information.

Example

```
TEST-RETURN.
    IF ITEMA NOT = ITEMB
         MOVE ITEMA TO ITEMB
         EXIT PROGRAM.
```

GENERATE

GENERATE — The GENERATE statement directs the Report Writer Control System (RWCS) to produce a report according to the Report Description entry (RD) in the Report Section of the Data Division.

General Format

GENERATE report-item

`report-item`

names either a report-name in a Report Description entry, or the group-data-name of a TYPE IS DETAIL report group.

Syntax Rules

1. If *report-item* references a group-data-name, it must name a TYPE DETAIL report group. Group-data-name can be qualified by report-name.

2. If *report-item* references a report-name, its report description must contain:

   - A CONTROL clause

   - At least one CONTROL HEADING, DETAIL, or CONTROL FOOTING report group

- No more than one DETAIL report group

General Rules

1. An INITIATE statement must be executed before a GENERATE statement is executed for a specific report.
2. The RWCS produces a summary report if all of the GENERATE statements for a report reference report-name. A summary report contains no TYPE IS DETAIL report groups.
3. The RWCS produces a detail report if a GENERATE statement references a DETAIL report group.
4. To detect and trigger control breaks for a specific report, the RWCS:
   - Saves the initial values within control data items as prior values when the GENERATE statement executes.
   - Compares the prior values to the current values of control data items when subsequent GENERATE statements execute. Only if the current values change does a control break occur. If a control break occurs, the current values are saved as prior values.
   - Repeats the preceding step until the last control break is processed.
5. The RWCS automatically processes any PAGE HEADING and PAGE FOOTING report groups when it must start a new page to present a CONTROL HEADING, DETAIL, or CONTROL FOOTING.
6. When the first GENERATE statement for a specific report is executed, the RWCS processes these report groups, if present in the report description, in this order:
   a. The REPORT HEADING report group.
   b. The PAGE HEADING report group.
   c. All CONTROL HEADING report groups from major to minor.
   d. For GENERATE group-data-name statements (detail reporting), the RWCS presents the specific DETAIL report group for processing.
   e. For GENERATE report-name statements (summary reporting), the RWCS does not present the DETAIL report group for processing; however, the RWCS does perform all other DETAIL report group functions.
7. When subsequent GENERATE statements are executed for a specific report, the RWCS:
   - Checks for control breaks. The rules governing the inequality of control data items are identical to the rules for relation conditions. If a control break occurs, the RWCS:
     a. Enables the CONTROL FOOTING USE procedures and CONTROL FOOTING SOURCE clauses. This allows program access to the control data item values that the RWCS uses to detect a given control break.
     b. Processes the CONTROL FOOTING report groups starting with the minor. Only CONTROL FOOTING report groups less major than the highest level at which a control break occurs are processed.

c. Processes the CONTROL HEADING report groups in the order major to minor. Only the CONTROL HEADING report groups less major than the highest level at which a control break occurs are processed.

- Processes the GENERATE statement. For GENERATE group-data-name statements (detail reporting), the RWCS processes the specific DETAIL report group. For GENERATE report-name statements (summary reporting), the RWCS does not present the DETAIL report group for processing; however, the RWCS does perform all other DETAIL report group functions.

8. No GENERATE statements can reference a file after executing a TERMINATE statement for the same file.

Additional References

- TYPE clause in Chapter 5
- USE statement

GO TO

GO TO — 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} … 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 Rules

1. 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.

2. If an ALTER statement refers to a paragraph, the paragraph must consist of only a paragraph header followed by a Format 1 GO TO statement.

3. A Format 1 GO TO statement without *proc-name* can only be in a single-statement paragraph.

General Rules

Format 1

1. The GO TO statement transfers control to *proc-name*.
2. If there is no *proc-name*, the GO TO statement cannot execute before an ALTER statement changes its destination.

Format 2

3. 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 label
2	SOPHOMORE label
3	JUNIOR label
4	SENIOR label
5	MOVE statement
0	MOVE statement
-10	MOVE statement

IF

IF — The IF statement evaluates a condition. The condition's truth value determines the program action that follows.

General Format

```
IF condition THEN { { stment-1 } . . .  } [ ELSE { stment-2 } . . . [ END-IF ] ]
                  {  NEXT SENTENCE      }  [ ELSE NEXT SENTENCE                ]
                                           [ END-IF                             ]
```

`stment-1`

is an imperative or conditional statement. An imperative statement can precede a conditional statement.

`stment-2`

is an imperative or conditional statement. An imperative statement can precede a conditional statement.

Syntax Rules

1. The ELSE NEXT SENTENCE phrase is optional if it immediately precedes a separator period.
2. If the END-IF phrase is specified, the NEXT SENTENCE phrase must not be specified.

General Rules

1. The scope of an IF statement ends with any of the following:
   - An END-IF phrase at the same nesting level
   - 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 or END-IF; this pairing proceeds from left to right. Thus, an ELSE or END-IF phrase applies to the first preceding unpaired IF.

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 ...
```

Input Values			Output Value
ITEMA	**ITEMC**	**Next Statement**	**ITEMB**
12	6	ADD	“Y”
12	12	ADD	“X”
8	8	PROC-A	?

5. END-IF:

   (In this example, the initial value of ITEMD is 5.)

```
IF ITEMA > 10
  IF ITEMA = ITEMC
    ADD 1 TO ITEMD
    MOVE "X" TO ITEMB
  END-IF
  ADD 1 TO ITEMD.
```

ITEMA	ITEMC	ITEMB	ITEMD
4	6	?	5
15	6	?	6
13	13	“X”	7
7	7	?	5

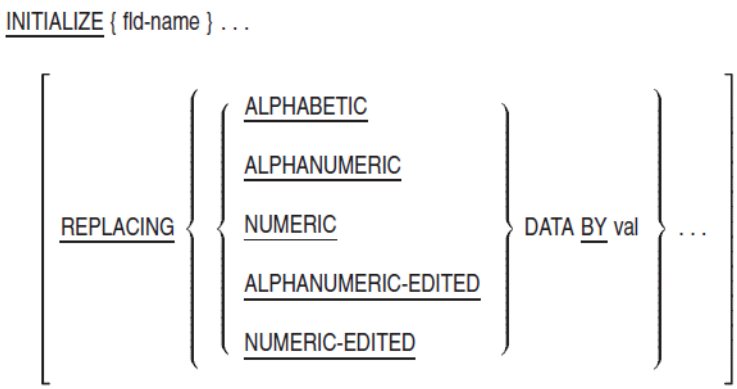
Additional References

- Section 6.1: Verbs, Statements, and Sentences
- Section 6.1.4: Scope of Statements
- Section 6.5: Conditional Expressions

INITIALIZE

INITIALIZE — The INITIALIZE statement sets selected types of data fields to predetermined values.

General Format



`fld-name`

is the identifier of the receiving area data item.

`val`

is the sending area. It can be a literal or the identifier of a data item.

Syntax Rules

1. The phrase after the word REPLACING is the category phrase.
2. The category of the data item referred to by *val* must be consistent with that in the category phrase. The combination of categories must allow execution of a valid MOVE statement.
3. The same category cannot be repeated in a REPLACING phrase.
4. The description of *fld-name* or any item subordinate to it cannot contain the OCCURS clause DEPENDING phrase.
5. Neither *fld-name* nor *val* can be index data items.
6. *fld-name* cannot contain a RENAMES clause.

General Rules

1. The key word that follows the word REPLACING corresponds to a category of data. (See the section on Categories and Classes of Data in the Data Division chapter.)
2. *fld-name* can be an elementary or group item. If it is a group item, the INITIALIZE statement operates on the elementary items within the group item. For a table within a group item, INITIALIZE operates on the elementary items within the table.
3. Whether *fld-name* is an elementary item or a group item, if the REPLACING phrase is specified, all data movement operations occur as if they resulted from a series of MOVE statements with elementary item receiving areas:
   - If the receiving area is a group item, INITIALIZE affects only those subordinate elementary items whose category matches a category phrase. General Rule 6 describes the effect on elementary items when there is no REPLACING phrase.
   - INITIALIZE affects all eligible elementary items, including all occurrences of table items in the group.
   - If the receiving area is an elementary item, that item is initialized only if it matches a category phrase.
4. INITIALIZE does not affect index data items and FILLER data items.
5. INITIALIZE does not affect items subordinate to *fld-name* that contain a REDEFINES clause. Nor does it affect data items subordinate to those items. However, *fld-name* itself can have a REDEFINES clause or be subordinate to a data item that does.
6. When there is a REPLACING phrase, *val* is the sending field for each of the implicit MOVE statements.
7. When there is no REPLACING phrase, the sending field for the implicit MOVE statements is as follows:

- SPACES, if the data item category is alphabetic, alphanumeric, or alphanumeric edited
- ZEROS, if the data item category is numeric or numeric edited

8. INITIALIZE operates on each *fld-name* in the order it appears in the statement. When *fld-name* is a group item, INITIALIZE operates on its eligible subordinate elementary items in the order they are defined in the group.
9. If *fld-name* occupies the same storage area as *val*, the execution result of this statement is undefined. (See the section on Overlapping Operands and Incompatible Data.)

Examples

In the examples' results, a hyphen (-) means that the value of the data item is unchanged; s represents the character space. The examples assume this data description:

```
01  ITEMA.
    03  ITEMB      PIC X(4).
    03  ITEMC.
        05  ITEMD  PIC 9(5).
        05  ITEME  PIC $$$9.99.
        05  ITEMF  PIC XX/XX.
    03  ITEMG.
        05  ITEMH  PIC 999.
        05  ITEMI  PIC XX.
        05  ITEMJ  PIC 99.9.
    03  ITEMK      PIC X(4) JUSTIFIED RIGHT.
```

1. `INITIALIZE ITEMA.`
2. `INITIALIZE ITEMB ITEMG.`
3. `INITIALIZE ITEMA REPLACING ALPHANUMERIC BY “ABCDE”.`
4. `INITIALIZE ITEMG REPLACING NUMERIC BY 9.`
5. `INITIALIZE ITEMA REPLACING NUMERIC-EDITED BY 16.`
6. `INITIALIZE ITEMA REPLACING ALPHANUMERIC-EDITED  BY “ABCD”.`
7. `INITIALIZE ITEMA REPLACING ALPHANUMERIC BY “99”.`

	ITEMB	ITEMD	ITEME	ITEMF	ITEMH	ITEMI	ITEMJ	ITEMK
1.	ssss	00000	ss$0.00	ss/ss	000	ss	00.0	ssss
2.	ssss	–	–	–	000	ss	00.0	–
3.	ABCD	–	–	–	–	AB	–	BCDE
4.	–	–	–	–	009	–	–	–
5.	–	–	s$16.00	–	–	–	16.0	–
6.	–	–	–	AB/CD	–	–	–	–
7.	99ss	–	–	–	–	99	–	ss99

Additional References

- MOVE statement

- Section 6.6.7: Overlapping Operands and Incompatible Data

INITIATE

INITIATE — The INITIATE statement causes the Report Writer Control System (RWCS) to begin processing a report.

General Format

INITIATE {report-name}...

`report-name`

names a report defined by a Report Description entry (RD) in the Report Section of the Data Division.

General Rules

1. The INITIATE statement does not automatically open a report file. The program must execute either an OPEN OUTPUT or an OPEN EXTEND statement before it can execute an INITIATE statement.
2. Upon execution of the INITIATE statement, the RWCS sets all sum counters, LINE-COUNTER, and PAGE-COUNTER to zero.
3. If the INITIATE statement has more than one *report-name*, the statement executes as if there were a separate INITIATE statement for each *report-name*.
4. A program must execute a TERMINATE statement before it can execute another INITIATE statement for the same *report-name*.

Additional Reference

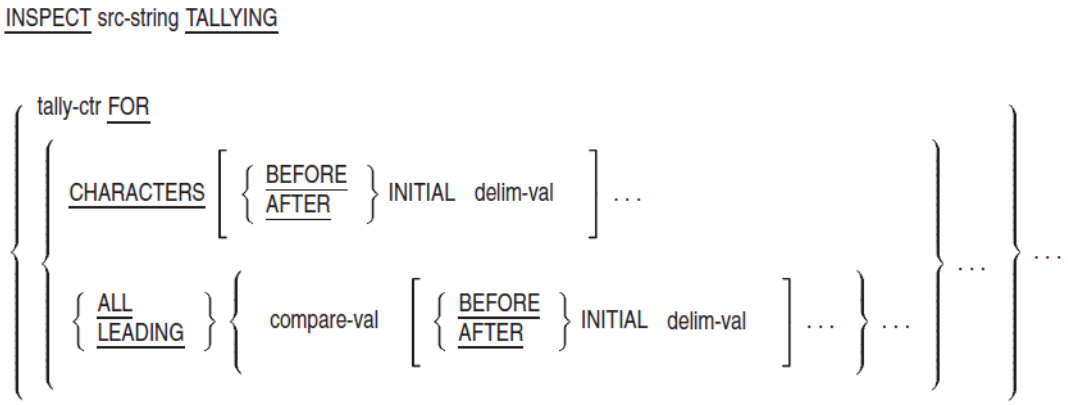
USE statement

INSPECT

INSPECT — The INSPECT statement counts or replaces occurrences of single characters or groups of characters in a data item.

General Format

Format 1



Format 2

```
INSPECT src-string REPLACING

{ CHARACTERS BY replace-char [ { BEFORE } INITIAL delim-val ] ...
                               { AFTER  }
  { ALL     }
  { LEADING }  { compare-val BY replace-val [ { BEFORE } INITIAL delim-val ] ... } ...
  { FIRST   }                               { AFTER  }
} ...
```

Format 3

```
INSPECT src-string TALLYING

{ tally-ctr FOR
  { CHARACTERS [ { BEFORE } INITIAL delim-val ] ...
                 { AFTER  }
  { ALL     }
  { LEADING }  { compare-val [ { BEFORE } INITIAL delim-val ] ... } ...
                               { AFTER  }
  } ...
} ...

REPLACING

{ CHARACTERS BY replace-char [ { BEFORE } INITIAL delim-val ] ...
                             { AFTER  }
  { ALL     }
  { LEADING }  { compare-val BY replace-val [ { BEFORE } INITIAL delim-val ] ... } ...
  { FIRST   }                               { AFTER  }
} ...
```

Format 4

```
INSPECT src-string CONVERTING compare-chars TO convert-chars
    [ { BEFORE } INITIAL delim-val ] ...
      { AFTER  }
```

`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.

`delim-val`

is the character-string that delimits the INSPECT operation. Its content restrictions are the same as those for *compare-val*.

`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.

`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*.

`compare-chars`

is the string that contains the individual characters that convert to those in *convert-chars*. It is the same kind of item as *compare-val*.

`convert-chars`

is the string that contains the individual characters to which the characters in *compare-chars* convert. It is the same kind of item as *compare-val*.

Syntax Rules

All Formats

1. If *compare-val*, *delim-val*, *replace-char*, or *compare-chars* is a figurative constant, it refers to an implicit one-character data item.
2. A *compare-val* of an ALL or LEADING phrase, and a CHARACTERS, FIRST, or CONVERTING phrase can have no more than one BEFORE and one AFTER phrase following it.

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.

Format 4

6. The sizes of the data referred to by *convert-chars* and *compare-chars* must be equal. When *convert-chars* is a figurative constant, its size equals that of the data referred to by *compare-chars*.

7. The same character cannot appear more than once in the data referred to by *compare-chars*.

General Rules

All Formats

1. Inspection includes: (a) comparison, (b) setting boundaries for the BEFORE and AFTER phrases, and (c) tallying 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 to 5.

2. If *src-string*, *compare-val*, *delim-val*, *replace-val*, *compare-chars*, or *convert-chars* 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. If the size of *src-string* is zero characters, inspection does not occur.

4. If the size of *compare-val* is zero characters, *compare-val* does not match in any *src-string* comparison.

5. If any identifier is subscripted or is a function-identifier, the subscript or function-identifier is evaluated only once as the first operation in the execution of the INSPECT statement.

6. 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)

7. The comparison operation determines which occurrences of *compare-val* are tallied 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. For each match, tallying, replacing, or both occur, as described in General Rules 9 to 17. 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 6a to 6d operates as if:

   - Inspection compares a one-character data item to each character in *src-string*
   - A match occurs for each comparison

8. The BEFORE phrase determines the character position in *src-string* that will be the final leftmost position in the comparison operation.

   a. Comparison occurs on *src-string* only:

      - From its leftmost character position
      - To, but not including, the first occurrence of *delim-val*

   b. The position of the first occurrence of *delim-val* in *src-string* is determined before the first comparison operation.

   c. If *delim-val* does not occur in *src-string*, the comparison operation proceeds as if there were no BEFORE phrase.

9. The AFTER phrase determines the character position in *src-string* that will be the first leftmost position in the comparison operation.

   a. Comparison occurs on *src-string* only:

      - From the character position to the immediate right of the rightmost character position of *delim-val*'s first occurrence
      - To the rightmost position of *src-string*

   b. The position of the first occurrence of *delim-val* in *src-string* is determined before the first comparison operation.

   c. If *delim-val* is not in *src-string*, no match occurs, and inspection causes no tallying or replacement.

Format 1

10. Executing the INSPECT statement does not initialize the value of *tally-ctr*.

11. If the ALL phrase is present, the value of *tally-ctr* is incremented by one for each occurrence of *compare-val* in *src-string*.

12. 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.

13. 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 6e).

Format 2

14. The adjectives ALL, LEADING, and FIRST apply to succeeding BY phrases until the next adjective appears.

15. If the CHARACTERS phrase is present, each character matched in *src-string* is replaced by *replace-char* (see General Rule 6e).

16. When ALL is present, each occurrence of *compare-val* in *src-string* is replaced by *replace-val*.

17. 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.

18. When FIRST is present, the leftmost occurrence of *compare-val* in *src-string* is replaced by *replace-val*.

Format 3

19. 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.

Format 4

20. A Format 4 statement executes as if:

    - It were a Format 2 INSPECT statement with a series of ALL phrases, one for each character of *compare-chars*

    - *compare-val* in each ALL phrase referred to a single character of *compare-chars*

- *replace-val* in each ALL phrase referred to a single character of *replace-chars*

The individual characters of *compare-chars* and *replace-chars* correspond by ordinal position in the data items.

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" "R"
       REPLACING LEADING "A" BY "E" AFTER INITIAL "L".
```

ITEMA	COUNT1	ITEMA
CALLAR	3	CALLAR
SALAMI	1	SALEMI
LATTER	2	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

8. CONVERTING phrase:

```
INSPECT ITEMA CONVERTING "SIR" TO "DTA"
       AFTER QUOTE BEFORE "@".
```

ITEMA	ITEMA
TIRMS "SRXIL@STAR	TIRMS "DAXTL@STAR

Additional Reference

- MOVE

MERGE

MERGE — 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

```
MERGE mergefile { ON { DESCENDING } KEY { mergekey } . . . } . . .
                   { ASCENDING }
    [ COLLATING SEQUENCE IS alpha ]
    USING  infile  { infile } . . .
    { OUTPUT PROCEDURE IS first-proc [ { THRU    } end-proc ] }
    {                                  { THROUGH } ]          }
    { GIVING { outfile } . . .                                }
```

`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 or paragraph-name of the output procedure's first (or only) section or paragraph.

`end-proc`

is the section-name or paragraph-name of the output procedure's last section or paragraph.

`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*.

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 equal 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.
    - Obtains 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 as follows:

    - 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 OUTPUT PROCEDURE can consist of any procedure needed to select, modify, or copy the next record made available by the RETURN statement in merged order from the file referenced by *mergefile*.

15. The range of the OUTPUT PROCEDURE additionally includes all statements executed as a result of a CALL, EXIT, GO TO, or PERFORM statement. The range of the OUTPUT PROCEDURE also includes all statements in the Declaratives Section that can be executed if control is transferred from statements in the range of the OUTPUT PROCEDURE.

16. The range of the OUTPUT PROCEDURE must not contain MERGE, SORT, or RELEASE statements.

17. If the MERGE statement is in a *fixed* segment, the OUTPUT PROCEDURE range must be either:

    - Completely in fixed segments
    - Completely contained in one independent segment

18. If the MERGE statement is in an *independent* segment, the OUTPUT PROCEDURE range must be either:

    - Completely in fixed segments
    - Completely contained in the same independent segment as the MERGE statement itself

19. 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.

20. 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*.

21. 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.

22. The MERGE statement begins *outfile* processing as if the program had executed an OPEN statement with the OUTPUT phrase.

23. 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.

24. The MERGE statement terminates *outfile* processing as if the program had executed a CLOSE statement with no optional phrases.

25. 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.

26. 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*.

Additional References

- Chapter 4
- USE statement

MOVE

MOVE — The MOVE statement transfers data to one or more data areas. The editing rules control data transfer.

General Format

Format 1

```
MOVE { src-item } TO { dest-item } . . .
     { lit      }
```

Format 2

```
MOVE { CORRESPONDING } src-item TO dest-item
     { CORR          }
```

`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 the following:
   - HIGH-VALUE
   - HIGH-VALUES
   - LOW-VALUE
   - LOW-VALUES
   - SPACE
   - SPACES
   - QUOTE
   - QUOTES
4. If *lit* is the figurative constant ALL literal and the usage of *dest-item* is COMP-1 or COMP-2, the MOVE statement uses only one occurrence of literal.
5. No operand can be an index data item.

General Rules

1. In Format 2, 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. In Format 1, 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. If *src-item* is reference-modified, subscripted, or indexed, or is a function-identifier, the reference modifier, subscript, index, or function-identifier is evaluated 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 each *dest-item* is evaluated immediately before the move to that item.

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).
```

Elementary Moves

8. 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.

9. These rules apply to elementary moves between categories:

   a. The figurative constant SPACE, or an 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, De-Editing, and Data Conversion During Elementary Moves

10. Editing, de-editing, or other required internal data conversions occur during elementary moves. They are controlled by the description of *dest-item*.

11. 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).

    If the sending operand is numeric and contains the PICTURE symbol (P), all digit positions specified with this symbol are considered to have the value zero and are counted in the size of the sending operand.

12. 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 sending item were described as an unsigned numeric integer.

    d. When *src-item* is numeric edited, the compiler de-edits it before moving it to *dest-item*. *Src-item* can be signed.

13. When *dest-item* is alphabetic, justification and space-filling occur according to the Standard Alignment Rules.

Nonelementary Moves

14. A nonelementary 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.

Summary

Table 6.13 summarizes the valid types of MOVE statements. References after slash marks show the applicable General Rule. For example, moving a numeric edited item to an alphabetic item is invalid because of General Rule 9b.

Table 6.13. Valid MOVE Statements

	Category of Receiving Data Item (dest-item)		
Category of Sending Data Item **(lit or src-item)**	**Alphabetic**	**Alphanumeric Edited** **Alphanumeric**	**Numeric Integer** **Numeric Noninteger** **Numeric Edited**
Alphabetic	Yes/13	Yes/11	No/9a
Alphanumeric	Yes/13	Yes/11	Yes/12

	Category of Receiving Data Item (dest-item)		
Category of Sending Data Item **(lit or src-item)**	**Alphabetic**	**Alphanumeric Edited** **Alphanumeric**	**Numeric Integer** **Numeric Noninteger** **Numeric Edited**
Alphanumeric Edited	Yes/13	Yes/11	No/9a
Numeric Integer	No/9b	Yes/11	Yes/12
Numeric Noninteger	No/9b	No/9c	Yes/12
Numeric Edited	No/9b	Yes/11	Yes/12

Examples

The following examples show the result of executing the statement:

```
MOVE ITEMA TO ITEMB.
```

An s indicates a space character.

- 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

- 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

- 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

- Numeric edited sending item:

	ITEMA PICTURE	ITEMA Value	ITEMB PICTURE	ITEMB Value
a.	Z,ZZZ.99-	1,234.56-	999.999-	234.560-
b.	ZZZ,ZZZ.99-	ss1,234.56-	$$$,$$$.99-	s$1,234.56-
c.	$$$,$$$.99CR	s$1,234.56CR	$$$,$$$.99-	s$1,234.56-
d.	$$$,$$$.99DB	s$1,234.56DB	ZZZ,ZZZ.99CR	ss1,234.56CR
e.	+++++.99	+1234.56	ZZZZZ.99+	s1234.56+
f.	++++++.99	#s-1234.56	ZZZZZZ.99-	ss1234.56-
g.	– – – – –.99	-1234.56	ZZZZZ.99DB	s1234.56DB
h.	– – – – – –.99	ss1234.56	$$,$$$.99	$1,234.56
i.	$$$$.99-	$123.45-	/ XXBXXBXXBXX/	/$1s23s.4s5-/
j.	$$$$.99-	$123.45-	/99B99B99B99/	/00s00s01s23/

Additional References

- Section 5.2.2: COBOL Standard Alignment Rules
- OCCURS clause in Chapter 5
- PICTURE clause in Chapter 5
- SIGN clause in Chapter 5
- Section 6.6.5: CORRESPONDING Phrase

MULTIPLY

MULTIPLY — The MULTIPLY statement multiplies two numeric operands and stores the product in one or more data items.

General Format

Format 1

MULTIPLY num BY { rsult [ROUNDED] } . . .

[ON SIZE ERROR stment]

[NOT ON SIZE ERROR stment2]

[END-MULTIPLY]

Format 2

MULTIPLY num BY num GIVING { rsult [ROUNDED] } . . .

[ON SIZE ERROR stment]

[NOT ON SIZE ERROR stment2]

[END-MULTIPLY]

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.

stment

is an imperative statement executed when an on size error condition has occurred.

stment2

is an imperative statement executed when no on size error condition has occurred.

General Rules

1. In Format 1, the value of *num* is multiplied by the value of the first *rsult*. The product replaces the current value of the first *rsult*. The process repeats for each succesive occurrence of *rsult*.
2. In Format 2, the values of the two operands before the word GIVING are multiplied together. The product replaces the current value of each *rsult*.

Examples

Each of the examples assume these data descriptions and beginning values:

INITIAL VALUES

```
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
```

1. Without GIVING phrase:

   RESULTS

```
MULTIPLY 2 BY ITEMB.                          ITEMB = -70
```

2. SIZE ERROR phrase:

   (When the SIZE ERROR condition occurs, the values of the affected resultant identifiers do not change.)

```
MULTIPLY 3 BY ITEMB
  ON SIZE ERROR                               ITEMB = -35
     MOVE 0 TO ITEMC.                         ITEMC =   0
```

3. NOT ON SIZE ERROR phrase:

```
MULTIPLY 2 BY ITEMB
  ON SIZE ERROR                               ITEMB = -70
     MOVE 0 TO ITEMC
  NOT ON SIZE ERROR
     MOVE 1 TO ITEMC.                         ITEMC =   1
```

4. END-MULTIPLY and MULTIPLY results with SIZE ERROR:

   (The *stment* in the SIZE ERROR phrase executes if any operation causes a size error condition. The first MULTIPLY statement terminates with END-MULTIPLY. The second MULTIPLY executes whether or not the SIZE ERROR condition occurs.)

```
MULTIPLY 4 BY ITEMA ITEMB ITEMC
  ON SIZE ERROR
    MOVE 1 TO ITEMD
END-MULTIPLY
MULTIPLY 2 BY ITEMA ITEMB ITEMC
  ON SIZE ERROR
    ADD 1 TO ITEMD
END-MULTIPLY.
```

After First MULTIPLY	**After Second MULTIPLY**
ITEMA = 16	ITEMA = 32
ITEMB = -35	ITEMB = -70
ITEMC = 40	ITEMC = 80
ITEMD = 1	ITEMD = 1

If the initial value of ITEMB had been -20, a SIZE ERROR condition would not have occurred during the first MULTIPLY. However, the second MULTIPLY would have caused the condition:

After First MULTIPLY	**After Second MULTIPLY**
ITEMA = 16	ITEMA =32
ITEMB = -80	ITEMB =-80
ITEMC = 40	ITEMC =80
ITEMD = 5	ITEMD =6

Additional References

- Section 6.1.4: Scope of Statements

- Section 6.6.1: Arithmetic Operations
- Section 6.6.3: ROUNDED Phrase
- Section 6.6.4: ON SIZE ERROR Phrase
- Section 6.6.7: Overlapping Operands and Incompatible Data
- Section 6.6.2: Multiple Receiving Fields in Arithmetic Statements

OPEN

OPEN — The OPEN statement creates an access stream to the file, makes the file available to the program, begins the processing of a file, and specifies file sharing.

General Format

Format 1—Sequential, Line Sequential, Relative, or Indexed Files

```
OPEN
  { { INPUT  }
    { OUTPUT }
      { file-name [ WITH NO REWIND ]
        [ WITH LOCK (Alpha, I64)
          [ ALLOWING { NO OTHERS
                     { { READERS | WRITERS | UPDATERS } }
                     { ALL                             } } ] } ...
    { { EXTEND }
    { { I-O    }
      { file-name
        [ WITH LOCK (Alpha, I64)
          [ ALLOWING { NO OTHERS
                     { { READERS | WRITERS | UPDATERS } }
                     { ALL                             } } ] } ... } ...
```

Format 2—Report Writer Files

```
OPEN { { OUTPUT { file-name [ WITH NO REWIND ] } ...
       { EXTEND { file-name } ...                   } ...
```

`file-name`

is the name of a file described in the Data Division. It cannot be the name of a sort or merge file.

Leading and trailing blanks are removed from file specifications on all platforms before an OPEN is attempted. Embedded blanks and tabs are removed on OpenVMS systems only.

Syntax Rules

Format 1—Sequential, Line Sequential (Alpha, I64), Relative, or Indexed Files

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.
3. The I-O phrase cannot be used with LINE SEQUENTIAL.
4. The EXTEND phrase can be used for sequential access mode files only.
5. The WITH LOCK phrase cannot be used with the ALLOWING phrase, because it is invalid to specify both X/Open standard (WITH [NO] LOCK or LOCK MODE) and VSI standard (LOCK-HOLDING, ALLOWING, or REGARDLESS) file sharing for the same file connector.

Format 2—Report Writer Files

6. *file-name* must be in a file description entry containing a REPORT clause.

General Rules

All Files

1. Successful OPEN statement execution:
   - Creates an access stream to the file
   - Makes the file available to the program
   - Puts the file in an open mode
   - Associates the file with *file-name* through the file connector
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 applies only to sequential single-reel/unit files. If the concept of rewinding does not apply to the file's storage medium, then the open is successful and an I-O status is set.
6. If the file's storage medium allows rewinding, and:

- There is neither an EXTEND nor a 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
8. If more than one *file-name* is in the OPEN statement, execution is the same as if there were a separate OPEN statement for each *file-name*.
9. A file's maximum record size is established when the file is created and must not subsequently be changed.

Format 1—Sequential, Line Sequential (Alpha, I64), Relative, or Indexed Files

10. A file is available if it is both:
    - Physically present
    - Recognized by the I-O system

    Table 6.14 shows the result of opening available and unavailable sequential, relative, and indexed files.

Table 6.14. Opening Available and Unavailable Sequential, Line Sequential (Alpha, I64), 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	Normal open	Error
I-O (Optional File)	Normal open	The OPEN creates the file
OUTPUT	Creates a new version of the file. See General Rule 24	The OPEN creates the file
EXTEND	Normal open	Error
EXTEND (Optional File)	Normal open	The OPEN creates the file

11. Successful OPEN statement execution makes the file's record area available to the program. If the file connector is an external file connector, the file has only one record area for the executable image.

12. 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

13. An OPEN statement for a file must successfully execute before any allowable input-output statement executes for the file. Table 6.15 shows allowable input-output statements by file organization, access mode, and open mode for sequential, line sequential, relative, and indexed files.

Table 6.15. Allowable Input-Output Statements for Sequential, Line Sequential (Alpha, I64), Relative, and Indexed Files

			Open Mode			
File Organization	**Access Mode**	**Statement**	**INPUT**	**OUTPUT**	**I-O**	**EXTEND**
SEQUENTIAL	SEQUENTIAL	READ	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		WRITE	No	Yes	No	Yes
		UNLOCK	Yes	Yes	Yes	Yes
LINE SEQUENTIAL (Alpha, I64)	SEQUENTIAL	READ	Yes	No	No	No
		REWRITE	No	No	No	No
		WRITE	No	Yes	No	Yes
		UNLOCK	Yes	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	Yes
		UNLOCK	Yes	Yes	Yes	Yes
	RANDOM	DELETE	No	No	Yes	No
		READ	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		WRITE	No	Yes	Yes	No
		UNLOCK	Yes	Yes	Yes	No
	DYNAMIC	DELETE	No	No	Yes	No

			Open Mode			
File Organization	**Access Mode**	**Statement**	**INPUT**	**OUTPUT**	**I-O**	**EXTEND**
		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
		UNLOCK	Yes	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	Yes
		UNLOCK	Yes	Yes	Yes	Yes
	RANDOM	DELETE	No	No	Yes	No
		READ	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		WRITE	No	Yes	Yes	No
		UNLOCK	Yes	Yes	Yes	No
	DYNAMIC	DELETE	No	No	Yes	No
		READ	Yes	No	Yes	No
		READ NEXT	Yes	No	Yes	No
		READ PRIOR	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		START	Yes	No	Yes	No
		WRITE	No	Yes	Yes	No
		UNLOCK	Yes	Yes	Yes	No

14. If the file opened with the INPUT phrase is an optional file that is not present, the OPEN statement sets the File Position Indicator to indicate this condition.

15. An OPEN statement with the EXTEND phrase positions the file immediately after its last logical record. The definition of last logical record differs by file organization:

    - For sequential and line sequential files, it is the last record written in the file.
    - For relative files, it is the currently existing record with the highest relative record number.
    - For indexed files in ascending sort order, it is the currently existing record with the highest prime record key value.

      For indexed files in descending sort order, it is the currently existing record with the lowest prime record key.
    - For Report Writer files, the last logical record is the last record written in the file.

16. Files for which the LINAGE clause has been specified must not be opened in the EXTEND mode.

17. The I-O phrase opens a mass storage file for both input and output operations.

18. The ALLOWING phrase specifies a file-sharing option for the file.

    Automatic record-locking is the system default.

    The ALLOWING phrase, which specifies VSI standard manual record-locking, must be used if the program names this file in the LOCK-HOLDING syntax of the I-O-CONTROL paragraph.

19. When LOCK MODE IS AUTOMATIC or LOCK MODE IS MANUAL is specified and WITH LOCK is not specified, the file is shareable, and can be opened by more than one access stream (except for files opened in OUTPUT mode, which cannot be shared).

20. The NO OTHERS option or WITH LOCK option specifies exclusive file access by this access stream. The access stream created by the OPEN ALLOWING NO OTHERS or OPEN WITH LOCK statement has exclusive access to the file and, therefore, no other concurrent access stream can access (or open) the file.

21. The READERS option permits read-only access to the file for concurrent access streams.

    However, on UNIX systems, the ALLOWING READERS phrase is minimally supported for indexed files, and should not be used. Refer to the description of file handling for indexed files in the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/], in the section on sharing files.

22. The ALL, WRITERS, and UPDATERS phrases allow concurrent access streams access to the file.

23. If there is no ALLOWING phrase or WITH LOCK phrase, the default file-sharing behavior for files depends on the open mode and whether X/Open standard (Alpha or I64) or VSI standard file sharing is in effect.

    For files opened in input mode:

    - VSI standard—The default is ALLOWING READERS (see General Rule 21 for the exception).
    - X/Open standard (Alpha, I64)—The default is to make the file fully shareable.

For files opened in modes other than input mode, the default is always to make the file exclusive. (Also see General Rule 24.)

The selection of X/Open (Alpha, I64) or VSI standard file-sharing default behavior is made as follows by the compiler:

- On Alpha and I64, if X/Open standard syntax (LOCK MODE or WITH [NO] LOCK) has been specified for *file-name* prior to the OPEN statement, the compiler interprets the statement according to the X/Open standard.
- If VSI standard syntax (LOCK-HOLDING, ALLOWING, or REGARDLESS) has been specified for *file-name* prior to the OPEN statement, the compiler interprets the statement according to the VSI standard.
- If no file-sharing syntax (LOCK-HOLDING, ALLOWING, REGARDLESS, LOCK MODE, or WITH [NO] LOCK) has been specified for *file-name* prior to the OPEN statement, then the compiler uses the /STANDARD=[NO]XOPEN qualifier on OpenVMS Alpha and I64 (or the UNIX equivalent `-std [no]xopen` flag) to determine whether the OPEN INPUT statement is interpreted as X/Open or VSI standard: a setting of `xopen` selects the X/Open standard, whereas a setting of `noxopen` selects the VSI standard.

Any subsequent I-O locking syntax for the same file connector in your program must be consistent: X/Open standard locking (Alpha, I64) and VSI standard locking (implicit or explicit) cannot be mixed for the same file connector.

24. On UNIX systems, files opened in OUTPUT mode adhere to the same file-sharing protocols as do files opened in the EXTEND and I-O modes. Access can be denied or granted depending on the file lock requested and the file lock held.

    On OpenVMS systems, file sharing is limited for OUTPUT mode. A higher-numbered version is always created by default.

    On Alpha and I64 systems, if X/Open standard file sharing is in effect, files opened in OUTPUT mode cannot be shared.

25. On UNIX systems, when two file connectors in one process concurrently access the same physical file, a file-locked condition is not generated.

    On OpenVMS systems, when two file connectors in one process concurrently access the same physical file, a file-locked condition might be generated.

26. For files specified with a MULTIPLE FILE TAPE clause:

    - The NO REWIND phrase, if specified, is ignored.
    - Any required rewinding or positioning of the reel (or device) is accomplished according to the relative position of the file as specified in the MULTIPLE FILE TAPE clause.

27. An OPEN OUTPUT statement for a file specified with a POSITION phrase of a MULTIPLE FILE TAPE clause is invalid unless the tape contains all the files at positions prior to the position specified.

28. An OPEN OUTPUT statement for a file specified with a POSITION phrase of a MULTIPLE FILE TAPE clause is invalid if the tape already contains a file at the position specified.

29. An OPEN INPUT statement for a file specified with a POSITION phrase of a MULTIPLE FILE TAPE clause is invalid unless the tape contains a file at that position, as well as all the files at the positions prior to the position specified.

30. A file specified in a MULTIPLE FILE TAPE clause cannot be opened in either I-O or EXTEND mode.

Format 2—Report Writer Files

31. A file is available if it is physically present and recognized by the I-O system.

    Table 6.16 shows the results of opening available and unavailable Report Writer files.

Table 6.16. Opening Available and Unavailable Report Writer Files

Open Mode	File Is Available	File Is Unavailable
OUTPUT	Creates a new version of the file	The OPEN creates the file
EXTEND	Normal OPEN	The OPEN is unsuccessful
EXTEND (optional file)	Normal OPEN	The OPEN creates the file

32. Successful OPEN statement execution makes the file's record area available to the Report Writer Control System. If the file connector is an external file connector, the file has only one record area for the executable image.

33. When a file is not in an open mode, no statement that references the file can execute either implicitly or explicitly, except for the OPEN statement.

34. An OPEN statement for a *file-name* must execute successfully before an INITIATE statement executes for the file. Table 6.17 shows allowable Report Writer statements by file organization and open mode for Report Writer files.

Table 6.17. Allowable Statements for Report Writer Files

	Open Mode	
Statement	**OUTPUT**	**EXTEND**
INITIATE	Yes	Yes
GENERATE	Yes	Yes
SUPPRESS	Yes	Yes
TERMINATE	Yes	Yes
All other I-O statements	No	No (for record I-O only)

Technical Notes

- OPEN statement execution can result in these FILE STATUS data item values:

File Status	Meaning
00	Open is successful.
05	Optional file not present.
07	No rewind on non-reel device.
35	File not found.

File Status	Meaning
37	An open in I-O mode is attempted for a nonmass storage file, or an open is attempted for nonmass storage file that was declared as a relative or indexed file.
38	An open is attempted on a file closed with lock.
39	A mismatch exists between the current program's description of an index key and the existent file's description of the key, or (for OpenVMS systems) there is a conflict of maximum record size or record type.
41	File is already open
91	Open is unsuccessful; file locked by another access stream.
95	No file space on device.
30	All other permanent errors.

You must use the I-O-CONTROL statement APPLY PREALLOCATION with a value greater than 0 (the default is 0) to enable the detection of "device full" (file status 95) with the OPEN statement.

- Attempts to specify both X/Open standard and VSI standard file-sharing syntax for the same file connector are invalid. When the compiler cannot detect such attempts because they occur in different compilation units, the run-time system detects and reports the violations (file status 30). This holds for explicit and implicit usage.

Additional References

- Chapter 4
- CLOSE statement
- USE statement

PERFORM

PERFORM — The PERFORM statement executes one or more intra-program procedures. It returns control to the end of the PERFORM statement when all procedures have completed execution.

General Format

Format 1

```
PERFORM [ first-proc [ { THRU } end-proc ] ]
                       { THROUGH }

  [ stment END-PERFORM ]
```

Format 2

```
PERFORM [ first-proc [ { THRU } end-proc ] ] repeat-count TIMES
                       { THROUGH }

  [ stment END-PERFORM ]
```

Format 3

```
PERFORM [ first-proc [ { THRU } end-proc ] ]
                       { THROUGH }

[ WITH TEST { BEFORE } ] UNTIL cond
            { AFTER  }

  [ stment END-PERFORM ]
```

Format 4

```
PERFORM [ first-proc [ { THRU } end-proc ] ] [ WITH TEST { BEFORE } ]
                     { THROUGH }                         { AFTER  }

  VARYING var FROM init BY increm UNTIL cond

  [ AFTER var FROM init BY increm UNTIL cond ] . . .

  [ stment END-PERFORM ]
```

`first-proc`

is a procedure-name that identifies a paragraph or section in the Procedure Division. The set of statements in *first-proc* is 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* is the last set of statements in the PERFORM range.

`stment`

is an imperative statement.

`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 integer 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 there is no *first-proc*, the PERFORM statement must contain *stment* and the END-PERFORM phrase. If there is a *first-proc*, the statement cannot have the END-PERFORM phrase.
2. If either *first-proc* or *end-proc* is placed in the Declaratives part of the Procedure Division, both must also be placed in the same DECLARATIVES section.
3. The words THRU and THROUGH are equivalent and interchangeable.

Formats 3 and 4

4. If there is no TEST BEFORE or TEST AFTER phrase, TEST BEFORE is the default.

Format 4

5. If there is no *first-proc*, there can be no AFTER phrase.
6. 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
7. 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 *first-proc* appears, the statement is an out-of-line PERFORM statement. Otherwise, it is an in-line PERFORM statement.
2. The statements in the range of *first-proc* to *end-proc* for an out-of-line PERFORM are the statement set. For an in-line PERFORM, the statement set is contained within the scope of the PERFORM...END-PERFORM syntax.
3. Unless restricted to in-line or out-of-line statements, all General Rules apply to both types of PERFORM statements. An in-line PERFORM statement operates according to the general rules for an out-of-line PERFORM, except for periods, which are not allowed within the body of the PERFORM. The statements in the in-line PERFORM execute in place of the statements in the range of *first-proc* to *end-proc*.
4. When the PERFORM statement executes, control transfers to the first statement of *first-proc*. However, for Format 2, 3, or 4 PERFORM statements, transfer of control depends on evaluation of the specified condition.

   Transfer of control occurs only once for each PERFORM statement executed. When transfer of control does occur, after the statement set executes, control implicitly transfers back to the to the end of the perform statement as follows:

- 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*.
- If the statement is an in-line PERFORM, execution ends after the last statement of the statement set.

5. *first-proc* and *end-proc* need not be related except that *first-proc* is the beginning and *end-proc* is the last in a consecutive series of operations.

   GO TO and PERFORM statements can occur between *first-proc* and *end-proc*. If there are two or more logical paths to the return point, *end-proc* can be a paragraph, consisting of the EXIT statement, to which all these paths must lead.

6. If a statement other than a PERFORM statement, transfers control to the statement set, at the end of the statement set, control transfers through the last statement of the set to the next executable statement following the set as if no PERFORM statement referenced the set.

7. 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.

8. The range of the PERFORM statement additionally includes all statements executed as a result of a CALL, EXIT, GO TO, or PERFORM statement. The range the PERFORM statement also includes all statements in the Declaratives Section that can be executed if control is transferred from statements in the range of the PERFORM statement.

9. Statements executed as the result of a control transfer caused by an EXIT PROGRAM statement are not part of the range when:

   - The EXIT PROGRAM statement is specified in the same program as the PERFORM statement, and
   - The EXIT PROGRAM statement is within the range of the PERFORM statement.

10. A PERFORM statement in a *fixed* segment can have only one of the following in its range:

    - Sections and paragraphs completely contained in one or more nonindependent segments
    - Sections and paragraphs completely contained in one independent segment

    However, the PERFORM statement range also includes any Declarative procedures activated during its execution.

11. A PERFORM statement in an *independent* [1] segment can have only one of the following in its range:

[1] Segmentation is described in Section 6.7. VSI COBOL supports segmentation for compatibility with existing applications only. VSI recommends that you do not use segmentation in new applications.

- Sections and paragraphs completely contained in one or more nonindependent segments
- Sections and paragraphs completely contained in the same independent segment as the PERFORM statement itself

However, the PERFORM statement range also includes any Declarative procedures activated during its execution.

12. *first-proc* and *end-proc* cannot name sections or paragraphs in any other program in the executable image. Statements in other programs are in a PERFORM statement's range only if the range includes a CALL statement.

13. 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.

    For example:

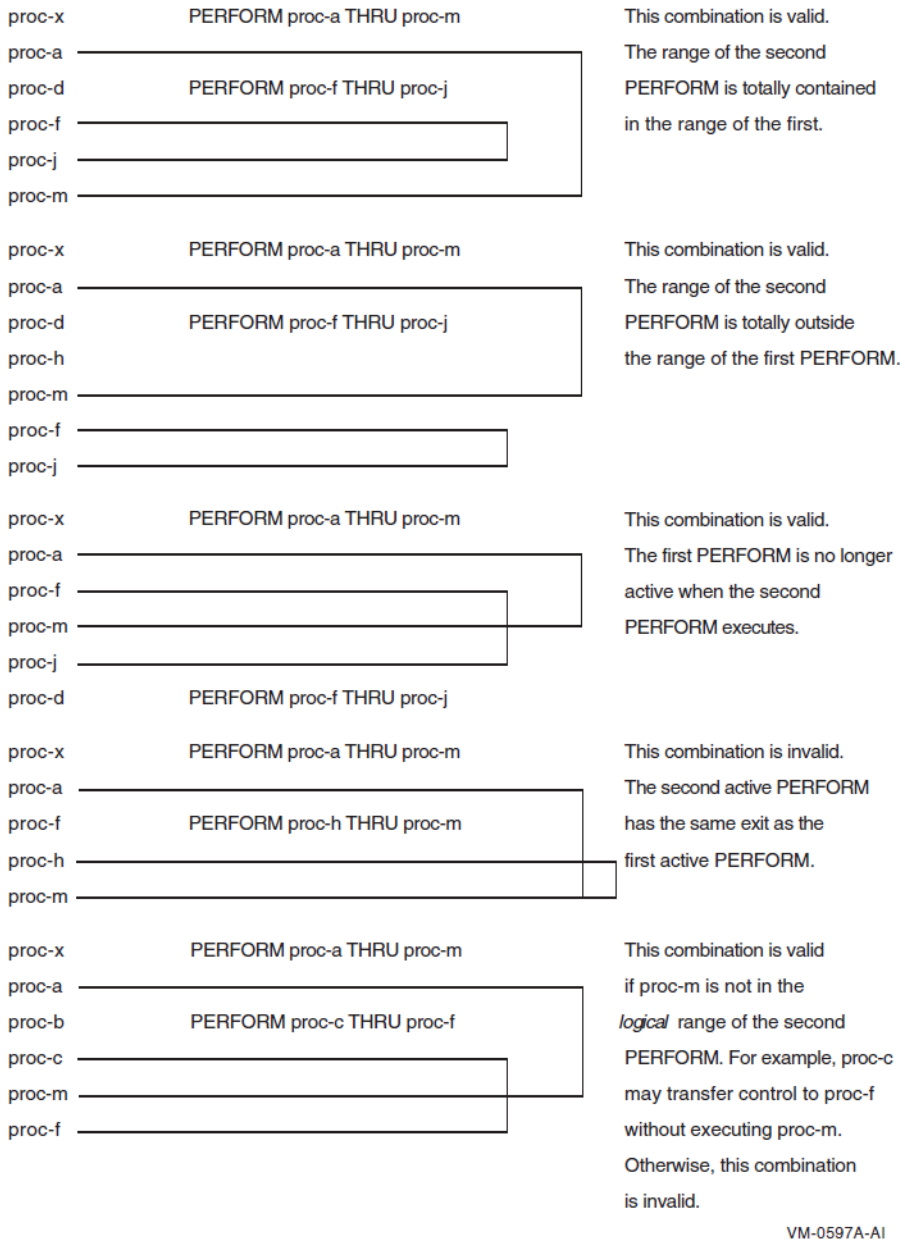
    - An active PERFORM statement whose execution point is in the range of another active PERFORM statement must not allow control to pass to the exit of the other active PERFORM.
    - Two or more active PERFORM statements cannot have a common exit.

    Use the `check` compiler option with the `perform` keyword to verify at run time that there is no recursive activation of a PERFORM.

    Figure 6.2 shows valid and invalid nested PERFORM statements.

14. Undocumented results might occur when *end-proc* precedes *first-proc* or when *first-proc* and *end-proc* are not in the same program segment.

Figure 6.2. Valid and Invalid Nested PERFORM Statements



Format 1

15. Format 1 is the basic PERFORM statement. The statement sets in the PERFORM range execute once. Control then passes to the end of the PERFORM statement.

Format 2

16. The statement sets execute 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 sets execute.

Format 3

17. The statement sets execute until *cond* is true. Control then transfers to the end of the PERFORM statement.

18. If *cond* is true when the PERFORM statement executes:

    - If there is a TEST BEFORE phrase or one is implied, there is no transfer to *first-proc*; control passes to the end of the PERFORM statement.

    - If there is a TEST AFTER phrase, the PERFORM statement tests *cond* after the statement set executes.

Format 4

19. The Format 4 PERFORM statement systematically changes the value of *var* during its execution.

20. If *var* is an index-name, its value, when the PERFORM statement execution begins, must equal the occurrence number of an element in its table.

21. 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.

22. *increm* must not be zero.

23. *init* must be positive when *var* is an index-name and *init* is an identifier.

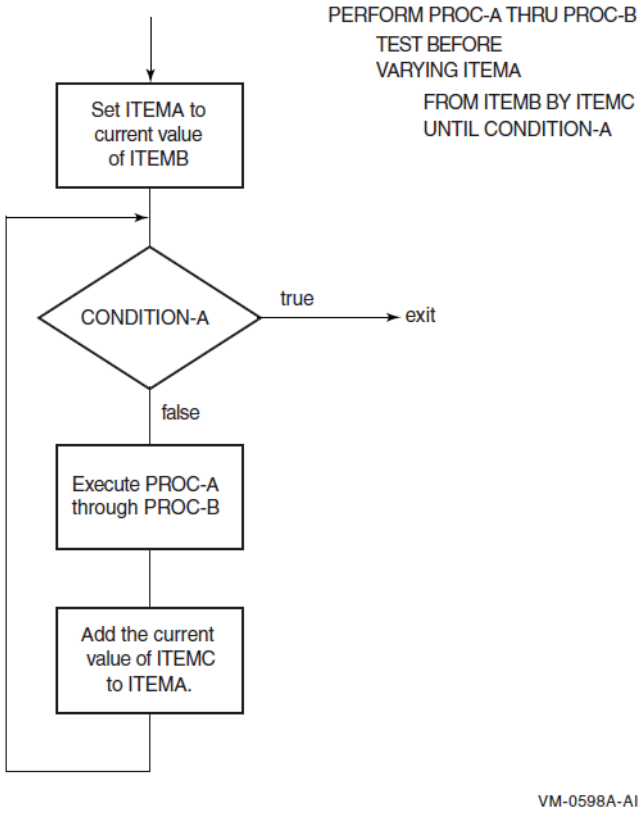
24. If there is a TEST BEFORE phrase (or one is implied), and one *var* is varied (see Figure 6.3):

    - *var* is set to the value of *init* when the PERFORM statement begins to execute.

    - 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.

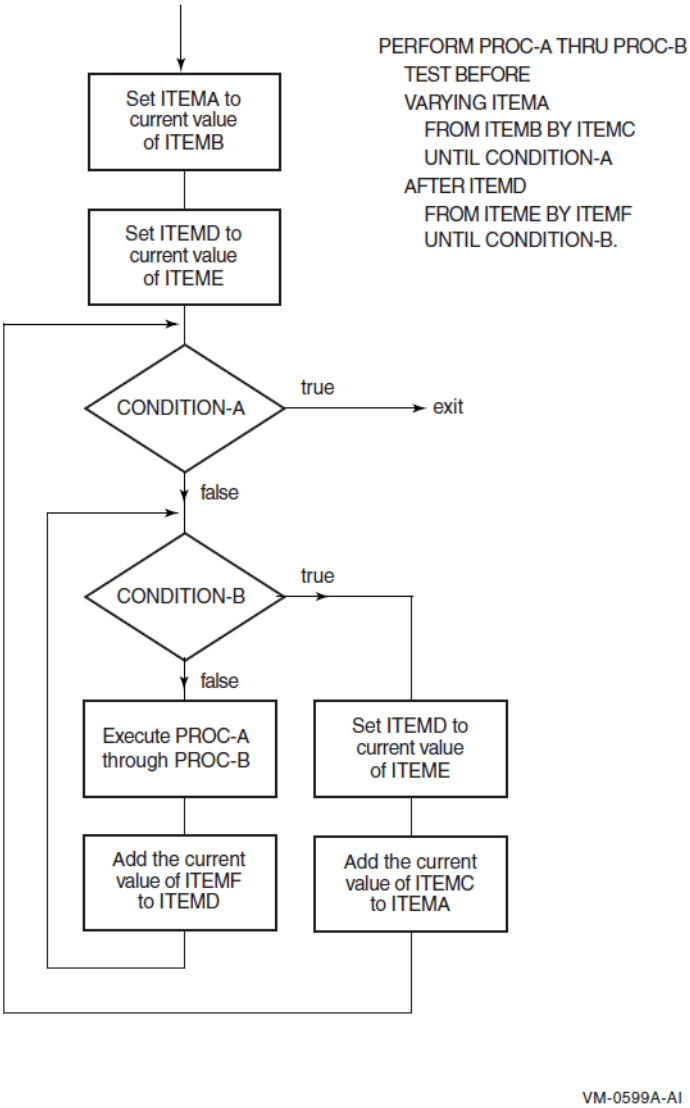
Figure 6.3. PERFORM … VARYING with the TEST BEFORE Phrase and One Condition



25. If there is a TEST BEFORE phrase (or one is implied), and the PERFORM statement has two *vars* (see Figure 6.4):

    - The first and second *vars* are set to the value of the first and second *init* when the PERFORM statement begins to execute.
    - 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 value of the first *var* changes by the value of the first *increm*, and the second *var* is set to the value of the second *init*. The first *cond* is reevaluated. The PERFORM statement ends if the first *cond* is true. Otherwise, the cycle continues until *cond* is true.

Figure 6.4. PERFORM … VARYING with the TEST BEFORE Phrase and Two Conditions



26. At the end of a PERFORM statement with the TEST BEFORE phrase:

- 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*.

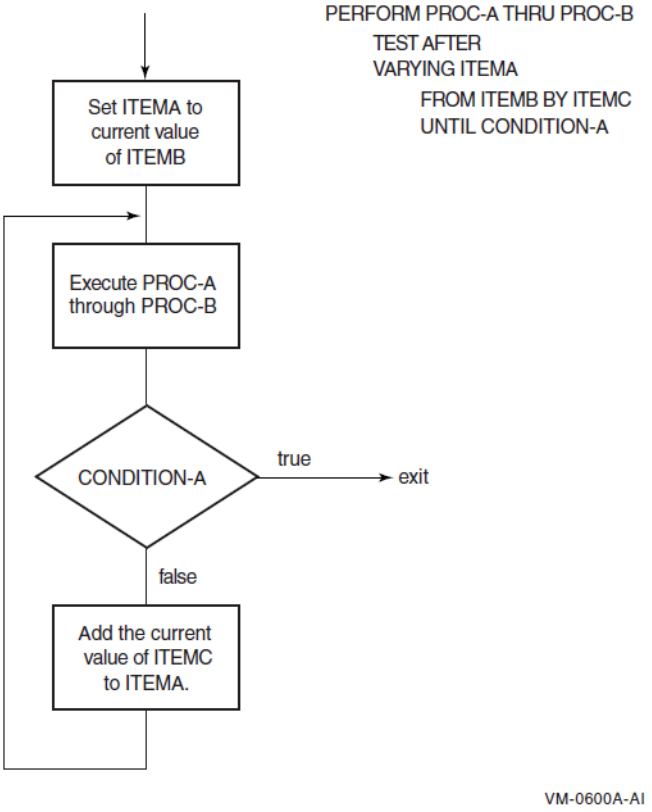
- The value of each other *var* equals the current value of its associated *init*.

27. If there is a TEST AFTER phrase and one *var* is varied (see Figure 6.5):

- *var* is set to the value of *init* when the PERFORM statement starts to execute.

- The statement set executes once. Then, *cond* is evaluated. If it is false, the value of *var* changes by the increment or decrement value (*increm*), and the statement set executes again. This cycle continues until *cond* is true. Control then transfers to the end of the PERFORM statement.

Figure 6.5. PERFORM … VARYING with the TEST AFTER Phrase and One Condition



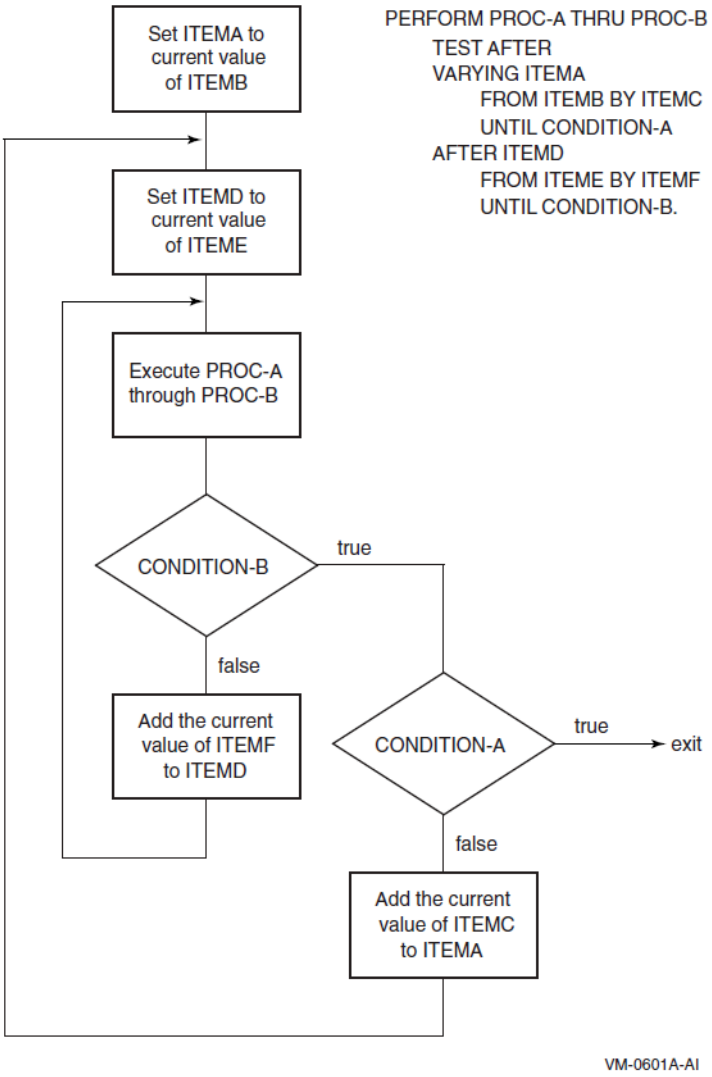
28. If there is a TEST AFTER phrase, and two *vars* are varied (see Figure 6.6):

- The first and second *vars* are set to the value of the first and second *init* when the PERFORM statement starts to execute.

- The statement set executes. The second *cond* is then evaluated. If it is false, the second *var* changes by the value of *increm*, and the statement set executes again. This cycle continues until the second *cond* is true.

- When the second *cond* is true, the first *cond* is evaluated. If it is false, the value of the first *var* changes by the value of the first *increm*, the second *var* is set to the value of the second *init*, and the statement set executes again. The PERFORM statement ends if the first *cond* is true. Otherwise, the cycle continues until *cond* is true.

Figure 6.6. PERFORM … VARYING with the TEST AFTER Phrase and Two Conditions



29. At the end of a PERFORM statement with the TEST AFTER phrase, the value of each *var* is the same as at the end of the most recent statement set execution.

30. During execution of the sets of statements in the range, any change to *var*, *increm*, or *init* affects PERFORM statement operation.

31. 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.

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.
   ```

   ITEMB = Asssssssss

2. Performing a range of procedures (Format 1):

   ```
   PERFORM PROC-A THRU PROC-B.
   ```

   ITEMB = AssssssssA

3. Performing a range of procedures (Format 2):

   ```
   PERFORM PROC-A THRU PROC-C
      3 TIMES.
   ```

   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.
   ```

   ITEMB = ABCDEssssE

   ITEMC = 06

5. Testing the UNTIL condition after execution (Format 4):

```
PERFORM PROC-A THRU PROC-B
     TEST AFTER
     VARYING ITEMC FROM 1 BY 1
  UNTIL ITEMC > 5.
```

ITEMB = ABCDEFsssF

ITEMC = 06

6. 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.
```

ITEMB = sssDEFGssD

ITEMC = 03

7. In-line PERFORM (Format 4):

```
PERFORM
  VARYING ITEMC FROM 1 BY 2
  UNTIL ITEMC > 7
   MOVE CHARA (ITEMC) TO CHARB (ITEMC)
   MOVE CHARA (ITEMC) TO CHARB (ITEMC + 3)
END-PERFORM.
```

ITEMB = AsCAECGEsG

8. 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.
```

ITEMG (1) = 01s02s03s04s05s

ITEMG (2) = 02s04s06s08s10s

ITEMG (3) = 03s06s09s12s15s

ITEMG (4) = 04s08s12s16s20s

Additional References

- Section 6.1.4: Scope of Statements
- Section 6.5: Conditional Expressions
- SET statement
- Online help, `check` compiler option

READ

READ — 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—Sequential, Line Sequential, Relative, or Indexed Files

```
READ file-name [ NEXT
                 PREVIOUS (Alpha, I64)
                 PRIOR (Alpha, I64) ] RECORD [ INTO dest-item ]

    [ WITH [NO] LOCK (Alpha, I64)
      REGARDLESS OF LOCK
      ALLOWING { UPDATERS
                 READERS
                 NO OTHERS } ]

  [ AT END stment ]

  [ NOT AT END stment2 ]

  [ END-READ ]
```

Format 2—Relative Files

```
READ file-name RECORD [ INTO dest-item ]

    [ WITH [NO] LOCK (Alpha, I64)
      REGARDLESS OF LOCK
      ALLOWING { UPDATERS
                 READERS
                 NO OTHERS } ]

  [ INVALID KEY stment ]

  [ NOT INVALID KEY stment2 ]

  [ END-READ ]
```

Format 3—Indexed Files

```
READ file-name RECORD [ INTO dest-item ]

    [ WITH [NO] LOCK (Alpha, I64)
      REGARDLESS OF LOCK
      ALLOWING { UPDATERS
                 READERS
                 NO OTHERS } ]

  [ KEY IS key-data ]

  [ INVALID KEY stment ]

  [ NOT INVALID KEY stment2 ]

  [ END-READ ]
```

`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 when the relevant condition (at end or invalid key) occurs.

`stment2`

is an imperative statement executed when the relevant condition (not at end or not invalid key) occurs.

`key-data`

is the data-name of a data item specified as a record key for *file-name* or the segmented-key name 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. READ file-name PRIOR and READ file-name PREVIOUS are equivalent syntax.
4. Format 2 can be used for random or dynamic access mode files to retrieve records randomly.
5. The KEY phrase can be used only for indexed files.
6. To use the REGARDLESS or ALLOWING options the program must specify these entries:
   - APPLY LOCK-HOLDING clause of the I-O-CONTROL paragraph
   - ALLOWING option of the OPEN statement
7. There must be an INVALID KEY or AT END phrase when there is no applicable USE AFTER EXCEPTION procedure for the file.
8. The storage area associated with *dest-item* and the record area associated with *file-name* cannot be the same storage area.
9. On Alpha and I64 systems, the WITH [NO] LOCK phrase is X/Open standard syntax. It is invalid to specify both X/Open standard and VSI standard (LOCK-HOLDING, ALLOWING, OR REGARDLESS) file-sharing syntax for the same file connector. Hence, the WITH [NO] LOCK phrase cannot be used with the ALLOWING or REGARDLESS phrase.

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. READ PRIOR can only be used with an INDEXED file whose organization is INDEXED and whose access mode is DYNAMIC. The file must be opened for INPUT or I-O.

4. Executing a Format 1 READ statement can cause the following to occur:
   - The record pointed to by the File Position Indicator becomes available in the file's record area.
   - For sequential and relative files, the File Position Indicator points to the file's next existing record.
   - For indexed files, the File Position Indicator points to the next existing record established by the file's Key of Reference.
   - If the file has no next record, the File Position Indicator indicates that no next logical record exists.
5. The READ statement updates the value of the FILE STATUS data item for the file.
6. A record is available before any statement executes after the READ.
7. 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.
8. 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.
9. During execution of a Format 2 READ statement, the File Position Indicator can indicate that an optional file is not present. The invalid key condition then exists, and READ statement execution is unsuccessful.
10. When a Format 1 READ statement executes, the File Position Indicator can indicate that:
    - There is no next logical record.
    - No valid next record has been established.
    - An optional file is not present.
    - The number of significant digits in the relative record number is larger than the relative key data items.

    When the READ statement detects the no valid next record condition, the READ is unsuccessful.

    When the READ statement detects one of the above conditions, not including the no valid next record condition:
    - It updates the FILE STATUS data item for the file to indicate the at end condition.
    - If the READ statement has an AT END phrase, control transfers to *stment*. No USE AFTER EXCEPTION procedure for the file executes.
    - 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.

11. 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 File Position Indicator is unchanged; otherwise, it indicates that no valid next record has been established. For indexed files, the Key of Reference is undefined.

12. READ PRIOR retrieves a record from an Indexed file which logically precedes the one which was made current by the previous file access operation, if such a logically previous record exists.

13. 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. For an indexed file, when the Key of Reference has ascending sort order, the next logical record is the next record with a key value equal to or *greater* than the previous key value. When the Key of Reference has descending sort order, the next logical record is the next record with a key value equal to or *less* than the previous key value.

14. 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.

15. For a relative file, a Format 2 READ statement sets the File Position Indicator 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.

16. When your program sequentially accesses an indexed file for records with duplicate record key values in the Key of Reference, 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.

17. 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.

18. 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.

19. 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. The READ statement sets File Position Indicator 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.

20. The Format 2 READ verb can use the KEY IS syntax to establish the key field within the file record which is the Key of Reference. An immediately subsequent READ PRIOR will follow the order of the Key of Reference to access the logically previous record in the file according to that Key of Reference. If the KEY IS syntax is not used, the Key of Reference is understood to be the file's primary key field.

21. When a successful READ PRIOR has occurred and the Key of Reference has ascending order, the record retrieved can have the same key value or a smaller key value than the preceding record for the Key of Reference. If the Key of Reference has descending order, the record retrieved can have

the same key value or a higher key value for the Key of Reference. The retrieved record can have the same key value if duplicate values for the Key of Reference exist on the file.

22. When a READ PRIOR has been executed and a logically previous record does not exist, a File Status value of 10 indicating END-OF-FILE is returned. A READ PRIOR which is done immediately after Opening the file will produce the END-OF-FILE status.

23. If the number of character positions in the record being read is less than the minimum size specified by the record description entries for the file, the record area to the right of the last valid character read is undefined.

    If the number of character positions in the record being read is greater than the maximum size specified by the record description entries for the file, the record is truncated on the right to the maximum size.

    In both cases, the READ operation is successful and the I-O status is set to indicate a record length conflict has occurred.

24. The REGARDLESS and ALLOWING options can be used only in a VSI standard manual record-locking environment. To create a manual record-locking environment, an access stream must specify the APPLY LOCK-HOLDING clause of the I-O-CONTROL paragraph.

25. On UNIX and OpenVMS, the REGARDLESS option enables an access stream to read a record regardless of any record locks held by other concurrent access streams. READ REGARDLESS holds no lock on the record read.

    This statement generates a soft record lock condition if the record is locked by another access stream. This condition results in a File Status value of 90 and invokes an applicable USE procedure, if any. Execution of the READ REGARDLESS statement is considered successful and program execution resumes at the next statement following the READ REGARDLESS statement.

    However, on UNIX systems, the soft lock condition (file status 90) is not recognized for indexed files. A READ REGARDLESS statement for a record locked by another process performs the requested read operation on the record and returns a file status of 00.

26. The ALLOWING UPDATERS and WITH NO LOCK options permit other concurrent access streams in a manual record-locking environment to simultaneously READ, DELETE, START, and REWRITE the current record. These options hold no locks on the current record.

27. The ALLOWING READERS option permits other concurrent access streams in a VSI standard, manual record-locking environment to simultaneously read the current record. This option holds a read-lock on each such record read. No access stream can update the current record until it is unlocked.

    On UNIX systems, for indexed files, the ALLOWING READERS phrase has some limitations, which are described in the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/] (see the section on indicating access allowed to other streams in the chapter on sharing files).

28. The ALLOWING NO OTHERS or WITH LOCK option locks the record read by the current access stream. No other concurrent access stream can access this record until it is unlocked. Only this access stream can update this record. This option applies to files opened in I-O mode. See general rule 29.

29. For files opened for input, a READ statement does not acquire a record lock, regardless of the locking syntax specified. This applies to X/Open standard and VSI standard locking.

30. If there is an applicable USE AFTER EXCEPTION procedure, it executes whenever an input condition occurs that would result in a nonzero value in the first character of 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.

31. If no exception condition exists, the record is made available in the record area. Control is transferred to the end of the READ statement; however, if *stment2* is specified, *stment2* executes before control is transferred to the end of the READ statement.

Technical Note

- READ statement execution can result in these FILE STATUS data item values:

File Status	File Organization	Access Method	Meaning
00	All	All	Read is successful; record is available; lock acquired as requested
02	Ind	All	Read is successful; duplicate key detected
04	All	All	Record read larger or smaller than record area
10	All	Seq	No next logical record (at end), optional file not present (at end), or no valid next record (at end)
14	Rel	Seq	Relative record number too large for relative key data item (at end)
23	Ind, Rel	Rand	Record not in file (invalid key) or optional file not present (invalid key)
46	All	Seq	No valid next record
47	All	All	File not open, or incompatible open mode
90	All	All	Record locked by another user; record is available in record area; no lock is acquired (soft lock for VSI standard locking only)
92	All	All	Record locked by another user; record is not available; no lock is acquired (hard lock)
30	All	All	All other permanent errors

- On Alpha and I64 systems, use START before initiating a sequence of either READ NEXT statements or READ PRIOR/READ PREVIOUS statements. You should use START, if you switch between READ NEXT and READ PRIOR/READ PREVIOUS or vice versa.

- On Alpha and I64 systems, the order of duplicate key values for a descending key is not necessarily the same as the order of duplicate key values for READ PRIOR/READ PREVIOUS used with an ascending key defined as the same file record location as the descending key.

Additional References

- Chapter 4

- Section 6.1.4: Scope of Statements

- Section 6.6.8: I-O Status

- Section 6.6.10: INVALID KEY Phrase
- Section 6.6.9: AT END Phrase
- Section 6.6.12: INTO Phrase
- OPEN statement
- UNLOCK statement
- USE statement

RECORD (OpenVMS Only)

RECORD (OpenVMS Only) — The RECORD statement creates an Oracle CDD/Repository dependency relationship between an VSI COBOL for OpenVMS program and the Oracle CDD/Repository entity referred to by the RECORD statement.

General Format

RECORD DEPENDENCY path-name [TYPE IS relation-type] [IN DICTIONARY]

path-name

is a partial or complete Oracle CDD/Repository path name. It specifies a dictionary entity in CDO format.

relation-type

is a valid Oracle CDD/Repository protocol. It specifies the type of relationship to be created between the VSI COBOL for OpenVMS program and the CDO dictionary entity specified in the path name. The default relationship type is CDD$COMPILED_DEPENDS_ON.

Syntax Rules

1. A space must precede the word RECORD.
2. The RECORD statement must be terminated by the separator period.

General Rules

1. *path-name* refers to the Oracle CDD/Repository path for a dictionary entity. The entity must be in CDO format.
2. The RECORD statement creates an Oracle CDD/Repository relationship between the VSI COBOL for OpenVMS program through a compiled module entity (see Technical Notes) and the dictionary entity specified in the path name. This relationship information is then stored in Oracle CDD/Repository.
3. The RECORD statement is ignored unless the /DEPENDENCY_DATA compiler option is specified.
4. If the RECORD statement is in a contained program, the relationship created is between the outermost containing program and the entity specified in the path name.

Technical Notes

1. The *path-name* can be a nonnumeric literal or COBOL word formed according to the rules for user-defined names. It represents a complete or partial Oracle CDD/Repository path name specifying an Oracle CDD/Repository entity. If *path-name* is not a literal, the compiler translates hyphens in the COBOL word to underscore characters.

   The resultant path name must conform to all rules for forming Oracle CDD/Repository path names.

2. The *relation-type* can be a nonnumeric literal or COBOL word formed according to the rules for user-defined names. It must be a valid Oracle CDD/Repository protocol type. For example:

   - CDD$COMPILED_DEPENDS_ON is an example of a COBOL word that is a valid Oracle CDD/Repository protocol type.

   - CDD$COMPILED_DERIVED_FROM is an example of a nonnumeric literal that is also a valid Oracle CDD/Repository protocol type.

3. The RECORD statement creates a relationship between an Oracle CDD/Repository compiled module dictionary entity and the dictionary entity specified in the path name. A compiled module entity is automatically created and stored in Oracle CDD/Repository when the /DEPENDENCY_DATA compiler option is specified. The name of the compiled module entity is the *program-name* from the PROGRAM-ID paragraph, with hyphens translated to underscores.

Additional References

- The description of Oracle CDD/Repository in the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/]

- Oracle CDD/Repository Documentation Set

RELEASE

RELEASE — 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 in the same program as the SORT verb. The input procedure must be associated with a SORT statement for the sort or merge file that contains *rec*.

2. If *src-area* is a function-identifier, it must reference an alphanumeric function. When *src-area* is not a function-identifier, *rec* and *src-area* must not reference the same storage area.

General Rules

1. See the FROM Phrase section for a list of rules.
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

- Chapter 4
- Section 6.6.11: FROM Phrase phrase

RETURN

RETURN — The RETURN statement obtains 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
    [ NOT AT END stment2 ]
    [ END-RETURN ]
```

`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 executed for an at end condition.

`stment2`

is an imperative statement executed for a not at end condition.

Syntax Rules

1. A RETURN statement can be used only in an output procedure in the same program as the SORT verb. 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. See the INTO Phrase section for a list of rules.

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 File Position Indicator is updated. It points to the record whose key values make it first in the file. If there are no records, the File Position Indicator indicates the at end condition.

4. The RETURN statement makes the next record (pointed to by the File Position Indicator) available in the record area for *smrg-file*.

5. The File Position Indicator is updated to point to the next record in *smrg-file*. The key values in the SORT or MERGE statement determine the next record.

6. If *smrg-file* has no next record, the File Position Indicator is updated to indicate the at end condition.

7. If the File Position Indicator indicates the at end condition when the RETURN statement executes, *stment* executes and control is transferred to the end of the RETURN statement. If the NOT AT END phrase is specified, it is ignored. The contents of the *smrg-file* record areas are undefined.

8. If the File Position Indicator does not indicate an at end condition when the RETURN statement executes, after the record is made available and after executing any implicit move resulting from the presence of an INTO phrase, control is transferred to *stment2*, if specified. Otherwise, control is transferred to the end of the RETURN statement.

9. When the at end condition occurs:

   - RETURN statement execution is unsuccessful.

   - The File Position Indicator is not changed.

10. See the Scope of Statements section for a description of scope terminators such as END-RETURN.

Additional References

- Section 6.1.4: Scope of Statements

- Chapter 4

- Section 6.6.9: AT END Phrase

- Section 6.6.12: INTO Phrase

REWRITE

REWRITE — The REWRITE statement logically replaces a mass storage file record.

General Format

Format 1—Sequential Files

```
REWRITE rec-name [ FROM src-item ]
    [ ALLOWING NO OTHERS ]
    [ END-REWRITE ]
```

Format 2—Relative or Indexed Files

```
REWRITE rec-name [ FROM src-item ]
    [ ALLOWING NO OTHERS ]
    [ INVALID KEY stment ]
    [ NOT INVALID KEY stment2 ]
    [ END-REWRITE ]
```

`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 executed for an invalid key condition.

`stment2`

is an imperative statement executed for a not invalid key condition.

Syntax Rules

1. To use the ALLOWING option, the program must include these entries:
   - APPLY LOCK-HOLDING clause of the I-O-CONTROL paragraph
   - ALLOWING clause of the OPEN statement
2. The INVALID KEY and the NOT INVALID KEY phrases cannot be specified in a REWRITE statement that refers to a sequential file or to a relative file with sequential access mode.
3. 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.
4. If *src-item* is a function-identifier, it must reference an alphanumeric function. When *src-item* is not a function-identifier, *rec-name* and *src-item* must not reference the same storage area.
5. The ALLOWING clause is VSI standard file-sharing syntax, and cannot be used for a file connector that has had X/Open standard file-sharing syntax (WITH [NO] LOCK or LOCK MODE) specified.

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.

   Because line sequential files (Alpha, I64) cannot be opened in I-O mode, REWRITE cannot be used with line sequential files.

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 READ must lock the record for the REWRITE statement to be successful.

4. The record in *rec-name* is no longer available after the REWRITE statement successfully executes unless the associated file-name is in a SAME RECORD AREA clause. In this case, the record is also available to record areas of other file-names in the SAME RECORD AREA clause.

5. The REWRITE statement does not affect the File Position Indicator.

6. The REWRITE statement updates the value of the FILE STATUS data item for the file.

7. The ALLOWING option can be used only in a VSI standard, manual record-locking environment. To create a manual record-locking environment, the program must OPEN file-name with an ALLOWING option and specify the APPLY LOCK-HOLDING phrase of the I-O-CONTROL paragraph.

8. The ALLOWING option locks the current record rewritten by the current access stream. No other concurrent access stream can access this record until it is unlocked.

   However, on UNIX systems, for indexed files the REWRITE statement with the ALLOWING clause does not acquire a record lock.

9. See the FROM Phrase section for a list of rules for the FROM phrase.

Sequential Files

10. The record named by *rec-name* must be the same size as the record being replaced.

Relative Files

11. 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

12. 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.

13. For a random access mode file, the prime record key specifies the record to replace. If the program specifies duplicates on the prime record key, then it can replace only the first occurrence of a key

value using random access mode. Replacing subsequent records with the same prime key value is done by sequentially positioning to the desired record in sequential or dynamic access mode.

14. For indexed files in dynamic access mode, the presence of DUPLICATES on the prime record key determines the behavior. If DUPLICATES are allowed, Rule 11 applies. If DUPLICATES are not allowed, Rule 12 applies.

15. 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.

16. 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.

17. 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 the first character position of 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.

    See the rules for the INVALID KEY phrase, Section 6.6.10.

18. The number of character positions in the record to be updated must not be less than the lowest or greater than the highest number of character positions allowed by the RECORD VARYING clause. In either case, the REWRITE statement is unsuccessful and the following occurs:

    - The updating operation does not take place.

    - The contents of the record area remain unaffected.

    - The I-O status of the file is set to a value that indicates the cause of the condition.

Technical Notes

- REWRITE statement execution can result in these FILE STATUS data item values:

File Status	File Organization	Access Method	Meaning
00	All	All	Rewrite is successful.

File Status	File Organization	Access Method	Meaning
02	Ind	All	Created duplicate primary or alternate key.
21	Ind	Seq	Primary key changed after read.
22	Ind	All	Duplicate primary or alternate key (invalid key).
23	Ind, Rel	Rand	Record not in file (invalid key).
43	All	Seq	No previous read or record not locked by prior READ or START.
44	All	All	Invalid record size.
49	All	All	File not open, or incompatible open mode.
92	Ind, Rel	All	Record locked by another user; record is not available.
30	All	All	All other permanent errors.

Additional References

- Chapter 4
- Section 6.1.4: Scope of Statements
- Section 6.6.8: I-O Status
- Section 6.6.10: INVALID KEY Phrase
- Section 6.6.11: FROM Phrase
- OPEN statement
- READ statement
- UNLOCK statement
- USE statement

SEARCH

SEARCH — 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 Formats

Format 1

```
SEARCH src-table [ VARYING pointr ] [ AT END stment ]
    { { WHEN cond stment } ... END-SEARCH
      { WHEN cond { stment ... [END-SEARCH] } } ...
                  { NEXT SENTENCE           }
    }
```

Format 2

```
SEARCH ALL src-table [ AT END stment ]

    WHEN { elemnt { IS EQUAL TO } arg }
         { cond-name { IS =      }     }

    [ AND { elemnt { IS EQUAL TO } arg } ] ...
          { cond-name { IS =      }     }

      { stment ... [ END-SEARCH ] }
      { NEXT SENTENCE             }
```

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.

Syntax Rules

Both Formats

1. *src_table* must not be subscripted, indexed, or reference modified. However, its description must contain an OCCURS clause with the INDEXED BY phrase.
2. If the END-SEARCH phrase is specified, the NEXT SENTENCE phrase must not be specified.

Format 2

3. *src_table* must contain the KEY IS phrase in its OCCURS clause.

4. 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*.

5. 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*

6. 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*

7. 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 single SEARCH statement can search only one dimension of a table; therefore, you must execute SEARCH statements repeatedly to search through 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. Example 3, "Serial search with two WHEN phrases," illustrates 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.

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 in the source table.)

   ```
   INITIALIZE-SEARCH.
       MOVE "NH" TO CUSTOMER-USPS-STATE.

       SEARCH ALL STATES
          AT END
                 MOVE 1 TO STATE-ERROR
                 GO TO SEARCH-END
          WHEN STATE-USPS-CODE (STATE-INDEX) = CUSTOMER-USPS-STATE
                 MOVE 0 TO STATE-ERROR
                 MOVE STATE-REGION (STATE-INDEX) TO CUSTOMER-REGION.

   SEARCH-END.
       DISPLAY " ".
       DISPLAY "Customer State index number = " STATE-INDEX WITH CONVERSION
        "   Region = " STATE-REGION (STATE-INDEX)
        "   State Error Code = " STATE-ERROR.
   ```

   Following are the results of the binary search:

   ```
   Customer State index number = 31   Region = 1   State Error Code = 0
   ```

2. Serial search with WHEN phrase:

   ```
   INITIALIZE-SEARCH.
       MOVE "2" TO CUSTOMER-REGION.
   SEARCH-LOOP.
       SEARCH STATES
   ```

```
        AT END
               MOVE 1 TO STATE-ERROR
               GO TO SEARCH-END
        WHEN STATE-REGION (STATE-INDEX) = CUSTOMER-REGION
               MOVE 0 TO STATE-ERROR
               DISPLAY STATE-USPS-CODE (STATE-INDEX)
                   " " STATE-INDEX WITH CONVERSION
                   " " STATE-ERROR.
    SET STATE-INDEX UP BY 1.
    GO TO SEARCH-LOOP.

SEARCH-END.
```

The following lists the results of this serial search:

```
 IA 13 0 IL 15 0 IN 16 0 KS 17 0 MI 23 0 MN 24 0 NE 30 0 OH 36 0 WI 49 0
```

3. Serial search with two WHEN phrases:

```
IA 13 0
IL 15 0
IN 16 0
KS 17 0
MI 23 0
MN 24 0
NE 30 0
OH 36 0
WI 49 0
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-USPS-CODE (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.
```

The following lists the results of the serial search with two WHEN phrases:

```
States in customer's region:
CT DC DE MA MD ME NJ NY PA RI VT

Customer state index number = 31
```

4. 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.
```

The following lists the results of updating a table in a SEARCH statement:

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

Additional References

- On Alpha and I64, SORT statement (Format 2, for table sorting), useful for SEARCH ALL (which presumes a sorted table)
- OCCURS statement
- Section 6.1.4: Scope of Statements

- Section 6.5: Conditional Expressions

SET

SET — The SET statement sets values of indexes associated with table elements. It can also change the value of a conditional variable, change the status of an external switch, and store the address of a COBOL identifier reference at run time.

General Format

Format 1

SET {rsult} … TO val

Format 2

SET {indx} … {UP BY|DOWN BY} increm

Format 3

SET {cond-name} … TO TRUE

Format 4

SET {{switch-name}… TO {ON|OFF}} …

Format 5

SET {pointer-id} … TO REFERENCE OF identifier

Format 6

SET status-code-id TO {SUCCESS|FAILURE}

`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 can 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 can be signed. It can also be the identifier of an elementary numeric data item described as an integer.

`cond-name`

is a condition-name that must be associated with a conditional variable.

`switch-name`

is the name of an external switch defined in the SPECIAL-NAMES paragraph.

`pointer-id`

is a data-name whose data description entry must contain the USAGE IS POINTER clause.

`identifier`

is a data item in the File, Working-Storage, Linkage, or Subschema Section.

`status-code-id`

is a word or longword integer data item represented by PIC S9(1) to S9(9) COMP or PIC 9(1) to 9(9) COMP.

Syntax Rule

No two occurrences of *cond-name* can refer to the same conditional variable.

General Rules

Formats 1 and 2

1. Index-names are associated with a table in the table's OCCURS clause INDEXED BY phrase.
2. 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.
3. 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*.
4. 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

5. 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.
6. If *rsult* is an index data item, *val* cannot be an integer. No conversion occurs when *rsult* is set to the value of *val*.
7. If *rsult* is not an index data item or an index-name, *val* can only be an index-name.
8. 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.
9. Table 6.18 shows the validity of operand combinations. An asterisk (*) means that no conversion occurs during the SET operation.

Table 6.18. 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

Sending Item	Receiving Item		
	Integer Data Item	**Index**	**Index Data Item**
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

10. The SET statement increments (UP) or decrements (DOWN) *indx* by a value that corresponds to the number of occurrences *increm* represents.

11. When there is more than one *indx*, SET uses the original value of *increm* in each operation.

Format 3

12. SET moves the literal in the VALUE clause for *cond-name* to its associated conditional variable. The transfer occurs according to the rules for elementary moves. If the VALUE clause contains more than one literal, the first is moved.

Format 4

13. SET changes the status of each *switch-name* in the statement.

14. The ON phrase changes the status of *switch-name* to on.

15. The OFF phrase changes the status of *switch-name* to off.

16. The SET statement changes the switch status only for the image in which it executes. When the image terminates, the status of each external switch is the same as when the image began.

Format 5

17. The address of *identifier* is evaluated and stored in *pointer-id*.

Format 6

18. Specifying the SUCCESS option sets *status-code-id* to the SUCCESS state (the low-bit of *status-code-id* is set to 1).

19. Specifying the FAILURE option sets *status-code-id* to the FAILURE state (the low-bit of *status-code-id* is set to 0).

Examples

The examples assume these Environment and Data Division entries:

```
 SPECIAL-NAMES.
    SWITCH 1 UPDATE-RUN ON STATUS IS DO-UPDATE
    SWITCH 3 REPORT-RUN ON STATUS IS DO-REPORT
        OFF STATUS IS SKIP-REPORT
    SWITCH 4 IS NEW-YEAR ON STATUS IS BEGIN-YEAR
        OFF IS CONTINUE-YEAR.
DATA DIVISION.
WORKING-STORAGE SECTION.
```

```
01    YEAR-LEVEL            PIC 99.
    88    FRESHMAN VALUE 1.
    88    SOPHOMORE VALUE 2.
    88    JUNIOR VALUE 3.
    88    SENIOR VALUE 4.
    88    FIRST-MASTERS VALUE 5.
    88    MASTERS VALUE 5,6.
    88    FIRST-DOCTORAL VALUE 7.
    88    DOCTORAL VALUE 7,8.
    88    NON-DEGREE-UNDERGRAD VALUE 9.
    88    NON-DEGREE-GRAD VALUE 10.
    88    UNDERGRAD VALUE 9, 1 THROUGH 4.
    88    GRAD VALUE 10, 5 THROUGH 8.
01    COURSES-AVAILABLE.
    02    OCCURS 100 TIMES INDEXED BY COURSE-INDEX.
        03    COURSE-NAME             PIC X(10).
        03    COURSE-INSTRUCTOR       PIC X(20).
        03    COURSE-LOCATION         PIC X(10).
        03    COURSE-CODE             PIC 9(5).
01    POINTER-VAL USAGE IS POINTER.
01    THREE-DIMENSIONAL-TABLE.
    02    X OCCURS 5 TIMES INDEXED BY I.
        03    Y OCCURS 7 TIMES INDEXED BY J.
            04    Z     PIC X(17) OCCURS 3 TIMES.
01    K                 PIC S9(9) COMP.
01    RETURN-STATUS     PIC S9(9) COMP.
01    DECREMENT-VALUE   PIC 9 VALUE 1.
```

1. Format 1—Initializing COURSE-INDEX.

   ```
   SET COURSE-INDEX TO 5.
   ```

2. Format 2—Adding to or subtracting from the index-name COURSE-INDEX.

   ```
   SET COURSE-INDEX UP BY 1.

   SET COURSE-INDEX DOWN BY DECREMENT-VALUE.
   ```

3. Format 3—Initializing a conditional variable:

   YEAR-LEVEL

   ```
   SET SOPHOMORE TO TRUE                                02
   SET MASTERS TO TRUE                                  05
   SET GRAD TO TRUE                                     10
   SET NON-DEGREE-GRAD TO TRUE                          10
   ```

4. Format 4—Setting external switches. The truth value shows the result of the IF statements:

   TRUTH VALUE

   ```
   SET UPDATE-RUN TO ON.
   SET REPORT-RUN TO OFF.
   SET NEW-YEAR TO ON.
   IF DO-UPDATE ...                                     true
   IF DO-REPORT ...                                     false
   IF CONTINUE-YEAR...                                  false
   SET REPORT-RUN TO ON.
   IF DO-REPORT ...                                     true
   ```

```
IF SKIP-REPORT ...                                  false
```

5. Format 5—Setting POINTER-VAR to the address of the subscripted table item named Z(I,J,K).

```
SET POINTER-VAR TO REFERENCE OF Z(I,J,K).
```

6. Format 6—On OpenVMS Alpha and I64, initializing RETURN-STATUS to FAILURE before calling subprogram SUBPROGA and a Run-Time Library Procedure, then checking for SUCCESS from each.

```
     .
     .
     .
    SET RETURN-STATUS TO FAILURE.
    CALL "SUBPROGA" GIVING RETURN-STATUS.
    IF RETURN-STATUS IS SUCCESS
        THEN
            GO TO A0200-PARA
        ELSE
            DISPLAY "SUBPROGA failed"
            STOP RUN.
A0200-PARA.
    SET RETURN-STATUS TO FAILURE.
    CALL "SCR$SET_CURSOR" USING BY VALUE 4, 22 GIVING RETURN-STATUS.
    IF RETURN-STATUS IS SUCCESS
        THEN
            DISPLAY "UPDATE ROUTINE COMPLETED"
        ELSE
            DISPLAY "Cursor positioning failed"
            STOP RUN.
     .
     .
     .

IDENTIFICATION DIVISION.
PROGRAM-ID. SUBPROGA.
     .
     .
     .
01    PROGRAM-STATUS    PIC S9(9) COMP.
     .
     .
     .
PROCEDURE DIVISION GIVING PROGRAM-STATUS.
A000-BEGIN.
     .
     .
     .

    IF ... SET PROGRAM-STATUS TO SUCCESS
      ELSE SET PROGRAM-STATUS TO FAILURE.
    EXIT PROGRAM.
```

Additional References

- Chapter 4
- Section 6.5.4: Switch-Status Condition

- Section 6.5.6: Success/Failure Condition
- MOVE statement
- PERFORM statement
- SEARCH statement

SORT

SORT — The SORT statement (Format 1) 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. SORT (Format 2) orders the elements in a table. This is especially useful for tables used with SEARCH ALL. The table elements are sorted based on the keys as specified in the OCCURS for the table unless you override them by specifying keys in the SORT statement. If no key is specified, the table elements are the SORT keys.

General Format

Format 1

```
SORT sortfile { ON { DESCENDING } KEY { sortkey } . . . } . . .
                   { ASCENDING }
    [ WITH DUPLICATES IN ORDER ]
    [ COLLATING SEQUENCE IS alpha ]
    { INPUT PROCEDURE IS first-proc [ { THRU } end-proc ] }
    { USING { infile } . . .        [ { THROUGH } ]       }
    { OUTPUT PROCEDURE IS first-proc [ { THRU } end-proc ] }
    { GIVING { outfile } . . .       [ { THROUGH } ]       }
```

Format 2 (Alpha, I64)

```
SORT table-name [ ON { DESCENDING } KEY { sortkey } . . . ] . . .
                     { ASCENDING }
    [ WITH DUPLICATES IN ORDER ]
    [ COLLATING SEQUENCE IS alpha ] .
```

`sortfile`

is a file-name described in a sort-merge file description (SD) entry in the Data Division.

`sortkey`

(Format 1) is the data-name of a data item in a record associated with *sortfile*.

(Format 2) is the data-name of a data item in the *table-name* table.

`first-proc`

is the section-name or paragraph-name of the first (or only) section or paragraph of the INPUT or OUTPUT procedure range.

`end-proc`

is the section-name or paragraph-name of the last section or paragraph 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.

`outfile`

is the file-name of the output file. It must be described in a file description (FD) entry in the Data Division.

`table-name (Alpha, I64)`

is a table described with OCCURS in the Data Division.

`alpha`

is an alphabet-name defined in the SPECIAL-NAMES paragraph of the Environment Division.

Syntax Rules

All Formats

1. You can use SORT statements anywhere in the Procedure Division except in:
   - Declaratives (Format 1)
   - SORT or MERGE statement input or output procedures
2. *sortkey* can be qualified.
3. *sortkey* cannot be in a group item that contains variable occurrence data items.
4. The *sortkey* description cannot contain an OCCURS clause or be subordinate to a data description entry that does.

Format 1

5. If *sortfile* contains variable-length records, *infile* records must not be smaller than the smallest in *sortfile* nor larger than the largest.
6. If *sortfile* contains fixed-length records, *infile* records must not be larger than the largest record described for *sortfile*.

7. If *outfile* contains variable-length records, *sortfile* records must not be smaller than the smallest in *outfile* nor larger than the largest.

8. If *outfile* contains fixed-length records, *sortfile* records must not be larger than the largest record described for *outfile*.

9. *sortfile* can have more than one record description. However, *sortkey* needs to 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*.

Format 2 (Alpha, I64)

12. *table-name* may be qualified and must have an OCCURS clause in its data description entry. If *table-name* is subject to more than one level of OCCURS clauses, subscripts must be specified for all levels with OCCURS INDEXED BY.

13. *table-name* is a key data-name, subject to the following rules:

    - The data item identified by a key data-name must be the same as, or subordinate to, the data item referenced by *table-name*.

    - Key data items may be qualified.

    - The data items identified by key data-names must not be variable-length data items.

    - If the data item identified by a key data-name is subordinate to *table-name*, it must not be described with an OCCURS clause, and it must not be subordinate to an entry that is also subordinate to *table-name* and contains an OCCURS clause.

14. The KEY phrase may be omitted only if the description of the table referenced by *table-name* contains a KEY phrase.

General Rules

All Formats

1. 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.

2. The ASCENDING phrase causes the sorted sequence to be from the lowest to highest *sortkey* value.

3. The DESCENDING phrase causes the sorted sequence to be from the highest to the lowest *sortkey* value.

4. Sort sequence follows the rules for relation condition comparisons.

5. The DUPLICATES phrase affects the return order of records or table elements 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.
- When table elements are returned, the order is the relative order of the contents of these table elements before sorting.

6. If there is no DUPLICATES phrase, the return order for records or table elements with equal corresponding *sortkey* values is unpredictable.
7. 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.

Format 1

8. 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*.
9. The INPUT PROCEDURE range consists of one or more sections or paragraphs 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 INPUT PROCEDURE range can consist of any procedure needed to select, modify, or copy the next record made available by the RELEASE statement to the file referenced by *sortfile*.
12. The range of the INPUT PROCEDURE additionally includes all statements executed as a result of a CALL, EXIT, GO TO, or PERFORM statement. The range of the INPUT PROCEDURE also includes all statements in the Declaratives Section that can be executed if control is transferred from statements in the range of the INPUT PROCEDURE.
13. The INPUT PROCEDURE range must not contain MERGE, RETURN, or SORT statements.
14. If there is an INPUT PROCEDURE phrase, control transfers 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 exception condition occurs.

18. OUTPUT PROCEDURE consists of one or more sections or paragraphs that:
    - Appear contiguously in the source program
    - Do not form part of an INPUT PROCEDURE range
19. When the SORT statement begins the OUTPUT PROCEDURE phrase, 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. 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.
21. The OUTPUT PROCEDURE can consist of any procedure needed to select, modify, or copy the next record made available by the RETURN statement in sorted order from the file referenced by *sortfile*.
22. The range of the OUTPUT PROCEDURE additionally includes all statements executed as a result of a CALL, EXIT, GO TO, or PERFORM statement. The range of the OUTPUT PROCEDURE also includes all statements in the Declarative USE procedures that can be executed if control is transferred from statements in the range of the OUTPUT PROCEDURE.
23. The OUTPUT PROCEDURE range must not include MERGE, RELEASE, or SORT statements.
24. 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.
25. If there is a GIVING phrase, the SORT statement writes all sorted records to each *outfile*. This transfer is an implied SORT output procedure. When the SORT statement executes, *outfile* must not be open.
26. 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.
27. The SORT statement obtains 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.

28. The SORT statement terminates *outfile* processing as if the program had executed a CLOSE statement with no optional phrases.

29. These implicit OPEN, WRITE, and CLOSE operations can cause associated USE procedures to execute if they are present. If a USE procedure is present, processing terminates after the USE procedure has completed execution. If a USE procedure is not present, processing terminates as if the program had executed a CLOSE statement with no optional phrases.

30. 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*.

31. If the SORT statement is in a *fixed* segment, its input and output procedures must be completely in either:

    - Fixed segments
    - One independent segment

32. If the SORT statement is in an *independent* segment, its input and output procedures must be completely in either:

    - Fixed segments
    - The same independent segment as the SORT statement itself

Format 2 (Alpha, I64)

33. The SORT statement sorts the table referenced by *table-name* and presents the sorted table in *table-name* either in the order determined by the ASCENDING or DESCENDING phrases, if specified, or in the order determined by the KEY phrase associated with *table_name*.

34. To determine the relative order in which the table elements are stored after sorting, the contents of corresponding key data items are compared according to the rules for comparison of operands in a relation condition, starting with the most significant key data item.

    - If the contents of the corresponding key data items are not equal and the key is associated with the ASCENDING phrase, the table element containing the key data item with the lower value has the lower occurrence number.
    - If the contents of the corresponding key data items are not equal and the key is associated with the DESCENDING phrase, the table element containing the key data item with the higher value has the lower occurrence number.

Examples (Alpha, I64)

The following examples all illustrate the use of table sorting (Format 2). For examples on Format 1 sorting, refer to the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/].

The first example is a simple sort in which the table is sorted by the key definitions in the OCCURS clause of data item *tabl*. *elem-item2* is the major key (ascending) and *elem-item1* is the secondary key (descending). A SEARCH ALL statement is used.

```
identification division.
program-id. EXAMPLE1.
```

```
data division.
working-storage section.
01 group-item.
   05 tabl occurs 10 times
 ascending elem-item2
 descending elem-item1
 indexed by ind.
      10 elem-item1 pic x.
      10 elem-item2 pic x.
procedure division.
1. display "Example 1".
 move "13n3m3p3o3x1x1x1x1x1" to group-item.
 sort tabl.
 search all tabl
     at end
  display "not found"
     when elem-item1 (ind) = "m"
  if (elem-item1 (ind - 1) = "n")
  and (elem-item1 (ind + 1) = "1")
      display "elem-item1 is descending order - 2nd key"
  else
      display "sort failed"
  end-if
     end-search.
 exit program.
end program EXAMPLE1.
```

The following example is also a simple sort in which the table is sorted by the key definitions in the OCCURS clause of data item *tabl*. *elem-item2* is the major key (ascending) and *elem-item1* is the secondary key (descending). A SEARCH ALL statement is used.

```
identification division.
program-id. EXAMPLE2.
data division.
working-storage section.
01 group-item.
   05 tabl occurs 10 times.
      10 elem-item1 pic x.
      10 elem-item2 pic x.
procedure division.
2. display "Example 2".
 move "13n3m3p3o3x1x1x1x1x1" to group-item.
 sort tabl ascending.
 if tabl (1) = "13"
 and tabl (2) = "m3"
     display "tabl is ascending order"
 else
     display "sort failed"
 end-if.
 exit program.
end program EXAMPLE2.
```

This following example is a simple sort in which the table is sorted in ascend- ing order using each entire element of the table (data item *tabl*) to determine the sequence.

```
identification division.
program-id. EXAMPLE3.
data division.
```

```
working-storage section.
01 group-item.
   05 tabl occurs 10 times
 ascending elem-item3
 descending elem-item1.
      10 elem-item1 pic x.
      10 elem-item2 pic x.
      10 elem-item3 pic x.
procedure division.
3. display "Example 3".
 move "13bn3cm3ap3do3fx1ex1ix1hx1gx1a" to group-item.
 sort tabl descending elem-item2 elem-item3.
 if tabl (1) = "o3f"
 and tabl (2) = "p3d"
     display "tabl is descending order"
 else
     display "sort failed"
 end-if.
 exit program.
end program EXAMPLE3.
```

The following example sorts only the third instance of *tabl2*, that is, *tabl1(3)*. The qualified data item, *elem-item1* of *group2* is its key. In normal PROCEDURE DIVISION reference, *elem-item1* of *group2* requires two levels of subscripting/indexing, whereas here it has none. Similarly, *tabl2* normally requires one level of subscripting, but cannot be subscripted as *data-name2* in the SORT statement. Instead it uses the value of *t1-ind* for determining which instance is sorted.

```
identification division.
program-id. EXAMPLE4.
data division.
working-storage section.
01 group-item.
   05 tabl1 occurs 3 times
 indexed by t1-ind t2-ind.
      10 tabl2 occurs 5 times.
         15 group1.
            20 elem-item1 pic x.
         15 group2.
            20 elem-item1 pic 9.
procedure division.
4. display "Example 4".
 move "x5z4y6z6x4a3b2b1a2c1j7j8k8l7j9" to group-item.
 set t1-ind to 3.
 sort tabl2 descending elem-item1 of group2.
 if group1 (3 1) = "j"
 and group2 (3 1) = "9"
 and tabl1 (1) = "x5z4y6z6x4"
 and tabl1 (2) = "a3b2b1a2c1"
     display "tabl1 (3) is descending order"
 else
     display "sort failed"
 end-if.
 exit program.
end program EXAMPLE4.
```

Additional References

- Chapter 4

- Section 6.5.1: Relation Conditions
- Section 6.7: Segmentation
- USE statement
- *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/], chapter on using SORT and MERGE statements

START

START — The START statement establishes the logical position of the File Position Indicator in an indexed or relative file. The logical position affects subsequent sequential record retrieval.

General Format

```
START file-name

[ KEY { IS EQUAL TO                                  } key-data ]
      { IS =                                         }
      { IS GREATER THAN                              }
      { IS >                                         }
      { IS NOT LESS THAN                             }
      { IS NOT <                                     }
      { IS GREATER THAN OR EQUAL TO                  }
      { IS >=                                        }
      { IS LESS THAN (Alpha, I64)                    }
      { IS < (Alpha, I64)                            }
      { IS LESS THAN OR EQUAL TO (Alpha, I64)        }
      { IS <= (Alpha, I64)                           }
      { IS NOT GREATER THAN (Alpha, I64)             }
      { IS NOT > (Alpha, I64)                        }

  [ REGARDLESS OF LOCK                               ]
  [ ALLOWING { UPDATERS } }                          ]
  [          { READERS  }                            ]
  [          { NO OTHERS }                           ]

[ INVALID KEY stment ]

[ NOT INVALID KEY stment2 ]

[ END-START ]
```

`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 one of the following:

- The data-name specified as a record key
- The segmented-key name specified as a record key
- The leftmost part of a record key
- The relative key for *file-name*

It can be qualified.

`stment`

is an imperative statement executed for an invalid key condition.

`stment2`

is an imperative statement executed for a not invalid key condition.

Syntax Rules

1. To use the REGARDLESS or ALLOWING options, the program must include these entries:
   - APPLY LOCK-HOLDING clause of the I-O-CONTROL paragraph
   - ALLOWING option of the OPEN statement
2. There must be an INVALID KEY phrase if *file-name* does not have an applicable USE AFTER EXCEPTION procedure.
3. For a relative file, *key-data* must be the file's RELATIVE KEY data item.
4. For an indexed file, *key-data* can be either:
   - A record key for the file.
   - A data item subordinate to the description of a record key for the file. The data item must have the same leftmost character position as the record key, and must be one of the following:
     - A group, alphanumeric, or alphabetic item
     - An unsigned numeric display item
     - A COMP-3 integer or a COMP integer

     All the data types in the preceding list except alphanumeric are VSI extensions.
5. The REGARDLESS and ALLOWING options are VSI standard syntax, and cannot be used for a file connector that has had (on Alpha and I64) X/Open standard syntax (WITH [NO] LOCK or LOCK MODE) specified.

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. 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.

5. START LESS can only be used with a file whose organization is INDEXED and whose access mode is DYNAMIC. The file must be opened for INPUT or I-O.

6. For indexed files, the file system compares the Key of Reference according to the native collating sequence and the sort order of the Key of Reference. The comparisons IS GREATER THAN, IS GREATER THAN OR EQUAL TO, and IS NOT LESS THAN refer to the logical record order, according to the sort order of the key. For example, if the sort order is descending, the KEY GREATER THAN *key-data* phrase positions the file at the next record whose key is less than *key-data*.

   All other numeric or nonnumeric comparison rules apply.

   The File Position Indicator 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 File Position Indicator denotes that no valid next record is established.

7. On Alpha and I64 systems, START LESS, LESS OR EQUAL, and NOT GREATER set the file position indicator by making reference to the logical record order in the same manner as START GREATER, GREATER OR EQUAL and NOT LESS.

8. The START verb can use the KEY IS syntax to establish the key field within the file record which is the Key of Reference. An immediately subsequent READ PRIOR will follow the order of the Key of Reference to access the logically previous record in the file according to that Key of Reference. If the KEY IS syntax is not used, the Key of Reference is understood to be the file's primary key field.

9. On Alpha and I64 systems, when a successful START LESS, LESS OR EQUAL or NOT GREATER has occurred and the Key of Reference has ascending order, the record pointed to by the file position indicator can have the same key value or a smaller key value than the preceding record for the Key of Reference. If the Key of Reference has descending order, the record pointed to can have the same key value or a higher key value for the Key of Reference. The record pointed to can have the same key value if duplicate values for the Key of Reference exist on the file.

10. On Alpha and I64 systems, when an unsuccessful START LESS, LESS OR EQUAL or NOT GREATER has occurred the key of reference is undefined and a File Status value of 23 is returned, which indicates the INVALID KEY condition, or record not found.

11. The START statement updates the FILE STATUS data item for the file.

12. If the File Position Indicator denotes that an optional file is not present when the START statement executes, the invalid key condition exists. START statement execution is then unsuccessful.

13. The REGARDLESS and ALLOWING options can be used only in a manual record-locking environment. To create a manual record-locking environment, an access stream must specify the APPLY LOCK-HOLDING clause of the I-O-CONTROL paragraph.

14. The REGARDLESS option allows an access stream to position to a record regardless of any record locks held by other concurrent access streams. The START REGARDLESS option holds no lock on the record positioned to.

    This statement generates a soft record lock condition if the record that is pointed to is locked by another access stream. This condition results in a File Status value of 90 and invokes an applicable USE procedure, if any. Execution of the START REGARDLESS statement is considered successful and execution resumes at the next statement following the START REGARDLESS statement.

    However, on UNIX systems, the soft lock condition (file status 90) is not recognized for indexed files. A START REGARDLESS statement for a record locked by another process performs the requested operation on the record and returns a file status of 00.

15. On OpenVMS, the ALLOWING UPDATERS option permits other concurrent access streams in the manual record-locking environment to simultaneously READ, DELETE, START, and REWRITE the current record. This option holds no lock on the current record.

16. The ALLOWING READERS option permits other concurrent access streams in the manual record-locking environment to simultaneously READ the current record. This option holds a read-lock on each such record read. No access stream can update the current record until it is unlocked.

17. On OpenVMS, the ALLOWING NO OTHERS option locks the current record. No other concurrent access stream can access this record until it is unlocked. Only this access stream can update this record.

18. On UNIX systems, for indexed files the START statement (with or without the ALLOWING phrase) does not detect or acquire a record lock on the current record.

19. On Alpha and I64 systems, if X/Open file sharing is in effect, the START statement does not detect or acquire a lock.

20. If VSI standard record locking is in effect and the ALLOWING or REGARDLESS option is not specified, the default behavior for a START statement is that a lock is acquired if the file is opened in I-O mode and locks are detected in any mode.

21. On Alpha and I64, if ALLOWING or REGARDLESS is not specified, there is potential for ambiguity regarding VSI standard record locking or X/Open standard record locking. The selection of X/Open standard (rule 19) or VSI standard (rule 20) behavior is made as follows by the compiler:

    - If (on Alpha and I64) X/Open standard syntax (LOCK MODE or WITH (NO) LOCK) has been specified for *file-name* prior to the START statement, the compiler interprets the statement according to the X/Open standard.

    - If VSI standard syntax (LOCK-HOLDING, ALLOWING, or REGARDLESS) has been specified for *file-name* prior to the START statement, the compiler interprets the statement according to the VSI standard.

    - If no file-sharing syntax (LOCK-HOLDING, ALLOWING, REGARDLESS, LOCK MODE, or WITH [NO] LOCK) has been specified for *file-name* prior to the START statement, then the compiler uses the /STANDARD=[NO]XOPEN qualifier on OpenVMS Alpha and I64 (or the UNIX equivalent `-std [no]xopen` flag) to determine whether the START statement

is interpreted as X/Open or VSI standard: a setting of `xopen` selects the X/Open standard, whereas a setting of `noxopen` selects the VSI standard.

Any subsequent I-O locking syntax for the same file connector in your program must be consistent: X/Open standard locking (Alpha, I64) and VSI standard locking (implicit or explicit) cannot be mixed for the same file connector.

Relative Files

22. 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

23. 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.

24. If there is a KEY phrase, the comparison described in General Rule 4 uses the contents of *key-data*.

25. 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.

26. If START statement execution is not successful, the Key of Reference is undefined.

27. 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 the first character of a FILE STATUS data item. However, it does not execute if the condition is invalid key and there is an INVALID KEY phrase.

    See the rules for the INVALID KEY phrase, Section 6.6.10.

Technical Notes

- START statement execution can result in these FILE STATUS data item values:

File Status	Meaning
00	Start is successful
23	Record not in file or optional file not present (invalid key)
47	File not open, or incompatible open mode
90	Record locked by another user; record available; soft lock

File Status	Meaning
92	Record locked by another user; record not available; hard lock
30	All other permanent errors

Additional References

- Chapter 4
- Section 6.1.4: Scope of Statements
- Section 6.5.1.1: Comparison of Numeric Operands
- Section 6.5.1.2: Comparison of Nonnumeric Operands
- Section 6.6.8: I-O Status
- Section 6.6.10: INVALID KEY Phrase
- OPEN statement
- READ statement
- UNLOCK statement
- USE statement

STOP

STOP — The STOP statement permanently terminates 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 terminates 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 before control returns to the operating system prompt.

2. STOP *disp* returns control to the operating system command language interpreter level without terminating the image as follows:

   - On UNIX systems, STOP *disp* resumes execution when a carriage return is entered.
   - On Open VMS systems, control returns to DCL. The user can continue image execution with a CONTINUE command, which returns control to the program at the next executable statement.

Additional Reference (OpenVMS)

Refer to the *VSI OpenVMS User's Manual* and the *VSI OpenVMS DCL Dictionary* for more information on the Digital Command Language (DCL).

STRING

STRING — The STRING statement concatenates the partial or complete contents of one or more data items into a single data item.

General Format

```
STRING { { src-string } ... DELIMITED BY { delim } } ...
                                         { SIZE  }
       INTO dest-string [ WITH POINTER pointr ]
     [ ON OVERFLOW stment ]
     [ NOT ON OVERFLOW stment2 ]
     [ END-STRING ]
```

`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. It cannot be reference modified. *dest-string* is the receiving area that contains the result of the concatenated *src-string*s.

`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 executed for an on overflow condition.

`stment2`

is an imperative statement executed for a not on overflow condition.

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 characters to delimit the move.
2. If the size of *delim* is zero characters, it never matches a *src-string* delimiter.
3. If *src-string* is a variable-length item, SIZE refers to the number of characters currently defined for it.
4. When *src-string* or *delim* is a figurative constant, its size is one character.
5. 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.
6. When the DELIMITED phrase contains *delim*:
   - The contents of each *src-string* are moved to *dest-string* in the sequence in which 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:
     a. Reaches the end of *src-string*
     b. Reaches the end of *dest-string*
     c. Detects the characters specified by *delim*
7. No data movement occurs if the size of *src-string* is zero characters.
8. When the DELIMITED phrase contains the SIZE phrase:
   - The contents of each *src-string* are moved to *dest-string* in the same sequence in which 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.

9. When the POINTER phrase appears, the program must set *pointr* to an initial value greater than zero before executing the STRING statement.

10. When there is no POINTER phrase, the STRING statement operates as if *pointr* were set to an initial value of 1.

11. 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.

12. 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.

13. Before it moves each character to *dest-string*, the STRING statement tests the value of *pointr*.

    If *pointr* is less than 1 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

    If *pointr* is not less than 1 or not greater than the number of character positions in *dest-string* after the data is transferred, the STRING statement:

    - Executes the NOT ON OVERFLOW phrase *stment2* and then transfers control to the end of the STRING statement
    - Transfers control to the end of the STRING statement if the NOT ON OVERFLOW phrase is not specified

14. Subscript evaluation for *dest-string* and *pointr* occurs at the beginning of the 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  CITY              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.
```

- 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.
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
```

- 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,
```

Additional References

- Section 6.1.4: Scope of Statements
- MOVE statement

SUBTRACT

SUBTRACT — The SUBTRACT statement subtracts one, or the sum of two or more, numeric items from one or more items. It stores the difference in one or more items.

General Format

Format 1

```
SUBTRACT { num } . . . FROM { rsult [ ROUNDED ] } . . .

    [ ON SIZE ERROR stment ]

    [ NOT ON SIZE ERROR stment2 ]

    [ END-SUBTRACT ]
```

Format 2

```
SUBTRACT { num } . . . FROM num GIVING { rsult [ ROUNDED ] } . . .

    [ ON SIZE ERROR stment ]

    [ NOT ON SIZE ERROR stment2 ]

    [ END-SUBTRACT ]
```

Format 3

```
SUBTRACT { CORRESPONDING } grp-1 FROM grp-2 [ ROUNDED ]
         { CORR        }

    [ ON SIZE ERROR stment ]

    [ NOT ON SIZE ERROR stment2 ]

    [ END-SUBTRACT ]
```

`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.

`stment`

is an imperative statement executed when a size error condition has occurred.

`stment2`

is an imperative statement executed when no size error condition has occurred.

`grp-1`

is the identifier of a group item.

`grp-2`

is the identifier of a group item.

Syntax Rule

CORR is an abbreviation for CORRESPONDING.

General Rules

1. In Format 1, the values of the operands before the word FROM are summed. This total is then subtracted from each *rsult*.

2. 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*.

3. In Format 3, data items in *grp-1* are subtracted from and stored in the corresponding data items in *grp-2*.

Examples

Each of the examples assume these data descriptions and initial values.

INITIAL VALUES

```
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
```

1. Without GIVING phrase: *RESULTS*

```
SUBTRACT 2 ITEMB FROM ITEMA.                  ITEMA = -89
```

2. SIZE ERROR clause:

   (When the size error condition occurs and the SIZE ERROR clause is specified, the values of the affected resultant identifiers do not change.)

```
SUBTRACT 14 FROM ITEMA, ITEME                 ITEMA = -99
  ON SIZE ERROR                               ITEME = -95
    MOVE 0 TO ITEMB.                          ITEMB = 0
```

3. NOT ON SIZE ERROR clause:

```
SUBTRACT 14 FROM ITEMA                        ITEMA = -99
  ON SIZE ERROR
    MOVE 9 TO ITEMB.
  NOT ON SIZE ERROR
    MOVE 1 TO ITEMB.                          ITEMB =  1
```

4. 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
```

5. GIVING phrase:

```
SUBTRACT ITEME ITEMD (ITEMB) FROM ITEMA        ITEMB = 8
     GIVING ITEMB.
```

6. END-SUBTRACT:

   (The first SUBTRACT terminates with END-SUBTRACT. If the SIZE ERROR condition had not occurred, the second SUBTRACT statement would have executed anyway: the value of ITEMA would have been -86.)

```
SUBTRACT 10 ITEMB FROM ITEMD (ITEMB)         ITEMD (2) = 2
  ON SIZE ERROR                              ITEMA = 0
      MOVE 0 TO ITEMA
  END-SUBTRACT.
SUBTRACT 1 FROM ITEMA.                     ITEMA = -1
```

   (The following example shows the usefulness of END-SUBTRACT inside an IF statement. Without it, there would be no way to code the DISPLAY statements.)

```
IF ITEMB < 3 AND > 1
   SUBTRACT 1 FROM ITEMD(ITEMB)
      ON SIZE ERROR
         MOVE 0 TO ITEMA
   END-SUBTRACT
   DISPLAY 'yes'
ELSE
   DISPLAY 'no'.
```

Additional References

- Section 6.1.4: Scope of Statements
- Section 6.6.1: Arithmetic Operations
- Section 6.6.2: Multiple Receiving Fields in Arithmetic Statements
- Section 6.6.3: ROUNDED Phrase
- Section 6.6.4: ON SIZE ERROR Phrase
- Section 6.6.5: CORRESPONDING Phrase
- Section 6.6.7: Overlapping Operands and Incompatible Data

SUPPRESS

SUPPRESS — The SUPPRESS statement causes the Report Writer Control System (RWCS) to inhibit the presentation of a report group.

General Format

SUPPRESS PRINTING

Syntax Rule

The SUPPRESS statement can appear only in a USE BEFORE REPORTING Declarative procedure.

General Rules

1. The SUPPRESS statement inhibits only the presentation of a report-group-name (a 01-level Report Group Description entry).
2. Each time the presentation of a report group is to be inhibited, the program must execute a SUPPRESS statement.
3. The SUPPRESS statement directs the Report Writer Control System (RWCS) to inhibit the processing of these report group functions:
   - The presentation of the print lines
   - The processing of all LINE clauses
   - The processing of the NEXT GROUP clause
   - The adjustment of LINE-COUNTER
4. The SUPPRESS statement does not inhibit the processing of sum counters or control breaks.

Example

```
PROCEDURE DIVISION.
DECLARATIVES.
DET SECTION.
    USE BEFORE REPORTING DETAIL-LINE.
DETA-1.
    IF SORTED-NAME = NAME
        ADD A TO B
        SUPPRESS PRINTING.
    IF NAME = SPACES SUPPRESS PRINTING.
END DECLARATIVES.
MAIN SECTION.
    .
    .
    .
```

Additional References

- Section 6.6.7: Overlapping Operands and Incompatible Data
- USE statement

TERMINATE

TERMINATE — The TERMINATE statement causes the Report Writer Control System (RWCS) to complete the processing of the specified report.

General Format

TERMINATE {report-name} …

`report-name`

names a report defined by a Report Description entry in the Report Section of the Data Division.

General Rules

1. If the TERMINATE statement includes more than one *report-name*, the statement executes as if there were a separate TERMINATE statement for each *report-name*.
2. The program cannot execute a TERMINATE statement unless an INITIATE statement was executed before the TERMINATE statement for that report, and the program did not already execute a TERMINATE statement for that report.
3. If the program did not execute a GENERATE statement, the execution of a TERMINATE statement does not cause the RWCS to produce any of its report groups or perform any of the related processing.
4. The TERMINATE statement causes the RWCS to:
   - Produce all CONTROL FOOTING report groups beginning with the minor CONTROL FOOTING report group.
   - Produce the REPORT FOOTING report group.

   The RWCS makes the prior set of control data item values available to these two report groups and to any associated USE procedure. This action simulates a control break at the most major level.
5. The RWCS automatically processes the PAGE HEADING and PAGE FOOTING report groups, if present, when it must advance the report to a new page to present a CONTROL HEADING, DETAIL, or CONTROL FOOTING report group.
6. The TERMINATE statement does not automatically close a report file; the program must close the file. The program must terminate the report before the CLOSE statement can close the report file.

Additional Reference

USE statement.

UNLOCK

UNLOCK — The UNLOCK statement removes a record lock from the current record or from all locked records in the file. On Alpha and I64 systems, the X/Open standard UNLOCK statement always removes the record lock from all locked records in the file.

General Format

Format 1—Hewlett-Packard Standard

`UNLOCK file-name` [RECORD | ALL RECORDS]

Format 2—X/Open Standard (Alpha, I64)

`UNLOCK file-name` [RECORD | RECORDS]

`file-name`

is the name of a sequential, relative, or indexed file described in the Data Division.

Syntax Rules

1. For Format 1, if the UNLOCK statement does not include the RECORD or the ALL RECORDS option, the singular RECORD option is the default. (However, see General Rule 3.)

2. For Format 2, the RECORD and RECORDS options have the same effect: to unlock all currently locked records. This behavior also is the default if neither option is specified.

General Rules

1. The first access stream to lock a record owns the record lock for that record.

2. Only the owner of a record lock can unlock the record.

3. For Format 1, implicitly (by default) or explicitly specifying the RECORD option unlocks the current record. Therefore, you must specify ALL RECORDS explicitly to unlock all the record locks held on *file-name*.

   The single exception to this rule for Format 1 is that for indexed files the RECORD option (implicitly or explicitly) is unsupported on UNIX systems. The ALL RECORDS phrase is assumed.

4. For Format 2, whether you specify the RECORD option or the RECORDS option, the effect is the same: to unlock all record locks held on *file-name* by the current access stream.

5. If an access stream attempts to unlock a record (or records) in a file containing no record locks, the statement is considered successful and execution resumes at the statement following the UNLOCK statement.

6. Because both formats of the UNLOCK statement include the `UNLOCK RECORD` and `UNLOCK` forms, the compiler determines whether to interpret these forms of the statement as X/Open standard (on Alpha and I64) or VSI standard as follows:

   - If on Alpha and I64 X/Open standard syntax (LOCK MODE or WITH (NO) LOCK) has been specified for *file-name* prior to the UNLOCK statement, the compiler interprets the statement according to the X/Open standard.

   - If VSI standard syntax (LOCK-HOLDING, ALLOWING, or REGARDLESS) has been specified for *file-name* prior to the UNLOCK statement, the compiler interprets the statement according to the VSI standard.

   - If no file-sharing syntax (LOCK-HOLDING, ALLOWING, REGARDLESS, LOCK MODE, or WITH [NO] LOCK) has been specified for *file-name* prior to the UNLOCK statement, then the compiler uses the /STANDARD=[NO]XOPEN qualifier on OpenVMS Alpha and I64 (or the UNIX equivalent `-std [no]xopen` flag) to determine whether the START statement is interpreted as X/Open or VSI standard: a setting of `xopen` selects the X/Open standard, whereas a setting of `noxopen` selects the VSI standard.

   Any subsequent I-O locking syntax for the same file connector in your program must be consistent: X/Open standard locking and VSI standard locking (implicit or explicit) cannot be mixed for the same file connector.

Technical Notes

- UNLOCK statement execution can result in these FILE STATUS data item values:

File Status	File Organization	Access Method	Meaning
00	All	All	Unlock is successful
93	All	All	No current record
94	All	All	File not open, or incompatible open mode
30	All	All	All other permanent errors

VSI Standard Examples

These examples assume only one access stream for the image. The following examples refer to this partial program:

```
CONFIGURATION SECTION.
FILE-CONTROL.
    SELECT MASTER-FILE ASSIGN TO "CLIENT.DAT"
    ORGANIZATION IS INDEXED
    ACCESS MODE IS DYNAMIC
    RECORD KEY IS MASTER-KEY
    FILE STATUS IS FILE-STAT.
I-O-CONTROL.

*
* This APPLY clause is required syntax for manual record locking
*

    APPLY LOCK-HOLDING ON MASTER-FILE.

DATA DIVISION.
FD  MASTER-FILE
    LABEL RECORDS STANDARD.
01  MASTER-RECORD.
      .
      .
      .
PROCEDURE DIVISION.
A100-BEGIN.

*
*  The ALLOWING phrase enables file sharing
*

    OPEN I-O MASTER-FILE ALLOWING ALL.
      .
      .
      .
A900-END-OF-JOB.
```

1. Unlocking the record lock on the current record by taking the default RECORD option:

```
        READ MASTER-FILE KEY IS MASTER-KEY
             ALLOWING NO OTHERS.
```

```
        REWRITE MASTER-RECORD ALLOWING NO OTHERS.
        UNLOCK MASTER-FILE.
```

2. Explicitly unlocking the record lock on the current record:

```
        READ MASTER-FILE KEY IS MASTER-KEY
             ALLOWING NO OTHERS.
        .
        .
        .
        UNLOCK MASTER-FILE RECORD.
```

3. Unlocking all records in MASTER-FILE:

```
        PERFORM A100-READ-MASTER UNTIL
                  MASTER-KEY = ID-KEY
               OR
                  MASTER-KEY > ID-KEY.
        .
        .
        .
        UNLOCK MASTER-FILE ALL RECORDS.
        .
        .
        .
    A100-READ-MASTER.
        READ MASTER-FILE ALLOWING NO OTHERS.
```

X/Open Standard Example (Alpha, I64)

The following example shows the use of X/Open standard syntax:

```
  SELECT employee-file ASSIGN TO "EMPFIL"
         ORGANIZATION IS INDEXED
         ACCESS MODE IS DYNAMIC
         RECORD KEY IS employee-id
         LOCK MANUAL LOCK ON MULTIPLE RECORDS
         FILE STATUS IS emp-stat.
    .
    .
    .
* The file is implicitly shareable via the SELECT specification.
    OPEN I-O employee-file.

    PERFORM UNTIL emp-stat = end-of-file
        READ employee-file NEXT RECORD
             WITH LOCK

        IF employee-job-code = peon-code
           PERFORM find-boss-record
        ENDIF
    .
    .
    .
        REWRITE employee-record

*   This will unlock this record and the boss's
*   record found earlier.
```

```
        UNLOCK employee-file RECORDS

     END-PERFORM.

FIND-BOSS-RECORD.
    START employee-file
          KEY > employee-job-code.
    READ employee-file NEXT WITH LOCK.
```

Additional References

- Chapter 4
- Technical Notes for DELETE statement
- OPEN statement

UNSTRING

UNSTRING — The UNSTRING statement separates contiguous data in a sending field and stores it in one or more receiving fields.

General Format

```
UNSTRING src-string
    [ DELIMITED BY [ ALL ] delim [ OR [ ALL ] delim ] ... ]
      INTO { dest-string [ DELIMITER IN delim-dest ] [ COUNT IN countr ] } ...
    [ WITH POINTER pointr ]
    [ TALLYING IN tally-ctr ]
    [ ON OVERFLOW stment ]
    [ NOT ON OVERFLOW stment2 ]
    [ END-UNSTRING ]
```

`src-string`

is the identifier of an alphanumeric class data item. It cannot be reference modified. *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 executed for an on overflow condition.

`stment2`

is an imperative statement executed for a not on overflow condition.

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 their PICTURE clauses.

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 in *src-string* is a single delimiter. No character 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 to 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, if there is a NOT ON OVERFLOW phrase, this phrase is ignored and the UNSTRING operation ends. If there is an ON OVERFLOW phrase, *stment* executes. Otherwise, control passes to the end of the UNSTRING statement.
25. At the end of the UNSTRING operation, when an overflow condition does not exist, the ON OVERFLOW phrase is ignored and the UNSTRING operation ends if a NOT ON OVERFLOW phrase does not exist. If there is a NOT ON OVERFLOW phrase, *stment2* executes. After *stment2* executes, control is passed to the end of the UNSTRING statement.
26. If there is a DELIMITED BY phrase and the size of *dest-string* is zero characters, no characters are moved. However, *delim-dest* contains the matched delimiter and *countr* contains the character count.
27. If there is no DELIMITED BY phrase and the size of *dest-string* is zero characters, no characters are moved. The value of *pointr* does not change. UNSTRING continues with the next *dest-string*.
28. If the size of *delim* is zero characters, *delim* does not match any characters in *src-string*.

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.
```

- 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.
```

Results

```
Enter a date: 6/13/87
870613
Enter a date: 6-13-87
870613
Enter a date: 6-13 87
870613
Enter a date: 6/13/87/2
000000
Enter a date: 1-2-3
030102
```

- 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: 6/13/87 8/15/87
870613   09   03
870815   21   03
Enter two dates in a row: 10 15 87-1 1 88
871015   10   03
```

```
880101   21   03
Enter two dates in a row: 6/13/87-12/31/87
870613   09   03
871231   21   03
Enter two dates in a row: 6/13/87-12/31
870613   09   03
001231   21   02
Enter two dates in a row: 6/13/87/12/31/87
870613   09   03
871231   21   03
```

- 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: 6/13/87 8/15/87
870613   09   03 : 01-02-02
870815   21   03 : 01-02-02
Enter two dates in a row: 10 15 87-1 1 88
871015   10   03 : 02-02-02
880101   21   03 : 01-01-02
Enter two dates in a row: 6/13/87-12/31/87
870613   09   03 : 01-02-02
871231   21   03 : 02-02-02
Enter two dates in a row: 6/13/87-12/31
870613   09   03 : 01-02-02
001231   21   02 : 02-02-00
Enter two dates in a row: 6/13/87/12/31/87
870613   09   03 : 01-02-02
871231   21   03 : 02-02-02
```

Additional References

- Section 6.1.4: Scope of Statements
- MOVE statement

USE

USE — The USE statement specifies Declarative USE procedures to handle input/output exceptions and errors. It can also specify procedures to be executed before the program processes a specific report group.

General Format

Format 1

```
USE | GLOBAL | AFTER STANDARD { EXCEPTION } PROCEDURE
                               { ERROR   }

        { { file-name } ... }
        {   INPUT           }
ON      {   OUTPUT          }  .
        {   I-O             }
        {   EXTEND          }
```

Format 2

USE [GLOBAL] BEFORE REPORTING group-data-name

`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.

`group-data-name`

is the name of a report group in a report group description entry in a Data Division. It must not appear in more than one USE statement.

Syntax Rules

All Formats

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 USE procedures.

2. The USE statement itself does not execute. It defines the conditions that cause execution of the USE procedure.

Format 1

3. The ERROR and EXCEPTION syntax are equivalent and interchangeable.

Format 2

4. Of the four Report Writer Procedure Division verbs (SUPPRESS, GENERATE, INITIATE, or TERMINATE), only the SUPPRESS statement can appear in a USE BEFORE REPORTING

procedure. A PERFORM statement in a USE BEFORE REPORTING procedure must not have GENERATE, INITIATE, or TERMINATE statements in its range.

The USE procedure must not alter the value of any control data item.

General Rules

All Formats

1. At run time, two special precedence rules apply for the selection of a USE procedure when a program is contained within another program. In applying these rules, only the first qualifying USE procedure is selected for execution. The order of precedence for the selection of a USE procedure is as follows:

   - First, select the applicable USE procedure within the program containing the statement that caused the qualifying condition.

   - If a USE procedure is not found in the program using the previous rule, the Run-Time System searches all programs directly or indirectly containing that program for a USE GLOBAL procedure. This search continues until the Run-Time System either: (a) finds an applicable USE GLOBAL procedure, or (b) finds the outermost containing program, if there is no applicable USE GLOBAL procedure. Either condition terminates the search.

2. A Declarative USE procedure cannot refer to a non-Declarative procedure. However, only the PERFORM statement can transfer execution control from:

   - A Declarative USE procedure to another Declarative USE procedure

   - A non-Declarative procedure to a Declarative USE procedure

3. After a USE procedure 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.

4. A program must not execute a statement in a USE procedure that would cause execution of a USE procedure that had been previously executed and had not yet returned control to the routine that invoked it.

Format 1

5. A USE procedure executes automatically:

   - After the system's input-output exception processing completes

   - When an invalid key or at end condition results from an input-output statement that has no INVALID KEY or AT END clause

6. 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 the first character of 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.

7. One input-output exception cannot cause more than one USE AFTER EXCEPTION procedure to execute.

8. More than one USE AFTER EXCEPTION procedure can relate 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. This rule applies only to USE procedures in the same program.

9. If no applicable USE procedures are found in the local program, then containing programs are searched upwards for: (a) USE GLOBAL procedures for the file, and then (b) for USE GLOBAL procedures for the input-output type.

10. A USE AFTER EXCEPTION procedure specifying an open mode applies to an input-output operation only when all of the following are true:

    - The open mode (INPUT, OUTPUT, I-O, or EXTEND) specified in the USE AFTER EXCEPTION procedure is identical to the open mode in effect (that is, the open mode established by the OPEN statement).
    - The file is open or in the process of being opened.
    - There is no *file-name* declarative procedure for that file within the same program.

11. If an input-output error occurs for a file that is not open or not in the process of being opened, the only applicable USE procedure is a *file-name* USE procedure.

Format 2

12. The Report Writer Control System (RWCS) executes the USE BEFORE REPORTING procedure before it processes the named *group-data-name* report group. Only during the processing of the report group does the RWCS change prior values, execute control breaks, adjust LINE-COUNTER and PAGE-COUNTER, and present the report group.

Example

```
*************************************************************
*  This example assumes that SELECT and FD statements exist
*  for FILE1-SEQ, FILE1-RAN, FILE1-DYN and FILE1-EXT.
*  All three USE procedures are local to the program
*  that hosts this fragment.
*  At run-time if there is an exception on opening FILE1-RAN
*  or FILE1-DYN, FILE1-ERR section can be invoked.
*  If there is an exception on opening FILE1-SEQ, INPUT-ERR
*  section can be invoked. Since there is no USE procedure
*  declared for the EXTEND mode or for FILE1-EXT,
*  an exception on opening that file will cause an abnormal
*  termination of the program. Also, since FILE1-SEQ in the
*  fragment is not opened for OUTPUT mode, the OUTPUT-ERR USE
*  procedure is not eligible to be invoked here.
*********************************************************

PROCEDURE DIVISION.
DECLARATIVES.
INPUT-ERR SECTION.
 USE AFTER STANDARD ERROR PROCEDURE ON INPUT.
INP-1.
 DISPLAY "INVOKED USE PROCEDURE FOR INPUT".
OUTPUT-ERR SECTION.
 USE AFTER STANDARD ERROR PROCEDURE ON OUTPUT.
OUT-1.
 DISPLAY "INVOKED USE PROCEDURE FOR OUTPUT".
```

```
FILE1-ERR SECTION.
 USE AFTER STANDARD ERROR PROCEDURE ON FILE1-RAN, FILE1-DYN.
FILE1-1.
 DISPLAY "INVOKED USE PROCEDURE FOR FILES".

END DECLARATIVES.
MAIN-PROGRAM SECTION.
P0. DISPLAY "***ENTERED USE TEST PROGRAM FRAGMENT***".

 OPEN INPUT FILE1-SEQ.

 OPEN OUTPUT FILE1-RAN.

 OPEN I-O FILE1-DYN.

 OPEN EXTEND FILE1-EXT.

 ...
```

Additional References

- Section 6.2.6: Scope of Names
- Section 6.3.1: Explicit and Implicit Procedure Division References
- Description of exception handling in the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/].

WRITE

WRITE — 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 Formats

Format 1—Sequential or Line Sequential Files

```
WRITE rec-name [ FROM src-item ]
 [ ALLOWING NO OTHERS ]
    [ { BEFORE } ADVANCING { advance-num [ LINE ] }
      { AFTER  }           { [ LINES ]         }
                           { top-of-page-name  }
                           { PAGE              } ]
    [ AT { END-OF-PAGE } stment ]
         { EOP         }
    [ NOT AT { END-OF-PAGE } stment2 ]
             { EOP         }
 [ END-WRITE ]
```

Format 2—Relative or Indexed Files

```
WRITE rec-name [ FROM src-item ]
      [ ALLOWING NO OTHERS ]
     [ INVALID KEY stment ]
     [ NOT INVALID KEY stment2 ]
[ END-WRITE ]
```

`rec-name`

is the name of a logical record described in the Data Division File Section. 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.

`top-of-page-name`

is a mnemonic-name equated to C01 in the SPECIAL-NAMES paragraph of the Environment Division. It represents top-of-page and is equivalent to the PAGE phrase.

`stment`

is an imperative statement executed when the relevant condition (end-of-page or invalid key) occurs.

`stment2`

is an imperative statement executed when the relevant condition (not at end-of-page or not invalid key) occurs.

Syntax Rules

1. Format 1 must be used for sequential files.
2. Format 2 must be used for relative and indexed files.
3. If the file description entry containing *rec-name* has a LINAGE clause, the WRITE statement cannot have an ADVANCING *top-of-page-name* phrase.
4. If there is an END-OF-PAGE phrase, the file description entry containing *rec-name* must have a LINAGE clause.
5. The words END-OF-PAGE and EOP are equivalent.
6. In Format 2, there must be an INVALID KEY phrase if there is no applicable USE AFTER EXCEPTION procedure for the file.
7. To use the ALLOWING option, the program must include these entries:

- LOCK-HOLDING clause of the I-O-CONTROL paragraph
- ALLOWING option of the OPEN statement

8. If *src-item* is a function-identifier, it must reference an alphanumeric function. When *src-item* is not a function-identifier, *rec-name* and *src-item* must not reference the same storage area.
9. The ADVANCING PAGE and END-OF-PAGE phrase cannot be used in the same WRITE statement.
10. ADVANCING cannot be used with LINE SEQUENTIAL (Alpha, I64).
11. The ALLOWING clause is VSI standard file-sharing syntax, and cannot be used for a file connector that has had X/Open standard file-sharing syntax (WITH [NO] LOCK or LOCK MODE) specified.

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. The FROM Phrase section lists the rules for the FROM phrase.
3. For mass storage files, the WRITE statement does not affect the File Position Indicator.
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. (See Technical Notes.)
7. WRITE statement execution releases a logical record to the I-O system.
8. The ALLOWING NO OTHERS option can be used only in a VSI standard manual record-locking environment. To create a manual record-locking environment, the program must open file-name with an ALLOWING option and specify the APPLY LOCK-HOLDING phrase of the I-O-CONTROL paragraph. If you use manual locking (APPLY LOCK-HOLDING), then the ALLOWING NO OTHERS clause on the WRITE statement is required.
9. The ALLOWING NO OTHERS option locks the current record. No other concurrent access stream can access this record until it is unlocked.

   However, on UNIX systems, for indexed files the WRITE statement with the ALLOWING clause does not acquire a record lock.
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 the first character of 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.

    See the rules for the INVALID KEY phrase, Section 6.6.10.

11. The number of character positions in the record to be written must not be less than the lowest or greater than the highest number of character positions allowed by the RECORD VARYING clause. In either case, the WRITE statement is unsuccessful and the following occurs:

    - The WRITE operation does not take place.
    - The contents of the record area remain unaffected.
    - The I-O status of the file is set to a value that indicates the cause of the condition.

Sequential or Line Sequential (Alpha, I64) Files

12. The file must be open in the OUTPUT or EXTEND mode when the WRITE statement executes. (See Table 6.15.)

13. 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.

14. 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.

15. When a program tries to write beyond a sequential file's externally defined boundaries (for example,attempting to write to a full disk device), 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.

16. 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.

17. If the program reaches the 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.

18. 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 statement 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.

    If this statement does not occur and the NOT END-OF-PAGE is specified, *rec-name* is written to the file, file status is updated, and control is transferred to *stment2*.

19. 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.

20. If there is no LINAGE clause FOOTING phrase, the WRITE statement operates as if the FOOTING phrase value was beyond the limits of the page. That is, the end-of-page condition occurs after the specified number of lines per page are written. No space is reserved for a footing.

21. 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

22. The file must be open in the OUTPUT, I-O, or EXTEND mode when the WRITE statement executes. (See Table 6.15.)

23. When a relative file with sequential access mode is open in the output mode, the WRITE statement releases a record to the I-O system. 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.

24. When a relative file with sequential access mode is open in the extend mode, the WRITE statement releases a record to the I-O system. The first record has a relative record number one greater than the highest relative record number existing in the file. Subsequent records have consecutively higher relative record numbers. If *rec-name* has an associated RELATIVE KEY data item, the WRITE statement places the relative record number of the released record into it.

25. 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 the I-O system.

26. 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 the I-O system.

27. The invalid key condition exists when:

    - 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.
    - The number of significant digits in the relative record number is larger than the size of the relative key data item described for the file.

28. 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.

  See the rules for the INVALID KEY phrase, Section 6.6.10.

Indexed Files

29. The file must be open in the OUTPUT, I-O, or EXTEND mode when the WRITE statement executes. (See Table 6.15.)
30. Executing a Format 2 WRITE statement releases a record to the I-O system. The contents of the record keys enable later record access based on any defined key.
31. When the file description entry has a RECORD KEY IS clause, the prime record key value is unique unless the DUPLICATES phrase is specified. When a program later accesses these records sequentially, the retrieval order is the same as the order in which they were written in the program.
32. The program must set the value of the prime record key data item before executing the WRITE statement.
33. If the file is open in the sequential access mode, the program must release records in ascending or descending order of prime record key values, depending on the sort order specified in the RECORD KEY clause. If the file is open in the extend mode, the first released record must have a prime record key value that logically follows the last record in the file according to the prime key sort order.
34. If the file is open in the random or dynamic access mode, the program can release records in any order.
35. When the file description entry has an ALTERNATE RECORD KEY clause, the alternate record key value is unique unless the program specifies the DUPLICATES phrase. When a program later accesses these records sequentially, the retrieval order is the same as the order in which they were written in the program.
36. The invalid key condition occurs for any of the following:
    - The file is open in the sequential access mode *and* in the OUTPUT or EXTEND mode, and the prime record key value does not logically follow the prime record key value of the previous record.
    - The file is open in the OUTPUT, EXTEND, or I-O mode, the prime record key value duplicates an existing record's prime record key value, and the program does not specify duplicates on the prime record key.
    - The file is open in the OUTPUT, EXTEND, 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.
37. 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.

  See the rules for the INVALID KEY phrase, Section 6.6.10.

Technical Notes

- WRITE statement execution can result in these FILE STATUS data item values:

File Status	File Organization	Access Method	Meaning
00	All	All	Write is successful
02	Ind	All	Created duplicate primary or alternate key
21	Ind	Seq	Attempted key value not in prime key sort order (invalid key)
22	Ind, Rel	All	Duplicate key (invalid key)
24	Ind, Rel	All	Boundary violation (relative or indexed files) or relative record number is too large for relative key data item (invalid key)
34	Seq	Seq	Boundary violation (sequential files)
44	All	All	Boundary violation. An attempt was made to write a record that is larger than the largest or smaller than the smallest record allowed
48	All	All	File not open, or incompatible open mode
92	Ind, Rel	All	Record locked by another process
30	All	All	All other permanent errors

  In order to detect "device full" (file status 34) on a sequential WRITE operation, each WRITE needs to be followed by a call to SYS$FLUSH to ensure that an attempt has been made to write any buffered records to disk. For more information, at the OpenVMS system prompt, type

```
HELP RMS $FLUSH
```

Additional References

- Chapter 4
- Section 6.1.4: Scope of Statements
- Section 6.6.8: I-O Status
- Section 6.6.10: INVALID KEY Phrase
- Section 6.6.11: FROM Phrase

- OPEN statement
- UNLOCK statement

END PROGRAM

END PROGRAM — The END PROGRAM header indicates the end of the named COBOL source program. Alternatively, the end of a named COBOL source program can be indicated by the end of the program's Procedure Division.

General Format

END PROGRAM program-name

program-name

must contain 1 to 31 characters and follow the rules for user-defined words. It must be identical to a program-name declared in a preceding PROGRAM-ID paragraph.

Syntax Rules

1. An inside PROGRAM-ID/END PROGRAM pair must be contained within the outside pair.
2. The END PROGRAM header must be present in every program that either contains or is contained within another program.
3. The END PROGRAM header indicates the end of a specific COBOL source program.
4. The END PROGRAM header starts in Area A.
5. The only COBOL statements that can follow an END PROGRAM header are as follows:
   - An Identification Division header of another program
   - Another END PROGRAM header

   The last END PROGRAM header must reference the outermost containing program.
6. If a program includes an END PROGRAM header and if it is not contained in another program, the next COBOL statement, if any, must be the Identification Division header of another program to be compiled.

Examples

1. This separately compiled program (PROG-NAME-A) ❶ contains one program (PROG-NAME-B) ❷.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-A.  ❶
PROCEDURE DIVISION.
         ...

IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-B.  ❷
PROCEDURE DIVISION.
```

```
            ...

END PROGRAM PROG-NAME-B.
END PROGRAM PROG-NAME-A.
```

2. This separately compiled program (PROG-NAME-A) ❶ contains eight other programs ❷ through ❾. Also, ❸ is contained within ❷, and ❹ is contained within ❸. ❻, ❼, and ❽ are contained within ❺. ❷, ❺, and ❾ are directly contained within ❶.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-A.   ❶
          …
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-B.   ❷
          …
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-C.   ❸
          …
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-D.   ❹
          …
END PROGRAM PROG-NAME-D.
END PROGRAM PROG-NAME-C.
END PROGRAM PROG-NAME-B.
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-F.   ❺
          …
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-G.   ❻
          …
END PROGRAM PROG-NAME-G.
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-H.   ❼
          …
END PROGRAM PROG-NAME-H.
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-I.   ❽
          …
END PROGRAM PROG-NAME-I.
END PROGRAM PROG-NAME-F.
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-J.   ❾
          …
END PROGRAM PROG-NAME-J.
END PROGRAM PROG-NAME-A.
```

Additional Reference

Chapter 3

Chapter 7. Intrinsic Functions

Data processing problems frequently require the use of values not directly accessible in the data storage associated with a program. These data values must be derived through operations on other data. Instead of having to write code to specify many common operations step by step, the programmer can use *intrinsic functions*. An intrinsic function is treated as a temporary elementary data item that contains a temporary data value to be derived automatically at the time of reference during execution of the program.

The uses of the intrinsic functions can be summarized briefly in the following listing by category:

CATEGORY	FUNCTIONS
Scientific/Mathematical	ACOS, ASIN, ATAN, COS, FACTORIAL, LOG, LOG10, MOD, REM, SIN, SQRT, SUM, TAN
Relational	MAX, MIN, ORD-MAX, ORD-MIN
String Manipulation	LOWER-CASE, NUMVAL, NUMVAL-C, REVERSE, UPPER-CASE
Date Manipulation	CURRENT-DATE, DATE-OF-INTEGER, DATE-TO-YYYYMMDD, DAY-TO-YYYYDDD, DAY-OF-INTEGER, INTEGER-OF-DATE, INTEGER-OF-DAY, TEST-DATE-YYYYMMDD, TEST-DAY-YYYYDDD, WHEN-COMPILED, YEAR-TO-YYYY
Statistical/Accounting	ANNUITY, MEAN, MEDIAN, MIDRANGE, PRESENT-VALUE, RANGE,STANDARD-DEVIATION, VARIANCE
Other	ARGCOUNT (OpenVMS), CHAR, INTEGER, INTEGER-PART, LENGTH, ORD, RANDOM

Later in this chapter (in : Function Descriptions) you will find a comprehensive table (Table 7.1) of functions, including their types, arguments, and values returned. Following the table are complete descriptions, including formats, of the individual functions in alphabetic order.

Intrinsic Function

Intrinsic Function — A call to an intrinsic function is constructed as a *function-identifier* made up of the word FUNCTION and a *name*, as well as any applicable *arguments* and *modifiers*. The name is one of those shown in Table 7.1. An argument (see the description in the *argument* section) is selected according to application requirements.

Description

A function-identifier is a syntactically correct combination of character strings and separators that uniquely references the data item resulting from the evaluation of a function. Although intrinsic functions are treated as elementary data items, they cannot be receiving operands.

A function-identifier that references an alphanumeric function can generally be specified wherever a sending identifier is permitted and wherever a reference to a function is not specifically prohibited by general-format rules. (For example, the rules for the CALL statement prohibit a function from being referenced in a CALL statement as an argument.) An integer or numeric function can be used anywhere an arithmetic expression (defined in Section 6.4) can be used.

General Format

FUNCTION function-name [({argument}…)] [reference-modifier]

function-name

is one of the names listed in the first column of Table 7.1. A *function-name* must be specified as part of a function-identifier. Most *function-names* are not reserved words, and can be used in a program outside the context of a function.

argument

is an identifier, a literal, or an arithmetic expression. It complies with the specific rules governing the number, class, and category of *arguments* for the function. If it is an identifier, it can be subscripted, qualified, or reference-modified, and it can be a function-identifier. Functions may have between 0 and 250 *arguments* as specified by the definition of each function. The *arguments* in an argument list may be separated by a comma. *Arguments* are evaluated individually, from left to right.

Most intrinsic functions require one or more *arguments*. The programmer must specify *arguments* of the proper type and number and within the legal constraints for the function; otherwise, the result of the statement may be undefined.

reference-modifier

can be specified only for alphanumeric functions. It specifies the beginning character position to be selected in the resulting data item and optionally the length of the resulting data item. (For more information on reference modification, see Section 6.2.3.)

Functions and Subscripting

An argument of an intrinsic function that permits a variable number of arguments can be a generically subscripted table or portion of a table. Generic, or ALL, subscripting (available only for intrinsic function arguments) is the use of the word ALL to specify all elements in one or more dimensions of a table. (A table element is a data item that contains or is subordinate to an OCCURS clause; if it is an OCCURS DEPENDING ON clause, the range of values is determined by the object of the clause.) Additional arguments, if any, of the function may or may not be table names. The evaluation of an ALL subscript must result in at least one argument; otherwise the result of executing the statement is undefined.

The order of the implicit specification of each occurrence of a table element is from left to right. This process is spelled out in detail in the following paragraph:

The first (or leftmost) specification is the identifier with each subscript specified by ALL replaced by one, and the next specification is the same identifier with the rightmost subscript specified by ALL incremented by one. This process continues with the rightmost ALL subscript being incremented by one for each implicit specification until the rightmost ALL subscript has been incremented through its range of values. If there are any additional ALL subscripts, the ALL subscript immediately to the left of the rightmost ALL is incremented by one, the rightmost ALL is reset to one, and the process of varying the rightmost ALL subscript is repeated. The ALL subscript to the left of the rightmost ALL subscript is incremented by one through its range of values. For each additional ALL subscript, this process is repeated in turn until the leftmost ALL subscript has been incremented by one through its range of values. If the ALL subscript is associated with an OCCURS DEPENDING ON clause, the range of values is determined by the object of that clause.

The reference modifier (if any) of an argument with an ALL subscript applies to each of the implicitly specified elements of the table. See Chapter 6: *Procedure Division* for the general format of ALL subscripting.

When one subscript of a multidimensional table is ALL, every other subscript must be one of the following:

Another ALL subscript
A positive integer literal
The data-name of a numeric integer elementary item (optionally followed by a plus or minus sign and an integer literal)
An index-name (optionally followed by a plus or minus sign and an integer literal)

The functions that permit generic subscripting of arguments are the following:

MAX
MEAN
MEDIAN
MIDRANGE
MIN
ORD-MAX
ORD-MIN
PRESENT-VALUE
RANGE
STANDARD-DEVIATION
SUM
VARIANCE

See MAX and SUM for examples of generically subscripted arguments.

Function Descriptions

There are three types of functions, based on the type of their resultant values, as follows:

- Alphanumeric—A function whose resultant value is composed of a string of one or more characters from the computer's character set, and whose implicit usage is DISPLAY
- Integer—A function whose resultant value is a number that cannot have any nonzero digits to the right of the decimal point
- Numeric—A function whose resultant value is a number that may have nonzero digits to the right of the decimal point

Table 7.1 lists the intrinsic functions, along with their types, their arguments, and the values they return. Complete descriptions of the functions, arranged alphabetically, follow.

Table 7.1. Intrinsic Functions

Function	Number and Type of Arguments	Function Type	Value Returned
ACOS	1 numeric, *num*	Numeric	Arccosine of *num*
ANNUITY	1 numeric, *num*, and 1 integer, *int*	Numeric	Ratio of annuity paid for each of *int* periods at interest of *num* to initial investment of one monetary unit
ARGCOUNT (OpenVMS)	None	Integer	Number of arguments passed to the COBOL program

Function	Number and Type of Arguments	Function Type	Value Returned
ASIN	1 numeric, *num*	Numeric	Arcsine of *num*
ATAN	1 numeric, *num*	Numeric	Arctangent of *num*
CHAR	1 integer, *int*	Alphanumeric	Character in position *int* of program collating sequence
COS	1 numeric, *num*	Numeric	Cosine of *num*
CURRENT-DATE	None	Alphanumeric	Current date and time
DATE-OF-INTEGER	1 integer	Integer	Standard date equivalent (YYYYMMDD) of integer date [5]
DATE-TO-YYYYMMDD	1 or 2 integer	Integer	YYYYMMDD date converted from YYMMDD date
DAY-OF-INTEGER	1 integer	Integer	YYYYDDD date equivalent of integer date [5]
DAY-TO-YYYYDDD	1 or 2 integer	Integer	YYYYDDD date converted from YYDDD date
FACTORIAL	1 integer, *int*	Integer	Factorial of *int*
INTEGER	1 numeric, *num*	Integer	The greatest integer not greater than *num*
INTEGER-OF-DATE	1 integer	Integer	Integer date [5] equivalent of standard date (YYYYMMDD)
INTEGER-OF-DAY	1 integer	Integer	Integer date [5] equivalent of date in YYYYDDD format
INTEGER-PART	1 numeric, *num*	Integer	Integer part of *num*
LENGTH	1 alphabetic or numeric or alphanumeric data item, or 1 nonnumeric literal	Integer	Length of argument
LOG	1 numeric, *num*	Numeric	Natural logarithm of *num*
LOG10	1 numeric, *num*	Numeric	Logarithm to base 10 of *num*
LOWER-CASE	1 alphabetic or 1 alphanumeric	Alphanumeric	All letters in the argument set to lowercase
MAX	1 or more alphabetic and/or alphanumeric, or 1 or more integer and/or numeric	Depends on arguments [6]	Value of maximum argument

Function	Number and Type of Arguments	Function Type	Value Returned
MEAN	1 or more numeric	Numeric	Arithmetic mean of arguments
MEDIAN	1 or more numeric	Numeric	Median of arguments
MIDRANGE	1 or more numeric	Numeric	Mean of minimum and maximum arguments
MIN	1 or more alphabetic and/or alphanumeric, or 1 or more integer and/or numeric	Depends on arguments [6]	Value of minimum argument
MOD	2 integer, *int-1* and *int-2*	Integer	Value of *int-1* modulo *int-2*
NUMVAL	1 alphanumeric	Numeric	Numeric value of simple numeric string
NUMVAL-C	1 or 2 alphanumeric	Numeric	Numeric value of numeric string with optional commas and currency sign
ORD	1 alphabetic or 1 alphanumeric	Integer	Ordinal position of the argument in collating sequence
ORD-MAX	1 or more alphabetic, or 1 or more numeric, or 1 or more alphanumeric	Integer	Ordinal position of maximum argument
ORD-MIN	1 or more alphabetic, or 1 or more numeric, or 1 or more alphanumeric	Integer	Ordinal position of minimum argument
PRESENT-VALUE	1 numeric, *num-1*; and 1 or more additional numeric, *num-2*	Numeric	Present value of a series of future period-end amounts, *num-2*, at a discount rate of *num-1*
RANDOM	1 integer or none	Numeric	Pseudo-random number
RANGE	1 or more integer, or 1 or more numeric	Integer or numeric depending on arguments	Value of maximum argument minus value of minimum argument
REM	2 numeric, *num-1* and *num-2*	Numeric	Remainder of *num-1/ num-2*
REVERSE	1 alphabetic or 1 alphanumeric	Alphanumeric	Reverse order of the characters of the argument
SIN	1 numeric, *num*	Numeric	Sine of *num*
SQRT	1 numeric, *num*	Numeric	Square root of *num*
STANDARD-DEVIATION	1 or more numeric	Numeric	Standard deviation of arguments

Function	Number and Type of Arguments	Function Type	Value Returned
SUM	1 or more integer or 1 or more numeric	Integer or numeric depending on arguments	Sum of arguments
TAN	1 numeric, *num*	Numeric	Tangent of *num*
TEST-DATE-YYYYMMDD	1 integer	Integer	0,1,2, or 3, indicating whether the date is a valid date in the Gregorian calendar, and reason if invalid
TEST-DAY-YYYYDDD	1 integer	Integer	0, 1, or 2, indicating whether the Julian date is a valid date in the Gregorian calendar, and reason if invalid
UPPER-CASE	1 alphabetic or 1 alphanumeric	Alphanumeric	All letters in the argument set to uppercase
VARIANCE	1 or more numeric	Numeric	Variance of argument
WHEN-COMPILED	None	Alphanumeric	Date and time program was compiled
YEAR-TO-YYYY	1 or 2 integer	Integer	4-digit year, converted from 2-digit year

[5]An integer date is a positive integer representing the number of days after December 31, 1600, in the Gregorian calendar.

[6]A function that has only alphabetic and/or alphanumeric arguments is type alphanumeric. A function that has only integer arguments is type integer. A function that has both integer and numeric arguments is type numeric.

ACOS

ACOS — The ACOS function returns a numeric value in radians that approximates the arccosine of the argument.

General Format

`FUNCTION ACOS` (arg)

`arg`

is a numeric argument with a value greater than or equal to -1 and less than or equal to +1.

Rules

1. The type of this function is numeric.
2. The returned value is the approximation of the arccosine of arg and is greater than or equal to 0 and less than or equal to π (approximately 3.14159).

Example

```
COMPUTE RSULT = FUNCTION ACOS (.85).
```

The value returned and stored in RSULT (a numeric data item) is a number that approximates the arccosine of .85.

ANNUITY

ANNUITY — The ANNUITY function (annuity immediate) returns a numeric value that approximates the ratio of an annuity paid at the end of each period for the number of periods specified (by the second argument) to an initial investment of one. Interest is earned at the rate specified (by the first argument), and is applied at the end of the period, before the payment.

General Format

FUNCTION ANNUITY (interest-rate num-periods)

interest-rate

is a numeric argument with a value greater than or equal to 0, representing the interest rate applied at the end of the period before the payment.

num-periods

is a positive integer argument representing the number of periods.

Rules

1. The type of this function is numeric.
2. When the value of interest-rate is 0, the value of the function is the approximation of 1 / num-periods.
3. When the value of interest-rate is not 0, the value of the function is the approximation of interest-rate / (1 –(1 + interest-rate) ** (– num-periods)).

Example

```
COMPUTE RSULT = FUNCTION ANNUITY (INTEREST-RATE, NUM-PERIODS).
```

INTEREST-RATE is a numeric data item, and NUM-PERIODS is a numeric integer data item.
If the value of INTEREST-RATE is 0 and the value of NUM-PERIODS is 6, RSULT has a value approximating 1/6. If the value of INTEREST-RATE is .11 and the value of NUM-PERIODS is 6, RSULT (a numeric data item) has a value of approximately 0.2364.

ARGCOUNT (OpenVMS Only)

ARGCOUNT (OpenVMS Only) — The ARGCOUNT function returns a numeric integer equal to the number of arguments passed to the VSI COBOL for OpenVMS program.

General Format

FUNCTION ARGCOUNT

Rules

1. The type of this function is integer.
2. The returned value represents the actual number of arguments passed to the VSI COBOL for OpenVMS program that contains the function.

Example

```
IF FUNCTION ARGCOUNT = 3
       PERFORM PROCESS-OPTIONAL-3RD-ARGUMENT.
```

If there are three arguments passed to the VSI COBOL for OpenVMS program containing the ARGCOUNT function, a third argument supplied with the COBOL program calling command will be parsed and processed.

Additional Reference

Refer to the Argument Information Register in the *VSI OpenVMS Calling Standard*.

ASIN

ASIN — The ASIN function returns a numeric value in radians that approximates the arcsine of the argument.

General Format

FUNCTION ASIN (arg)

arg

is a numeric argument with a value greater than or equal to -1 and less than or equal to +1.

Rules

1. The type of this function is numeric.
2. The returned value is the approximation of the arcsine of arg and is greater than or equal to - π/2 and less than or equal to + π/2. (π is approximately 3.14159.)

Example

```
COMPUTE RSULT = FUNCTION ASIN (.675).
```

The value returned and stored in RSULT (a numeric data item) is a number that approximates the arcsine of .675.

ATAN

ATAN — The ATAN function returns a numeric value in radians that approximates the arctangent of the argument.

General Format

FUNCTION ATAN (arg)

arg

is a numeric argument.

Rules

1. The type of this function is numeric.
2. The returned value is the approximation of the arctangent of arg and is greater than - π/2 and less than + π/2. (π is approximately 3.14159.)

Example

```
COMPUTE RSULT = FUNCTION ATAN (MEASUREMENT-IN-RADIANS).
```

MEASUREMENT-IN-RADIANS and RSULT are numeric data items. The value returned and stored in RSULT is a number that approximates the arctangent of the value of MEASUREMENT-IN-RADIANS.

CHAR

CHAR — The CHAR function returns a one-character alphanumeric value that is a character in the program collating sequence having the ordinal position equal to the value of the argument.

General Format

FUNCTION CHAR (position)

position

is a positive integer argument representing the ordinal position of the desired character in the program collating sequence, and having a value less than or equal to the number of positions in the collating sequence.

Rules

1. The type of this function is alphanumeric.
2. The character returned as the function value is the character in the program collating sequence. (See the information on the ALPHABET clause in Chapter 4.)
3. If more than one character has the same position in the program collating sequence, the character returned as the function value is that of the first literal specified for that character position in the ALPHABET clause.

Example

```
MOVE FUNCTION CHAR (13) TO FORM-FEED-CH.
```

The character occupying the 13th position in the program collating sequence (that is, a form feed in the default collating sequence) is returned and stored in FORM-FEED-CH, which is an alphanumeric data item one character in length. (The numeric representation of a character is not the same as its ordinal position. Numeric representation starts at 0, whereas ordinals start at 1. Always add 1 to the numeric value of the desired character.)

COS

COS — The COS function returns a numeric value that approximates the cosine of an angle or arc, expressed in radians, that is specified by the argument.

General Format

`FUNCTION COS` (angle)

`angle`

is a numeric argument having the value of the measurement in radians of an angle or arc.

Rules

1. The type of this function is numeric.
2. The returned value is the approximation of the cosine of angle, and is greater than or equal to -1 and less than or equal to +1.

Example

```
COMPUTE COSIN-RSLT = FUNCTION COS (X).
```

X and COSIN-RSULT are numeric data items. If the value of X is 3, the approximate cosine of an angle of 3 radians is moved to COSIN-RSLT.

CURRENT-DATE

CURRENT-DATE — The CURRENT-DATE function returns a 21-character alphanumeric value that represents the calendar date and the time of day.

General Format

`FUNCTION CURRENT-DATE`

Rules

1. The type of this function is alphanumeric.
2. The contents of the character positions returned, numbered from left to right, are as follows:

Character Positions	Contents
1-4	Four numeric digits of the year in the Gregorian calendar.

Character Positions	Contents
5-6	Two numeric digits of the month of the year, in the range 01 through 12.
7-8	Two numeric digits of the day of the month, in the range 01 through 31.
9-10	Two numeric digits of the hours past midnight, in the range 00 through 23.
11-12	Two numeric digits of the minutes past the hour, in the range 00 through 59.
13-14	Two numeric digits of the seconds past the minute, in the range 00 through 59.
15-16	Two numeric digits of the hundredths of a second past the second, in the range 00 through 99.
17-21	The value 00000. (Reserved for future use.)

Example

The COBOL syntax for this function (similar to the example) is common to all platforms:

```
MOVE FUNCTION CURRENT-DATE TO RSULT.
```

```
19970110165231320000 0
```

This is a sample value returned by the example CURRENT-DATE function. Reading from left to right, it shows

- The year, 1997
- The month, January
- The day of the month, the 10th
- The time of day, 16:52 (4:52 P.M.)
- The seconds, 31, and the hundredths of seconds, 32, after 16:52:31

DATE-OF-INTEGER

DATE-OF-INTEGER — The DATE-OF-INTEGER function converts a date from an integer date form representing the number of days after December 31, 1600, to standard date form (YYYYMMDD).

General Format

FUNCTION DATE-OF-INTEGER (num-days)

num-days

is a positive integer argument that represents a number of days succeeding December 31, 1600, in the Gregorian calendar.

Rules

1. The type of this function is integer.

2. The returned value represents the ISO Standard date, in the form YYYYMMDD, that is equivalent to the integer specified. YYYY is an integer in the range 1601 through 9999. MM is an integer in the range 1 through 12. DD is an integer in the range 1 through 31.

Example

```
COMPUTE RSULT = FUNCTION DATE-OF-INTEGER (20).
```

The value returned and stored in RSULT (a numeric integer data item) is

```
16010120
```

This value represents January 20, 1601, which is 20 days after December 31, 1600.

DATE-TO-YYYYMMDD

DATE-TO-YYYYMMDD — The DATE-TO-YYYYMMDD function converts a date in the form YYMMDD to the form YYYYMMDD. An optional second argument, when added to the current year (at the time the program executes), defines the ending year of a 100-year interval. This interval determines to what century the two-digit year belongs.

General Format

FUNCTION DATE-TO-YYYYMMDD (arg-1 [arg-2])

arg-1

is a nonnegative integer between 0 and 999999.

arg-2

is an integer. Its value, when added to the current year, must be between 1700 and 9999. If it is omitted, the default value is 50.

Rules

1. The type of this function is integer.

2. The returned value is an integer representing YYYYMMDD and is calculated as follows:

```
YY = int(arg-1 / 10000)
mmdd = mod (arg-1, 10000)
return FUNCTION YEAR-TO-YYYY(YY, arg-2) * 10000 + mmdd
```

Example

```
IF FUNCTION DATE-TO-YYYYMMDD (801123, 50 )   = 19801123
   DISPLAY "correct".
IF FUNCTION DATE-TO-YYYYMMDD (801123, 100 )  = 20801123
   DISPLAY "correct".
IF FUNCTION DATE-TO-YYYYMMDD (801123, -100 ) = 18801123
   DISPLAY "correct".
```

DATE-TO-YYYYMMDD implements a sliding window algorithm. To use it for a fixed window, you can specify *arg-2* as follows:

```
(fixed-ending-year - function numval (function current-date (1:4)))
```

If fixed-ending-year is 2100, then in 1999 *arg-2* has the value 101. If *arg-1* is 501123, the returned-value is 20501123. If *arg-1* is 991123, the returned-value is 20991123.

DAY-OF-INTEGER

DAY-OF-INTEGER — The DAY-OF-INTEGER function converts a date from an integer date form representing the number of days succeeding December 31, 1600, to a date form (sometimes called "Julian") representing year and days (YYYYDDD).

General Format

`FUNCTION DAY-OF-INTEGER` (num-days)

`num-days`

is a positive integer argument that represents a number of days succeeding December 31, 1600, in the Gregorian calendar.

Rules

1. The type of this function is integer.
2. The returned value is an integer of the form YYYYDDD, where YYYY represents a year in the Gregorian calendar and DDD represents the day of that year. YYYY is an integer in the range 1601 through 9999. DDD is an integer in the range 1 through 366.

Example

```
COMPUTE RSULT = FUNCTION DAY-OF-INTEGER (28).
```

The value returned and stored in RSULT (a numeric integer data item) shows the 28th day of the year 1601 (that is, 28 days succeeding December 31, 1600), as follows:

```
1601028
```

DAY-TO-YYYYDDD

DAY-TO-YYYYDDD — The DAY-TO-YYYYDDD function converts a date in the form YYDDD to the form YYYYDDD. An optional second argument, when added to the current year (at the time the program executes), defines the ending year of a 100-year interval. This interval determines to what century the two-digit year belongs.

General Format

`FUNCTION DAY-TO-YYYYDDD` (arg-1 [arg-2])

`arg-1`

is a nonnegative integer between 0 and 99999.

`arg-2`

is an integer. Its value, when added to the current year, must be between 1700 and 9999. If it is omitted, the default value is 50.

Rules

1. The type of this function is integer.
2. The returned value is an integer representing YYYYDDD and is calculated as follows:

```
YY = int(arg-1 / 1000)
ddd = mod (arg-1, 1000)
return FUNCTION YEAR-TO-YYYY(YY, arg-2) * 1000 + ddd
```

Example

```
IF FUNCTION DAY-TO-YYYYDDD (80111, 50 )   = 1980111
   DISPLAY "correct".
IF FUNCTION DAY-TO-YYYYDDD (80111, 100 )  = 2080111
   DISPLAY "correct".
IF FUNCTION DAY-TO-YYYYDDD (80111, -100 ) = 1880111
   DISPLAY "correct".
```

DAY-TO-YYYYDDD implements a sliding window algorithm. To use it for a fixed window, you can specify *arg-2* as follows:

```
(fixed-ending-year - function numval (function current-date (1:4)))
```

If fixed-ending-year is 2100, then for 1999 *arg-2* has the value 101. If *arg-1* is 50111, the returned-value is 2050111. If *arg-1* is 99111, the returned-value is 2099111.

FACTORIAL

FACTORIAL — The FACTORIAL function returns an integer that is the factorial of the argument specified.

General Format

FUNCTION FACTORIAL (num)

num

is 0 or a positive integer argument whose value is less than or equal to 19.

Rules

1. The type of this function is integer.
2. If the value of the argument is 0, the value 1 is returned.
3. If the value of the argument is positive, its factorial is returned.

Example

```
COMPUTE RSULT = FUNCTION FACTORIAL (NUM).
```

NUM and RSULT are numeric integer data items. If NUM has the value of 5, the value returned and stored in RSULT is 120.

(5! = 5 * 4 * 3 * 2 * 1 = 120.)

INTEGER

INTEGER — The INTEGER function returns the greatest integer value that is less than or equal to the argument.

General Format

`FUNCTION INTEGER` (num)

`num`

is a numeric argument.

Rule

The type of this function is integer.

Example

```
COMPUTE RSULT = FUNCTION INTEGER (NUM).
```

If the value of NUM (a numeric data item) is -1.5, the value returned and stored in RSULT (a numeric integer data item) is -2, because -2 is the greatest integer that is less than or equal to -1.5. If the value of NUM is +1.5, the value returned and stored in RSULT is +1, because +1 is the greatest integer that is less than or equal to +1.5.

INTEGER-OF-DATE

INTEGER-OF-DATE — The INTEGER-OF-DATE function converts a date from standard date form (YYYYMMDD) to an integer date form representing the number of days after December 31, 1600.

General Format

`FUNCTION INTEGER-OF-DATE` (num)

`num`

is an integer argument of the form YYYYMMDD representing a date subsequent to December 31, 1600.

Rules

1. The type of this function is integer.
2. The value of the argument is obtained from the calculation (YYYY * 10,000) + (MM * 100) + DD. YYYY represents the year in the Gregorian calendar, and must be an integer in the range 1601

through 9999. MM represents a month and is an integer in the range 1 through 12. DD represents a day and is an integer in the range 1 through 31; the value of DD must be valid for the specified month and year combination.

3. The returned value is an integer that is the number of days the specified date succeeds December 31, 1600, in the Gregorian calendar.

Examples

1. ```
   COMPUTE RSULT = FUNCTION INTEGER-OF-DATE (NUM).
   ```

   NUM and RSULT are numeric integer data items. If NUM has the value 16010215 (that is, February 15, 1601), the value returned and stored in RSULT is 46 (the 46th day after December 31, 1600).

2. ```
   COMPUTE DAYS-IN-BILLING-CYCLE =
       FUNCTION INTEGER-OF-DATE(THIS-ENDING-DATE) -
       FUNCTION INTEGER-OF-DATE(LAST-ENDING-DATE)
   ```

   DAYS-IN-BILLING-CYCLE, THIS-ENDING-DATE, and LAST-ENDING-DATE are numeric integer items. If THIS-ENDING-DATE has the value 19970301 (representing March 1, 1997), and LAST-ENDING-DATE has the value 19970201 (representing February 1, 1997), the value returned is 28.

INTEGER-OF-DAY

INTEGER-OF-DAY — The INTEGER-OF-DAY function converts a date in the Gregorian calendar from year-day (YYYYDDD) form (sometimes called "Julian") to an integer date form representing the number of days after December 31, 1600.

General Format

FUNCTION INTEGER-OF-DAY (num)

num

is an integer argument of the form YYYYDDD representing a date subsequent to December 31, 1600.

Rules

1. The type of this function is integer.

2. The value of the argument is obtained from the calculation (YYYY * 1000) + DDD. YYYY represents the year in the Gregorian calendar, and must be an integer in the range 1601 through 9999. DDD represents a day and is an integer in the range 1 through 366; the value of DDD must be valid for the specified year.

3. The returned value is an integer that is the number of days the specified date succeeds December 31, 1600, in the Gregorian calendar.

Example

```
COMPUTE RSULT = FUNCTION INTEGER-OF-DAY (1601365).
```

The value returned and stored in RSULT (a numeric integer data item) is 365, which is the number of days succeeding December 31, 1600, and which represents December 31, 1601.

INTEGER-PART

INTEGER-PART — The INTEGER-PART function returns an integer that is the integer portion of the argument.

General Format

`FUNCTION INTEGER-PART` (num)

`num`

is a numeric argument.

Rules

1. The type of this function is integer.
2. If the value of the argument is 0, the returned value is 0.
3. If the value of the argument is positive, the returned value is the greatest integer less than or equal to the value of the argument.
4. If the value of the argument is negative, the returned value is the least integer greater than or equal to the value of the argument.

Example

```
COMPUTE RSULT = FUNCTION INTEGER-PART (NUM).
```

NUM is a numeric data item, and RSULT is a numeric integer data item. If NUM has the value 0, the value returned is 0. If NUM has the value +1.5, the value returned is +1. If NUM has the value -1.5, the value returned is -1 (the least integer greater than or equal to the value of -1.5).

LENGTH

LENGTH — The LENGTH function returns an integer equal to the length of the argument in character positions.

General Format

`FUNCTION LENGTH` (arg)

`arg`

is a nonnumeric literal or a data item of any class or category.

Rules

1. The type of this function is integer.

2. The value returned is an integer equal to the length of the argument in character positions. However, if the argument is a group data item containing a variable occurrence data item, the returned value is an integer determined by evaluation of the data item specified in the DEPENDING phrase of the OCCURS clause for that variable occurrence data item. This evaluation is accomplished according to the rules in the OCCURS clause dealing with the data item as a sending data item.

3. The returned value includes implicit FILLER characters, if any.

4. For items that are not USAGE DISPLAY, the returned value represents the allocated physical storage in bytes as described in Tables 5.12 and 5.13.

Examples

1. ```
   COMPUTE RSULT = FUNCTION LENGTH ("J. R. Donaldson").
   ```

   The value 15 is returned and stored in RSULT (a numeric integer data item).

2. ```
   01  RECORD-SIZE      PIC 9(9).
   01  RECORD1.
       05   REC-TYPE    PIC 9(4)  VALUE 23.
       05   REC-CNT     PIC 9(4)  VALUE 50.
       05   A-REC       PIC X(30) OCCURS 1 TO 100 TIMES
                                  DEPENDING ON REC-CNT.
      .
      .
      .
   COMPUTE RECORD-SIZE = FUNCTION LENGTH (RECORD1).
   CALL 'SUBR' USING RECORD1, RECORD-SIZE.
   ```

   RECORD-SIZE is a numeric integer data item. The value returned by the function and stored in RECORD-SIZE is 1508. (The computation is 4 + 4 + (50 * 30) = 1508.)

LOG

LOG — The LOG function returns a numeric value that approximates the logarithm to the base e (natural log) of the argument.

General Format

FUNCTION LOG (num)

num

is a positive numeric argument.

Rules

1. The type of this function is numeric.

2. The returned value is an approximation of the logarithm to the base e of the argument.

Example

```
COMPUTE RSULT = FUNCTION LOG (NUM).
```

NUM and RSULT are numeric data items; the value of NUM must be greater than 0. The value returned and stored in RSULT is an approximation of the logarithm to the base e of NUM.

LOG10

LOG10 — The LOG10 function returns a numeric value that approximates the logarithm to the base 10 of the argument.

General Format

FUNCTION LOG10 (num)

num

is a positive numeric argument.

Rules

1. The type of this function is numeric.
2. The returned value is an approximation of the logarithm to the base 10 of the argument.

Example

```
COMPUTE RSULT = FUNCTION LOG10 (NUM).
```

NUM and RSULT are numeric data items; the value of NUM must be greater than 0. The value returned and stored in RSULT is an approximation of the logarithm to the base 10 of NUM.

LOWER-CASE

LOWER-CASE — The LOWER-CASE function returns a character string that is the same length as the argument with each uppercase letter in the argument replaced by the corresponding lowercase letter.

General Format

FUNCTION LOWER-CASE (string)

string

is an alphabetic or alphanumeric argument at least one character in length.

Rules

1. The type of this function is alphanumeric.
2. The returned value is the same character string as the argument, except that each uppercase letter in the argument is replaced by the corresponding lowercase letter.

Example

```
MOVE FUNCTION LOWER-CASE (STR) TO LC-STR.
```

If STR (an alphanumeric data item six characters in length) contains the value "Autumn" the value returned and stored in LC-STR (also an alphanumeric data item six characters in length) is "autumn"; if STR contains "fall98" the value returned is unchanged ("fall98").

MAX

MAX — The MAX function returns the contents of the argument that contains the maximum value.

General Format

`FUNCTION MAX` ({argument}…)

`argument`

is an alphabetic, alphanumeric, integer, or numeric argument.

Rules

1. The arguments must be all alphabetic, all alphanumeric, all integer, or all numeric, except that integer and numeric arguments can be mixed and alphabetic and alphanumeric arguments can be mixed.

2. The type of the function depends on the arguments, as follows:

Arguments	Function Type
Alphabetic and/or alphanumeric	Alphanumeric
Integer (all arguments)	Integer
Numeric (some arguments might be integer)	Numeric

3. The returned value consists of the contents of the argument having the greatest value, as determined by comparisons made according to the rules for simple conditions. (See Chapter 6.)

4. If more than one argument has the same value, and that value is the maximum, the returned value consists of the contents of the leftmost of these arguments.

5. If there is only one argument, the returned value consists of the contents of that argument.

6. If the type of the function is alphanumeric, the size of the returned value is the same as the size of the argument selected as the maximum.

Examples

1.
```
MOVE FUNCTION MAX ("A", "B", "C") TO MAX-LETTER-OUT.
MOVE FUNCTION MAX (1, 2, 3) TO MAX-NUMBER-OUT.
```

   MAX-LETTER-OUT is alphabetic or alphanumeric, and receives the value "C"; MAX-NUMBER-OUT is integer and receives the value 3.

2.
```
COMPUTE ITEMC = (ITEMA + FUNCTION MAX (ITEMB, 10)).
```

   If ITEMA and ITEMB both contain the value 1, this statement results in ITEMC having the value 11.

```
IF FUNCTION MAX (A, B, C) > 100 …
```

This is equivalent to the following more complex code:

```
IF A >= B
   IF A >= C
      MOVE A TO TMP
   ELSE
      MOVE C TO TMP
ELSE
   IF B >= C
      MOVE B TO TMP
   ELSE
      MOVE C TO TMP.
IF TMP > 100 ...
```

3. The following example shows generic subscripting with reference modification:

```
05  TABLE1 PIC X(7) OCCURS 3 TIMES.
   .
   .
   .
MOVE "XAAAAAQ" TO TABLE1(1).
MOVE "XBBBBBQ" TO TABLE1(2).
MOVE "XCCCCCQ" TO TABLE1(3).
MOVE FUNCTION MAX(TABLE1(ALL)(2:5)) TO RSULT.
```

The value "CCCCC" is returned and stored in RSULT, an alphanumeric data item. The reference modifier, (2:5), applies to each element implicitly specified by the ALL subscript. Thus,

```
FUNCTION MAX(TABLE1(ALL)(2:5))
```

is equivalent to

```
FUNCTION MAX(TABLE1(1)(2:5),
             TABLE1(2)(2:5),
             TABLE1(3)(2:5))
```

MEAN

MEAN — The MEAN function returns a numeric value that is the arithmetic mean (average) of its arguments.

General Format

FUNCTION MEAN ({arg} …)

arg

is a numeric argument.

Rules

1. The type of this function is numeric.

2. The return value is the arithmetic mean of the arguments in the argument list; that is, it is the sum of the arguments divided by the number of arguments.

Examples

1. `COMPUTE AVERAGE-VALUE = FUNCTION MEAN (9, 2, 6, 7, 1).`

   The value returned and stored in AVERAGE-VALUE (a numeric data item) is 5 (the sum of the arguments divided by the number of arguments).

2. `COMPUTE MEAN-ANSWER = FUNCTION MEAN(A, B, C).`

   MEAN-ANSWER, A, B, and C are numeric data items. This code is equivalent to

   ```
   COMPUTE MEAN-ANSWER = (A + B + C ) / 3.
   ```

MEDIAN

MEDIAN — The MEDIAN function returns the median value of a list of numbers, represented by the arguments. This value is such that at least half of the values are greater than or equal to the returned value, and at least half are less than or equal.

General Format

FUNCTION MEDIAN ({num}…)

num

is a numeric argument.

Rules

1. The type of this function is numeric.
2. If the number of arguments is odd, the returned value is the middle occurrence in the sorted list.
3. If the number of arguments is even, the returned value is the arithmetic mean of the values referenced by the two middle occurrences in the sorted list.
4. The comparisons used to arrange the argument values in sorted order are made according to the rules for simple conditions. (See Chapter 6.)

Examples

1. `COMPUTE RSULT = FUNCTION MEDIAN (1, 1, 9, 2, 1).`

   The value returned and stored in RSULT (a numeric data item) is 1.

2. `COMPUTE RSULT = FUNCTION MEDIAN (1, 1, 9, 2).`

   The value returned and stored in RSULT is 1.5.

MIDRANGE

MIDRANGE — The MIDRANGE (middle range) function returns a numeric value that is the arithmetic mean (average) of the values of the minimum argument and the maximum argument.

General Format

FUNCTION MIDRANGE ({num}…)

num

is a numeric argument.

Rules

1. The type of this function is numeric.
2. The returned value is the arithmetic mean of the greatest argument value and the least argument value. The comparisons used to determine the greatest and least values are made according to the rules for simple conditions. (See Chapter 6.)
3. The values of the arguments that are neither the greatest nor the least in value do not affect the value returned.

Example

```
COMPUTE RSULT = FUNCTION MIDRANGE (1, 2, 50, 4, 3).
```

The value returned and stored in RSULT (a numeric data item) is 25.5, which is the arithmetic mean of the greatest and least arguments; that is, the sum of 50 and 1 divided by 2.

MIN

MIN — The MIN function returns the content of the argument that contains the minimum value.

General Format

FUNCTION MIN ({argument}…)

argument

is an alphabetic, alphanumeric, integer, or numeric argument.

Rules

1. The arguments must be all alphabetic, all alphanumeric, all integer, or all numeric, except that integer and numeric arguments can be mixed and alphabetic and alphanumeric arguments can be mixed.
2. The type of the function depends on the arguments, as follows:

Arguments	Function Type
Alphabetic and/or alphanumeric	Alphanumeric
Integer (all arguments)	Integer
Numeric (some arguments might be integer)	Numeric

3. The returned value consists of the contents of the argument having the least value, as determined by comparisons made according to the rules for simple conditions. (See Chapter 6.)

4. If more than one argument has the same value, and that value is the minimum, the returned value consists of the contents of the leftmost of these arguments.
5. If there is only one argument, the returned value consists of the contents of that argument.
6. If the type of the function is alphanumeric, the size of the returned value is the same as the size of the argument selected as the minimum.

Example

```
COMPUTE ITEMC = (ITEMA + FUNCTION MIN (ITEMB, 10)).
```

If ITEMA and ITEMB both contain the value 1, this statement results in ITEMC having the value 2.

MOD

MOD — The MOD function returns the value of argument-1 modulo argument-2.

General Format

FUNCTION MOD (argument-1 argument-2)

argument-1

is an integer argument.

argument-2

is an integer argument whose value cannot be 0.

Rules

1. The type of this function is integer.
2. The returned value is an integer value and is defined as the following:

   ```
   argument-1  -  (argument-2 * FUNCTION INTEGER (argument-1 / argument-2))
   ```

   (The INTEGER function returns the greatest integer value that is less than or equal to the argument. See INTEGER for more information.)

Example

```
COMPUTE RSULT = FUNCTION MOD (ARGUMENT-1, ARGUMENT-2).
```

ARGUMENT-1 and ARGUMENT-2 are numeric integer data items. Following are the expected results for some values of ARGUMENT-1 and ARGUMENT-2:

ARGUMENT-1	ARGUMENT-2	RETURN
11	5	1
-11	5	4
11	-5	-4

ARGUMENT-1	ARGUMENT-2	RETURN
-11	-5	-1

NUMVAL

NUMVAL — The NUMVAL function returns the numeric value represented by the character string specified by the argument. Leading and trailing spaces are ignored.

General Format

`FUNCTION NUMVAL` (arg)

`arg`

is an alphanumeric argument whose content has one of the following two formats:

Format 1

[space] { + | - } [space] { digit [. [digit]] | . digit } [space]

Format 2

[space] { digit [. [digit]] | . digit } [space] { + | - | CR | DB } [space]

where space is a string of 0 or more spaces, and digit is a string of 1 to 18 digits.

Rules

1. The type of this function is numeric.
2. The total number of digits in the argument must not exceed 18.
3. If the DECIMAL-POINT IS COMMA clause is specified in the SPECIAL-NAMES paragraph, a comma must be used in the argument rather than a decimal point.
4. The returned value is the numeric value represented by the argument.
5. The number of digits returned is 18.

Examples

1. `COMPUTE RSULT = FUNCTION NUMVAL ("4540").`

   The value returned and stored in RSULT (a numeric data item) is 4540.

2. ```
   MOVE "-123.49" TO OLD-ID.
   COMPUTE NEW-ID = 2 + FUNCTION NUMVAL (OLD-ID).
   ```

   OLD-ID is an alphanumeric data item, and NEW-ID is a numeric data item. The value returned by the function is the numeric value -123.49, which is added to 2, giving the sum -121.49, which is stored in NEW-ID.
   ```

NUMVAL-C

NUMVAL-C — The NUMVAL-C function returns the numeric value represented by the character string specified by the first argument. Any currency sign found in the character string and any commas preceding the decimal point are ignored in determining the result.

General Format

FUNCTION NUMVAL-C (arg-1 [arg-2])

arg-1

is an alphanumeric argument whose content has one of the following two formats:

Format 1

```
[space] + -  [space] [cs] [space] digit [, [digit] …[. [digit]] . digit
 [space]
```

Format 2

```
[space] [cs] [space] digit [, [digit] …[. [digit]] . digit [space] + -   CR
    DB   [space]
```

where *space* is a string of 0 or more spaces, *cs* is a string of 1 or more characters specified by *arg-2,* and *digit* is a string of 1 or more digits.

arg-2

(if specified) is an alphanumeric argument.

Rules

1. The type of this function is numeric.
2. The total number of digits in the first argument must not exceed 18.
3. If the DECIMAL-POINT IS COMMA clause is specified in the SPECIAL-NAMES paragraph, the functions of the comma and decimal point in the first argument are reversed.
4. If the optional second argument is not specified, the character used for *cs* is the currency symbol specified for the program.
5. The returned value is the numeric value represented by the first argument.
6. The number of digits returned is 18.

Example

```
COMPUTE RSULT = FUNCTION NUMVAL-C ("#1,000.00", "#").
```

The numeric value returned and stored in RSULT (a numeric data item) is 1000.

ORD

ORD — The ORD function returns an integer value that is the ordinal position of the argument in the collating sequence for the program. The lowest ordinal position is 1.

General Format

FUNCTION ORD (arg)

arg

is an alphabetic or alphanumeric argument one character in length.

Rules

1. The type of this function is integer.
2. The value returned is the ordinal position of the specified character in the program collating sequence. (See the information on the ALPHABET clause in Chapter 4.)

Example

```
COMPUTE POSITION = FUNCTION ORD (SINGLE-CHAR).
```

If SINGLE-CHAR (an alphabetic or alphanumeric data item) has the value "A", the integer representing the ordinal position of "A" in the program collating sequence (66 for native) is the value returned and stored in POSITION (a numeric integer data item). (The numeric representation of a character is not the same as its ordinal position. Numeric representation starts at 0, whereas ordinals start at 1. Thus, the ordinal value of a character is always 1 greater than its numeric value.)

ORD-MAX

ORD-MAX — The ORD-MAX function returns a value that is the ordinal number of the argument that contains the maximum value.

General Format

FUNCTION ORD-MAX ({arg}…)

arg

is an alphabetic, alphanumeric, integer, or numeric argument.

Rules

1. The type of this function is integer.
2. The arguments must be all alphabetic, all alphanumeric, all integer, or all numeric, except that integer and numeric arguments can be mixed and alphabetic and alphanumeric arguments can be mixed.
3. The returned value is the ordinal number that corresponds to the position of the argument having the greatest value in the argument series.

4. The comparisons used to determine the greatest value are made according to the rules for simple conditions. (See Chapter 6.)
5. If more than one argument has the same greatest value, the number returned corresponds to the position of the leftmost argument having that value.
6. If there is only one argument, the value returned is 1.

Example

```
COMPUTE RSULT = FUNCTION ORD-MAX (A, B, C).
```

A, B, and C are alphanumeric data items one character in length. If the value "A" is in A, "B" is in B, and "C" is in C, the value returned and stored in RSULT (a numeric data item) is 3, because the third argument, C, has the greatest value.

ORD-MIN

ORD-MIN — The ORD-MIN function returns a value that is the ordinal number of the argument that contains the minimum value.

General Format

FUNCTION ORD-MIN ({arg}…)

arg

is an alphabetic, alphanumeric, integer, or numeric argument.

Rules

1. The type of this function is integer.
2. The arguments must be all alphabetic, all alphanumeric, all integer, or all numeric, except that integer and numeric arguments can be mixed and alphabetic and alphanumeric arguments can be mixed.
3. The returned value is the ordinal number that corresponds to the position of the argument having the least value in the argument series.
4. The comparisons used to determine the least value are made according to the rules for simple conditions. (See Chapter 6.)
5. If more than one argument has the same least value, the number returned corresponds to the position of the leftmost argument having that value.
6. If there is only one argument, the value returned is 1.

Example

```
COMPUTE RSULT = FUNCTION ORD-MIN (A, B, C).
```

A, B, and C are alphanumeric data items one character in length. If the value "A" is in A, "B" is in B, and "C" is in C, the value returned and stored in RSULT (a numeric data item) is 1, because the first argument, A, has the least value.

PRESENT-VALUE

PRESENT-VALUE — The PRESENT-VALUE function returns a value that approximates the present value of a series of future period-end amounts at a discount rate. The discount rate is specified by the first argument, and the future period-end amount(s) by one or more subsequent arguments.

General Format

FUNCTION PRESENT-VALUE (rate {amt}…)

rate

is a numeric argument greater than -1 representing the discount rate.

amt

is a numeric argument representing a future period-end amount.

Rules

1. The type of this function is numeric.
2. The period-end amounts specified must be for periods of equal duration, and the discount rate must be the rate per period (for example, if each period is a year, then use the annual discount rate).
3. The returned value is an approximation of the summation of a series of calculations with each term in the following form:

   ```
   amt / (1 + rate) ** n
   ```

   There is one term for each occurrence of amt. The exponent, n, is incremented from 1 by 1 for each term in the series. If there are four arguments (rate, amt-1, amt-2, amt-3), the calculation is as follows:

   ```
   present-value = amt-1 / (1 + rate) ** 1 +  amt-2 / (1 + rate) ** 2 +
    amt-3 / (1 + rate) ** 3
   ```

Example

```
COMPUTE RSULT = FUNCTION PRESENT-VALUE (DISCOUNT-RATE, 2000).
```

DISCOUNT-RATE and RSULT are numeric data items. If DISCOUNT-RATE has the value 0.08, the value returned and stored in RSULT is approximately 1851.85.

RANDOM

RANDOM — The RANDOM function returns a numeric value that is a pseudo-random number from a rectangular distribution.

General Format

FUNCTION RANDOM [(seed)]

seed

is an optional integer argument with the value of 0 or a positive integer, used as the seed value to generate a sequence of pseudo-random numbers. The range of seed values that results in unique sequences of pseudo-random numbers is 0 through 2147483647.

Rules

1. The type of this function is numeric.
2. If the optional seed argument is not specified by the first reference to this function in the run unit, the seed value is 0.
3. If any subsequent reference to this function in the run unit does not specify the seed argument, the value returned is the next number in the current sequence of pseudo-random numbers.
4. If a subsequent reference to this function in the run unit does specify the seed argument, a new sequence of pseudo-random numbers is started.
5. The returned value is greater than or equal to 0 and less than 1.
6. For a given seed value, the sequence of pseudo-random numbers is always the same.

Example

```
COMPUTE RSULT-1 = FUNCTION RANDOM (12345).
   .
   .
   .
COMPUTE RSULT-2 = FUNCTION RANDOM.
   .
   .
   .
COMPUTE RSULT-3 = FUNCTION RANDOM (12345).
```

RSULT-1, RSULT-2, and RSULT-3 are numeric data items. Assuming the three sentences in the example are in the same run unit, the values returned and stored in RSULT-1 and RSULT-3 are the same. RSULT-2 has a different value consisting of the next number in the sequence that was started by the first reference to the function.

RANGE

RANGE — The RANGE function returns a value that is equal to the value of the maximum argument minus the value of the minimum argument.

General Format

FUNCTION RANGE ({num}...)

num

is a numeric or integer argument.

Rules

1. The type of this function depends upon the argument types, as follows:

Arguments	Function Type
Integer (all arguments)	Integer
Numeric (some arguments might be integer)	Numeric

2. The returned value is equal to the greatest value in the series of arguments minus the least value in the series.

3. The comparisons used to determine the greatest and least values are made according to the rules for simple conditions. (See Chapter 6.)

4. If only one argument is specified, the value returned is 0.

Example

```
COMPUTE RSULT = FUNCTION RANGE (4, 8, 10).
```

The value returned and stored in RSULT (a numeric integer data item) is 6.

REM

REM — The REM function returns a numeric value that is the remainder of the first argument divided by the second argument.

General Format

FUNCTION REM (arg-1 arg-2)

arg-1

is a numeric or integer argument.

arg-2

is a numeric or integer argument whose value cannot be 0.

Rules

1. The type of this function is numeric.

2. The returned value is the remainder of the first argument divided by the second argument, and is defined as the following expression:

   ```
   arg-1  -  (arg-2 * FUNCTION INTEGER-PART (arg-1 / arg-2))
   ```

   (The INTEGER-PART function returns an integer that is the integer portion of its argument. See INTEGER-PART.)

Examples

1. ```
   COMPUTE RSULT = FUNCTION REM (3, 2).
   ```
   ```

The value returned and stored in RSULT (a numeric data item) is 1.

2. `COMPUTE RSULT = FUNCTION REM (4, 2).`

   The value returned and stored in RSULT is 0.

REVERSE

REVERSE — The REVERSE function returns a character string of exactly the same length as the argument and whose characters are exactly the same as those of the argument, except that they are in reverse order.

General Format

`FUNCTION REVERSE` (arg)

`arg`

is an alphabetic or alphanumeric argument at least one character in length.

Rules

1. The type of this function is alphanumeric.
2. If the argument is a character string of length n, the returned value is a character string of length n.
3. When 1 is less than or equal to j and j is less than or equal to n, the character in position j of the returned value is the character from position (n–j)+1 of the argument.

Example

```
MOVE FUNCTION REVERSE (STR) TO RSULT.
```

STR and RSULT are alphanumeric data items four characters in length. If STR contains the value "ABCD" then "DCBA" is the value returned and stored in RSULT.

If the value "AB" is moved to the four-character data item STR, then STR will actually contain "AB " with two trailing spaces. Then the REVERSE function returns the value " BA" with two leading spaces.

SIN

SIN — The SIN function returns a numeric value that approximates the sine of an angle or arc, expressed in radians, that is specified by the argument.

General Format

`FUNCTION SIN` (angle)

`angle`

is a numeric argument having the value of the measurement in radians of an angle or arc.

Rules

1. The type of this function is numeric.
2. The returned value is the approximation of the sine of angle, and is greater than or equal to -1 and less than or equal to +1.

Example

```
COMPUTE SIN-RSLT = FUNCTION SIN (X).
```

If the value of X is 3, the approximate sine of an angle of 3 radians is moved to SIN-RSLT (a numeric data item).

SQRT

SQRT — The SQRT function returns a numeric value that approximates the square root of the argument.

General Format

FUNCTION SQRT (num)

num

is a numeric or integer argument whose value must be 0 or positive.

Rules

1. The type of this function is numeric.
2. The returned value is the absolute value of the approximation of the square root of the argument.

Example

```
COMPUTE RSULT = FUNCTION SQRT (NUM).
```

NUM and RSULT are numeric data items. If NUM has the value 4, the value returned and stored in RSULT is 2.

STANDARD-DEVIATION

STANDARD-DEVIATION — The STANDARD-DEVIATION function returns a numeric value that approximates the standard deviation of its arguments.

General Format

FUNCTION STANDARD-DEVIATION ({arg}…)

arg

is a numeric or integer argument.

Rules

1. The type of this function is numeric.
2. The returned value is the approximation of the standard deviation of the argument series.
3. The returned value is calculated as follows:
   a. The difference between each argument's value and the arithmetic mean (average) of the argument series is calculated and squared.
   b. The values obtained are then added together. This sum is divided by the number of values in the argument series.
   c. The square root of the quotient obtained is then calculated. The returned value is the absolute value of this square root.
4. If the argument series consists of only one value, the returned value is 0.

Example

```
COMPUTE RSULT = FUNCTION STANDARD-DEVIATION (A, B, C).
```

A, B, C, and RSULT are numeric data items. If A has the value 1, B has 2, and C has 12, the standard deviation of these values (approximately 4.96655) is returned and stored in RSULT.

SUM

SUM — The SUM function returns a value that is the sum of the arguments.

General Format

FUNCTION SUM ({arg}…)

arg

is an integer or numeric argument.

Rules

1. The type of this function depends on the argument types, as follows:

Arguments	Function Type
Integer (all arguments)	Integer
Numeric (some arguments might be integer)	Numeric

2. The returned value is the sum of the arguments.

Examples

1. ```
   COMPUTE RSULT = FUNCTION SUM (A, B, C).
   ```

   A, B, C, and RSULT are numeric or numeric integer data items. If A has the value +4, B -2, and C +1, the sum of +3 is the value returned and stored in RSULT.

2. ```
   COMPUTE TOTAL-OUT =
      FUNCTION SUM(FUNCTION SQRT(X),
                   FUNCTION MOD(Y, Z),
                   A * B,
                   FUNCTION ACOS(1)).
   ```

   This example shows functions used as arguments to another function. The data items are all numeric or numeric integer. The value returned and stored in TOTAL-OUT is the approximate value of the result of adding the values returned by the functions SQRT, MOD, and ACOS to another arithmetic expression, A * B.

3. The following example shows two arguments that are tables, with generic (ALL) subscripting, and a third argument that is a literal:

   ```
   FUNCTION SUM(A(ALL), B(ALL, 2), 4)
   ```

   The number of subscripts shows that A is a one-dimensional table and B is a two-dimensional table. If A has three occurrences, then A(ALL) is a set consisting of the elements A(1), A(2), and A(3). If B has two occurrences in its outer dimension, then B(ALL, 2) is a set consisting of the elements in B(1, 2) and B(2, 2).

   If A has three elements altogether with the values 2 in A(1), 3 in A(2), and 3 in A(3), *and* if B has the values 9 in B(1, 2) and 3 in B(2, 2), then the value returned is 24—the sum of 2, 3, 3 (from table A), 9, 3 (from table B), and 4 (the third argument).

TAN

TAN — The TAN function returns a numeric value that approximates the tangent of an angle or arc, expressed in radians, that is specified by the argument.

General Format

`FUNCTION TAN` (arg)

`arg`

is a numeric or integer argument.

Rules

1. The type of this function is numeric.

2. The returned value is the approximate tangent of the angle specified.

Example

```
COMPUTE TAN-RSLT = FUNCTION TAN (X).
```

X and TAN-RSULT are numeric data items. If the value of X is 3, the approximate tangent of an angle of 3 radians is moved to TAN-RSLT.

TEST-DATE-YYYYMMDD

TEST-DATE-YYYYMMDD — The TEST-DATE-YYYYMMDD function tests whether a standard date in the form YYYYMMDD is a valid date in the Gregorian calendar.

General Format

FUNCTION TEST-DATE-YYYYMMDD (arg)

arg

is an integer.

Rules

1. The type of this function is integer.
2. If the year is not within the range 1601 through 9999, the function returns a 1.

   Otherwise, if the month is not within the range 1 through 12, the function returns a 2.

   Otherwise, if the number of days is invalid for the given month, the function returns a 3.

   Otherwise, the function returns a 0 to indicate the date is a valid date in the form YYYYMMDD.

Example

```
IF FUNCTION TEST-DATE-YYYYMMDD (123456789) = 1
   DISPLAY "correct - invalid year (12345)".
IF FUNCTION TEST-DATE-YYYYMMDD (19952020)  = 2
   DISPLAY "correct - invalid mm  (20)".
IF FUNCTION TEST-DATE-YYYYMMDD (19950229)  = 3
   DISPLAY "correct - invalid dd  (29)".
IF FUNCTION TEST-DATE-YYYYMMDD (20040229)  = 0
   DISPLAY "correct - valid YYYYMMDD".
```

TEST-DAY-YYYYDDD

TEST-DAY-YYYYDDD — The TEST-DAY-YYYYDDD function tests whether a Julian date in the form YYYYDDD is a valid date in the Gregorian calendar.

General Format

FUNCTION TEST-DAY-YYYYDDD (arg)

arg

is an integer.

Rules

1. The type of this function is integer.
2. If the year is not within the range 1601 through 9999, the function returns a 1.

   Otherwise, if the number of days is invalid for the given year, the function returns a 2.

   Otherwise, the function returns a 0 to indicate the date is a valid date in the form YYYYDDD.

Example

```
IF FUNCTION TEST-DAY-YYYYDDD (12345678) = 1
   DISPLAY "correct - invalid year (12345)".
IF FUNCTION TEST-DAY-YYYYDDD (1995366)  = 2
   DISPLAY "correct - invalid ddd (366)".
IF FUNCTION TEST-DAY-YYYYDDD (2004366)  = 0
   DISPLAY "correct - valid YYYYDDD".
```

UPPER-CASE

UPPER-CASE — The UPPER-CASE function returns a character string that is the same length as the argument with each lowercase letter in the argument replaced by the corresponding uppercase letter.

General Format

`FUNCTION UPPER-CASE` (string)

`string`

is an alphabetic or alphanumeric argument at least one character in length.

Rules

1. The type of this function is alphanumeric.
2. The returned value is the same character string as the argument, except that each lowercase letter in the argument is replaced by the corresponding uppercase letter.

Examples

1. `MOVE FUNCTION UPPER-CASE (STR) TO UC-STR.`

   If STR (an alphanumeric data item six characters in length) contains the value "Autumn" the value returned and stored in UC-STR (also an alphanumeric data item six characters in length) is "AUTUMN"; if STR contains "FALL98" the value returned is unchanged ("FALL98").

2. ```
   ACCEPT NAME-FIELD.
   WRITE RECORD-OUT
         FROM FUNCTION UPPER-CASE(NAME-FIELD).
   ```

   The value in NAME-FIELD is made all-uppercase, unless it was already all-uppercase, in which case it is unchanged. Any nonalphabetic characters remain unchanged.
   ```

VARIANCE

VARIANCE — The VARIANCE function returns a numeric value that approximates the variance of its arguments.

General Format

FUNCTION VARIANCE ({arg}...)

arg

is an integer or numeric argument.

Rules

1. The type of this function is numeric.
2. The returned value is the approximation of the variance of the argument series, and is defined as the square of the standard deviation of the argument series. (For a definition of standard deviation, see the description of the STANDARD-DEVIATION function.)
3. If the argument series consists of only one value, the returned value is 0.

Examples

1. `COMPUTE RSULT = FUNCTION VARIANCE (A).`

   The value returned and stored in RSULT is 0, because there is only one argument.

2. `COMPUTE RSULT = FUNCTION VARIANCE (A, B, C).`

   If A has the value 1, B has 2, and C has 12, the value returned and stored in RSULT is approximately 24.667. This represents the variance, which is the square of the standard deviation of these arguments; the calculation is described in the description of the STANDARD-DEVIATION function. In the above examples, A, B, C, and RSULT are numeric data items.

WHEN-COMPILED

WHEN-COMPILED — The WHEN-COMPILED function returns the date and time the program was compiled.

General Format

FUNCTION WHEN-COMPILED

Rules

1. The type of this function is alphanumeric.
2. The returned value is the date and time of compilation of the source program that contains this function. If the program is a contained program, the returned value is the compilation date and time associated with the separately compiled program in which it is contained.

3. The returned value denotes the same time as the compilation date and time provided in the listing of the source program and in the generated object code for the source program. The representation differs, and the precision can differ, as shown in the second example.

4. The contents of the character positions returned, numbered from left to right, are as follows:

Character Positions	Contents
1-4	Four numeric digits of the year in the Gregorian calendar.
5-6	Two numeric digits of the month of the year, in the range 01 through 12.
7-8	Two numeric digits of the day of the month, in the range 01 through 31.
9-10	Two numeric digits of the hours past midnight, in the range 00 through 23.
11-12	Two numeric digits of the minutes past the hour, in the range 00 through 59.
13-14	Two numeric digits of the seconds past the minute, in the range 00 through 59.
15-16	Two numeric digits of the hundredths of a second past the second, in the range 00 through 99.
17-21	The value 00000. (Reserved for future use.)

Examples

```
MOVE FUNCTION WHEN-COMPILED TO VERSION-STAMP.
```

The value returned and stored in VERSION-STAMP (an alphanumeric data item) is the date and time of the program's compilation.

```
19970110165231320000 0
```

This is a sample value returned by the WHEN-COMPILED function. Reading from left to right, it shows:

- The year, 1997
- The month, January
- The day of the month, the 10th
- The time of day, 16:52 (4:52 P.M.)
- The seconds, 31, and the hundredths of seconds, 32, after 16:52:31

This compilation date and time as shown on the compiler listing (which does not show hundredths of seconds) is as follows:

```
10-Jan-1997 16:52:31
```

YEAR-TO-YYYY

YEAR-TO-YYYY — The YEAR-TO-YYYY function converts a two-digit year to a four-digit year. An optional second argument, when added to the current year (at the time the program executes), defines the ending year of a 100-year interval. This interval determines to what century the two-digit year belongs.

General Format

FUNCTION YEAR-TO-YYYY (arg-1 [[arg-2])

arg-1

is a nonnegative integer between 0 and 99.

arg-2

is an integer. Its value, when added to the current year, must be between 1700 and 9999. If it is omitted, the default value is 50.

Rules

1. The type of this function is integer.
2. The returned value is an integer representing a four-digit year calculated as follows:

```
max-year = current-yyyy + arg-2
if mod(max-year, 100) >= arg-1
   return (arg-1 + 100 * int(max-year / 100))
else
   return (arg-1 + 100 * int(max-year / 100) - 1)
```

Example

```
IF FUNCTION YEAR-TO-YYYY (80, 50 )   = 1980
   DISPLAY "correct".
IF FUNCTION YEAR-TO-YYYY (80, 100 )  = 2080
   DISPLAY "correct".
IF FUNCTION YEAR-TO-YYYY (80, -100 ) = 1880
   DISPLAY "correct".
```

YEAR-TO-YYYY implements a sliding window algorithm. To use it for a fixed window, you can specify *arg-2* as follows:

```
(fixed-ending-year - function numval (function current-date (1:4)))
```

If fixed-ending-year is 2100, then for 1999 *arg-2* has the value 101. If *arg-1* is 50, the returned-value is 2050. If *arg-1* is 99, the returned-value is 2099.

Chapter 8. Source Text Manipulation

Source programs can copy frequently used COBOL text from a UNIX directory containing library files, an OpenVMS Librarian file, a COBOL library file, or (for OpenVMS) Oracle CDD/Repository. The COPY statement can include text without change, or it can change the text as it is copied into the source program.

The COPY statement REPLACING phrase changes text in the copying process. It matches arguments against the text to determine which text to replace. The matching procedure operates on text-words.

The REPLACE statement changes text in the source program. It matches source text to the pseudo-text specified in the REPLACE statement and changes the specified text when a match is detected.

8.1. Text-Word Definition Rules

A **text-word** is a character or sequence of characters in a COBOL library, source program, pseudo-text, or repository. It can be any of the following:

1. A literal, including the opening and closing quotation marks for nonnumeric literals
2. A hexadecimal literal, including the opening and closing delimiters
3. A separator other than:

   A space
   A pseudo-text delimiter
   The opening and closing quotation marks of a nonnumeric literal

4. Any other sequence of contiguous characters, bounded by separators, except:

   Comment lines
   Separators

Examples

These examples show how the compiler interprets COBOL text in terms of text-words. The rule letters refer to the text-word definition rules.

Text	Interpretation
MOVE	One text-word (Rule 4).
MOVE ITEMA TO ITEMB	Four text-words.
MOVE ITEMA TO ITEMB.	Five text-words. The separator period is a text-word (Rule 3).
PIC S9(4)V9(6)	Nine text-words. Each parenthesis is a separator, and therefore a text-word. The nine text-words are PIC, S9, (, 4,), V9, (, 6, and).
“PIC S9(4)V9(6)”	One text-word (Rule 1).
X “4865784C6974”	One text-word (Rule 2).
ITEMA.	Two text-words. ITEMA and the separator period are text words.

Text	Interpretation
==ITEMA. #==	Two text-words. The pseudo-text delimiters are not text-words (Rule 3). However, the separator period is a text-word.
==ITEMA.==	One text-word. The pseudo-text delimiters are not text-words. The punctuation character period is part of the character-string "ITEMA."; the period is not a separator because a space does not follow it.

COPY

COPY — The COPY statement includes text in a COBOL program.

General Formats

```
COPY text-name [ { OF } library-name ]
                 { IN }

[ REPLACING
    { { ==pseudo-text-1== }  BY { ==pseudo-text-2== } }
    { { identifier-1      }     { identifier-2      } }
    { { literal-1         }     { literal-2         } }
    { { word-1            }     { word-2            } } ... ] .
```

`text-name`

is the name of a COBOL library file available during compilation; or, if *library-name* is specified, is the name of a text record within the library file. (See Technical Notes.)

`library-name`

is the directory that contains library files on the UNIX system; or, on OpenVMS, is the name of the OpenVMS Librarian library file that contains *text-name*. (See Technical Notes.)

`pseudo-text-1`

are text-matching arguments that the compiler compares against text-words in the library text.

[pseudo-text-2]

are replacement items that the compiler inserts into the source program.

`record-name (OpenVMS)`

is a partial or complete Oracle CDD/Repository pathname. It specifies the Oracle CDD/Repository record description to be copied into the source program. (See Technical Notes.)

Syntax Rules

1. A Format 1 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. On OpenVMS, a Format 2 COPY statement can appear only in the File, Working-Storage, or Linkage Sections.

3. A space must precede the word COPY.
4. The COPY statement must be terminated by the separator period.
5. *pseudo-text-1* must contain at least one text-word.
6. *pseudo-text-2* can contain zero, one, or more text-words.
7. *word-1* or *word-2* can be any COBOL word.
8. *pseudo-text-1* must not consist entirely of a separator comma or a separator semicolon.

General Rules

Format 1

1. On UNIX, when both *text-name* and *library-name* are specified, *library-name* refers to the directory containing library files; *text-name* identifies a specific file within the directory.
2. On OpenVMS, when both *text-name* and *library-name* are specified, *library-name* refers to an OpenVMS Librarian library file; *text-name* identifies a text record within the library file.
3. When only *text-name* is used, it identifies a file that contains library text.
4. Library text must follow the source reference format rules. Library text and source program text formats must be the same; that is, both must be ANSI format, or both must be terminal format.
5. On UNIX, the COPY statement references source text from a directory containing library files or from a COBOL library file.

Format 2 (OpenVMS)

6. *record-name* refers to a record description stored in Oracle CDD/Repository.
7. The compiler translates the record description associated with *record-name* to COBOL source text. If the source program containing the COPY statement is in terminal format, the translated record description is in terminal format; otherwise, the record description is translated to ANSI format.

Both Formats

8. On OpenVMS, the COPY statement references source text from an OpenVMS Librarian file, a COBOL library file, or the Oracle CDD/Repository.
9. The compiler evaluates the COBOL source program after processing all COPY statements.
10. The COPY statement does not change the original source program text file.
11. The COPY statement causes the compiler to copy the source text associated with *text-name* into the program. The source text logically replaces the COPY statement, beginning with the word COPY and ending with the punctuation character period (inclusive).
12. If there is no REPLACING phrase, the compiler copies the source text without modification.
13. If there is a REPLACING phrase, the compiler changes the source text as it copies it. The compiler replaces each successfully matched occurrence of a text-matching argument in the source text by the corresponding replacement item.

14. For the purposes of matching, the compiler treats each text-matching argument as pseudo-text that contains *identifier-1*, *word-1*, or *literal-1*.

15. The comparison operation starts with the leftmost source text text-word and the first text-matching argument. The compiler compares the entire text-matching argument to an equivalent number of consecutive source text text-words.

16. A text-matching argument matches the source text only if the ordered sequence of text-words that forms the text-matching argument is equal, character for character, to the ordered sequence of source text text-words.

    In the matching operation, the compiler treats each occurrence or combination of the following items in source text as a single space:

    - Separator comma
    - Separator semicolon
    - A sequence of one or more separator spaces
    - A blank line
    - A comment line

17. If no match occurs, the compiler repeats the comparison with each successive text-matching argument in the REPLACING phrase until either:

    - A match occurs.
    - There are no more text-matching arguments.

18. If no match occurs after the compiler compares all text-matching arguments, the compiler copies the leftmost source text text-word into the source program. The next source text text-word then becomes the leftmost text-word for the next cycle. The comparison cycle resumes with the first text-matching argument in the REPLACING phrase.

19. If a match occurs between a text-matching argument and the source text, the compiler inserts the replacement item into the source program. The source text-word immediately after the rightmost replaced text-word then becomes the leftmost text-word for the next cycle. The comparison cycle resumes with the first text-matching argument in the REPLACING phrase.

20. The comparison cycle continues until the rightmost text-word in the source text has been either:

    - Matched and replaced
    - Used as the leftmost library text-word in a comparison cycle

21. The rules for Reference Format determine the sequence of text-words in the source text and the text-matching arguments.

22. When the compiler inserts *pseudo-text-2* into the source program, any comment lines or blank lines within *pseudo-text-2* are inserted without modification. (See Example 5.)

23. The compiler copies any comment lines and blank lines in the source text into the source program unchanged (see Example 1). However, the compiler does not copy a comment line or blank line from the source text if it is in the sequence of text-words that matches the text-matching argument.

24. The resultant source program cannot contain a COPY statement after the compiler processes a COPY statement.
    - Text copied from a source text cannot contain a COPY statement unless the replacement operation changes the word COPY in the resultant source text.
    - The replacement item in the REPLACING phrase must not insert a COPY statement into the source text.
25. The compiler cannot determine the syntactic correctness of source text, or the source program, until all COPY statements are processed.
26. When the compiler copies a text-word from the source text, it places it in the source program beginning in the same area as in the source text. That is, a text-word that begins in Area A in the source text begins in Area A of the source program after the copy operation. Similarly, a text-word that begins in Area B in the source text begins somewhere in Area B of the source program.
27. When the compiler inserts a text-word from *pseudo-text-2*, it places it in the source program beginning in the same area as in *pseudo-text-2*.
28. When the compiler inserts text from *identifier-2*, *literal-2*, or *word-2*, it places the first text-word in the source program beginning in the same area as the leftmost library text-word that matches the argument. It places all other replacement text-words in the source program beginning in the same area as they appear in the COPY statement.
29. Pseudo-text insertion can change parts of a single character-string. An unmatched text-word and a replaced text-word can combine to form a character-string. For example, the COPY statement can replace part of a PICTURE character-string. (See Example 3.)
30. Conditional compilation lines are permitted within the library text and pseudo-text. Text-words within a conditional compilation line participate in the matching process as if the indicator area character of the line on which they began was not present. A conditional compilation line is specified within pseudo-text if it begins in the source program after the opening pseudo-text delimiter, but before the matching closing pseudo-text delimiter.
31. The resultant text can occur on conditional compilation lines according to the following precedence rules:
    - If a COPY statement begins on a conditional compilation line, each line of the resulting text appears on the same kind of line.
    - If a library text-word that is not involved in a match begins on a conditional compilation line, it appears in resulting text on the same kind of line.
    - If the first library text-word that is involved in a match begins on a conditional compilation line, the *identifier-2*, *literal-2*, *word-2*, or *pseudo-text-2* that replaces the first library text-word appears on the same kind of line.
    - If text-words within *pseudo-text-2* begin on a conditional compilation line, resulting text appears on the same kind of line.

Technical Notes

Format 1

1. If there is a *library-name* phrase, *text-name* is *tr-name*.

On OpenVMS systems, *tr-name* is defined as a user-defined name or nonnumeric literal that matches the name of a text record in *library-name*.

2. *library-name* (or *text-name*, when there is no *library-name* phrase) represents a complete or partial file specification, defined to be *f-name*: *f-name* is a nonnumeric literal or a COBOL word formed according to the rules for user-defined names.

3. On Alpha and I64 systems, the COBOL command-line qualifier /INCLUDE (or `-include`) can be used at compile time to set up a search list for the COPY command. Assume that the following conditions are all true:

   - *library-name* is not used.
   - *text-name* does not include a directory specification.
   - /INCLUDE is specified.

   Then the compiler searches for the *text-name* file in the following order:

   a. The current working directory when the compiler is invoked

   b. The directory specified after /INCLUDE (or `-include`)

      If more than one directory is specified, they are searched in left to right order.

   On Alpha and I64 systems, if *library-name* is used, and *library-name* does not include a directory specification, the /INCLUDE qualifier causes a search for a .TLB file, in the same search order.

   The first file encountered that matches the *text-name* terminates the search.

   On UNIX, /INCLUDE (or `-include`) can be used to set up a search list for a *text-name* without a directory specification only if *library-name* is not specified.

   If the default, /NOINCLUDE, is in effect, a file without a directory specification is searched for only in the current default directory at compile time.

   The pathname(s) specified with /INCLUDE can be relative or absolute directory specifications, logical names on OpenVMS Alpha or I64, or environment variables on UNIX.

4. If *f-name* or *tr-name* is not a literal, the compiler translates hyphens in the word to underscore characters and treats it as if it were enclosed in quotation marks.

5. When the COPY statement executes, the I/O system:

   - Removes leading and trailing spaces and tab characters from *f-name* and from *tr-name*
   - Translates lowercase letters to uppercase in *f-name* and in *tr-name*

6. The default file type for *text-name*, when *text-name* is *f-name*, is LIB. For example, COPY CUSTFILE becomes COPY CUSTFILE.LIB.

7. On OpenVMS, the default file type for *library-name* is TLB. For example, COPY "ACCOUNTS" OF CUSTLIB becomes COPY "ACCOUNTS" OF CUSTLIB.TLB.

8. On all platforms, file names must conform to the rules of the operating system where compilation occurs. For example:

- On UNIX systems:

```
COPY "/usr/proj/empl/empl_file".

COPY "empl_file" IN "/usr/proj/empl".
```

- On OpenVMS systems:

```
COPY "COPYDIR:EMPL_FILE.LIB".

COPY "EMPL_FILE" IN "EMPLLIB.TLB".
```

Format 2 (OpenVMS)

11. *record-name* can be a nonnumeric literal or COBOL word formed according to the rules for user-defined names. It represents a complete or partial Oracle CDD/Repository pathname specifying the Oracle CDD/Repository record description to be copied into the source program. If *record-name* is not a literal, the compiler translates hyphens in the COBOL word to underline characters.

    The resultant pathname must conform to all rules for forming Oracle CDD/Repository pathnames.

12. Table 8.1 shows the representation of Oracle CDD/Repository data types in the VSI COBOL for OpenVMS compiler. It lists the data types that can be specified using CDO with the corresponding COBOL data item picture. Note that COBOL does not have an equivalent specification for some data types.

Table 8.1. Oracle CDD/Repository Data Types and VSI COBOL for OpenVMS Equivalents (OpenVMS)

Oracle CDD/Repository Data Type	VSI COBOL for OpenVMS Equivalent
BIT l	No equivalent [1]
SIGNED BYTE l s	No equivalent [1]
UNSIGNED BYTE l s	No equivalent [1]
D_FLOATING s	COMP-2 (with /FLOAT=D_FLOAT)
D_FLOATING COMPLEX s	No equivalent [1]
DATE	No exact equivalent [2]
F_FLOATING s	COMP-1 (with /FLOAT=D_FLOAT or /FLOAT=G_FLOAT)
F_FLOATING COMPLEX s	No equivalent [1]
G_FLOATING s	COMP-2 (with /FLOAT=G_FLOAT) on Alpha, I64; no equivalent on VAX
G_FLOATING COMPLEX s	No equivalent [1]
H_FLOATING s	No equivalent [1]
H_FLOATING COMPLEX s	No equivalent [1]
IEEE S_FLOATING	COMP-1 (with /FLOAT=IEEE_FLOAT) on Alpha, I64; no equivalent on VAX
l The total number of digits in the item. **s The decimal offset to l.**	

Oracle CDD/Repository Data Type	VSI COBOL for OpenVMS Equivalent
IEEE T_FLOATING	COMP-2 (with /FLOAT=IEEE_FLOAT) on Alpha, I64; no equivalent on VAX
SIGNED LONGWORD l s	S9 (9) COMP
UNSIGNED LONGWORD l s	No exact equivalent [3]
UNSIGNED NUMERIC l s	9 (m)V9 (n)
SIGNED NUMERIC LEFT SEPARATE l s	S9 (m)V9 (n) LEADING SEPARATE
SIGNED NUMERIC LEFT OVERPUNCHED l s	S9 (m)V9 (n) LEADING
SIGNED NUMERIC RIGHT SEPARATE l s	S9 (m)V9 (n) TRAILING SEPARATE
SIGNED NUMERIC RIGHT OVERPUNCHED l s	S9 (m)V9 (n) TRAILING
SIGNED OCTAWORD l s	S9 (31) COMP on Alpha, I64; no equivalent on VAX
UNSIGNED OCTAWORD l s	No exact equivalent [3]
PACKED NUMERIC l s	S9 (m)V9 (n) COMP-3
SIGNED QUADWORD l s	S9(18) COMP
UNSIGNED QUADWORD l s	No exact equivalent [3]
TEXT m CHARACTERS	X (m)
UNSPECIFIED m BYTES	X (m)
VARYING STRING m CHARACTERS	No equivalent [1]
VIRTUAL FIELD	Ignored [4]
SIGNED WORD l s	S9 (4) COMP
UNSIGNED WORD l s	No exact equivalent [3]
POINTER	POINTER
SEGMENTED STRING	No equivalent [1]
ZONED	No equivalent [1]
l The total number of digits in the item. **s The decimal offset to l.**	

[1]COBOL has no equivalent for this data type. A warning diagnostic will be issued for such an item that is part of a record description entry. The compiler will treat that item as if it had been specified as an alphanumeric data item that occupies that same number of bytes.

[2]COBOL has no exact equivalent for this data type. A warning diagnostic will be issued for such an item that is part of a record description entry. The compiler will treat that item as if it had been specified as PIC S9(11)V9(7) COMP. (This gives the item units of seconds.)

[3]COBOL has no exact equivalent for this data type. A warning diagnostic will be issued for such an item that is part of a record description entry. The compiler will treat that item as if it had been specified as the corresponding unsigned COMP data type.

[4]The VSI COBOL for OpenVMS compiler ignores this data item and all its phrases.

The method for describing the assumed decimal point is different in the two products. In a COBOL picture, the decimal position is directly indicated by the symbol V or implied by the symbol P. In CDO, scaled numbers are specified by two integers: (1) the first integer represents the total number of decimal digits that the item represents, and (2) the second integer represents the decimal offset to the first integer. These are indicated in Table 8.1 by l and s, respectively.

For example, the COBOL data item described by PIC 9(4)V99 is equivalent to the CDO entry UNSIGNED NUMERIC 6 DIGITS SCALE -2. Similarly, the CDO entry SIGNED NUMERIC LEFT SEPARATE NUMERIC 6 DIGITS SCALE 2 is equivalent to the COBOL description PIC

S9(6)PP SIGN IS LEADING SEPARATE. You can also represent digits to the right of the decimal point in CDO with the FRACTIONS phrase. For example, instead of UNSIGNED NUMERIC 6 DIGITS SCALE -2, you can also use UNSIGNED NUMERIC 6 DIGITS 2 FRACTIONS.

13. One of the primary goals of Oracle CDD/Repository is to describe data in such a way that data definitions can be shared among many different processors. Many languages have different semantic interpretations for the same physical data. Record descriptions in Oracle CDD/Repository must be able to describe the physical characteristics of data unambiguously. In other words, the logical view of the data must be separated from the physical description if different processors are to access the same record description.

    VSI COBOL for OpenVMS expects numeric literals and PICTURE character-strings to be obtained from Oracle CDD/Repository in standard representation. Whether or not a particular COBOL source program uses the DECIMAL-POINT IS COMMA clause or the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph, the record description that was stored in Oracle CDD/Repository must have used the period (.) to represent the decimal point in numeric literals and PICTURE character-strings, the comma (,) to represent the comma in PICTURE character-strings, and the currency symbol ($) to represent the currency symbol in PICTURE character-strings.

    When the COBOL source program contains the DECIMAL-POINT IS COMMA clause, the VSI COBOL for OpenVMS compiler substitutes commas for decimal points in numeric literals and PICTURE character-strings obtained from Oracle CDD/Repository. It substitutes decimal points for commas in PICTURE character-strings obtained from Oracle CDD/Repository.

    When the COBOL source program contains the CURRENCY SIGN clause, the VSI COBOL for OpenVMS compiler substitutes the currency symbol for the currency sign in PICTURE character-strings obtained from Oracle CDD/Repository.

Examples Using Format 1

The examples that follow copy library text from two library files:

- Contents of CUSTFILE.LIB:

```
01[Tab]CUSTOMER-REC.
[Tab]03  CUST-REC-KEY[Tab]PIC X(03) VALUE "KEY".
[Tab]03  CUST-NAME[Tab]PIC X(25).
[Tab]03  CUST-ADDRESS.
[Tab]    05  CUST-CUST-STREET[Tab]PIC X(20).
[Tab]    05  CUST-CITY[Tab]PIC X(20).
[Tab]    05  CUST-STATE[Tab]PIC XX.
[Tab]    05  CUST-ZIP[Tab]PIC 9(5).
* THE COMPILER IGNORES COMMENT LINES AND BLANK LINES

* FOR MATCHING PURPOSES
[Tab]03  CUST-ORDERS OCCURS XYZ TIMES.
[Tab]    05  CUST-ORDER[Tab]PIC 9(6).
[Tab]    05  CUST-ORDER-DATE[Tab]PIC 9(6).
[Tab]    05  CUST-ORDER-AMT[Tab]PIC 9(R)V99.
```

- Contents of CPROC01.LIB:

```
[Tab]ADD CUST-ORDER-AMT (X) TO TOTAL-ORDERS.
[Tab]COMPUTE AVERAGE-ORDER = (TOTAL-ORDERS - CANCELED-ORDERS)
[Tab]  / NUMBER-ORDERS.
[Tab]MOVE CUST-REC-KEY
```

```
   [Tab]  OF CUSTOMER-REC TO CUST-ID (X).
   [Tab]MOVE CUST-REC-KEY
   [Tab]  OF KEY-HOLD TO NEW-KEY.
```

In the following examples, the original source program text is shown in lowercase text. The text that is copied is shown in uppercase. (The messages in these examples are in OpenVMS Alpha and I64 format.)

Example 8.1 shows the results of a COPY statement with no REPLACING phrase. The compiler copies the library text without change. In this example, syntax errors result from invalid library text.

Example 8.1. COPY with No REPLACING Phrase

```
             1 identification division.
             2 program-id. cust01.
             3 data division.
             4 working-storage section.
             5 copy custfile.
L            6 01  CUSTOMER-REC.
L            7     03  CUST-REC-KEY        PIC X(03) VALUE "KEY".
L            8     03  CUST-NAME   PIC X(25).
L            9     03  CUST-ADDRESS.
L           10         05  CUST-CUST-STREET        PIC X(20).
L           11         05  CUST-CITY       PIC X(20).
L           12         05  CUST-STATE      PIC XX.
L           13         05  CUST-ZIP        PIC 9(5).
L           14 * THE COMPILER IGNORES COMMENT LINES AND BLANK LINES
L           15
L           16 * FOR MATCHING PURPOSES
L           17     03  CUST-ORDERS OCCURS XYZ TIMES.
                                          1        2
%COBOL-F-SYN5  121, (1) Invalid OCCURS clause
%COBOL-W-RESTART  297, (2) Processing of source program resumes at this
 point
L           18         05  CUST-ORDER      PIC 9(6).
L           19         05  CUST-ORDER-DATE PIC 9(6).
L           20         05  CUST-ORDER-AMT  PIC 9(R)V99.
                                                1
%COBOL-F-ERROR  178, (1) Invalid repetition factor
```

Example 8.2 shows the results of replacing a word (“xyz”) by a literal (6).

Example 8.2. Replacing a Word with a Literal

```
            22 copy custfile replacing xyz by 6.
L           23 01  CUSTOMER-REC.
L           24     03  CUST-REC-KEY        PIC X(03) VALUE "KEY".
L           25     03  CUST-NAME   PIC X(25).
L           26     03  CUST-ADDRESS.
L           27         05  CUST-CUST-STREET        PIC X(20).
L           28         05  CUST-CITY       PIC X(20).
L           29         05  CUST-STATE      PIC XX.
L           30         05  CUST-ZIP        PIC 9(5).
L           31 * THE COMPILER IGNORES COMMENT LINES AND BLANK LINES
L           32
L           33 * FOR MATCHING PURPOSES
LR          34     03  CUST-ORDERS OCCURS 6   TIMES.
L           35         05  CUST-ORDER      PIC 9(6).
L           36         05  CUST-ORDER-DATE PIC 9(6).
L           37         05  CUST-ORDER-AMT  PIC 9(R)V99.
```

```
                                                1
%COBOL-F-PICREPEAT  178, (1) Invalid repetition factor
```

Example 8.3 shows the results of replacing a word (“xyz”) by a literal (6), and pseudo-text by pseudo-text. The compiler recognizes R as a text-word because parentheses enclose it. The other R characters are not text-words; they are part of other text-words.

Example 8.3. Replacing a Word by a Literal and Pseudo-Text by Pseudo-Text

```
           39 copy custfile replacing xyz by 6, ==r== by ==4==.
L          40 01  CUSTOMER-REC.
L          41     03  CUST-REC-KEY        PIC X(03) VALUE "KEY".
L          42     03  CUST-NAME   PIC X(25).
L          43     03  CUST-ADDRESS.
L          44         05  CUST-CUST-STREET        PIC X(20).
L          45         05  CUST-CITY       PIC X(20).
L          46         05  CUST-STATE      PIC XX.
L          47         05  CUST-ZIP        PIC 9(5).
L          48 * THE COMPILER IGNORES COMMENT LINES AND BLANK LINES
L          49
L          50 * FOR MATCHING PURPOSES
LR         51     03  CUST-ORDERS OCCURS 6   TIMES.
L          52         05  CUST-ORDER      PIC 9(6).
L          53         05  CUST-ORDER-DATE PIC 9(6).
LR         54         05  CUST-ORDER-AMT  PIC 9(4)V99.
```

Example 8.4 shows the results of matching a nonnumeric literal. The opening and closing quotation marks are part of the text-word.

Example 8.4. Matching a Nonnumeric Literal

```
           129 copy custfile replacing xyz by 6, ==r== by ==4==
           130    "KEY" by "abc".
L          131 01  CUSTOMER-REC.
LR         132     03  CUST-REC-KEY        PIC X(03) VALUE "abc" .
L          133     03  CUST-NAME   PIC X(25).
L          134     03  CUST-ADDRESS.
L          135         05  CUST-CUST-STREET        PIC X(20).
L          136         05  CUST-CITY       PIC X(20).
L          137         05  CUST-STATE      PIC XX.
L          138         05  CUST-ZIP        PIC 9(5).
L          139 * THE COMPILER IGNORES COMMENT LINES AND BLANK LINES
L          140
L          141 * FOR MATCHING PURPOSES
LR         142     03  CUST-ORDERS OCCURS 6   TIMES.
L          143         05  CUST-ORDER      PIC 9(6).
L          144         05  CUST-ORDER-DATE PIC 9(6).
LR         145         05  CUST-ORDER-AMT  PIC 9(4)V99.
```

Example 8.5 shows the results of a multiple-line pseudo-text replacement item. The replacement item starts after the pseudo-text delimiter on line 167 and ends before the delimiter on line 169. The continuation area on the new line (172) contains the same characters as line 168 in the pseudo-text replacement item. This example is not a recommended use of the COPY statement. It only shows the mechanics of the statement.

Example 8.5. Multiple-Line Pseudo-Text Replacement Item

```
         166  copy custfile replacing xyz by 6, ==r== by ==4==
```

```
         167      "KEY" by =="abc".
         168 * cust-number is a new field
         169      03  cust-number pic 9(8)==.
L        170  01  CUSTOMER-REC.
LR       171      03  CUST-REC-KEY        PIC X(03) VALUE "abc".
LR       172 * cust-number is a new field
LR       173      03  cust-number pic 9(8).
L        174      03  CUST-NAME   PIC X(25).
L        175      03  CUST-ADDRESS.
L        176          05  CUST-CUST-STREET        PIC X(20).
L        177          05  CUST-CITY       PIC X(20).
L        178          05  CUST-STATE      PIC XX.
L        179          05  CUST-ZIP        PIC 9(5).
L        180 * THE COMPILER IGNORES COMMENT LINES AND BLANK LINES
L        181
L        182 * FOR MATCHING PURPOSES
LR       183      03  CUST-ORDERS OCCURS 6   TIMES.
L        184          05  CUST-ORDER      PIC 9(6).
L        185          05  CUST-ORDER-DATE PIC 9(6).
LR       186          05  CUST-ORDER-AMT  PIC 9(4)V99.
```

Example 8.6 shows the results of matching pseudo-text that includes separators.

The replacement phrase in line 210 fails to match the library text in line 212. The text-matching argument contains *one* text-word: the 13 characters beginning with c and ending with a period (.). The period is not a separator period, because it is not followed by a space. This argument fails to match the *two* text-words on line 212. The two text-words are: (1) CUSTOMER-REC and (2) the separator period.

The replacement phrase in line 211 replaces library text on line 215. The text-matching argument contains the same two text-words that are in the library text: (1) CUST-ADDRESS and (2) the separator period.

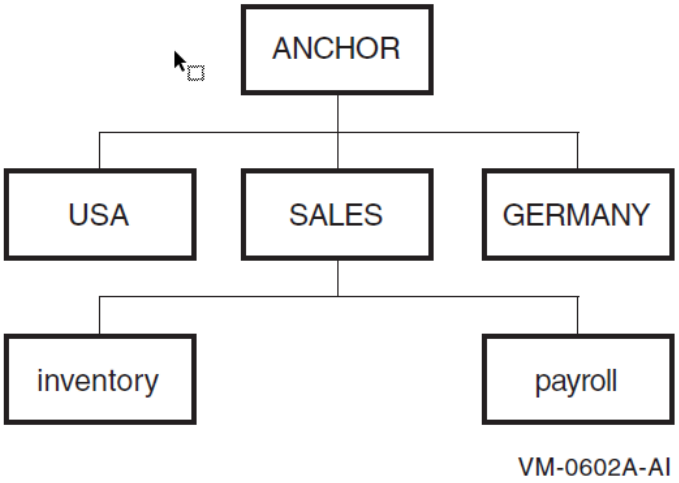
Example 8.6. Matching Pseudo-Text That Includes Separators

```
           209 copy custfile replacing xyz by 6, ==r== by ==4==
           210    ==customer-rec.== by ==record-a.==
           211    ==cust-address. == by ==customer-address.==.
L          212 01  CUSTOMER-REC.
L          213     03  CUST-REC-KEY        PIC X(03) VALUE "KEY".
L          214     03  CUST-NAME   PIC X(25).
LR         215     03  customer-address.
L          216         05  CUST-CUST-STREET        PIC X(20).
L          217         05  CUST-CITY       PIC X(20).
L          218         05  CUST-STATE      PIC XX.
L          219         05  CUST-ZIP        PIC 9(5).
L          220 * THE COMPILER IGNORES COMMENT LINES AND BLANK LINES
L          221
L          222 * FOR MATCHING PURPOSES
LR         223     03  CUST-ORDERS OCCURS 6   TIMES.
L          224         05  CUST-ORDER      PIC 9(6).
L          225         05  CUST-ORDER-DATE PIC 9(6).
LR         226         05  CUST-ORDER-AMT  PIC 9(4)V99.
           227
```

Examples Using Format 2 (OpenVMS)

Figure 8.1 represents a hierarchical repository structure for Examples 8.7, 8.8, and 8.9. It contains one repository directory and two repository objects.

Figure 8.1. Hierarchical Repository Structure (OpenVMS)



In Figure 8.1, the repository is named SALES (USA and GERMANY are not used). ANCHOR is the starting directory for the full repository pathname. Repository directories are analogous to OpenVMS Alpha and I64 subdirectories. They catalog other repository directories or repository objects, and they are labeled by the paths through the hierarchy that lead to them.

The repository objects are named PAYROLL and INVENTORY. These objects are the named record descriptions stored in Oracle CDD/Repository, and they form the end-points of the repository hierarchy branches. The examples that follow copy these record descriptions.

The full repository pathname provides a unique designation for every directory and object in Oracle CDD/Repository hierarchy. It traces the paths from ANCHOR to the directory or object.

For information on how to create and maintain a hierarchical structure in Oracle CDD/Repository, refer to the Oracle CDD/Repository documentation set.

Note

Not all Oracle CDD/Repository data types are valid VSI COBOL for OpenVMS data types. See the Technical Notes.

Example 8.7 shows how to use a command file to create the repository directories and objects shown in Figure 8.1 using CDO.

Example 8.7. Command File That Creates Oracle CDD/Repository Directories and Objects in Figure 8-1 (OpenVMS)

```
define field name
    datatype is text
    size 30.
define field address
    datatype is text
    size is 40.
define field salesman_id
    datatypes is text
    size is 5.
define record salesman.
    name.
    address.
    salesman_id.
```

```
end record.
define field ytd_sales
    datatype is right overpunched numeric
    size is 11 digits
    scale -2.
define field ytd_commission
    datatype is right overpunched numeric
    size is 11 digits
    scale -2.
define field curr_month_sales
    datatype is right overpunched numeric
    size is 11 digits
    scale -2.
define field curr_month_commission
    datatype is right overpunched numeric
    size is 11 digits
    scale -2.
define field curr_week_sales
    datatype is right overpunched numeric
    size is 11 digits
    scale -2.
define field curr_week_commission
    datatype is right overpunched numeric
    size is 11 digits
    scale -2.
define record payroll_record.
    salesman.
    ytd_sales.
    ytd_commission.
    curr_month_sales.
    curr_month_commission.
    curr_week_sales.
    curr_week_commission.
end record.
define field part_number
    datatype is right overpunched numeric
    size is 6 digits.
define field quantity_on_hand
    datatype is right overpunched numeric
    size is 9 digits.
define field quantity_on_order
    datatype is right overpunched numeric
    size is 9 digits.
define field retail_price
    datatype is right overpunched numeric
    size is 8 digits
    scale -2.
define field wholesale_price
    datatype is right overpunched numeric
    size is 8 digits
    scale -2.
define field supplier
    datatype is text
    size is 5 characters.
define record inventory_record.
    part_number.
    quantity_on_hand.
    quantity_on_order.
```

```
    retail_price.
    wholesale_price.
    supplier.
end record.
```

Example 8.8 shows the results of copying the repository object PAYROLL in Figure 8.1. The program defines the logical name payroll to be equivalent to the full Oracle CDD/Repository pathname DEVICE: [DIRECTORY.ANCHOR]. Line 27 of the program shows the DCL command used to define the logical name and line 30 contains the COPY FROM DICTIONARY statement.

On OpenVMS Alpha and I64 systems, the COPY statement produces lines 31 to 44 in your program listing if you include the /COPY_LIST compiler option. Line 32 is the resulting full Oracle CDD/Repository pathname used by the compiler. Lines 31 and 33 are separator comment lines. Lines 34 to 44 are the COBOL compiler-translated record description entries taken from the PAYROLL repository object in Oracle CDD/Repository.

Example 8.8. Using a Logical Name in a COPY Statement (OpenVMS)

```
              1 IDENTIFICATION DIVISION.
              2 PROGRAM-ID.  TEST-CDD.
              3 *
              4 *   Copy from CDD/Repository
              5 *   FILE SECTION
              6 *           Records:    PERSONNEL
              7 *                       INVENTORY
              8 *                       PAYROLL
              9 *
             10 *   WORKING-STORAGE SECTION
             11 *           Records:    SYDNEY
             12 *                       MAPLE
             13 *                       FRENCH
             14 *
             15 ENVIRONMENT DIVISION.
             16 INPUT-OUTPUT SECTION.
             17 FILE-CONTROL.
             18     SELECT SALES-CDD-FILE
             19     ASSIGN TO "CDD.TMP".
             20 DATA DIVISION.
             21 FILE SECTION.
             22 FD SALES-CDD-FILE.
             23 *
             24 *   To create a logical name entry for the repository
 object
             25 *   PAYROLL, use this command:
             26 *
             27 *   $ DEFINE PAYROLL_RECORD "DEVICE:
[DIRECTORY.ANCHOR]SALES.PAYROLL"
             28 *
             29 *
             30     COPY PAYROLL FROM DICTIONARY.
L            31 *
L            32 * _DEVICE:[DIRECTORY.ANCHOR]PAYROLL_RECORD
L            33 *
L            34 01  PAYROLL_RECORD.
L            35     02  SALESMAN.
L            36         03  NAME            PIC X(30).
L            37         03  ADDRESS         PIC X(40).
```

```
L            38         03  SALESMAN_ID     PIC X(5).
L            39     02  YTD_SALES           PIC S9(9)V9(2) SIGN TRAILING.
L            40     02  YTD_COMMISSION      PIC S9(9)V9(2) SIGN TRAILING.
L            41     02  CURR_MONTH_SALES    PIC S9(9)V9(2) SIGN TRAILING.
L            42     02  CURR_MONTH_COMMISSION PIC S9(9)V9(2) SIGN TRAILING.
L            43     02  CURR_WEEK_SALES     PIC S9(9)V9(2) SIGN TRAILING.
L            44     02  CURR_WEEK_COMMISSION PIC S9(9)V9(2) SIGN TRAILING.
             45
             46     COPY "DEVICE:[DIRECTORY.ANCHOR]INVENTORY_RECORD" FROM
 DICTIONARY.
L            47 *
L            48 * _DEVICE:[DIRECTORY.ANCHOR]INVENTORY_RECORD
L            49 *
L            50 01  INVENTORY_RECORD.
L            51     02  PART_NUMBER         PIC S9(6) SIGN TRAILING.
L            52     02  QUANTITY_ON_HAND    PIC S9(9) SIGN TRAILING.
L            53     02  QUANTITY_ON_ORDER   PIC S9(9) SIGN TRAILING.
L            54     02  RETAIL_PRICE        PIC S9(6)V9(2) SIGN TRAILING.
L            55     02  WHOLESALE_PRICE     PIC S9(6)V9(2) SIGN TRAILING.
L            56     02  SUPPLIER            PIC X(5).
             57
             58
        ...
```

Example 8.9 shows the results of copying a repository object INVENTORY by specifying its full Oracle CDD/Repository pathname.

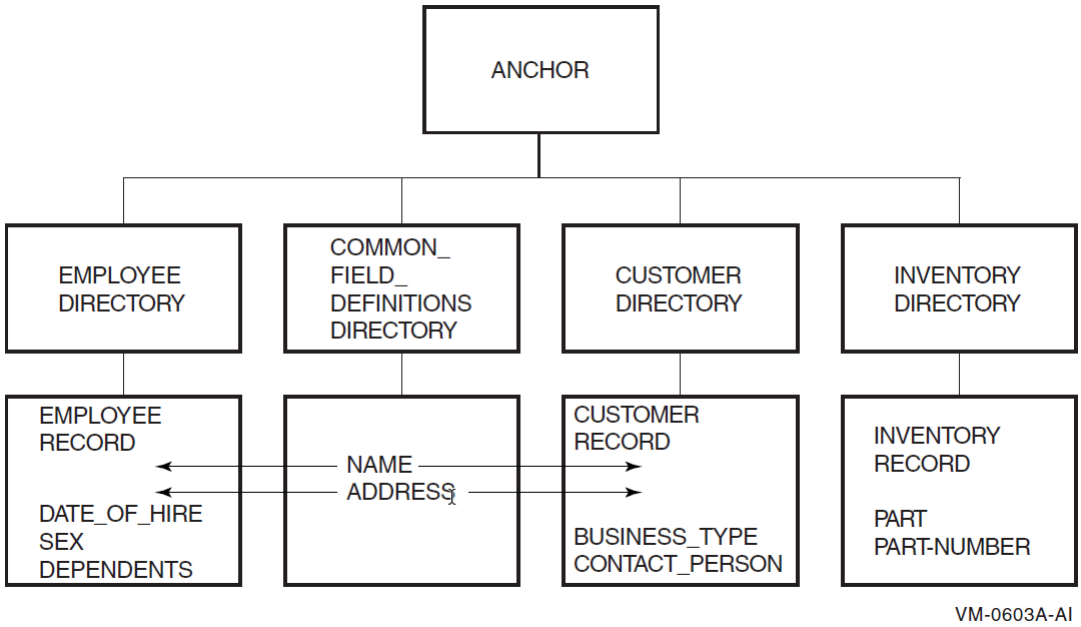
In Example 8.9, line 44 contains the COPY FROM DICTIONARY statement. On OpenVMS Alpha and I64 systems, this COPY statement produces lines 45 to 54 in your program listing if you include the /COPY_LIST compiler option. Line 46 is the resulting full Oracle CDD/Repository pathname used by the compiler. Lines 45 and 47 are separator comment lines. Lines 48 to 54 are the compiler-translated record description entries taken from the inventory repository object in Oracle CDD/Repository.

Example 8.9. Using a Full Pathname in a COPY Statement (OpenVMS)

```
       44      COPY "DEVICE:[DIRECTORY.ANCHOR]SALES.INVENTORY" FROM
 DICTIONARY.
L      45 *
L      46 * DEVICE:[DIRECTORY.ANCHOR]SALES.INVENTORY
L      47 *
L      48 01  INVENTORY_RECORD.
L      49      02  PART_NUMBER              PIC 9(6).
L      50      02  QUANTITY_ON_HAND         PIC S9(9) SIGN TRAILING.
L      51      02  QUANTITY_ON_ORDER        PIC S9(9) SIGN TRAILING.
L      52      02  RETAIL_PRICE             PIC S9(6)V9(2) SIGN TRAILING.
L      53      02  WHOLESALE_PRICE          PIC S9(6)V9(2) SIGN TRAILING.
L      54      02  SUPPLIER                 PIC X(5).
```

Figure 8.2 shows a nonhierarchical repository structure. In this example, fields NAME and ADDRESS are used by both the EMPLOYEE-RECORD and the CUSTOMER-RECORD. As such, they are defined in a separate directory (COMMON_FIELD_DEFINITIONS). The fields PART and PART_NUMBER are used exclusively by the INVENTORY_RECORD. As such, they are defined in the INVENTORY directory. This functionality is only available in CDO formatted repositories.

Figure 8.2. Nonhierarchical Repository Structure (OpenVMS)



Example 8.10 shows how to use a CDO command file to create the directories and objects shown in Figure 8.2 using CDO. The CDO file is executed from within CDO using the following command:

```
$ REPOSITORY
CDO>@FILENAME.CDO
```

Example 8.10. Command File That Creates Oracle CDD/Repository Directories and Objects in Figure 8-2 (OpenVMS)

```
DEFINE DICTIONARY DEVICE:[DIRECTORY.ANCHOR].
SET DEFAULT DEVICE:[DIRECTORY.ANCHOR]
DEFINE DIRECTORY EMPLOYEE.
DEFINE DIRECTORY CUSTOMER.
DEFINE DIRECTORY INVENTORY.
DEFINE DIRECTORY COMMON_FIELD_DEFINITIONS.
SET DEFAULT DEVICE:[DIRECTORY.ANCHOR]COMMON_FIELD_DEFINITIONS
DEFINE FIELD NAME DATATYPE IS TEXT SIZE IS 25 CHARACTERS.
DEFINE FIELD ADDRESS DATATYPE IS TEXT SIZE IS 47 CHARACTERS.
SET DEFAULT DEVICE:[DIRECTORY.ANCHOR]EMPLOYEE
DEFINE FIELD DATE_OF_HIRE DATATYPE IS UNSIGNED NUMERIC SIZE IS 8 DIGITS.
DEFINE FIELD SEX DATATYPE IS TEXT SIZE IS 1 CHARACTER.
DEFINE FIELD DEPENDENTS DATATYPE IS UNSIGNED NUMERIC SIZE IS 2 DIGITS.
DEFINE RECORD EMPLOYEE_RECORD.
[DIRECTORY.ANCHOR]COMMON_FIELD_DEFINITIONS.NAME.
[DIRECTORY.ANCHOR]COMMON_FIELD_DEFINITIONS.ADDRESS.
DATE_OF_HIRE.
SEX.
DEPENDENTS.
END RECORD.
SET DEFAULT DEVICE:[DIRECTORY.ANCHOR]CUSTOMER
DEFINE FIELD BUSINESS_TYPE DATATYPE IS TEXT SIZE IS 25 CHARACTERS.
DEFINE FIELD CONTACT_PERSON DATATYPE IS TEXT SIZE IS 25 CHARACTERS.
DEFINE RECORD CUSTOMER_RECORD.
[DIRECTORY.ANCHOR]COMMON_FIELD_DEFINITIONS.NAME.
[DIRECTORY.ANCHOR]COMMON_FIELD_DEFINITIONS.ADDRESS.
```

```
BUSINESS_TYPE.
CONTACT_PERSON.
END RECORD.
SET DEFAULT DEVICE:[DIRECTORY.ANCHOR]INVENTORY
DEFINE FIELD PART DATATYPE IS TEXT SIZE IS 25 CHARACTERS.
DEFINE FIELD PART_NUMBER DATATYPE IS TEXT SIZE IS 10 CHARACTERS.
DEFINE RECORD INVENTORY_RECORD.
PART.
PART_NUMBER.
END RECORD.
```

Additional References

- Section 1.3: Source Reference Format
- Oracle CDD/Repository documentation set

REPLACE

REPLACE — The REPLACE statement is used to replace source program text.

General Format

Format 1

REPLACE {==pseudo-text-1== BY ==pseudo-text-2==} ...

Format 2

REPLACE OFF

pseudo-text-1

is a text-matching argument that the compiler compares against text-words in the source text.

pseudo-text-2

is a replacement item that the compiler inserts into the source program.

Syntax Rules

1. A REPLACE statement can be inserted anywhere that a character-string can be used. This statement must be preceded by a separator period unless it is the first statement in a separately compiled program.
2. A REPLACE statement must be terminated by the separator period.
3. *pseudo-text-1* must contain at least one text-word.
4. *pseudo-text-2* can contain zero, one, or more text-words.
5. Character-strings within *pseudo-text-1* and *pseudo-text-2* can be continued.

6. *pseudo-text-1* must not consist entirely of a separator comma or a separator semicolon.

7. The word REPLACE is considered part of a comment-entry if it appears in the comment-entry or in the place where a comment-entry can appear.

General Rules

Format 1

1. Each matched occurrence of *pseudo-text-1* in the source program is replaced by the corresponding *pseudo-text-2*.

Format 2

2. Any text replacement currently in effect is discontinued.

Both Formats

3. A REPLACE statement remains in effect until the next occurrence of a REPLACE statement or until the end of a separately compiled program has been reached.

4. Any occurrence of a REPLACE statement in a source program is processed after all COPY statements in the source program have been processed.

5. *pseudo-text-2* must not contain a REPLACE statement.

6. The comparison operation starts with the leftmost source text word and the first text-matching argument. The compiler compares the entire text-matching argument to an equivalent number of consecutive source text-words.

7. A text-matching argument matches the source text only if the ordered sequence of text-words that forms the text-matching argument is equal, character for character, to the ordered sequence of source text-words.

   In the matching operation, the compiler treats each occurrence or combination of the following items in source text as a single space:

   - Separator comma
   - Separator semicolon
   - A sequence of one or more separator spaces

8. If no match occurs, the compiler repeats the comparison operation with each successive text-matching argument until a match is found or there are no more text-matching arguments.

9. If no match occurs after the compiler has compared all of the text-matching arguments, the next successive source text-word becomes the leftmost text-word, and the comparison resumes with the first occurrence of *pseudo-text-1*.

10. If a match occurs between a text-matching argument and the source program text, the compiler inserts the replacement text into the source program. The source text-word immediately following the rightmost replaced text-word becomes the leftmost text-word for the next cycle. The comparison cycle resumes with the first occurrence of *pseudo-text-1*.

11. The comparison cycles continue until the rightmost text-word in the source text that is within the scope of the REPLACE statement has been either:

    - Matched and replaced
    - Used as the leftmost source text-word in a comparison cycle

12. The rules for Reference Format determine the sequence of text-words in the source text and the text-matching arguments.

13. The compiler ignores comment lines and blank lines in the source program and in *pseudo-text-1* for matching.

14. When the compiler inserts *pseudo-text-2* in the source program, it inserts comment lines and blank lines in *pseudo-text-2* without modification.

15. Debugging lines are permitted in *pseudo-text-1* and *pseudo-text-2*. The compiler treats the comparison of debugging lines as if the conditional compilation character does not appear in the indicator area.

16. The compiler cannot determine the syntactic correctness of source text or the source program until all COPY and REPLACE statements have been processed.

17. Text words that are inserted as a result of a processed REPLACE statement are placed in the source program according to the rules for Reference Format.

18. When the compiler inserts text words of *pseudo-text-2* into the source program, additional spaces may be introduced between text words where spaces already exist (including the assumed space between source lines).

19. If additional lines are added to the source program as a result of a REPLACE operation, the indicator area of the added lines contains the same character as the line on which the text being replaced begins (unless that line contains a hyphen, in which case the introduced line contains a space).

    If a literal within *pseudo-text-2* cannot be contained on a single line without a continuation to another line in the resultant program and the literal is not being placed on a debugging line, additional continuation lines are introduced that contain the remainder of the literal. If replacement requires the continued literal to be continued on a debugging line, the program is in error.

Examples

In the following examples, uppercase words represent text-words that have been replaced.

1. REPLACE statement with multiple replacement items:

```
         8 working-storage section.
         9 replace ==alpha== by ==NUM-1==
        10         ==num== by ==ALPHA-1==.
R       11 01  NUM-1  pic 9(10).
R       12 01  ALPHA-1
        13            pic x(10).
        14 procedure division.
```

2. Multiple REPLACE statements:

A given occurrence of the REPLACE statement is in effect from the point at which it is specified until the next occurrence of the REPLACE statement. The new REPLACE statement supersedes the text-matching established by the previous REPLACE statement.

```
         7 working-storage section.
         8 01  total           pic 9(4)v99.
         9 replace ==class== by ==CLASS1==
        10         ==total== by ==ORDER-AMT==.
        11 01  customer-rec.
 R      12     03  CLASS1      pic x(02).
        13     03  name        pic x(25).
        14     03  address.
        15         05  street  pic x(20).
        16         05  city    pic x(20).
        17         05  state   pic xx.
        18         05  zip     pic 9(5).
        19     03  orders occurs 6 times.
        20         05  order-numb  pic 9(6).
        21         05  order-date  pic 9(6).
 R      22         05  ORDER-AMT   pic 9(4)v99.
        23 procedure division.
        24 replace ==class== by ==CLASS1==.
        25 p0.  add order-amt of orders(3) to total.
```

In the previous example, the word total on line 25 is not replaced because the REPLACE statement on line 24 reestablished the text-matching arguments.

3. REPLACE OFF:

   Any text-matching currently in effect is turned off.

```
       11 working-storage section.
       12 replace ==add== by ==PIC 9(18)==.
 R     13 01  a1              PIC 9(18).
 R     14 01  a2              PIC 9(18).
       15 procedure division.
       16     replace off.
       17 p0. add a1 to a2.
```

   In the previous example, the word add on line 17 is not replaced because the REPLACE statement on line 16 turned off all text-matching arguments.

4. COPY interaction:

   In the following example, library text is copied from the library file DATAFILE.LIB:

```
Contents of "DATAFILE.LIB":
01      customer-rec.
        03  class       pic x(02).
        03  name        pic x(25).
        03  address.
            05  street  pic x(20).
            05  city    pic x(20).
            05  state   pic xx.
            05  zip     pic 9(5).
        03  orders occurs 6 times.
            05  order-number    pic 9(6).
```

```
            05  order-date      pic 9(6).
            05  order-amt       pic 9(4)v99.
```

The text-matching specified by an active REPLACE statement occurs after COPY (and COPY REPLACING) processing is complete.

```
         7 working-storage section.
         8 replace ==class== by ==CLASS1==.
         9 copy datafile.
L       10 01  customer-rec.
L       11     03  CLASS1      pic x(02).
L       12     03  name        pic x(25).
L       13     03  address.
L       14         05  street  pic x(20).
L       15         05  city    pic x(20).
L       16         05  state   pic xx.
L       17         05  zip     pic 9(5).
L       18     03  orders occurs 6 times.
L       19         05  order-number    pic 9(6).
L       20         05  order-date      pic 9(6).
L       21         05  order-amt       pic 9(4)v99.
        22 procedure division.
```

Additional Reference

See Section 1.3: Source Reference Format.

Appendix A. VSI COBOL for OpenVMS Reserved Words

VSI COBOL Reserved Words			
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	AUTO	AUTOMATIC
AUTOTERMINATE			
BACKGROUND-COLOR	BATCH	BEFORE	BEGINNING
BELL	BINARY	BIT	BITS
BLANK	BLINK	BLINKING	BLOCK
bold	BOOLEAN	BOTTOM	BY
CALL	CANCEL	CD	CF
CH	CHARACTER	CHARACTERS	CLASS
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	COMP-X	COMPUTATIONAL
COMPUTATIONAL-1	COMPUTATIONAL-2	COMPUTATIONAL-3	COMPUTATIONAL-4
COMPUTATIONAL-5	COMPUTATIONAL-6	COMPUTATIONAL-X	COMPUTE
CONCURRENT	CONFIGURATION	CONNECT	CONTAIN
CONTAINS	CONTENT	CONTINUE	CONTROL
CONTROLS	CONVERSION	CONVERTING	COPY
CORR	CORRESPONDING	COUNT	CRT
CURRENCY	CURRENT	CURSOR	
DATA	DATE	DATE-COMPILED	DATE-WRITTEN
DAY	DAY-OF-WEEK	DB	DB-ACCESS-CONTROL-KEY

VSI COBOL Reserved Words			
DB-CONDITION	DB-CURRENT-RECORD-ID	DB-CURRENT-RECORD-NAME	DB-EXCEPTION
DBKEY	DB-KEY	DB-RECORD-NAME	DB-SET-NAME
DB-STATUS	DB-UWA	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	DEPENDENCY	DEPENDING	DESCENDING
DESCRIPTOR	DESTINATION	DETAIL	DICTIONARY
DISABLE	DISCONNECT	DISPLAY	DISPLAY-6
DISPLAY-7	DISPLAY-9	DIVIDE	DIVISION
DOES	DOWN	DUPLICATE(S)	DYNAMIC
ECHO	EDITING	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	EOL
EOP	EOS	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	FOREGROUND-COLOR	FREE	FROM
FULL	FUNCTION		

VSI COBOL Reserved Words			
GENERATE	GET	GIVING	GLOBAL
GO	GREATER	GROUP	
HEADING	HIGHLIGHT	HIGH-VALUE(S)	
IDENT	IDENTIFICATION	IF	IN
INCLUDING	INDEX	INDEXED	INDICATE
INITIAL	INITIALIZE	INITIATE	INPUT
INPUT-OUTPUT	INSPECT	INSTALLATION	INTO
INVALID	I-O	I-O-CONTROL	IS
JUST	JUSTIFIED		
KEEP	KEY		
LABEL	LAST	LD	LEADING
LEFT	LENGTH	LESS	LIMIT
LIMITS	LINAGE	LINAGE-COUNTER	LINE
LINE-COUNTER	LINES	LINKAGE	LOCALLY
LOCK	LOCK-HOLDING	LOWLIGHT	LOW-VALUE(S)
MANUAL	MATCH	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		
PACKED-DECIMAL	PADDING	PAGE	PAGE-COUNTER

VSI COBOL Reserved Words			
PERFORM	PF	PH	PIC
PICTURE	PLUS	POINTER	POSITION
POSITIVE	PRINTING	PRIOR	PROCEDURE
PROCEDURES	PROCEED	PROGRAM	PROGRAM-ID
PROTECTED	PURGE	PREVIOUS	
QUEUE	QUOTE(S)		
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
REQUIRED	RERUN	RESERVE	RESET
RETAINING	RETRIEVAL	RETURN	RETURN-CODE
REVERSED	REVERSE-VIDEO	REWIND	REWRITE
RF	RH	RIGHT	RMS-CURRENT-FILENAME
RMS-CURRENT-STS	RMS-CURRENT-STV	RMS-FILENAME	RMS-STS
RMS-STV	ROLLBACK	ROUNDED	RUN
SAME	SCREEN	SD	SEARCH
SECTION	SECURE	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	STREAM	
TABLE	TALLYING	TAPE	TENANT

VSI COBOL Reserved Words			
TERMINAL	TERMINATE	TEST	TEXT
THAN	THEN	THROUGH	THRU
TIME	TIMES	TO	TOP
TRAILING	TRUE	TYPE	
UNDERLINE	UNDERLINED	UNEQUAL	UNIT
UNLOCK	UNSTRING	UNTIL	UP
UPDATE	UPDATERS	UPON	USAGE
USAGE-MODE	USE	USING	
VALUE	VALUES	VARYING	VFU-CHANNEL
WAIT	WHEN	WHERE	WITH
WITHIN	WORDS	WORKING-STORAGE	WRITE
WRITERS			
ZERO	ZEROES	ZEROS	
+	-	*	/
**	>	<	=
>=	<=		

The reserved words listed in this appendix are both the default reserved words and the words that on Alpha and I64 systems are reserved only if activated by the COBOL command-line qualifier /RESERVED_WORDS=FOREIGN_EXTENSIONS or /RESERVED_WORDS=200X.

The XOPEN reserved words, which on Alpha and I64 systems are reserved by default, can be deactivated by the /RESERVED_WORDS=NOXOPEN qualifier.

These three categories of Alpha- and I64-only reserved words, which are activated or deactivated by command-line qualifiers, are marked in this appendix as follows:

[FOREIGN]	Reserved only if activated by /RESERVED_WORDS=FOREIGN_EXTENSIONS
[200X]	Reserved only if activated by /RESERVED_WORDS=200X
[XOPEN]	Reserved by default, but not reserved if deactivated by /RESERVED_WORDS=NOXOPEN

Reserved Words

ACCEPT
ACCESS
ADD
ADDRESS **[FOREIGN]** (Alpha, I64)
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
AUTO **[XOPEN]** (Alpha, I64)
AUTOMATIC
AUTOTERMINATE
BACKGROUND-COLOR **[XOPEN]** (Alpha, I64)
BATCH
BEFORE
BEGINNING
BELL **[XOPEN]** (Alpha, I64)
BINARY
BINARY-CHAR **[200X]** (Alpha, I64)
BINARY-DOUBLE **[200X]** (Alpha, I64)
BINARY-LONG **[200X]** (Alpha, I64)
BINARY-SHORT **[200X]** (Alpha, I64)
BIT
BITS
BLANK
BLINK **[XOPEN]** (Alpha, I64)
BLINKING
BLOCK
bold
BOOLEAN
BOTTOM
BY
CALL
CANCEL
CD
CF
CH
CHANGED **[FOREIGN]** (Alpha, I64)
CHARACTER
CHARACTERS

CLASS
CLOCK-UNITS
CLOSE
COBOL
CODE
CODE-SET
COL **[200X]** (Alpha, I64)
COLLATING
COLUMN
COMMA
COMMIT
COMMON
COMMUNICATION
COMP
COMP-1
COMP-2
COMP-3
COMP-4
COMP-5
COMP-6
COMP-X
COMPUTATIONAL
COMPUTATIONAL-1
COMPUTATIONAL-2
COMPUTATIONAL-3
COMPUTATIONAL-4
COMPUTATIONAL-5
COMPUTATIONAL-6
COMPUTATIONAL-X
COMPUTE
CONCURRENT
CONFIGURATION
CONNECT
CONTAIN
CONTAINS
CONTENT
CONTINUE
CONTROL
CONTROLS
CONVERSION
CONVERTING
COPY
CORE-INDEX **[FOREIGN]** (Alpha, I64)
CORR
CORRESPONDING
COUNT
CRT
CURRENCY
CURRENT
CURSOR
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-KEY
DB-RECORD-NAME
DB-SET-NAME
DB-STATUS
DB-UWA
DBCS **[FOREIGN]** (Alpha, I64)
DBKEY
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
DEPENDENCY
DEPENDING
DESCENDING
DESCRIPTOR
DESTINATION
DETAIL
DICTIONARY
DISABLE
DISCONNECT
DISP **[FOREIGN]** (Alpha, I64)
DISPLAY
DISPLAY-1 **[FOREIGN]** (Alpha, I64)

DISPLAY-6
DISPLAY-7
DISPLAY-9
DIVIDE
DIVISION
DOES
DOWN
DUPLICATE
DUPLICATES
ECHO
EDITING
EGI
EJECT **[FOREIGN]** (Alpha, I64)
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
ENTRY **[FOREIGN]** (Alpha, I64)
ENVIRONMENT
EOL **[XOPEN]** (Alpha, I64)
EOP
EOS **[XOPEN]** (Alpha, I64)
EQUAL
EQUALS
ERASE **[XOPEN]** (Alpha, I64)
ERROR
ESI
EVALUATE
EVERY
EXAMINE **[FOREIGN]** (Alpha, I64)
EXCEEDS
EXCEPTION
EXCLUSIVE
EXHIBIT **[FOREIGN]** (Alpha, I64)
EXIT
EXOR
EXTEND
EXTERNAL
FAILURE
FALSE
FD
FETCH
FILE
FILE-CONTROL
FILLER
FINAL
FIND
FINISH
FIRST
FLOAT-EXTENDED **[200X]** (Alpha, I64)
FLOAT-LONG **[200X]** (Alpha, I64)
FLOAT-SHORT **[200X]** (Alpha, I64)
FOOTING
FOR
FOREGROUND-COLOR **[XOPEN]** (Alpha, I64)
FREE
FROM
FULL **[XOPEN]** (Alpha, I64)
FUNCTION
GENERATE
GET
GIVING
GLOBAL
GO
GOBACK **[FOREIGN]** (Alpha, I64)

GREATER
GROUP
HEADING
HIGH-VALUE
HIGH-VALUES
HIGHLIGHT **[XOPEN]** (Alpha, I64)
I-O
I-O-CONTROL
ID **[FOREIGN]** (Alpha, I64)
IDENT
IDENTIFICATION
IF
IN
INCLUDING
INDEX
INDEXED
INDICATE
INITIAL
INITIALIZE
INITIATE
INPUT
INPUT-OUTPUT
INSPECT
INSTALLATION
INTO
INVALID
IS
JUST
JUSTIFIED
KANJI **[FOREIGN]** (Alpha, I64)
KEEP
KEY
LABEL
LAST
LD
LEADING
LEFT
LENGTH
LESS
LIMIT
LIMITS
LINAGE
LINAGE-COUNTER
LINE
LINE-COUNTER
LINES
LINKAGE
LOCALLY
LOCK
LOCK-HOLDING
LOW-VALUE
LOW-VALUES

LOWLIGHT **[XOPEN]** (Alpha, I64)
MANUAL
MATCH
MATCHES
MEMBER
MEMBERSHIP
MEMORY
MERGE
MESSAGE
MODE
MODIFY
MODULES
MOVE
MULTIPLE
MULTIPLY
NAMED **[FOREIGN]** (Alpha, I64)
NATIVE
NEGATIVE
NEXT
NO
NON-NULL
NOT
NOTE **[FOREIGN]** (Alpha, I64)
NULL
NUMBER
NUMERIC
NUMERIC-EDITED
OBJECT-COMPUTER
OCCURS
OF
OFF
OFFSET
OMITTED
ON
ONLY
OPEN
OPTIONAL
OPTIONS **[200X]** (Alpha, I64)
OR
ORDER
OTHERWISE **[FOREIGN]** (Alpha, I64)
PACKED-DECIMAL
PADDING
PAGE
PAGE-COUNTER
PASSWORD **[FOREIGN]** (Alpha, I64)
PERFORM
PF
PH
PIC
PICTURE
PLUS

POINTER
POSITION
POSITIONING **[FOREIGN]** (Alpha, I64)
POSITIVE
PREVIOUS
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
RECORD-OVERFLOW **[FOREIGN]** (Alpha, I64)
RECORDING **[FOREIGN]** (Alpha, I64)
RECORDS
REDEFINES
REEL
REFERENCE
REFERENCE-MODIFIER
REFERENCES
REGARDLESS
RELATIVE
RELEASE
RELOAD **[FOREIGN]** (Alpha, I64)
REMAINDER
REMARKS **[FOREIGN]** (Alpha, I64)
REMOVAL
RENAMES
REORG-CRITERIA **[FOREIGN]** (Alpha, I64)
REPLACE
REPLACING
REPORT
REPORTING
REPORTS
REQUIRED **[XOPEN]** (Alpha, I64)
RERUN

RESERVE
RESET
RETAINING
RETRIEVAL
RETURN
RETURN-CODE **[XOPEN]** (Alpha, I64)
RETURNING **[FOREIGN]** (Alpha, I64)
REVERSE-VIDEO **[XOPEN]** (Alpha, I64)
REVERSED
REWIND
REWRITE
RF
RH
RIGHT
RMS-CURRENT-FILENAME
RMS-CURRENT-STS
RMS-CURRENT-STV
RMS-FILENAME
RMS-STS
RMS-STV
ROLLBACK
ROUNDED
RUN
SAME
SCREEN **[XOPEN]** (Alpha, I64)
SD
SEARCH
SECTION
SECURE **[XOPEN]** (Alpha, I64)
SECURITY
SEGMENT
SEGMENT-LIMIT
SELECT
SEND
SENTENCE
SEPARATE
SEQUENCE
SEQUENCE-NUMBER
SEQUENTIAL
SERVICE **[FOREIGN]** (Alpha, I64)
SET
SETS
SIGN
SIGNED **[200X]** (Alpha, I64)
SIZE
SKIP1 **[FOREIGN]** (Alpha, I64)
SKIP2 **[FOREIGN]** (Alpha, I64)
SKIP3 **[FOREIGN]** (Alpha, I64)
SORT
SORT-MERGE
SOURCE
SOURCE-COMPUTER

SPACE
SPACES
SPECIAL-NAMES
STANDARD
STANDARD-1
STANDARD-2
START
STATUS
STOP
STORE
STREAM
STRING
SUB-QUEUE-1
SUB-QUEUE-2
SUB-QUEUE-3
SUB-SCHEMA
SUBTRACT
SUCCESS
SUM
SUPPRESS
SYMBOL **[200X]** (Alpha, I64)
SYMBOLIC
SYNC
SYNCHRONIZED
TABLE
TALLYING
TAPE
TENANT
TERMINAL
TERMINATE
TEST
TEXT
THAN
THEN
THROUGH
THRU
TIME
TIMES
TO
TOP
TRACE **[FOREIGN]** (Alpha, I64)
TRAILING
TRANSFORM **[FOREIGN]** (Alpha, I64)
TRUE
TYPE
UNDERLINE **[XOPEN]** (Alpha, I64)
UNDERLINED
UNEQUAL
UNIT
UNLOCK
UNSIGNED **[200X]** (Alpha, I64)
UNSTRING

UNTIL
UP
UPDATE
UPDATERS
UPON
USAGE
USAGE-MODE
USE
USING
VALUE
VALUES
VARYING
VFU-CHANNEL
WAIT
WHEN
WHERE
WITH
WITHIN
WORDS
WORKING-STORAGE
WRITE
WRITERS
ZERO
ZEROES
ZEROS
+
-
*
/
**
>
<=
>=
<=

Appendix B. Character Sets

		ASCII		EBCDIC		NATIVE	
Position	**Character**	**Dec**	**Hex**	**Dec**	**Hex**	**Dec**	**Hex**
001	NUL	000	00	000	00	000	00
002	SOH	001	01	001	01	001	01
003	STX	002	02	002	02	002	02
004	ETX	003	03	003	03	003	03
005	EOT	004	04	055	37	004	04
006	ENQ	005	05	045	2D	005	05
007	ACK	006	06	046	2E	006	06
008	BEL	007	07	047	2F	007	07
009	BS	008	08	022	16	008	08
010	HT	009	09	005	05	009	09
011	LF	010	0A	037	25	010	0A
012	VT	011	0B	011	0B	011	0B
013	FF	012	0C	012	0C	012	0C
014	CR	013	0D	013	0D	013	0D
015	SO	014	0E	014	0E	014	0E
016	SI	015	0F	015	0F	015	0F
017	DLE	016	10	016	10	016	10
018	DC1	017	11	017	11	017	11
019	DC2	018	12	018	12	018	12
020	DC3	019	13	019	13	019	13
021	DC4	020	14	060	3C	020	14
022	NAK	021	15	061	3D	021	15
023	SYN	022	16	050	32	022	16
024	ETB	023	17	038	26	023	17
025	CAN	024	18	024	18	024	18
026	EM	025	19	025	19	025	19
027	SUB	026	1A	063	3F	026	1A
028	ESC	027	1B	039	27	027	1B
029	FS	028	1C	028	1C	028	1C
030	GS	029	1D	029	1D	029	1D
031	RS	030	1E	030	1E	030	1E
032	US	031	1F	031	1F	031	1F

		ASCII		EBCDIC		NATIVE	
Position	**Character**	**Dec**	**Hex**	**Dec**	**Hex**	**Dec**	**Hex**
033	space	032	20	064	40	032	20
034	!	033	21	090	5A	033	21
035	"	034	22	127	7F	034	22
036	#	035	23	123	7B	035	23
037	$	036	24	091	5B	036	24
038	%	037	25	108	6C	037	25
039	&	038	26	080	50	038	26
040	'	039	27	125	7D	039	27
041	(	040	28	077	4D	040	28
042	)	041	29	093	5D	041	29
043	*	042	2A	092	5C	042	2A
044	+	043	2B	078	4E	043	2B
045	,	044	2C	107	6B	044	2C
046	-	045	2D	096	60	045	2D
047	.	046	2E	075	4B	046	2E
048	/	047	2F	097	61	047	2F
049	0	048	30	240	F0	048	30
050	1	049	31	241	F1	049	31
051	2	050	32	242	F2	050	32
052	3	051	33	243	F3	051	33
053	4	052	34	244	F4	052	34
054	5	053	35	245	F5	053	35
055	6	054	36	246	F6	054	36
056	7	055	37	247	F7	055	37
057	8	056	38	248	F8	056	38
058	9	057	39	249	F9	057	39
059	:	058	3A	122	7A	058	3A
060	;	059	3B	094	5E	059	3B
061	<	060	3C	076	4C	060	3C
062	=	061	3D	126	7E	061	3D
063	>	062	3E	110	6E	062	3E
064	?	063	3F	111	6F	063	3F

		ASCII		EBCDIC		NATIVE	
Position	**Character**	**Dec**	**Hex**	**Dec**	**Hex**	**Dec**	**Hex**
065	@	064	40	124	7C	064	40
066	A	065	41	193	C1	065	41
067	B	066	42	194	C2	066	42
068	C	067	43	195	C3	067	43
069	D	068	44	196	C4	068	44
070	E	069	45	197	C5	069	45
071	F	070	46	198	C6	070	46
072	G	071	47	199	C7	071	47
073	H	072	48	200	C8	072	48
074	I	073	49	201	C9	073	49
075	J	074	4A	209	D1	074	4A
076	K	075	4B	210	D2	075	4B
077	L	076	4C	211	D3	076	4C
078	M	077	4D	212	D4	077	4D
079	N	078	4E	213	D5	078	4E
080	O	079	4F	214	D6	079	4F
081	P	080	50	215	D7	080	50
082	Q	081	51	216	D8	081	51
083	R	082	52	217	D9	082	52
084	S	083	53	226	E2	083	53
085	T	084	54	227	E3	084	54
086	U	085	55	228	E4	085	55
087	V	086	56	229	E5	086	56
088	W	087	57	230	E6	087	57
089	X	088	58	231	E7	088	58
090	Y	089	59	232	E8	089	59
091	Z	090	5A	233	E9	090	5A
092	[	091	5B			091	5B
093	\	092	5C	224	E0	092	5C
094	]	093	5D			093	5D
095	^	094	5E	095	5F	094	5E
096	_	095	5F	109	6D	095	5F
097		096	60	121	79	096	60

		ASCII		EBCDIC		NATIVE	
Position	**Character**	**Dec**	**Hex**	**Dec**	**Hex**	**Dec**	**Hex**
098	a	097	61	129	81	097	61
099	b	098	62	130	82	098	62
100	c	099	63	131	83	099	63
101	d	100	64	132	84	100	64
102	e	101	65	133	85	101	65
103	f	102	66	134	86	102	66
104	g	103	67	135	87	103	67
105	h	104	68	136	88	104	68
106	i	105	69	137	89	105	69
107	j	106	6A	145	91	106	6A
108	k	107	6B	146	92	107	6B
109	l	108	6C	147	93	108	6C
110	m	109	6D	148	94	109	6D
111	n	110	6E	149	95	110	6E
112	o	111	6F	150	96	111	6F
113	p	112	70	151	97	112	70
114	q	113	71	152	98	113	71
115	r	114	72	153	99	114	72
116	s	115	73	162	A2	115	73
117	t	116	74	163	A3	116	74
118	u	117	75	164	A4	117	75
119	v	118	76	165	A5	118	76
120	w	119	77	166	A6	119	77
121	x	120	78	167	A7	120	78
122	y	121	79	168	A8	121	79
123	z	122	7A	169	A9	122	7A
124	{	123	7B	192	C0	123	7B
125	|	124	7C	106	6A	124	7C
126	}	125	7D	208	D0	125	7D
127	~	126	7E	161	A1	126	7E
128	DEL	127	7F	007	07	127	7F
129						128	80
130						129	81

		ASCII		EBCDIC		NATIVE	
Position	**Character**	**Dec**	**Hex**	**Dec**	**Hex**	**Dec**	**Hex**
131						130	82
132						131	83
133						132	84
134						133	85
135						134	86
136						135	87
137						136	88
138						137	89
139						138	8A
140						139	8B
141						140	8C
142						141	8D
143						142	8E
144						143	8F
145						144	90
146						145	91
147						146	92
148						147	93
149						148	94
150						149	95
151						150	96
152						151	97
153						152	98
154						153	99
155						154	9A
156						155	9B
157						156	9C
158						157	9D
159						158	9E
160						159	9F
161						160	A0
162						161	A1
163						162	A2

		ASCII		EBCDIC		NATIVE	
Position	**Character**	**Dec**	**Hex**	**Dec**	**Hex**	**Dec**	**Hex**
164						163	A3
165						164	A4
166						165	A5
167						166	A6
168						167	A7
169						168	A8
170						169	A9
171						170	AA
172						171	AB
173						172	AC
174						173	AD
175						174	AE
176						175	AF
177						176	B0
178						177	B1
179						178	B2
180						179	B3
181						180	B4
182						181	B5
183						182	B6
184						183	B7
185						184	B8
186						185	B9
187						186	BA
188						187	BB
189						188	BC
190						189	BD
191						190	BE
192						191	BF
193						192	C0
194						193	C1
195						194	C2
196						195	C3

		ASCII		EBCDIC		NATIVE	
Position	**Character**	**Dec**	**Hex**	**Dec**	**Hex**	**Dec**	**Hex**
197						196	C4
198						197	C5
199						198	C6
200						199	C7
201						200	C8
202						201	C9
203						202	CA
204						203	CB
205						204	CC
206						205	CD
207						206	CE
208						207	CF
209						208	D0
210						209	D1
211						210	D2
212						211	D3
213						212	D4
214						213	D5
215						214	D6
216						215	D7
217						216	D8
218						217	D9
219						218	DA
220						219	DB
221						220	DC
222						221	DD
223						222	DE
224						223	DF
225						224	E0
226						225	E1
227						226	E2
228						227	E3
229						228	E4

		ASCII		EBCDIC		NATIVE	
Position	**Character**	**Dec**	**Hex**	**Dec**	**Hex**	**Dec**	**Hex**
230						229	E5
231						230	E6
232						231	E7
233						232	E8
234						233	E9
235						234	EA
236						235	EB
237						236	EC
238						237	ED
239						238	EE
240						239	EF
241						240	F0
242						241	F1
243						242	F2
244						243	F3
245						244	F4
246						245	F5
247						246	F6
248						247	F7
249						248	F8
250						249	F9
251						250	FA
252						251	FB
253						252	FC
254						253	FD
255						254	FE
256						255	FF

Appendix C. File Status 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.

You may receive different file status values depending upon whether you use the `standard` compiler option with the `v3` or `85` setting. Table C.1 lists all file status values in numeric order for the default `85` setting. Table C.2 lists the corresponding file status values for the `v3` and `85` settings.

For more information about the `standard` compiler option, on a UNIX system, refer to the COBOL man page. On an OpenVMS system, invoke the online help for COBOL.

Table C.1. I-O File Status Values for the Default -std 85 Flag or /STANDARD=85 Qualifier Option

File Status	Input/Output Statements	File Organization	Access Mode	Meaning
00	All	All	All	Successful
02	REWRITE WRITE	Ind	All	Created duplicate alternate key
02	READ	Ind	All	Detected alternate duplicate key
04	READ	All	All	Record not size of user's buffer
05	OPEN	All	All	Optional file not present
07	CLOSE OPEN	All	All	Invalid file organization or device
10	READ	All	Seq	No next logical record or option file not present (at end)
14	READ	Rel	All	Relative record number too large
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	Ran	Duplicate key (invalid key)
23	DELETE READ REWRITE START	Ind, Rel	Ran	Record not in file; optional file not present (invalid key)
24	WRITE	Ind, Rel	All	Boundary violation or relative record number too large (invalid key)
30	All	All	All	All other permanent errors
34	WRITE	Seq	Seq	Boundary violation
35	OPEN	All	All	File not found

File Status	Input/Output Statements	File Organization	Access Mode	Meaning
37	OPEN	All	All	Inappropriate device type
38	OPEN	All	All	File previously closed with lock
39	OPEN	All	All	Conflict of file attributes
41	OPEN	All	All	File already opened
42	CLOSE	All	All	File not opened
43	DELETE REWRITE	All	Seq	No previous READ or START
44	REWRITE WRITE	All	All	Invalid record size
46	READ	All	Seq	No valid next record (at end)
47	READ START	All	All	File not open, or incompatible open mode
48	WRITE	All	All	File not open, or incompatible open mode
49	DELETE REWRITE	All	All	File not open, or incompatible open mode
90	All	All	All	Record locked by another user (record available)
91	OPEN	All	All	Open is unsuccessful; file locked by another access stream
92	DELETE READ REWRITE START WRITE	All	All	Record locked by another user (record not available)
93	UNLOCK	All	All	No current record
94	UNLOCK	All	All	File not open, or incompatible open mode
95	OPEN	All	All	No file space on device

Table C.2. I-O File Status Values for the V3 and 85 Options

I-O Error Condition	Status Value	
	V3	**85**
READ successful – record shorter than fixed file attribute.	00	04
CLOSE reel/unit attempted on nonreel/unit device.	00	07
READ fails – relative key digits exceed relative key.	00	14

I-O Error Condition	Status Value	
	V3	**85**
WRITE fails – relative key digits exceed relative key.	00	24
OPEN I-O on file that is not mass storage.	00	37
WRITE fails – attempt to write a record of a different size than in the file description.	00	44
READ fails – no next logical record (EOF detected).	13	10
READ fails – no next logical record (EOF on OPTIONAL file).	15	10
READ fails – no valid next record (already at EOF).	16	10
READ NEXT or sequential READ – no valid next record pointer.	16 [1]	46 [1]
READ or START fails – optional input file not present.	25	23
READ successful – record longer than fixed file attribute.	30	04
OPEN on relative or indexed file that is not mass storage.	30	37
REWRITE fails – attempt to rewrite record of different size.	30	44
CLOSE fails – file not currently open.	94	42
DELETE or REWRITE fails – previous I-O not successful READ.	93	43
OPEN fails – file previously closed with LOCK.	94	38
OPEN fails – file created with different organization.	94	39
OPEN fails – file created with different prime record key.	94	39
OPEN fails – file created with different alternate record keys.	94	39
OPEN fails – file currently open.	94	41
READ or START fails – file not opened INPUT or I-O.	94	47
WRITE fails – file not opened OUTPUT, EXTEND, or I-O.	94	48
DELETE or REWRITE fails – file not opened I-O.	94	49
OPEN INPUT on a nonoptional file – file not found.	97	35

[1] Refer to the description of the /STANDARD qualifier in the COBOL online help file, or the *VSI COBOL User Manual* [https://docs.vmssoftware.com/vsi-cobol-for-openvms-user-guide/] for the description of the *-std* flag, for information about the No Valid Next Record Condition.

Appendix D. Report Writer Presentation Rules and Tables

The tables and rules in this appendix specify the following:

- The permissible combinations of LINE NUMBER and NEXT GROUP clauses for each type of report group
- The requirements for the use of these clauses
- The interpretation that the Report Writer Control System (RWCS) gives to these clauses

D.1. Organization

There is an individual presentation rules table for each of the following types of report groups: REPORT HEADING, PAGE HEADING, PAGE FOOTING, and REPORT FOOTING. In addition, DETAIL report groups, CONTROL HEADING report groups, and CONTROL FOOTING report groups are treated jointly in the Body Group Presentation Rules Table.

Columns 1 and 2 of a presentation rules table list all of the permissible combinations of LINE NUMBER and NEXT GROUP clauses for the designated report group type. Consequently, to identify the set of presentation rules that applies to a particular combination of LINE NUMBER and NEXT GROUP clauses, read a presentation rules table from left to right along the selected row.

The applicable rules columns of a presentation rules table are divided into two parts. The first part specifies the rules that apply if the report description contains a PAGE clause, and the second part specifies the rules that apply if the PAGE clause is omitted. The explanation of the rules named in the applicable rules columns follows:

- Upper-limit rules and lower-limit rules:

  These rules specify the vertical subdivisions of the page within which the RWCS may present the specified report group when the PAGE clause is used.

- Fit test rules:

  The fit test rules are applicable only to body groups when the PAGE clause is included in the Report Description entry. Therefore, fit test rules are specified only within the Body Group Presentation Rules Table. The RWCS applies the fit test rules to determine whether the designated body group can be presented on the page on which the report is currently positioned.

- First print line position rules:

  The first print line position rules specify where on the page the RWCS presents the first print line of the given report group.

  The presentation rules tables do not specify where on the page the RWCS presents the second and subsequent print lines (if any) of a report group; this is determined by the general rules of the LINE NUMBER clause.

- Next group rules:

  The next group rules relate to the proper use of the NEXT GROUP clause.

- Final LINE-COUNTER setting rules:

  These rules specify the values that the RWCS places in LINE-COUNTER after presenting report groups.

D.2. LINE NUMBER Clause Notation

Column 1 of the presentation rules table uses a shorthand notation to describe the sequence of LINE NUMBER clauses that may appear in the description of a report group. The meaning of the abbreviations used in column 1 is as follows:

1. The letter A represents one or more absolute LINE NUMBER clauses that appear in consecutive order within the sequence of LINE NUMBER clauses in the Report Group Description entry. None of the absolute LINE NUMBER clauses may have a NEXT PAGE phrase.

2. The letter R represents one or more relative LINE NUMBER clauses that appear in consecutive order within the sequence of LINE NUMBER clauses in the Report Group Description entry.

3. The letters NP represent one or more absolute LINE NUMBER clauses that appear in consecutive order within the sequence of LINE NUMBER clauses within the NEXT PAGE phrase appearing in the first, and only the first, LINE NUMBER clause.

4. When two abbreviations appear together, they refer to a sequence of LINE NUMBER clauses that consist of the two specified consecutive sequences. For example, A R refers to a Report Group Description entry within which the A sequence (defined in rule 1) is immediately followed by the R sequence (defined in rule 2).

5. A blank entry indicates that the clause is absent from the Report Group Description entry.

D.3. LINE NUMBER Clause Sequence Substitutions

Where A R is a permissible sequence in the presentation rules table, A is also permissible, and the same presentation rules apply.

Where NP R is a permissible sequence in the presentation rules table, NP is also permissible, and the same presentation rules apply.

D.4. Saved-Next-Group-Integer Description

Saved-next-group-integer is a data item that is addressable only by the RWCS. When an absolute NEXT GROUP clause specifies a vertical positioning value that cannot be accommodated on the current page, the RWCS stores that value in saved-next-group-integer. After page-advance processing, the RWCS positions the next body group using the value stored in saved-next-group-integer.

D.5. REPORT HEADING Group Presentation Rules

Figure D.1 points to the appropriate presentation rules for all permissible combinations of LINE NUMBER and NEXT GROUP clauses in a REPORT HEADING report group.

Figure D.1. REPORT HEADING Group Presentation Rules

**		*** Applicable Rules						
		If the PAGE Clause Is Specified					If the PAGE Clause Is Omitted	
*Sequence of LINE NUMBER Clauses	NEXT GROUP Clause	Upper Limit	Lower Limit	First Print Line Position	Next Group	Final LINE-COUNTER Setting	First Print Line Position	Final LINE-COUNTER Setting
A R	Absolute	1	2a	3a	4a	5a	Invalid Combination +	
A R	Relative	1	2a	3a	4b	5b	Invalid Combination +	
A R	NEXT PAGE	1	2b	3a	4c	5c	Invalid Combination +	
A R		1	2a	3a		5d	Invalid Combination +	
R	Absolute	1	2a	3b	4a	5a	Invalid Combination + +	
R	Relative	1	2a	3b	4b	5b	3d	5b
R	NEXT PAGE	1	2b	3b	4c	5c	Invalid Combination + +	
R		1	2a	3b		5d	3d	5d
				3c		5e	3c	5e

* See the LINE NUMBER Clause Notation section for a description of the abbreviations in column 1.

** A blank entry in column 1 or column 2 indicates that the named clause is totally absent from the Body Group Description entry.

*** A blank entry in an applicable rules column indicates the absence of the named rule for the given combination of LINE NUMBER and NEXT GROUP clauses.

+ See the section on the LINE NUMBER clause.

++ See the section on the NEXT GROUP clause.

VM-0604A-AI

REPORT HEADING Group Presentation Rules

1. Upper-limit rule:

   The first line number on which the REPORT HEADING report group can be presented is the line number specified by the HEADING phrase of the PAGE clause.

2. Lower-limit rules:

   a. The last line number on which the REPORT HEADING report group can be presented is the line number that is obtained by subtracting 1 from the first-detail-line value of the FIRST DETAIL phrase of the PAGE clause.

   b. The last line number on which the REPORT HEADING report group can be presented is the line number specified by page-size of the PAGE clause.

3. First print line position rules:

a. The first print line of the REPORT HEADING report group is presented on the line number specified by the integer of its LINE NUMBER clause.

b. The first print line of the REPORT HEADING report group is presented on the line number obtained by adding the integer of the first LINE NUMBER clause and the value obtained by subtracting 1 from the heading-line value of the HEADING phrase of the PAGE clause.

c. The REPORT HEADING report group is not presented.

d. The first print line of the REPORT HEADING report group is presented on the line number obtained by adding the contents of its LINE-COUNTER (in this case, zero) to the integer of the first LINE NUMBER clause.

4. Next group rules:

a. The NEXT GROUP integer must be greater than the line number on which the final print line of the REPORT HEADING report group is presented. In addition, the NEXT GROUP integer must be less than the line number specified by first-detail-line of the FIRST DETAIL phrase of the PAGE clause.

b. The sum of the NEXT GROUP integer and the line number on which the final print line of the REPORT HEADING report group is presented must be less than the value of first-detail-line of the FIRST DETAIL phrase of the PAGE clause.

c. NEXT GROUP NEXT PAGE signifies that the REPORT HEADING report group will appear by itself on the first page of the report. The RWCS processes no other report group while positioned at the first page of the report.

5. Final LINE-COUNTER setting rules:

a. After the REPORT HEADING report group is presented, the RWCS places the NEXT GROUP integer into LINE-COUNTER as the final LINE-COUNTER setting.

b. After the REPORT HEADING report group is presented, the RWCS places the sum of these two items into LINE-COUNTER as the final LINE-COUNTER setting:

- The NEXT GROUP integer
- The line number on which the final print line of the REPORT HEADING report group was presented

c. After the REPORT HEADING report group is presented, the RWCS places zero into LINE-COUNTER as the final LINE-COUNTER setting.

d. After the REPORT HEADING report group is presented, the final LINE-COUNTER setting is the line number on which the final print line of the REPORT HEADING report group was presented.

e. LINE-COUNTER is unaffected by the processing of a nonprintable report group.

D.6. PAGE HEADING Group Presentation Rules

Figure D.2 shows the appropriate presentation rules for all permissible combinations of LINE NUMBER and NEXT GROUP clauses in a PAGE HEADING report group.

Figure D.2. PAGE HEADING Group Presentation Rules Table

**		*** Applicable Tables **** If the PAGE Clause Is Specified				
* Sequence of LINE NUMBER Clauses	NEXT GROUP Clause	Upper Limit	Lower Limit	First Print Line Position	Next Group	Final LINE-COUNTER Setting
A R		1	2	3a		4a
R		1	2	3b		4a
				3c		4b

* See the section on LINE NUMBER Clause Notation for a description of the abbreviations in column 1.

** A blank entry in column 1 or column 2 indicates that the named clause is absent from the PAGE HEADING Report Group Description entry.

*** A blank entry in an applicable rules column indicates the absence of the named rule for the given combination of LINE NUMBER and NEXT GROUP clauses.

**** If the PAGE clause is omitted from the Report Description entry, then a PAGE HEADING report group cannot be defined. (See the TYPE clause in the Data Division chapter.)

VM-0605A-AI

PAGE HEADING Group Presentation Rules

1. Upper-limit rule:

   If a REPORT HEADING report group has been presented on the page on which the PAGE HEADING report group is to be presented, then the first line number on which the PAGE HEADING report group can be presented is one greater than the final LINE-COUNTER setting established by the REPORT HEADING.

   Otherwise, the first line number on which the PAGE HEADING report group can be presented is the line number specified by the HEADING phrase of the PAGE clause.

2. Lower-limit rule:

   The last line number on which the PAGE HEADING report group can be presented is the line number obtained by subtracting 1 from the first-detail-line value of the FIRST DETAIL phrase of the PAGE clause.

3. First print line position rules:

   a. The first print line of the PAGE HEADING report group is presented on the line number specified by the integer of its LINE NUMBER clause.

   b. If a REPORT HEADING report group has been presented on the page on which the PAGE HEADING report group is to be presented, then the sum of the following two items defines the line number on which the first print line of the PAGE HEADING report group is presented:

      - The final LINE-COUNTER setting established by the REPORT HEADING report group

      - The integer of the first LINE NUMBER clause of the PAGE HEADING report group

      Otherwise, the sum of the following two items defines the line number on which the first print line of the PAGE HEADING report group is presented:

      - The integer of the first LINE NUMBER clause of the PAGE HEADING report group

      - The value obtained by subtracting 1 from the heading-line value of the HEADING phrase of the PAGE clause

   c. The PAGE HEADING report group is not presented.

4. Final LINE-COUNTER setting rules:

   a. The final LINE-COUNTER setting is the line number on which the final print line of the PAGE HEADING report group was presented.

   b. LINE-COUNTER is unaffected by the processing of a nonprintable report group.

D.7. Body Group Presentation Rules

Figure D.3 points to the appropriate presentation rules for all permissible combinations of LINE NUMBER and NEXT GROUP clauses in CONTROL HEADING, DETAIL, and CONTROL FOOTING report groups.

Figure D.3. Body Group Presentation Rules

**		*** Applicable Rules							
		If the PAGE Clause Is Specified						If the PAGE Clause Is Omitted	
*Sequence of LINE NUMBER Clauses	NEXT GROUP Clause	Upper Limit	Lower Limit	Fit Test	First Print Line Position	Next Group	Final LINE-COUNTER Setting	First Print Line Position	Final LINE-COUNTER Setting
A R	Absolute	1	2	3a	4a	5	6a	+ Invalid Combination	
A R	Relative	1	2	3a	4a		6b	+ Invalid Combination	
A R	NEXT PAGE	1	2	3a	4a		6c	+ Invalid Combination	
A R		1	2	3a	4a		6d	+ Invalid Combination	
R	Absolute	1	2	3b	4b	5	6a	+ + Invalid Combination	
R	Relative	1	2	3b	4b		6b	4d	6f
R	NEXT PAGE	1	2	3b	4b		6c	+ + Invalid Combination	
R		1	2	3b	4b		6d	4d	6d
NP R	Absolute	1	2	3c	4a	5	6a	+ Invalid Combination	
NP R	Relative	1	2	3c	4a		6b	+ Invalid Combination	
NP R	NEXT PAGE	1	2	3c	4a		6c	+ Invalid Combination	
NP R		1	2	3c	4a		6d	+ Invalid Combination	
					4c		6e	4c	6e

* See the LINE NUMBER Clause Notation section for a description of the abbreviations in column 1.

** A blank entry in column 1 or column 2 indicates that the named clause is absent from the REPORT HEADING Group Description entry.

*** A blank entry in an applicable rules column indicates the absence of the named rule for the given combination of LINE NUMBER and NEXT GROUP clauses.

+ See the section on the LINE NUMBER clause.

++ See the section on the NEXT GROUP clause.

VM-0606A-AI

Body Group Presentation Rules

1. Upper-limit rule:

   The first line number on which a body group can be presented is the first-detail-line value in the FIRST DETAIL phrase of the PAGE clause.

2. Lower-limit rules:

The last line number on which a CONTROL HEADING report group or DETAIL report group can be presented is the last-detail-line value in the LAST DETAIL phrase of the PAGE clause.

The last line number on which a CONTROL FOOTING report group can be presented is the line number specified by the footing-line value in the FOOTING phrase of the PAGE clause.

3. Fit test rules:

   a. If the value in LINE-COUNTER is less than the integer of the first absolute LINE NUMBER clause, then the body group will appear on the page on which the report is currently positioned.

      Otherwise, the RWCS executes page-advance processing. After the PAGE HEADING report group (if defined) has been processed, the RWCS determines whether the saved-next-group-integer location was set when the final body group was presented on the preceding page. (See final LINE-COUNTER setting rule 6a.) If saved-next-group-integer was not so set, the body group will be presented on the page on which the report is currently positioned. If saved-next-group-integer was so set, the RWCS:

      - Moves the saved-next-group-integer into LINE-COUNTER
      - Resets saved-next-group-integer to zero
      - Reapplies fit test rule 3a

   b. If a body group has been presented on the page on which the report is positioned, the RWCS computes a trial sum in a work location. The trial sum is computed by adding:

      - The contents of LINE-COUNTER
      - The integers of all LINE NUMBER clauses of the report group

      If the trial sum is not greater than the body group's lower-limit integer, then the report group is presented on the current page. If the trial sum exceeds the body group's lower-limit integer, then the RWCS executes page-advance processing. After the PAGE HEADING report group (if defined) has been processed, the RWCS reapplies fit test rule 3b.

      If no body group has yet been presented on the page on which the report is currently positioned, the RWCS determines whether the saved-next-group-integer location was set when the final body group was presented on the preceding page. (See final LINE-COUNTER setting rule 6a.)

      If saved-next-group-integer was not set, the body group appears on the page on which the report is currently positioned.

      If saved-next-group-integer was set, the RWCS:

      - Moves the saved-next-group-integer into LINE-COUNTER
      - Resets saved-next-group-integer to zero
      - Computes a trial sum in a work location

      The trial sum is computed by adding:

      - The contents of LINE-COUNTER

- The integer 1
- The integers of all but the first LINE NUMBER clause of the body group

If the trial sum is not greater than the body group's lower-limit integer, then the body group is presented on the current page. If the trial sum exceeds the body group's lower-limit integer, then the RWCS executes page-advance processing. After the PAGE HEADING report group (if defined) has been processed, the RWCS presents the body group on that page.

c. If a body group has been presented on the page on which the report is currently positioned, the RWCS executes page-advance processing. After the PAGE HEADING report group (if defined) has been processed, the RWCS reapplies Fit Test rule 3c.

   If no body group has yet been presented on the page on which the report is currently positioned, the RWCS determines whether the saved-next-group-integer location was set when the final body group was presented on the preceding page. (See final LINE-COUNTER setting rule 6a). If saved-next-group-integer was not set, the body group will be presented on the page on which the report is currently positioned. If saved-next-group-integer was set, the RWCS moves saved-next-group-integer into LINE-COUNTER and resets saved-next-group-integer to zero.

   If the value in LINE-COUNTER is less than the integer of the first absolute LINE NUMBER clause, the RWCS presents the body group on the page on which the report is currently positioned. Otherwise, the RWCS executes page-advance processing. After the PAGE HEADING report group (if defined) has been processed, the RWCS presents the body group on that page.

4. First print line position rules:

   a. The first print line of the body group appears on the line number specified by the integer of its LINE NUMBER clause.

   b. The RWCS presents the first print line of the current body group on the line immediately following the line indicated by the value contained in LINE-COUNTER if these two conditions are true:

      - The value in LINE-COUNTER is equal to or greater than the line number specified by the first-detail-line value in the FIRST DETAIL phrase of the PAGE clause.
      - No body group has previously been presented on the page on which the report is currently positioned.

      The RWCS presents the first print line of the current body group on the line that is obtained by adding the contents of LINE-COUNTER and the integer of the first LINE NUMBER clause of the current body group if these two conditions are true:

      - The value in LINE-COUNTER is equal to or greater than the line number specified by the first-detail-line value in the FIRST DETAIL phrase of the PAGE clause.
      - A body group has previously been presented on the page to which the report is currently positioned.

      If the value in LINE-COUNTER is less than the line number specified by the first-detail-line value in the FIRST DETAIL phrase of the PAGE clause, then the RWCS presents the first print line of the body group on the line specified by the FIRST DETAIL phrase.

c. The body group is not presented.

d. The line number on which the RWCS presents the first print line is the sum of the contents of:

- LINE-COUNTER
- The integer of the first LINE NUMBER clause

5. Next group rule:

   The integer of the absolute NEXT GROUP clause (next-group-line-num) must specify a line number that is: (a) not less than that specified in the FIRST DETAIL phrase of the PAGE clause, and (b) not greater than that specified in the FOOTING phrase of the PAGE clause.

6. Final LINE-COUNTER setting rules:

   a. If the body group that has just been presented is a CONTROL FOOTING report group and if the CONTROL FOOTING report group is not associated with the highest level at which the RWCS detected a control break, then the final LINE-COUNTER setting is the line number on which the final print line of the CONTROL FOOTING report group was presented.

      If the line number on which the final print line of the body group was presented is less than the integer of the NEXT GROUP clause, then the RWCS places the NEXT GROUP integer into LINE-COUNTER as the final LINE-COUNTER setting. If the line number on which the final print line of the body group was presented is equal to or greater than the integer of the NEXT GROUP clause, then the RWCS places the line number specified by the FOOTING phrase of the PAGE clause into LINE-COUNTER as the final LINE-COUNTER setting. In addition, the RWCS places the NEXT GROUP integer into the saved-next-group-integer location.

   b. If the body group that has just been presented is a CONTROL FOOTING report group, and if the CONTROL FOOTING report group is not associated with the highest level at which the RWCS detected a control break, then the final LINE-COUNTER setting is the line number on which the final print line of the CONTROL FOOTING report group was presented.

      For all other cases the RWCS computes a trial sum in a work location. The trial sum is computed by adding:

      - The integer of the NEXT GROUP clause
      - The line number on which the final print line of the body group was presented

      If the sum is less than the line number specified by the footing-line value in the FOOTING phrase of the PAGE clause, then the RWCS places that sum into LINE-COUNTER as the final LINE-COUNTER setting.

      If the sum is equal to or greater than the line number specified by the footing-line value in the FOOTING phrase of the PAGE clause, then the RWCS places that line number into LINE-COUNTER as the final LINE-COUNTER setting.

   c. The final LINE-COUNTER setting is the line number on which the final print line of the CONTROL FOOTING report group was presented if:

      - The body group that has just been presented is a CONTROL FOOTING report group.

- The CONTROL FOOTING report group is not associated with the highest level at which the RWCS detected a control break.

For all other cases the RWCS places the line number specified by the footing-line value in the FOOTING phrase of the PAGE clause into LINE-COUNTER as the final LINE-COUNTER setting.

d. The final LINE-COUNTER setting is the line number on which the final print line of the body group was presented.

e. LINE-COUNTER is unaffected by the processing of a nonprintable body group.

f. The final LINE-COUNTER setting is the line number on which the RWCS presents the final print line of the CONTROL FOOTING report group if:

- The body group that has just been presented is a CONTROL FOOTING report group.
- The CONTROL FOOTING report group is not associated with the highest level at which the RWCS detected a control break.

For all other cases the RWCS uses the sum of these two items as the final LINE-COUNTER setting:

- The line number on which the final print line was presented
- The NEXT GROUP integer

D.8. PAGE FOOTING Group Presentation Rules

Figure D.4 shows the appropriate presentation rules for all permissible combinations of LINE NUMBER and NEXT GROUP clauses in a PAGE FOOTING report group.

Figure D.4. PAGE FOOTING Group Presentation Rules

**		*** Applicable Tables **** If the PAGE Clause Is Specified				
* Sequence of LINE NUMBER Clauses	NEXT GROUP Clause	Upper Limit	Lower Limit	First Print Line Position	Next Group	Final LINE-COUNTER Setting
A R	Absolute	1	2	3a	4a	5a
A R	Relative	1	2	3a	4b	5b
A R		1	2	3a		5c
				3b		5d

* See the section on LINE NUMBER Clause Notation for a description of the abbreviations in column 1.

** A blank entry in column 1 or column 2 indicates that the named clause is absent from the PAGE FOOTING Report Group Description entry.

*** A blank entry in an applicable rules column indicates the absence of the named rule for the given combination of LINE NUMBER and NEXT GROUP clauses.

**** If the PAGE clause is omitted from the Report Description entry, then a PAGE FOOTING report group cannot be defined. (See the TYPE clause in the Data Division chapter.)

VM-0607A-AI

The PAGE FOOTING Group Presentation Rules are:

1. Upper-limit rule:

   The first line number on which the PAGE FOOTING report group can be presented is the line number obtained by adding:

   - The integer 1
   - The value of footing-line in the FOOTING phrase of the PAGE clause

2. Lower-limit rule:

   The last line number on which the PAGE FOOTING report group can be presented is the line number specified by page-size of the PAGE clause.

3. First print line position rules:

   a. The first print line of the PAGE FOOTING report group is presented on the line specified by the integer of its LINE NUMBER clause.

   b. The PAGE FOOTING report group is not presented.

4. Next group rules:

a. The NEXT GROUP integer must be greater than the line number on which the final print line of the PAGE FOOTING report group is presented. In addition, the NEXT GROUP integer must not be greater than the line number specified by the page-size value of the PAGE clause.

b. The sum of the following two items must not be greater than the line number specified by page-size of the PAGE clause:

   - The NEXT GROUP integer
   - The line number on which the final print line of the PAGE FOOTING report group is presented

5. Final LINE-COUNTER setting rules:

   a. The final LINE-COUNTER setting after the RWCS presents the PAGE FOOTING report group is the NEXT GROUP integer.

   b. The final LINE-COUNTER setting after the RWCS presents the PAGE FOOTING report group is the sum of:

      - The NEXT GROUP integer
      - The line number on which the final print line of the PAGE FOOTING report group was presented

   c. After the PAGE FOOTING report group is presented, the final LINE-COUNTER setting is the line number on which the final print line of the PAGE FOOTING report group was presented.

   d. LINE-COUNTER is unaffected by the processing of a nonprintable report group.

D.9. REPORT FOOTING Group Presentation Rules

Figure D.5 points to the appropriate presentation rules for all permissible combinations of LINE NUMBER and NEXT GROUP clauses in a REPORT FOOTING report group.

Figure D.5. REPORT FOOTING Group Presentation Rules

**		*** Applicable Rules						
		If the PAGE Clause Is Specified					If the PAGE Clause Is Omitted	
*Sequence of LINE NUMBER Clauses	NEXT GROUP Clause	Upper Limit	Lower Limit	First Print Line Position	Next Group	Final LINE-COUNTER Setting	First Print Line Position	Final LINE-COUNTER Setting
A R		1a	2	3a		4a	+ Invalid Combination	
R		1a	2	3b		4a	3d	4a
NP R		1b	2	3c		4a	+ Invalid Combination	
				3e		4b	3e	4b

* See the LINE NUMBER Clause Notation section for a description of the abbreviations in column 1.

** A blank entry in column 1 or column 2 indicates that the named clause is totally absent from the REPORT FOOTING Group Description entry.

*** A blank entry in an applicable rules column indicates the absence of the named rule for the given combination of LINE NUMBER and NEXT GROUP clauses.

+ See the section on the LINE NUMBER clause.

VM-0608A-AI

REPORT FOOTING Group Presentation Rules

1. Upper-limit rules:

   a. The first line number on which the REPORT FOOTING report group can be presented is one greater than the final LINE-COUNTER setting established by the PAGE FOOTING report group if a PAGE FOOTING report group has been presented on the page on which the report is positioned.

      Otherwise, the first line number on which the REPORT FOOTING report group can be presented is the line number obtained by adding 1 and the footing-line value of the PAGE clause.

   b. The first line number on which the REPORT FOOTING report group can be presented is the line number specified by the HEADING phrase of the PAGE clause.

2. Lower-limit rule:

   The last line number on which the REPORT FOOTING report group can be presented is the line number specified by the page-size value of the PAGE clause.

3. First print line position rules:

   a. The first print line of the REPORT FOOTING report group is presented on the line specified by the integer of its LINE NUMBER clause.

   b. If the RWCS presents a PAGE FOOTING report group on the page to which the report is positioned, then the sum of the following two items defines the line number on which the RWCS presents the first print line of the REPORT FOOTING report group:

      - The final LINE-COUNTER setting established by the PAGE FOOTING report group
      - The integer of the first LINE NUMBER clause of the REPORT FOOTING report group

      Otherwise, the sum of the following two items defines the line number on which the RWCS presents the first print line of the REPORT FOOTING report group:

      - The integer of the first LINE NUMBER clause of the REPORT FOOTING report group
      - The line number specified by the footing-line value of the FOOTING phrase of the PAGE clause

   c. The NEXT PAGE phrase in the first absolute LINE NUMBER clause directs the REPORT FOOTING report group to appear on a page on which no other report group has been presented. The first print line of the REPORT FOOTING report group is presented on the line number specified by the integer of its LINE NUMBER clause.

   d. The line number on which the RWCS presents the first print line is the sum of:

      - The contents of LINE-COUNTER
      - The integer of the first LINE NUMBER clause

   e. The REPORT FOOTING report group is not presented.

4. Final LINE-COUNTER setting rules:

   a. The final LINE-COUNTER setting is the line number on which the RWCS presents the final print line of the REPORT FOOTING report group.

   b. LINE-COUNTER is unaffected by the processing of a nonprintable report group.

Appendix E. RTL Routines for Accessing the RAB and FAB Structures (OpenVMS Alpha and I64 Only)

In VSI COBOL for OpenVMS for OpenVMS Alpha and OpenVMS I64, when a file is successfully opened, the file's RAB pointer is placed in its RMS_STV field and its FAB pointer is placed in the FABPTR field of the RAB. The two RTL routines documented here (DCOB$RMS_CURRENT_RAB and DCOB$RMS_CURRENT_FAB) enable VSI COBOL for OpenVMS programmers to access the RAB and FAB data structures on OpenVMS Alpha and I64. However, the content and format of the RAB and FAB are not covered by any external standards (such as the ANSI standard for COBOL) and are subject to change.

Description

DCOB$RMS_CURRENT_FAB returns the address of the RMS FAB structure for the most recently used COBOL file connector. Some fields of the FAB are filled in by the RMS SYS$OPEN routine.

This routine can be used to obtain the address of the RMS FAB structure. The FAB is filled in by SYS $OPEN to reflect the actual attributes of the file in the cases where the actual attributes differ from the attributes specified by the COBOL file connector.

The FAB is a structure used internally by the VSI COBOL for OpenVMS run-time system to implement COBOL semantics. Modification of the FAB can result in abnormal program behavior, including unexpected program termination.

Example

See the example for DCOB$RMS_CURRENT_RAB.

Description

DCOB$RMS_CURRENT_RAB returns the address of the RMS RAB structure for the most recently used COBOL file connector.

This routine can be used to obtain the address of the RMS RAB structure. The RAB describes attributes of the connection to a file.

The RAB is a structure used internally by the VSI COBOL for OpenVMS run-time system to implement COBOL semantics. Modification of RAB fields can result in abnormal program behavior, including unexpected program termination.

Example

1.
```
*+
* PROGRAM : RMSEXAMPLE
*
* PROGRAM DESCRIPTION:
*
```

```
*       This program is an example of use of DCOB$RMS_CURRENT_FAB
*   and DCOB$RMS_CURRENT_RAB.
*
*-
IDENTIFICATION DIVISION.
PROGRAM-ID. RMSEXAMPLE.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT EMPLOYEE-FILE
     ASSIGN TO "employee.dat"
     ORGANIZATION IS SEQUENTIAL.


DATA DIVISION.
FILE SECTION.
FD EMPLOYEE-FILE
    BLOCK CONTAINS 2048 CHARACTERS.
01 EMPLOYEE.
    03 NAME PIC X(30).
    03 OFFICE PIC X(10).
    03 PHONE PIC X(10).

WORKING-STORAGE SECTION.
01  EMPLOYEE-FAB USAGE IS POINTER.
01  EMPLOYEE-RAB USAGE IS POINTER.

PROCEDURE DIVISION.
P0.
*
* Open the file to establish EMPLOYEE-FILE as the current file.
*
    OPEN INPUT EMPLOYEE-FILE.

*
* Get the pointer to the RMS FAB structure for EMPLOYEE-FILE.  Store
* the pointer in EMPLOYEE-FAB.  Do the same for the RAB.
*
    CALL "DCOB$RMS_CURRENT_FAB" GIVING EMPLOYEE-FAB.
    CALL "DCOB$RMS_CURRENT_RAB" GIVING EMPLOYEE-RAB.

*
* Pass the address of the FAB to a subroutine that will use
* the contents.
*
    CALL "PRINT-FIELDS" USING BY VALUE EMPLOYEE-FAB EMPLOYEE-RAB.

*
* CLOSE the file before exiting.
*
    CLOSE EMPLOYEE-FILE.
    STOP RUN.
END PROGRAM RMSEXAMPLE.
IDENTIFICATION DIVISION.
PROGRAM-ID. PRINT-FIELDS.

DATA DIVISION.
```

```
WORKING-STORAGE SECTION.
01 TEMP-W.
    03 TEMP-WORD PIC S9(4) COMP.
01 TEMP-W1 REDEFINES TEMP-W.
    03 TEMP-B1 PIC X.
    03 TEMP-B2 PIC X.

LINKAGE SECTION.
01 FAB.
  05 FAB_ID     PIC S9(9) COMP VALUE 20483.
  05 FOP        PIC S9(9) COMP.
  05 STS        PIC S9(9) COMP.
  05 STV        PIC S9(9) COMP.
  05 ALQ        PIC S9(9) COMP.
  05 DEQ        PIC S9(4) COMP.
  05 FAC        PIC X.
  05 SHR        PIC X.
  05 CTX        PIC S9(9) COMP.
  05 RTV        PIC X.
  05 ORG        PIC X.
  05 RAT        PIC X.
  05 RFM        PIC X.
  05 JNL        PIC S9(9) COMP.
  05 XAB-ADD    USAGE IS POINTER.
  05 NAM-ADD    USAGE IS POINTER.
  05 FNA        USAGE IS POINTER.
  05 DNA        USAGE IS POINTER.
  05 FNS        PIC X.
  05 DNS        PIC X.
  05 MRS        PIC S9(4) COMP.
  05 MRN        PIC S9(9) COMP.
  05 BLS        PIC S9(4) COMP.
  05 FILLER     PIC X(18).
01 RAB.
  05 RAB_ID PIC S9(9) COMP VALUE 17409.
  05 ROP PIC S9(9) COMP.
  05 STS PIC S9(9) COMP.
  05 STV PIC S9(9) COMP.
  05 RFA PIC S9(4) COMP OCCURS 3 TIMES.
  05 RESERVED   PIC S9(4) COMP.
  05 CTX PIC S9(9) COMP.
  05 RAC PIC X.
  05 TMO PIC X.
  05 USZ PIC S9(4) COMP.
  05 RSZ PIC S9(4) COMP.
  05 UBF PIC S9(9) COMP.
  05 RBF PIC S9(9) COMP.
  05 RHB PIC S9(9) COMP.
  05 KBF PIC S9(9) COMP.
  05 KSZ PIC X.
  05 KRF PIC X.
  05 MBF        PIC X.
  05 MBC PIC X.
  05 BKT PIC S9(9) COMP.
  05 FABPTR PIC S9(9) COMP.
  05 XAB PIC S9(9) COMP.

PROCEDURE DIVISION USING FAB RAB.
```

```
*
* Convert the MBF PIC X value to a PIC S9(5) COMP VALUE.
*
    MOVE 0 TO TEMP-WORD.
    MOVE MBF TO TEMP-B1.

*
* Display the multibuffer count and file allocation.
*
    DISPLAY "Multibuffer count is "
       TEMP-WORD WITH CONVERSION " blocks".
    DISPLAY "File allocation is " ALQ WITH CONVERSION.

    EXIT PROGRAM.
END PROGRAM PRINT-FIELDS.
```

Output:

2.
```
$
$ LINK RMSEXAMPLE
$ RUN RMSEXAMPLE
File allocation is       1206
$
```