Series 100
Cobol by Microsoft®
Reference Manual

Manual Part No.
45448-90001

Printed in U.S.A.
2/84

Microsoft Corporation

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy Microsoft COBOL on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

© Copyright Microsoft Corporation, 1980, 1983

If you have comments about the software or these manuals, please send your comments to:

Hewlett Packard
Personal Software Division
3410 Central Expressway
Santa Clara, California 95051

Microsoft and the Microsoft logo are registered trademarks of Microsoft Corporation.

MS is a trademark of Microsoft Corporation.

Part Number 45448-90001

Acknowledgment

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

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

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

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

"The authors and copyright holders of the copyrighted material used herein (FLOW-MATIC (Trademark of Sperry Rand Corporation), Programming for the UNIVAC (R) I and II, Data Automation Systems, copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator, Form No. F28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell) have specifically authorized the use of this material in whole or in part in the COBOL specification, in programming manuals or similar publications."

from the ANSI COBOL STANDARD
(X3.23-1974)

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

Contents

Introduction

How to Use This Manual Introduction-2
General Format and Syntax Notation Introduction-3
Learning More About COBOL Introduction-5
Microsoft COBOL and ANSI 74 Standard COBOL Introduction-6
Microsoft COBOL Extensions Introduction-9

Chapter 1 - Language Elements

Source Coding Rules 1-2
Character Set 1-3
Punctuation 1-4
Reserved Words 1-5
Names 1-6
Literals 1-9
Statements 1-12
Arithmetic Statements 1-14
Arithmetic Expressions 1-15

Chapter 2 - Structure of a COBOL Program

Level Numbers and Data-Items ... 2-4
Compiler Directing Statements ... 2-6

Chapter 3 - IDENTIFICATION DIVISION

AUTHOR Paragraph ... 3-2
DATE-COMPILED Paragraph ... 3-3
DATE-WRITTEN Paragraph ... 3-3
IDENTIFICATION DIVISION Header ... 3-3
INSTALLATION Paragraph ... 3-4
PROGRAM-ID Paragraph ... 3-4
SECURITY Paragraph ... 3-4

Chapter 4 - ENVIRONMENT DIVISION

ASSIGN Clause ... 4-4
CONFIGURATION SECTION Header ... 4-5
ENVIRONMENT DIVISION Header ... 4-6
FILE-CONTROL Paragraph ... 4-7
INPUT-OUTPUT SECTION Header ... 4-8
I-O-CONTROL Paragraph ... 4-9
OBJECT-COMPUTER Paragraph ... 4-10
SAME AREA Clause ... 4-11
SELECT Clause ... 4-12
SOURCE-COMPUTER Paragraph ... 4-15
SPECIAL-NAMES Paragraph ... 4-16

Chapter 5 - DATA DIVISION

Data-Items and Data Descriptions 5-7
DATA DIVISION Limitations 5-15
Sections 5-15
Clauses 5-25

Chapter 6 - PROCEDURE DIVISION

Arithmetic Statements 6-5
I-O Error Handling 6-8
Dynamic Debugging Statements 6-9
PROCEDURE DIVISION Statements 6-10

Chapter 7 - Inter-Program Communication

CALL Statement 7-2
EXIT PROGRAM Statement 7-3
CHAIN Statement 7-3
PROCEDURE DIVISION Header With CALL and CHAIN 7-4

Chapter 8 - Table Handling by the Indexing Method

Index-Names and Index-Items 8-1
SET Statement 8-2
Relative Indexing 8-2
Format 1 SEARCH Statement 8-3
Format 2 SEARCH Statement 8-6

Chapter 9 - SEQUENTIAL Files

Definition of Sequential File Organization 9-1
Syntax Considerations 9-3
File Status Reporting.. 9-4
PROCEDURE DIVISION Statements for Sequential Files........... 9-5

Chapter 10 - INDEXED Files

Definition of INDEXED File Organization....................... 10-1
Syntax Considerations 10-3
PROCEDURE DIVISION Statements for INDEXED Files 10-5

Chapter 11 - RELATIVE Files

Definition of RELATIVE File Organization 11-1
Syntax Considerations 11-2
File Status Reporting... 11-3
PROCEDURE DIVISION Statement for RELATIVE Files 11-4

Chapter 12 - Declaratives and the USE Sentence

Chapter 13 - Segmentation

Appendix A - Advanced Forms of Conditions

Evaluation Rules for Compound Conditions A-1
Parenthesized Conditions A-3
Abbreviated Conditions A-3
NOT, the Logical Negation Operator A-4

Appendix B - Table of Permissible MOVE Operands

Appendix C - Nested IF Statements

Appendix D - ASCII Character Set for ANSI 74 COBOL

Appendix E - Reserved Words

Appendix F - PERFORM With VARYING and AFTER Clauses

Introduction

Microsoft® COBOL (MS®-COBOL) provides the most extensive implementation of COBOL for microcomputers. With all of Level 1 and most Level 2 ANSI 74 Standard COBOL features, MS-COBOL has been validated by GSA as a low-intermediate implementation of COBOL. Special features such as advanced verbs, trace-style debugging, and formatted screen ACCEPT/DISPLAY assure you that MS-COBOL will not only perform to specification, but will also provide you with the tools you need to create better business programs.

How to Use This Manual

This manual is a reference for all implementations of Microsoft COBOL. The manual is divided into an introduction, the main technical portion of the manual, and appendices.

This introduction explains how to use the manual: it outlines the organization, describes the syntax notation used throughout the manual, and provides a list of sources for learning more about COBOL programming. This introduction also compares Microsoft COBOL with ANSI 74 Standard COBOL.

The main body of the *Reference Manual* covers the following topics:

Chapters 1 and 2 discuss elements of the MS-COBOL language and explain the structure of a COBOL program. These chapters include definitions of terms, an outline of the MS-COBOL general format, and a brief description of the COPY and USE commands.

Chapters 3 through 6 discuss the four divisions of an MS-COBOL program, including any statements or clauses normally placed in those divisions. The first page of each chapter gives the general format for the appropriate division, with the arrangement that would normally appear in an MS-COBOL source program. Individual portions of the general format are then discussed in alphabetical order.

Chapters 7 through 13 present advanced MS-COBOL topics, including interprogram communication, table handling, file organizations, declaratives, and segmentation.

The appendices provide a table of permissible MOVE operands, a list of ASCII characters, and a list of COBOL reserved words.

General Format and Syntax Notation

Whenever the general format for a statement is shown, the following conventions apply:

CAPS

All words shown in CAPS are MS-COBOL reserved words.

Reserved words that are not underlined are optional. If included, they are used solely for improving the readability of the program.

CAPS

All underlined reserved words are key words, and are required unless the portion of the format containing them is itself optional. A missing key word or incorrect spelling of a key word results in an error.

lower-case

Words printed in lower-case letters in formats represent terms for which the user must substitute a valid entry.

< > =

The characters less than (<), greater than (>), and equals (=), although not underlined, are required when they appear in a general format.

Special Characters

Punctuation and special characters are required where shown in a general format. Additional punctuation can be inserted, according to the rules for punctuation specified in Chapter 1.

In general, terminal periods are shown in formats, because they are required; semicolons and commas are not usually shown in the formats, because they are optional. To be considered separators, all commas, semicolons, and periods must be followed by a space or blank.

[ ]

Any part of a statement or data description entry that is enclosed in brackets is optional.

( )

Optional elements may be indicated by parentheses instead of brackets, if no ambiguity results.

{ }

When braces enclose parts of a statement, one — and only one — of the options must be used. Braces also delimit the portion of a statement that applies to a subsequent ellipsis.

|	Alternate options may also be separated by a vertical line (e.g., AREA | AREAS is equivalent to enclosure within braces).
lower-case with suffix	When more than one occurrence of a term is included in a format, a digit or letter may be used as a suffix. The suffix is for clarification only, and does not change the meaning of the term (e.g., data-name-1, data-name-2).
. . .	The ellipsis (. . .) indicates that the immediately preceding unit may occur once or any number of times in succession. A unit is either a single lower-case word, or a group of lower-case words and one or more reserved words enclosed in brackets or braces. If repetition occurs, the entire unit must be repeated.

If necessary, the text accompanying the general format will contain comments, restrictions, or clarification on use of the format.

Any clauses (e.g., BLOCK clause) or statements (e.g., PERFORM statement) mentioned in a format will be described elsewhere in the text.

Learning More About COBOL

If you are new to COBOL programming, you will probably want to learn more about the language before using this manual. The following texts are all COBOL tutorials, written for the novice programmer:

Abel, Peter. *COBOL Programming: A Structured Approach.* Reston, Virginia: Reston Publishing Co., 1980.

McCracken, Daniel D. *A Simplified Guide to Structured COBOL Programming.* New York: John Wiley and Sons, Inc., 1976.

Parkin, Andrew. *COBOL for Students.* Edward Arnold, Ltd., 1978.

Microsoft COBOL and ANSI 74 Standard COBOL

Guidelines established by the American National Standards Institute (ANSI) are used to compare COBOL compilers. The latest standards, adopted in 1974, contain twelve modules, each with two levels of implementation. The standard states that the first three modules (Nucleus, Table Handling, and Sequential I-O) must be implemented at least to Level 1. The other nine modules are optional. Microsoft COBOL, with all Level 1 features and many features from Level 2, is validated as low-intermediate.

Nucleus

MS-COBOL includes all Level 1 features. It includes all features of Level 2 except that:

1. A figurative constant used as a literal in ALL "literal" cannot be longer than one character (e.g., PIC A(9) VALUE IS ALL "ABC" is not valid; PIC A(9) VALUE IS ALL "Q" is valid).
2. In the ENVIRONMENT DIVISION, names cannot be qualified.
3. In the SPECIAL-NAMES paragraph, the alphabet-name must be ASCII, and the literal phrase cannot be used.
4. In the DATA DIVISION:
   a. The OCCURS DEPENDING ON clause is not supported.
   b. Level 88 statements may have either a list of items or a range, but not both.
   c. COMP-0 data items always require 2 bytes. Therefore:
      i. PIC 9(5) only allows a range of −32768 to 32767
      ii. PIC 9, 99, 999, and 9999 are equivalent to PIC 9(5) for COMP-0 items
      iii. An error message is given when more than 5 digits are specified.

d. Unsigned COMP-0 data items are not supported (e.g., PIC 9 is equivalent to PIC S9).

   (Note that in MS-COBOL for the Z80/8080 microprocessor, COMP data items are treated the same as COMP-0 data items unless the /V switch is used at compile time. In this case, COMP data items are treated as display data items. In non-Z80/8080 versions of MS-COBOL, COMP data items are treated as if they were defined with USAGE IS DISPLAY. See the *Microsoft COBOL Compiler User's Guide* for more information on the /V compiler switch.)

e. The RENAMES phrase is not supported.

5. In the PROCEDURE DIVISION:

   a. The CORRESPONDING option is not supported for MOVE, ADD, and SUBTRACT statements.

   b. Arithmetic statements may not have multiple destinations.

   c. Division remainders are not supported.

   d. INSPECT (Level 2) is not supported.

   e. An ALTER statement can contain only one procedure-name.

Table Handling

Same level of implementation as Nucleus.

Sequential I-O

The Level 1 Rerun facility is not included, because most microcomputer operating systems do not support it.

All Level 2 features are supported.

Microsoft COBOL uses special language for tape handling:

1. An optional tape file may be specified with SELECT OPTIONAL filename.
2. A fully functional RESERVE INTEGER AREA(S) clause allows input/output buffering.
3. Multi-file tapes may be specified with the MULTIPLE FILE TAPE CONTAINS clause.
4. Fully functional BLOCK CONTAINS and RECORD CONTAINS clauses are allowed in the FD entry for tape files, thereby giving the programmer control over blocking fixed-length and variable-length records.

5. Fully implemented OPEN and CLOSE statements allow multi-reel files, tape reversal, and tape positioning.

Relative I-O

Same level of implementation as Sequential I-O.

Indexed I-O

Same level of implementation as Sequential I-O.

Inter-program Communication

All features of Level 1 are implemented.

Library

All features of Level 1 are implemented.

Debug

Not implemented. However, Microsoft COBOL does include the trace-style debug extensions to ANSI 74 Standard COBOL and an interactive debug facility.

Report-Writer

Not implemented.

Segmentation

All features of Level 1 are implemented.

Sort/Merge

Full Level 2 implementations for all versions for which the MS-SORT package is available.

Microsoft COBOL Extensions

Microsoft COBOL includes the following extensions to ANSI 74 Standard COBOL.

1. Special options are available with the ACCEPT and DISPLAY statements (see Chapter 6, "Procedure Division," for discussion of these statements).
2. The COMP-3 data format is available. This format allows numeric data to be packed two digits to the byte, so that mass storage requirements are reduced.
3. Lower-case characters are treated as if they were upper-case, unless made part of a non-numeric (quoted) literal.
4. The dynamic debugging statements, READY TRACE, RESET TRACE, and EXHIBIT allow the display of procedure names or data items during program execution.

Chapter 1

LANGUAGE ELEMENTS

This chapter defines the terms that are used throughout this manual to refer to parts of an MS-COBOL program. It also gives the rules for coding an MS-COBOL source program and outlines the naming conventions recognized by MS-COBOL.

Source Coding Rules

Source programs may be written on standard coding sheets or on a terminal. The rules given below apply to both methods; though the actual column numbers may differ slightly for a particular terminal, the relative positions remain the same.

The following rules apply to coding an MS-COBOL program:

1. Each line of code may have a six-digit line number in columns 1 through 6. These line numbers must be in ascending order. Blanks are also permitted in columns 1 through 6.

2. If an asterisk (*) is placed in column 7 of the line, the line will be treated as an explanatory comment. It will be shown on the source listing but will otherwise be ignored. If a slash (/) appears in column 7, the line will be treated as a comment and the printer will space to the top of a new page before printing the line in the source listing.

3. If the character "D" is placed in column 7 of the line, the line will be treated as a comment unless the WITH DEBUGGING MODE clause of the SOURCE-COMPUTER paragraph is used. See Chapter 4, "SOURCE-COMPUTER Paragraph," for information concerning the SOURCE-COMPUTER paragraph.

4. If a hyphen (-) is placed in column 7, the line is treated as a continuation of the previous line. Except when non-numeric literals are continued, all trailing spaces on the preceding line and all leading spaces on the continuation line are ignored. (See under this chapter's "Non-Numeric Literals," for special rules on continuing non-numeric literals.) Area A (columns 8 through 11) of the continuation line must be blank.

5. Reserved words for division, section, and paragraph headers must begin in Area A. Definitions of procedure-names must also begin in Area A, as must level numbers 01 and 77 and level indicator FD. Other level numbers must begin in Area B (columns 12 through 72).

6. All other program elements should be confined to Area B. Rules of statement punctuation must be observed.

7. Columns 73 through 80 are ignored by the compiler.

8. Tab characters in a line are expanded as specified in the *Microsoft COBOL Compiler User's Guide.* Care should be taken that tab characters do not cause illegal placement of program elements.

Character Set

The MS-COBOL language character set consists of the following characters:

Letters A through Z, a through z
Blank or space
Digits 0 through 9
Special characters:

- + Plus sign
- - Minus sign
- * Asterisk
- = Equal sign
- > Relational sign (greater than)
- < Relational sign (less than)
- $ Dollar sign
- , Comma
- ; Semicolon
- . Period or decimal point
- " Quotation mark
- (Left parenthesis
-) Right parenthesis
- ' Apostrophe (alternate of quotation mark)
- / Slash

For non-numeric (quoted) literals and comments, the MS-COBOL character set is expanded to include the computer's entire character set.

Punctuation

The following characters are used for punctuation:

(Left parenthesis
) Right parenthesis
, Comma
. Period
; Semicolon

The following general rules of punctuation apply in writing source programs:

1. When used as punctuation, a period, semicolon, or comma should not be preceded by a space, but must be followed by a space.
2. At least one space must appear between two successive words and/or literals. Two or more successive spaces are treated as a single space, except in non-numeric literals.
3. Relational characters should always be preceded by a space and followed by another space.
4. When the plus or minus characters, period, or comma are used in the PICTURE clause, they are governed solely by rules for report (numeric edited) items (see Chapter 5, "PICTURE Clause," for discussion of report items).
5. A comma may be used as a separator between successive operands of a statement, or between two subscripts. It must be followed by a space (e.g., 10, 20).
6. A semicolon or comma may be used to separate a series of statements or clauses. The punctuation must be followed by a space (e.g., SUBTRACT A FROM X; MOVE X TO Y).

Reserved Words

Reserved words are words with specific meanings within the COBOL language or within Microsoft COBOL. They appear in upper-case letters in general formats. They may contain the letters A through Z, a through z, digits 0 through 9, or the hyphen (-). The maximum length is 30 characters. Many are verbs (e.g., ADD, SUBTRACT, MOVE) or descriptive phrases (e.g., PICTURE, VALUE IS). Reserved words may not be used for programmer-assigned names.

See Appendix E, "Reserved Words," for a complete list of reserved words.

Names

Any word that is not an MS-COBOL reserved word can be used as a programmer-assigned name, as long as it meets the naming conventions listed in the following section.

Naming Conventions

Names may be up to 30 characters long and must contain only letters A through Z, a through z, digits 0 through 9, or the hyphen (-). In addition:

1. All names except procedure-names must contain at least one letter or hyphen. Procedure-names may consist entirely of digits.
2. A name may not begin or end with a hyphen. However, a name may contain more than one hyphen, and consecutive hyphens are permitted.
3. A name is ended by a space or by appropriate punctuation.
4. If a programmer-supplied name is not unique, it must be used with qualifiers. Qualifiers are described in the following section on "Qualification of Names."

Data-Names

A data-name is a word assigned by the user to identify a data item referenced in a program. Data-names are defined in the DATA DIVISION of the program. A data-name always refers to a region that contains data, rather than to a particular value, because an item often assumes a number of different values during the course of a program.

If some of the characters in a record are not referenced in the processing steps of a program, a data-name need not be assigned. Instead, the word FILLER is used to set aside the appropriate amount of space.

A data-name must begin with an alphabetic character. A data-name or the key word FILLER must be the first word following the level number in each data-name entry, as shown in the following general format:

```
             { data-name-1 }
level-number { FILLER      }
```

See Chapter 2, "Structure of a COBOL Program," for discussion of level numbers. See also Chapter 5, "DATA DIVISION," for more information on assigning data-names.

File-Names

A file is a collection of data records, such as a printed listing or a region of floppy disk, containing individual records of a similar class or application. A file-name is defined by an FD entry in the FILE SECTION of the DATA DIVISION. The format is:

```
FD file-name
```

Rules for composition of the file-name are identical to those for data-names. References to file-names appear in the ENVIRONMENT DIVISION and in CLOSE, OPEN, and READ statements.

Condition-Names

A condition-name is a name assigned to a specific value, set, or range of values within the complete set of values that a data-item may assume. Condition-names are defined in level 88 entries within the DATA DIVISION. For example, a level 03 item "CLASS-NO" might be followed by the following subordinate level 88 entries:

```
88 VALID-NO     VALUE IS '1'.
88 INVALID-NO   VALUE IS '2'.
```

An IF statement could then reference either the literal value '1' or '2', or the condition-names VALID-NO or INVALID-NO.

Rules for forming condition-names are the same as those for data-names (see following discussion on "Data-Names"). Condition-names and their uses are explained more fully in Chapter 5, "DATA DIVISION," and Chapter 6, "PROCEDURE DIVISION."

Mnemonic-Names

A mnemonic-name assigns a user-defined word to an implementation-specific name, such as PRINTER. The rules for naming are the same as those for data-names (see following discussion on "Data-Names"). Mnemonic-names are assigned in the ENVIRONMENT DIVISION and are referenced by ACCEPT and DISPLAY statements.

Procedure-Names

Procedure-names are assigned to paragraphs or sections that are executed with the PERFORM or GO TO statements or by falling through from another part of the program. Procedure-names are declared in the PROCEDURE DIVISION. They must conform to the rules for data-names (see following discussion on "Data-Names"), except that a procedure-name may consist entirely of digits.

Index-Names and Index-Data-Items

Index-names and index-data-items are used for table handling by indexing. An index-name is declared implicitly by its appearance in the "INDEXED BY index-name" appendage to an OCCURS clause. Index-data-items are defined by the USAGE IS INDEX phrase. See Chapter 8, "Table Handling by the Indexing Method," for further discussion.

Qualification of Names

When a data-name, condition-name or paragraph-name (see Chapter 2, "Structure of a COBOL Program," for a description of paragraphs) is not unique, a specific instance of the name may be referenced by using qualifiers.

For example, if there were two or more items named YEAR, the qualified reference

```
YEAR OF HIRE
```

might differentiate between YEAR fields in HIRE and TERMINATION.

Qualifiers are data-names or condition-names preceded by "OF" or "IN". The qualifiers must designate broader-level groups that contain all the names in the reference. For example, HIRE must be a group item (or filename) containing an item called YEAR. Paragraph-names may be qualified by a section-name. (See Chapter 2, "Structure of a COBOL Program," for discussion of hierarchy.)

The maximum number of qualifiers allowed is: one for a paragraph-name; five for a data-name or condition-name. Filenames and mnemonic-names must be unique.

A qualified name may only be written in the SCREEN SECTION or PROCEDURE DIVISION. A reference to a multiply defined paragraph-name need not be qualified when the reference is made within the same section.

Literals

A literal is a constant. It is not assigned a data-name in a program, but is referred to only by its value, which does not change. Literals can be numeric, non-numeric (quoted), or figurative.

Numeric Literals

A numeric literal must contain at least one and not more than 18 digits. A numeric literal may consist of the characters 0 through 9, optionally preceded by a sign, and the decimal point. It may contain only one sign character and only one decimal point. The sign, if present, must appear as the left-most character in the numeric literal. If a numeric literal is unsigned, it is assumed to be positive.

A decimal point may appear anywhere within the numeric literal except as the right-most character. If a numeric literal does not contain a decimal point, it is considered to be an integer.

The following are examples of numeric literals:

```
72    +1011    3.14159    -6    -.333    0.5
```

European notation (period and comma interchanged) can be specified by including the DECIMAL-POINT IS COMMA entry in the Special Names portion of the ENVIRONMENT DIVISION. In European notation, for example, the numeric literal pi would be written as 3,14159.

Non-Numeric Literals

A non-numeric literal is also called a "quoted" literal. It is delimited by quotation marks or apostrophes, and may consist of any combination of characters in the ASCII set. Generally, if the literal is delimited by apostrophes, quotation marks may be used within the literal, and vice versa. However, the delimiter character can be used as a character within the literal if two such characters are consecutive. In such a case, the two characters are considered as one representation of the delimiter within the literal. For instance,

```
"THE DATA-NAME ""VALID-NO"" IS ACCEPTED HERE"
```

would be interpreted as

```
THE DATA-NAME "VALID-NO" IS ACCEPTED HERE
```

All spaces enclosed by the delimiters are included as part of the literal and are counted when the length is checked. Delimiters are not included in the length, which must be in the range 1 through 120.

The following are examples of non-numeric literals:

```
"ILLEGAL CONTROL CARD"

'CHARACTER-STRING'

"DO'S & DON'T'S"
```

Non-numeric literals may be continued from one line to the next. The following rules apply to the continuation line:

1. Column 7 of the continuation line must contain a hyphen.
2. Area A of the continuation line must be blank.
3. A delimiter must be entered in Area B, followed by the continuation of the literal.
4. All spaces at the end of the previous line and any spaces from the delimiter to the end of the continuation line are considered to be part of the literal.

Figurative Constants

A figurative constant is a special type of literal. It represents a value or character to which a reserved data-name has been assigned by MS-COBOL. When the program is compiled, that value or character will be provided as needed. For example, the figurative constant SPACE clears its entire field to blanks; LOW-VALUE enters the computer's lowest value. A figurative constant is not bounded by quotation marks.

In MS-COBOL, the reserved words listed below are figurative constants. The plural forms of the words are accepted by the compiler but are equivalent to the singular forms.

`ZERO`	may be used in many places in a program as a numeric literal. It may also be used in alphanumeric fields.
`SPACE`	represents the blank character.
`LOW-VALUE`	represents the computer's lowest value.
`HIGH-VALUE`	represents the computer's highest value.
`QUOTE`	represents the quotation mark.
`ALL literal`	indicates one or more instances of the literal, which must be a one-character non-numeric literal or a figurative constant. If the literal is a figurative constant, ALL is not necessary but is usually included for readability.

A figurative constant may be used anywhere a literal is called for in a general format, except where the literal is numeric only. In this case, the only figurative constant that can be used is ZERO.

Statements

Statements specify actions to be taken by the compiler. They usually consist of a verb, such as ACCEPT or MOVE, followed by operands that are data-names or literals. The compiler recognizes three kinds of statements: imperative, conditional, and compiler directing.

Imperative Statements

An imperative statement specifies an unconditional action to be taken by the program. Imperative statements appear only in the PROCEDURE DIVISION of the program. The verbs that can be used in imperative statements are:

```
ACCEPT      MOVE
ADD*        MULTIPLY*
CALL        OPEN
CLOSE       PERFORM
COMPUTE*    READ*
DELETE*     REWRITE*
DISPLAY     SET
DIVIDE*     START*
EXIT        STOP
GO          SUBTRACT*
INSPECT     WRITE*
```

*SIZE ERROR, INVALID KEY, or AT END clauses cannot be included in imperative statements.

See Chapter 6, "PROCEDURE DIVISION," for discussion of individual statements.

Conditional Statements

Conditional statements test for conditions and provide alternate paths of program execution. Conditional statements occur only in the PROCEDURE DIVISION of a program. The following verbs can be used for conditional statements:

```
ADD*          READ*
COMPUTE*      REWRITE*
DELETE*       START*
DIVIDE*       SUBTRACT*
IF            WRITE*
MULTIPLY*
```

*These verbs form conditional statements when used with SIZE ERROR, INVALID KEY, or AT END clauses.

Compiler Directing Statements

A compiler directing statement is a command to the compiler itself, rather than a functional part of the program. Compiler directing statements can be used anywhere in the ENVIRONMENT, DATA, or PROCEDURE DIVISIONS. The two verbs used as compiler directing statements are COPY and USE. COPY reads source lines from another file and inserts them in the original program. USE specifies procedures to be executed when input-output errors occur. See Chapter 2, "Structure of a COBOL Program," for discussion of these two verbs.

Arithmetic Statements

There are five arithmetic statements: ADD, SUBTRACT, MULTIPLY, DIVIDE and COMPUTE. Any arithmetic statement may be either imperative or conditional. Three optional clauses are available with arithmetic statements. They are: ON SIZE ERROR, ROUNDED, and GIVING. See Chapter 6, "PROCEDURE DIVISION," for detailed discussion of individual arithmetic statements and optional clauses.

When an arithmetic statement includes an ON SIZE ERROR specification, the entire statement is termed conditional, because the size-error condition is data-dependent. For example, the following arithmetic statement is conditional:

```
ADD 1 TO RECORD-COUNT
    ON SIZE ERROR
        MOVE ZERO TO RECORD-COUNT
        DISPLAY "LIMIT 99 EXCEEDED".
```

If a size error occurs (in this case, it is apparent that RECORD-COUNT has PICTURE 99, and cannot hold a value of 100), subsequent statements (i.e., MOVE and DISPLAY) are executed.

All data-names used in arithmetic statements must be elementary numeric data items that are defined in the DATA DIVISION of the program, except that operands of the GIVING option may be report (numeric edited) items. Index-names and index-items are not allowed in these arithmetic statements. (See Chapter 5, "DATA DIVISION," for definition of elementary items.)

Decimal point alignment is supplied automatically throughout arithmetic computations. Intermediate result fields are generated for the evaluation of arithmetic expressions. These intermediate fields assure the accuracy of the result, except where high-order truncation is necessary.

Arithmetic Expressions

An arithmetic expression is a combination of numeric literals, data-names, arithmetic operators, and parentheses. In general, the data-names in an arithmetic expression must designate numeric data. Consecutive data-names (or literals) must be separated by an arithmetic operator, and there must be one or more blanks on either side of the operator. The operators are:

```
 + for addition
 - for subtraction
 * for multiplication
 / for division
** for exponentiation to an integral power
```

When more than one operation is to be executed using a given variable or term, the order in which the operations are performed is:

1. the unary operators plus and minus (involving one variable only)
2. exponentiation
3. multiplication and division
4. addition and subtraction

Parentheses may be used to change the standard order of evaluation. Expressions within parentheses are evaluated first, and parentheses may be nested to any level. When parentheses are used in an expression, the following punctuation rules should be observed:

1. A left parenthesis is preceded by one or more spaces.
2. A right parenthesis is followed by one or more spaces.

The following examples illustrate the evaluation process:

Example 1: The expression

```
A + B / (C - D * E)
```

is evaluated in the following sequence:

1. compute the product D times E, considered as intermediate result R1
2. compute intermediate result R2 as the difference C - R1
3. divide B by R2, providing intermediate result R3
4. the final result is computed by addition of A to R3

Example 2: The same expression, without parentheses,

```
A + B / C - D * E
```

is evaluated as follows:

1. R1 = B / C
2. R2 = D * E
3. R3 = A + R1
4. the final result is R3 - R2

Example 3: The expression

```
A - B - C
```

is evaluated as:

(A - B) - C

Chapter 2

STRUCTURE OF A COBOL PROGRAM

Every COBOL source program is divided into four divisions.

The four divisions are:

1. IDENTIFICATION DIVISION, which names and documents the program.
2. ENVIRONMENT DIVISION, which indicates the computer equipment and features to be used in the program.
3. DATA DIVISION, which defines the names and characteristics of data to be processed.
4. PROCEDURE DIVISION, which consists of statements that direct the processing of data at execution time.

Each division must begin with a division header, and divisions must appear in the program in the preceding list.

Each program division consists of a particular arrangement of "grammatical" parts. In hierarchical order, these parts fit together as follows, with "division" as the highest level part:

Division
Region
Section
Paragraph
Sentence | Entry
Statement | Clause
Phrase | Option

When one of the levels shown in the preceding list contains multiple terms separated by a vertical bar (e.g., Sentence | Entry), the term that is used depends on which part of the program it occurs in. For example, the progression in the ENVIRONMENT DIVISION is: Division, Section, Paragraph; while the progression in the DATA DIVISION is: Division, Section, Entry, Clause. See Chapters 3 through 6 for general formats for each division.

The following list defines these terms and other MS-COBOL terms associated with them. The list starts with the lowest level in the hierarchy. For a more comprehensive glossary of COBOL terms, see the 1974 ANSI Standards document, American National Standard Programming Language COBOL, ANSI X3.23-1974, ISO 1989-1978, corrected edition July 1978.

1. Phrase

   A group of words that performs part of a procedural statement or clause. For example, the WRITE statement contains an optional INVALID KEY phrase which specifies a procedure that will be performed if an INVALID KEY condition exists.

2. Option

   Because most phrases are optional (as denoted by brackets in the general format), they are often referred to as options.

3. Statement

   A statement is an action that is to be performed. It includes a verb, one or more operands (data-names or literals) that are to be acted on, and any necessary phrases. The three kinds of statements — imperative, conditional, and compiler directing — are defined in Chapter 1, "Language Elements."

   All statements except COPY appear in the PROCEDURE DIVISION. The COPY statement may appear anywhere except the IDENTIFICATION DIVISION.

4. Clause

   A group of words that specify an attribute, or characteristic, of an entry in the DATA DIVISION. An entry may have multiple clauses.

5. Sentence

   A group of one or more statements. The last statement in the sentence is followed by a period (.) and a space. Like statements, sentences appear only in the PROCEDURE DIVISION.

6. Entry

   Entry is often used as a general term for anything that is "entered" in a particular place in a program. However, it does have a specific meaning in COBOL: an entry is a descriptive set of clauses, ending with a period. Unless stated otherwise, the specific meaning will be used in this manual. Entries may occur in the IDENTIFICATION, ENVIRONMENT, and DATA DIVISIONS.

7. Paragraph

   A paragraph is a group of related sentences (in the PROCEDURE DIVISION) or entries (in the IDENTIFICATION, ENVIRONMENT, or DATA DIVISIONS). A paragraph always starts with a paragraph-name or paragraph header.

   In some cases, a group of entries will constitute a section, rather than a paragraph. This happens, for example, in the FILE SECTION of the DATA DIVISION.

8. Section

   A set of related paragraphs or entries. A section always starts with a section header.

9. Region

   A set of PROCEDURE DIVISION sections. The only region in MS-COBOL is the DECLARATIVES REGION in the PROCEDURE DIVISION. It starts with the DECLARATIVES header and ends with END DECLARATIVES.

10. Division

    One of the four major functional parts of a COBOL program. A division always starts with a division header.

Level Numbers and Data-Items

In the DATA DIVISION of an MS-COBOL program, all entries except the FD (file description) entry are names and descriptions of data-items used in the program. These data-items can be group items, elementary items, or conditions. A group item has subordinate items within it; an elementary item does not.

Level numbers are a form of outline that shows how the data-items are related to each other. Group items can be at any level from 01 to 49. An item's status as a group or elementary item is determined solely by whether another item is subordinate to it. An item is a group item if a higher numbered level exists within the program before a lower or equal level is found. Subordinate items may themselves be either group items or elementary items.

Subordinate levels need not be consecutive, and numbers can be skipped to allow for later insertions.

The following example shows two levels of subordination:

```
01 TIME-CARD.
   02  NAME.
       03 LAST-NAME    PICTURE X(18).
       03 FIRST-INIT   PICTURE X.
       03 MIDDLE-INIT  PICTURE X.
   02  EMPLOYEE-NUM  PICTURE 99999.
   02  WEEKS-END-DATE.
       05  MONTH           PIC 99.
       05  DAY-NUMBER      PIC 99.
       05  YEAR            PIC 99.
   02  HOURS-WORKED  PICTURE 99V9.
```

In this example, all level 03 items are subordinate to the group item NAME (02), and all level 05 items are subordinate to group item WEEKS-END-DATE (02). The level 02 items are, in turn, subordinate to level 01.

Therefore, level 01 is a group item, all items in levels 03 and 05 are elementary items, level 02 items NAME and WEEKS-END-DATE are group items, and level 02 items EMPLOYEE-NUM and HOURS-WORKED are elementary items.

The level numbers assigned to various types of data-items are:

01-49	group and elementary items
77	items that are noncontiguous, are not subordinates of other items, and do not have subordinates
88	condition-names and conditions

The following rules apply to level numbers and data-items:

1. When a PROCEDURE DIVISION statement refers to a group item, the reference applies to the area reserved for the entire group, including all subordinate items.
2. In the FILE SECTION, consecutive 01 level numbers subordinate to any given file represent implicit redefinitions of the same area. In the WORKING-STORAGE SECTION, however, each 01 level number defines its own memory area, unless the REDEFINES clause is used.
3. When data-items are coded, level numbers 01, 77, and 88 are placed in Area A. The data-names for these items, and the level numbers and data-names for all subordinate items, begin in Area B.
4. All elementary items must be described with a PICTURE or USAGE IS INDEX clause.

See Chapter 5, "DATA DIVISION," for description of the various types of data-items.

Compiler Directing Statements

Two statements, COPY and USE, are included in this section of the manual because they are not confined to one division of a COBOL program. These statements are not really part of the source program; rather, they are direct instructions to the compiler.

COPY Statement

The COPY statement is used to logically embed the text of a disk file (other than the source file) in the source program. The COPY statement may be used anywhere in the ENVIRONMENT, DATA, or PROCEDURE DIVISIONS.

The format of the COPY statement is:

```
COPY file-name.
```

where filename is a disk filename in the format required by the operating system. For example, suppose BDEF.COB is a text file containing the following source code:

```
05 B.
    10 B1 PIC X.
    10 B2 PIC X.
```

Then a source file containing:

```
05 A.
    10 A1 PIC 9.
COPY BDEF.COB.
05 C.
    10 C1 PIC Z.
```

will compile exactly as if the following had been coded:

```
05 A.
    10 A1 PIC 9.
05 B.
    10 B1 PIC X.
    10 B2 PIC X.
05 C.
    10 C1 PIC Z.
```

The COPY statement should be the last or only statement on the line. Note that the entire statement containing the COPY verb, including the terminal period (.), is replaced by the contents of filename, so that any periods desired must be present in the copied file.

USE Statement

The USE statement specifies procedures for input-output error handling that are in addition to the standard procedures provided by the system. The USE statement itself is never executed; it merely defines the procedures that are to be executed under certain conditions.

See Chapter 12, "Declaratives and the USE Sentence," for details on the USE statement.

Chapter 3

IDENTIFICATION DIVISION

Every MS-COBOL program begins with the IDENTIFICATION DIVISION, which names the program and its author and describes other characteristics of the program.

Purpose To state the program name, author, and other characteristics.

Format The IDENTIFICATION DIVISION is divided into a header and accompanying paragraphs. The general format is:

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

Remarks Only the division header and the PROGRAM-ID paragraph are required. The other paragraphs are included only for documentation.

A period (.) is required at the end of the division header and at the end of each paragraph header.

Example The following example shows a typical IDENTIFICATION DIVISION, with the paragraphs in the order in which they are usually entered.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INVENTORY.
AUTHOR. M A HOWELL.
INSTALLATION. SHIPPING AND RECEIVING DIV.
DATE-WRITTEN. 1-15-83.
DATE-COMPILED. 1-20-83.
SECURITY. DEPT USE ONLY.
```

The paragraphs in the IDENTIFICATION DIVISION are discussed, in alphabetical order, in the remainder of this chapter.

AUTHOR Paragraph

Purpose The AUTHOR paragraph tells who wrote the program.

Format The general format is:

[AUTHOR. [comment-entry] ...]

Remarks This paragraph is optional, and is used for documentation only.

The name may not contain embedded periods (.).

Example

```
AUTHOR. M A HOWELL.
```

DATE-COMPILED Paragraph

Purpose Tells when the program was first compiled.

Format The general format is:

[DATE-COMPILED. [comment-entry] ...]

Remarks This paragraph is optional, and is used for documentation only.

Example

```
DATE-COMPILED. 1-20-83.
```

DATE-WRITTEN Paragraph

Purpose Tells when the program was written.

Format The general format is:

[DATE-WRITTEN. [comment-entry] ...]

Remarks This paragraph is optional, and is used for documentation only.

Example

```
DATE-WRITTEN. 1-15-80.
```

IDENTIFICATION DIVISION Header

Purpose Indicates the beginning of the program.

Format The general format is:

IDENTIFICATION DIVISION.

Remarks The IDENTIFICATION DIVISION header must be the first line of any MS-COBOL program.

The period (.) is required.

Example

```
IDENTIFICATION DIVISION.
```

INSTALLATION Paragraph

Purpose Tells how the program is used.

Format The general format is:

[INSTALLATION. [comment-entry] ...]

Remarks This paragraph is optional, and is used for documentation only.

Example

```
INSTALLATION. SHIPPING AND RECEIVING DIV.
```

PROGRAM-ID Paragraph

Purpose Tells the name of the object program created by the compiler.

Format The general format is:

PROGRAM-ID. program-name.

where program-name can be any alphanumeric string of characters, except that the first character must be a letter. Embedded periods (.) are not allowed. If the name contains more than one word, they must be separated by hyphens, rather than by spaces. Only the first six characters of the program-name are retained by the compiler.

Remarks This paragraph is required. It must be the first paragraph in the IDENTIFICATION DIVISION.

Example

```
PROGRAM-ID. INVENTORY.
```

SECURITY Paragraph

Purpose Tells the security level of the program.

Format The general format is:

[SECURITY. [comment-entry] ...].

Remarks This paragraph is optional, and is used for documentation only.

Example

```
SECURITY. DEPT USE ONLY.
```

Chapter 4

ENVIRONMENT DIVISION

The ENVIRONMENT DIVISION specifies the aspects of an MS-COBOL program that depend on the physical characteristics of the computer. This division is required in every program.

Purpose To specify aspects of the program that depend on the physical characteristics of the computer.

Format The ENVIRONMENT DIVISION always begins with a division header. The division may contain two sections: an optional CONFIGURATION SECTION, and an INPUT-OUTPUT SECTION that is required unless the program has no data files. Each of these sections is further divided into paragraphs, which may, in turn, be divided into clauses.

The general format is:

```
ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

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

OBJECT-COMPUTER. computer-name

    [ , MEMORY SIZE integer { WORDS      } ]
    [                       { CHARACTERS } ]
    [                       { MODULES    } ]

    [ , PROGRAM COLLATING SEQUENCE IS alphabet-name].

[SPECIAL-NAMES.

    [, PRINTER IS mnemonic-name]

    [, alphabet-name IS { STANDARD-1        } ]
    [                   { NATIVE            } ]
    [                   { implementor-name  } ]

    [, CURRENCY SIGN IS literal]

    [, DECIMAL-POINT IS COMMA]

    [ SWITCH-n IS comment-ID          ]
    [                                 ]
    [ { ON  } IS condition-name       ] .]
    [ { OFF }                         ]

[INPUT-OUTPUT SECTION.

FILE-CONTROL.

    {file-control-entry} ...

[I-O-CONTROL.

[ SAME [ RECORD     ] AREA FOR file-name-3 { , file-name-4 } ... ] ... .
[      [ SORT       ]                                            ]
[      [ SORT-MERGE ]                                            ]
```

Remarks The remainder of this chapter presents, in alphabetical order, the division and section headers, each of the paragraphs in the ENVIRONMENT DIVISION, and the SELECT, ASSIGN, and SAME AREA clauses.

Example

The following example shows a typical ENVIRONMENT DIVISION, with sections and paragraphs given in their order of appearance in the program.

```
ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER. IBM-PC.
OBJECT-COMPUTER. IBM-PC.
SPECIAL-NAMES. PRINTER IS LPRINTER.

INPUT-OUTPUT SECTION.

FILE-CONTROL.
    SELECT INVENTORY-MASTER-FILE
       ASSIGN TO DISK
       FILE STATUS IS MASTER-STATUS.
    SELECT INVENTORY-REPORT-FILE
       ASSIGN TO PRINTER.
I-O-CONTROL.
    SAME RECORD AREA FOR
        INVENTORY-MASTER-FILE,
        INVENTORY-REPORT-FILE.
```

ASSIGN Clause

Purpose Specifies that a file is to be used with a particular input or output device.

Format The ASSIGN clause always appears as part of the SELECT clause. It may be entered on the same line as the SELECT clause, but is generally entered on the following line and indented for readability.

The general format for SEQUENTIAL or LINE SEQUENTIAL file organization is:

```
          { DISK    }
ASSIGN TO { PRINTER }
```

The period (.) appears at the end of the sentence that comprises the entire SELECT clause. This means that if the ASSIGN clause is followed by optional clauses, the period will appear at the end of the entire sequence, rather than after ASSIGN.

The general format for RELATIVE and INDEXED files and for use with the Microsoft SORT Sorting Facility (MS-SORT) is:

```
ASSIGN TO DISK
```

See the entry for "SELECT Clause," for discussion of other clauses that appear in the SELECT clause.

CONFIGURATION SECTION Header

Purpose

Indicates the beginning of the CONFIGURATION SECTION. The type of computer being used and any special characteristics or names are specified in the CONFIGURATION SECTION.

Format

CONFIGURATION SECTION.

Remarks

The CONFIGURATION SECTION is optional.

The header must be entered as shown above, including the period (.). The header must begin in Area A.

The CONFIGURATION SECTION may contain three paragraphs:

```
SOURCE-COMPUTER
OBJECT-COMPUTER
SPECIAL-NAMES
```

The contents of the SOURCE-COMPUTER and OBJECT-COMPUTER paragraphs are treated as comments, except for the WITH DEBUGGING MODE clause of the SOURCE-COMPUTER paragraph. The SPECIAL-NAMES paragraph assigns user-defined names to system names such as PRINTER, and changes default editing characters. If any of these paragraphs are included in the program, the CONFIGURATION SECTION header must be entered.

For more information on these paragraphs, see the individual descriptions which follow in this chapter.

Example

```
ENVIRONMENT DIVISION.
  .
  .
  .
CONFIGURATION SECTION.
  .
  .
  .
```

ENVIRONMENT DIVISION Header

Purpose To indicate the beginning of the ENVIRONMENT DIVISION.

Format ENVIRONMENT DIVISION.

Remarks The ENVIRONMENT DIVISION is required, and always follows the IDENTIFICATION DIVISION. The ENVIRONMENT DIVISION header is also required. It must be entered exactly as shown above, including the period (.). It must begin in Area A.

See the introduction to this chapter for the general format of the ENVIRONMENT DIVISION.

Example

```
IDENTIFICATION DIVISION.
  .
  .
  .
ENVIRONMENT DIVISION.
  .
  .
  .
```

FILE-CONTROL Paragraph

Purpose Names the files to be processed in the program and associates them with specific input or output devices.

Format Each file whose records are described in the FILE SECTION of the DATA DIVISION must also be described in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION.

The general format for a FILE-CONTROL paragraph with SEQUENTIAL or LINE SEQUENTIAL file organization is:

```
FILE-CONTROL.

SELECT file-name

    ASSIGN TO { DISK    }
              { PRINTER }

    [; RESERVE integer [AREA ]]
                       [AREAS]

    [; ORGANIZATION IS [LINE] SEQUENTIAL]

    [; ACCESS MODE IS SEQUENTIAL]

    [; FILE STATUS IS data-name-1].
```

If MS-SORT is part of the user's MS-COBOL package, the following general format can be used:

```
FILE-CONTROL.

SELECT file-name

    ASSIGN TO DISK

    [SORT STATUS IS data-name-1].
```

Two other formats are available for INDEXED and RELATIVE file organizations. See Chapters 10 and 11 for these formats.

Remarks For the two formats given here, the SELECT and ASSIGN clauses are required; all other clauses are optional. See "ASSIGN Clause," and "SELECT Clause," for discussion of all clauses in the FILE-CONTROL paragraph.

Example

```
FILE-CONTROL.
    SELECT INVENTORY-RECORDS
        ASSIGN TO DISK.
    SELECT INVENTORY-REPORT
        ASSIGN TO PRINTER.
```

INPUT-OUTPUT SECTION Header

Purpose Indicates the beginning of the INPUT-OUTPUT SECTION.

Format

```
INPUT-OUTPUT SECTION.
```

This section header must be the first entry in the INPUT-OUTPUT SECTION, and must be entered exactly as shown above, including the period (.). The header must start in Area A.

Remarks The INPUT-OUTPUT SECTION is required unless the program has no data files. The section begins with the section header and contains two paragraphs, the FILE-CONTROL paragraph and the I-O-CONTROL paragraph. These two paragraphs define the file assignment parameters, including buffering. For more information on these paragraphs, see the individual listings in this chapter.

Example

```
ENVIRONMENT DIVISION.

CONFIGURATION SECTION.
 .
 .
 .
INPUT-OUTPUT SECTION.
FILE-CONTROL.
 .
 .
 .
I-O-CONTROL.
 .
 .
 .
```

I-O-CONTROL Paragraph

Purpose Specifies that physical buffer space is to be shared between two or more files.

Format

```
[I-O-CONTROL.

[ SAME [ RECORD     ] AREA FOR file-name-3 {, file-name-4} ... ] ...
       [ SORT       ]
       [ SORT-MERGE ]
```

Remarks The I-O-CONTROL paragraph is optional. It contains only the SAME AREA clause, which specifies the names of files that are to share the same logical record area so that memory space can be conserved.

The I-O-CONTROL paragraph header must be entered exactly as shown above, including the period (.). The header must begin in Area A. The SAME AREA clause usually begins in Area B for readability.

For further details see "SAME AREA clause."

Example

```
I-O-CONTROL.
    SAME RECORD AREA FOR
        INVENTORY-MASTER-FILE,
        INVENTORY-REPORT-FILE.
```

OBJECT-COMPUTER Paragraph

Purpose Identifies the computer on which the program is to be executed.

Format

```
OBJECT-COMPUTER. computer-name

    [ , MEMORY SIZE integer { WORDS      } ]
    [                       { CHARACTERS } ]
    [                       { MODULES    } ]

    [ , PROGRAM COLLATING SEQUENCE IS alphabet-name].
```

Remarks This paragraph is used for documentation only. The header must be entered exactly as shown above, including the period (.). The period must be followed by at least one space.

The MEMORY SIZE and PROGRAM COLLATING SEQUENCE clauses are optional. The memory size depends on the implementation of the specific computer. If the PROGRAM COLLATING SEQUENCE clause is used, the collating sequence associated with "alphabet-name" is used to determine the truth of non-numeric comparisons that are:

1. explicitly specified in relation conditions
2. explicitly specified in condition-name conditions
3. implicitly specified by the presence of a CONTROL clause in a report description

If the PROGRAM COLLATING SEQUENCE is not specified, the native collating sequence is used (the native sequence for MS-COBOL is ASCII).

Example

```
OBJECT-COMPUTER. IBM-PC,
    MEMORY SIZE 65535 CHARACTERS,
    PROGRAM COLLATING SEQUENCE IS ASCII.
```

SAME AREA Clause

Purpose

Specifies that two or more files are to use the same memory area during processing. This clause is generally used to save memory space.

Format

```
[ SAME [ RECORD     ] AREA FOR file-name-3 { , file-name-4 } ... ] ... .
       [ SORT       ]
       [ SORT-MERGE ]
```

Remarks

The SAME AREA clause comprises the entire I-O-CONTROL paragraph. This clause, and therefore the entire I-O-CONTROL paragraph, is optional.

The files named in a SAME AREA clause need not have the same organization or access. However, no file may be listed in more than one SAME AREA clause. More than one SAME AREA clause may be included in an I-O-CONTROL paragraph.

The SORT and SORT-MERGE options are available only with the MS-SORT facility.

Example

```
I-O-CONTROL.
    SAME RECORD AREA FOR
        INVENTORY-MASTER-FILE,
        INVENTORY-REPORT-FILE.
```

SELECT Clause

Purpose Specifies each file that will be accessed and that is described in the FILE SECTION of the DATA DIVISION.

Format Four general formats are available with the SELECT clause. The formats for SEQUENTIAL/LINE SEQUENTIAL organization and for the MS-SORT facility are given here. See Chapter 10 and 11 for the general formats for INDEXED and RELATIVE file organization.

The general format for SEQUENTIAL and LINE SEQUENTIAL files is:

```
SELECT file-name

    ASSIGN TO { DISK    }
              { PRINTER }

    [ ; RESERVE integer [ AREA  ] ]
                        [ AREAS ]

    [ ; ORGANIZATION IS [ LINE ] SEQUENTIAL ]

    [ ; ACCESS MODE IS SEQUENTIAL ]

    [ ; FILE STATUS IS data-name-1 ] .
```

The general format for use with the MS-SORT facility is:

```
SELECT file-name

    ASSIGN TO DISK

    [ SORT STATUS IS data-name-1 ] .
```

The SELECT clause must begin in Area B. After the SELECT clause is entered, the other clauses may be entered in any order. The preceding format shows them in the usual arrangement; they are described alphabetically below.

The following discussion applies to SEQUENTIAL and LINE SEQUENTIAL files. For discussion of the SELECT clause in INDEXED or RELATIVE files, see Chapters 10 and 11, respectively.

ACCESS MODE

The ACCESS MODE clause is optional for SEQUENTIAL files. This clause must begin in Area B, and is generally indented from the SELECT clause for readability.

ASSIGN

The ASSIGN clause is required. It specifies which device is to receive the file. The possibilities are DISK or PRINTER. This clause may appear on the same line with the SELECT clause, but is generally indented on a separate line for readability.

FILE STATUS

In the FILE STATUS clause, data-name refers to a two-character, alphanumeric item in the WORKING-STORAGE or LINKAGE SECTIONS of the DATA DIVISION. File status information will be placed in this item after an I-O statement. The left-hand character of data-name assumes the following values:

0 for successful completion
1 for end-of-file condition
2 for INVALID KEY (only
for INDEXED and RELATIVE files)
3 for a nonrecoverable (I-O) error
9 for special cases

The right-hand character of data-name is set to zero if no further status information exists for the previous I-O operation.

The following combinations of values are possible:

Left Character	**Right Character**	**Meaning**
0	0	OK
1	0	EOF
3	0	Permanent Error
3	4	Disk Space Full

For values of status-right when status-left has a value of 2 or 9, see Chapters 10 and 11.

In an OPEN INPUT or OPEN I-O statement, a File Status of 30 means "File Not Found."

ORGANIZATION

The ORGANIZATION clause specifies whether the file organization is SEQUENTIAL or LINE SEQUENTIAL. Both forms assume the records in the file are variable-length. With SEQUENTIAL organization, a two-byte count of the record length is followed by the actual record, for as many records as exist in the file. With LINE SEQUENTIAL organization, the record is followed by a carriage return/line feed delimiter, for as many records as exist in the file.

No COMP, COMP-0, or COMP-3 information should be written into a LINE SEQUENTIAL file because these data items may contain the same binary codes used for carriage return and line feed. This duplication would cause problems when the file was subsequently read.

RESERVE

The RESERVE clause is not functional in MS-COBOL, but is scanned for correct syntax. One physical block buffer is always allocated to the logical record area assigned to the RESERVE clause. This allows logical records to be spanned over physical block boundaries. For files assigned to PRINTER, the logical record area is used as the physical buffer as well.

Example

```
SELECT INVENTORY-MASTER-FILE
   ASSIGN TO DISK
   ORGANIZATION IS LINE SEQUENTIAL
   FILE STATUS IS MASTER-STATUS.
```

SOURCE-COMPUTER Paragraph

Purpose Specifies the computer on which the program is to be compiled.

Format SOURCE-COMPUTER. computer-name [WITH DEBUGGING MODE].

Remarks Except for the WITH DEBUGGING MODE clause, the contents of this paragraph are used for documentation only.

If the WITH DEBUGGING MODE clause is included, source program lines with "D" in column 7 (indicating a debug statement) are compiled. If the WITH DEBUGGING MODE clause is not included, these lines are ignored. See Chapter 6, "Dynamic Debugging Statements," for more information on debug statements.

Example

```
SOURCE-COMPUTER.
    IBM-PC WITH DEBUGGING MODE.
```

SPECIAL-NAMES Paragraph

Purpose

Assigns user-defined names to standard implementor names, such as PRINTER. This paragraph can also be used to change editing characters.

Format

```
[SPECIAL-NAMES.

    [, PRINTER IS mnemonic-name]

                           { STANDARD-1          }
    [, alphabet-name IS    { NATIVE              } ]
                           { implementor-name    }

    [, CURRENCY SIGN IS literal]

    [, DECIMAL-POINT IS COMMA]

    [ SWITCH-n IS comment-ID              ]
    [                                     ]
    [ { ON  } IS condition-name           ] .]
    [ { OFF }                             ]
```

Remarks

The SPECIAL-NAMES paragraph is optional, as is each individual clause within it.

The clauses in this paragraph are:

ALPHABET-NAME

The ALPHABET-NAME clause specifies the language conventions that are used. In MS-COBOL, the default is ASCII IS NATIVE. In MS-COBOL, STANDARD-1 and NATIVE are equivalent. Implementor-name refers to a name specified by the manufacturer of the computer.

The ASCII character set is given in Appendix D.

CURRENCY SIGN

In MS-COBOL, the default currency sign is the dollar sign ($). The user may change this sign by specifying a single-character, non-numeric literal in the CURRENCY SIGN clause. The designated character may not be a quotation mark, a digit (0-9), or any of the characters defined for PICTURE representations.

DECIMAL-POINT

The DECIMAL-POINT IS COMMA clause may be included to specify European notation. In European notation, the decimal point and comma are interchanged, so that the representation for pi, for example, is 3,14159.

PRINTER

The PRINTER IS clause allows a user-defined name to be used in the DISPLAY statement with the UPON phrase.

SWITCH-n

The SWITCH-n clause allows switches to be set at runtime. The maximum number of switches that can be set is 8. The user is prompted at runtime to enter the switch settings; the condition-name may then be used in a condition statement in the PROCEDURE DIVISION. The default setting is OFF.

Example

```
SPECIAL NAMES.
    PRINTER IS LPRINTER
    ASCII IS NATIVE
    CURRENCY SIGN IS "L"
    DECIMAL-POINT IS COMMA
    SWITCH-2 is TEST2
    ON IS ON2
    OFF IS OFF2.
```

Chapter 5

DATA DIVISION

The DATA DIVISION describes the data that the object program uses, creates, and produces as output.

Purpose Describes the data that will be used or produced by the object program.

Format The DATA DIVISION contains four sections:

```
FILE SECTION
WORKING-STORAGE SECTION
LINKAGE SECTION
SCREEN SECTION
```

The individual sections are discussed individually under "Sections" in this chapter.

The general format for the DATA DIVISION is:

```
DATA DIVISION.
[FILE SECTION.
FD file-name
    [; BLOCK CONTAINS [integer-1 TO] integer-2 {RECORDS   }]
                                               {CHARACTERS}]
    [; RECORD CONTAINS [integer-3 TO] integer-4 CHARACTERS]
    ; LABEL {RECORD IS  } {STANDARD}
            {RECORDS ARE} {OMITTED }
    [; VALUE OF FILE-ID IS {data-name-1}]
                           {literal-1  }]
    [; DATA {RECORD IS  } data-name-2 [, data-name-3] ...]
            {RECORDS ARE}
```

```
[; LINAGE IS {data-name-4} LINES [, WITH FOOTING AT {data-name-5}]]
             {integer-5  }                          {integer-6  }

    [, LINES AT TOP {data-name-6}] [, LINES AT BOTTOM {data-name-7}]
                    {integer- 7 }                     {integer-8  }

[ ; CODE-SET IS alphabet-name ] .

[record-description-entry] ...] ... ]

[SD file-name

[ ; RECORD CONTAINS [integer-1 TO] integer-2 CHARACTERS]

[; DATA {RECORD IS   } data-name-1 [, data-name-2] ... ]
        {RECORDS ARE }

[; VALUE OF FILE-ID IS {data-name-1}] .
                       {literal-1  }

[record-description-entry] ...] ... ]

[WORKING-STORAGE SECTION.

[77-level-description-entry]
[record-description-entry  ] ...]

[LINKAGE SECTION.

[77-level-description-entry]
[record-description-entry  ] ...]

[SCREEN SECTION.

level-number [screen-name]

    [BLANK SCREEN]

    [LINE NUMBER IS [PLUS] integer-1]

    [COLUMN NUMBER IS [PLUS] integer]

    [BLANK LINE]

    [BELL]

    [{UNDERLINE     }]
    [{REVERSE  VIDEO}]
    [{HIGHLIGHT     }]
    [{BLINK         }]

    { [ [VALUE] IS literal-1 ]                                                                 }
    { [{PICTURE} IS character-string {[FROM {literal-2   }] [TO identifier-2]}]               }
    { [{PIC    }                     {      {identifier-1}                   }]               }
    {                                {[USING identifier-3]                   }                }

        [BLANK WHEN ZERO]

        [{JUSTIFIED} RIGHT]
        [{JUST     }      ]

        [AUTO]

        [SECURE]

        [REQUIRED]

        [FULL]
```

where the general format for a record or data description entry, including level-77 entries, is:

```
               { data-name-1 }
level-number   { FILLER      }

     [ ; REDEFINES data-name-2 ]

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

     [                 { COMPUTATIONAL-0 } ]
     [                 { COMP-0          } ]
     [                 { COMPUTATIONAL   } ]
     [ ; [USAGE IS]    { COMP            } ]
     [                 { COMPUTATIONAL-3 } ]
     [                 { COMP-3          } ]
     [                 { DISPLAY         } ]
     [                 { INDEX           } ]

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

[ ; OCCURS integer-1 TIMES

     [ { ASCENDING  }  KEY IS data-name-4 [, data-name-5]... ] ...
     [ { DESCENDING }                                        ]

     [ INDEXED BY index-name-1 [, index-name-2] ...] ]

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

[ ; { JUSTIFIED }  RIGHT ]
[   { JUST      }        ]

[ ; BLANK WHEN ZERO ]

[ ; VALUE IS literal ] .
```

or, in the case of a level 88 entry:

```
                     { VALUE IS    }  { literal-1 [literal-2 ...]                    }
88 condition-name;   { VALUES ARE  }  { literal-1 [ { THROUGH }  literal-2 ]         }
                                      {            [ { THRU    }            ]         }
```

Remarks

The DATA DIVISION is a required part of an MS-COBOL program. It describes the data that were listed in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION. It also describes data used in the program that are not part of the input/output sections of the program (i.e., that are in the WORKING-STORAGE, LINKAGE, and SCREEN SECTIONS). These data are arranged in logical records. A logical record can be further divided into fields, or data-items. For example, an "Inventory-Master-File" declared in a FILE-CONTROL paragraph could contain one record for each piece of equipment inventoried. Each record could be further divided into data-items representing part-number, date-acquired, etc.

Within the DATA DIVISION, records are given level numbers of 01, and are declared in "record-description entries." Data-items are given level numbers 02 through 49 and are declared in "data-description entries." Note that a record-description entry includes all the data-description entries for that record.

For naming purposes, records are considered as data-items and follow the rules given in Chapter 1, "Language Elements," for data-names.

Level 77 and 88 entries may also be used in the DATA DIVISION. Level 77 entries describe "noncontiguous" data-items that do not fit into the group-elementary hierarchy. Level 88 entries describe conditions.

MS-COBOL meets all ANSI 74 Level 1 requirements and most of those from Level 2 for the FILE, WORKING-STORAGE, and LINKAGE SECTIONS. The COMMUNICATION and REPORT SECTIONS are not implemented. However, MS-COBOL provides, as an extension to the ANSI 74 Standard, the SCREEN SECTION, which describes the data used to set up an entire terminal screen.

This chapter, "DATA DIVISION," is arranged as follows:

The first part defines the various types of data-items that may be used in a program. These definitions are presented at the beginning of the chapter because they apply to all sections of the DATA DIVISION.

The physical limitations that apply to the DATA DIVISION in general are given in the next section. The last two parts of this chapter discuss the individual sections and clauses that are used in the DATA DIVISION.

The third section presents the sections, in the order in which they appear in a program.

The clauses that make up these sections are described alphabetically in the last section.

Example

```
DATA DIVISION.

FILE SECTION.

FD  INVENTORY-MASTER-FILE
    LABEL RECORDS ARE STANDARD
    VALUE OF FILE-ID IS "MASTER.DAT".

01  MASTER-RECORD.
    05  MSTR-KEY               PIC X(10).
    05  MSTR-DESCRIPTION       PIC X(25).
    05  MSTR-AMT-ON-HAND       PIC S9(5).
    05  MSTR-WARNING-LEVEL     PIC S9(5).

FD  INVENTORY-WARNING-FILE
    LABEL RECORDS ARE STANDARD
    VALUE OF FILE-ID IS "WARNING.DAT".

01  WARNING-RECORD            PIC X(45).

FD  INVENTORY-REPORT-FILE
    LABEL RECORDS ARE OMITTED
    LINAGE IS 56 LINES.

01  REPORT-RECORD            PIC X(80).

WORKING-STORAGE SECTION.

01  WORK-FIELDS.
    05  MASTER-STATUS      PIC XX
        VALUE SPACES.
    05  WARNING-STATUS     PIC XX
        VALUE SPACES.
    05  REC-COUNT          PIC S9(5)
        VALUE ZERO.
    05  WARNING-COUNT      PIC S9(5)
        VALUE ZERO.
    05  END-OF-FILE-SW     PIC X
        VALUE "N".
        88 END-OF-FILE
           VALUE "Y".
```

Data-Items and Data Descriptions

Chapter 1, "Language Elements," explains that data-items can be either group items having subordinate data elements, or elementary items, which do not have subordinates. Elementary items can be further classified and defined by their content as alphanumeric, numeric, report, or noncontiguous (level 77) items. Level 88 entries, which describe conditions, are also considered elementary items.

In the DATA DIVISION, each record-item or data-item in the program must be described in a separate record-description or data-description entry. Data-items must be entered in the order in which the items appear in the record.

For convenience, we will generally use the term "data-description entry" to mean both record-description entry and data-description entry. Note, however, that a record-description entry is a specific type of data-description entry. It always refers to data that begins with level number 01. We will use the term "record-description entry" when the data to be described must begin with level number 01.

The general format for a data-description entry is given in the introduction to this chapter. Every entry must contain a level number, data-name or the word "FILLER", and a series of clauses, followed by a period (.). All the clauses used in this division are given in the general format at the beginning of this chapter. Specific types of data-items, however, require certain clauses and cannot contain others. The following descriptions define the various types of group and elementary items and list the clauses that are mandatory and optional in the general format for each.

Group Items

A group item is any item that is further subdivided into elementary items or subordinate group items. The maximum size of a group item, including its subordinate items, is 4095 characters.

Optional clauses that may be used in the general format of a group item are:

```
REDEFINES
USAGE
OCCURS
SIGN
```

Example:

```
01 GROUP-NAME.
   02 FIELD-A PICTURE X.
   02 FIELD-B PICTURE X.
```

In this example, the level 02 elementary items are subordinate to the level 01 group item. The 01 level number also indicates the beginning of a new record; all items will be part of that record until another 01 level number is encountered.

Elementary Items

An elementary item is one that contains no subordinate items. Elementary items may be alphanumeric, numeric, report, noncontiguous, or conditions items.

Alphanumeric Items

An alphanumeric item consists of any combination of characters making a "character string" data field. If the associated PICTURE clause contains any of the editing characters discussed below, the item is an alphanumeric edited item.

An alphanumeric item must contain the PICTURE clause. The following clauses are optional:

```
OCCURS
REDEFINES
VALUE
USAGE
JUSTIFIED
SYNCHRONIZED
```

Example:

```
02 MISC-1 PICTURE X(53).
02 MISC-2 PICTURE BXXXBXXB.
```

In this example, MISC-1 may contain any combination of characters,

with a maximum of 53 characters. The "B" in MISC-2 is an edit character representing a space.

Report (Numeric Edited) Items

A report item is a receiving field for numeric data. It cannot be used as a numeric item itself in numeric calculations. For example, it might be a field named SALES-TOTAL where the calculated figure representing total sales is stored.

A report item contains only digits and/or special editing characters such as commas, dollar signs, etc. For this reason, it is sometimes called a numeric edited item. The maximum number of characters is 30.

Optional clauses for use with report items are:

```
REDEFINES
USAGE
OCCURS
VALUES
PICTURES
SYNCHRONIZED
BLANK WHEN ZERO
```

Example: `02 SALES-TOTAL PICTURE $$$$,$$9.99-.`

The minus (-) in the PICTURE clause represents the location of the operational sign of the calculated result. The dollar sign ($) will "float," i.e., only one dollar sign will appear in the result, one position to the left of the leftmost non-zero digit in SALES-TOTAL.

Numeric Items

A numeric item is an elementary item that contains numeric data only. There are four kinds of numeric items:

external decimal items
internal decimal items
binary items
index-data-items

These classifications are based on how the items are stored in memory.

1. External decimal item

   An external decimal item is one in which one character (byte) represents one digit. The maximum number of characters is 18.

   The exact number of digits in an item is defined by the PICTURE

clause in the item. For example, PICTURE 999 defines a three-digit item whose maximum value is 999.

If the value of PICTURE begins with the letter "S", the item will also contain an algebraic operational sign. This means that any data stored in the field as the result of a MOVE statement or an arithmetic statement will contain the algebraic sign of the result.

The sign does not occupy a separate byte unless the SEPARATE form of the SIGN clause is used in the general format.

The USAGE of an external decimal item is DISPLAY.

The PICTURE clause is required with external decimal items. The following clauses are optional:

```
REDEFINES
OCCURS
SIGN
USAGE
VALUE
SYNCHRONIZED
```

Examples:
```
02 HOURS-WORKED PICTURE 99V9
   USAGE IS DISPLAY.

02 HOURS-SCHED PICTURE S99V9
   SIGN IS SEPARATE TRAILING.
```

The "V" in the PICTURE clause represents an implied decimal point, and "S" represents an operational sign.

2. Internal decimal item

   An internal decimal item is one that is stored in packed decimal format. The USAGE IS COMPUTATIONAL-3 form of the USAGE clause specifies this format.

   An internal decimal item defined by n 9s in its PICTURE clause occupies 1/2 of (n + 2) (rounded down) bytes in memory. All bytes except the right-most contain a pair of digits, and each digit is represented by the binary equivalent of a valid digit value from 0 to 9. The item's low order digit and the operational sign are stored in the right-most byte of a packed item. For this reason, the compiler considers a packed item to have an arithmetic sign, even if the original PICTURE clause lacked an S-character.

The USAGE IS COMPUTATIONAL-3 clause is required for an internal decimal item. The optional clauses are:

```
REDEFINES
OCCURS
SIGN
VALUE
SYNCHRONIZED
```

Example:
```
05 TAX-RATE PICTURE S99V999
   VALUE 1.375
   USAGE IS COMPUTATIONAL-3.
```

3. Binary item

   A binary item uses the base 2 system to represent an integer in the range −32768 to 32767. A binary item occupies one 16-bit word. The left-most bit of the reserved area is the operational sign.

   A binary item is specified by the USAGE IS COMPUTATIONAL-0 form of the USAGE clause.

   The PICTURE and USAGE IS COMPUTATIONAL-0 clauses are required for binary items. The following clauses are optional:

```
REDEFINES
OCCURS
VALUE
SYNCHRONIZED
```

Note

In MS-COBOL for the Z80/8080 microprocessor, items with USAGE IS COMPUTATIONAL are also treated as binary items, unless the /V (Validation) compiler switch is used. In that case, and in the case of all other versions of MS-COBOL, items with USAGE IS COMPUTATIONAL are treated as external decimal (display) items.

Example:
```
03 YEAR-TO-DATE PICTURE S9(5)
   USAGE IS COMPUTATIONAL-0.
```

4. Index-data-items and index-names

   Index-data-names and index-names are used in table handling. An index-name is defined in the INDEXED BY phrase of the OCCURS clause. It is not declared in a separate WORKING-STORAGE SECTION entry. An index-name is associated with the table whose definition contains the OCCURS clause, and it cannot be used with any other table.

   An index-data-item is defined in a data-description entry with the USAGE IS INDEX clause. The PICTURE and VALUE IS clauses are not used in the index-data-item definition. An index-data-item is not associated with a particular table.

   Index-data-items and index-names have an implicit USAGE IS COMPUTATIONAL-0 (binary item) clause.

   See Chapter 8, "Table Handling by the Indexing Method," for more information about using index-data-items and index-names.

   In the general format of an index-data-item, the following clauses are optional:

```
REDEFINES
SIGN
SYNCHRONIZED
JUSTIFIED
BLANK
```

   Examples:

```
05 TABLE-ENTRY OCCURS 10 TIMES
   PIC 9
   INDEXED BY SUB-VAL.

05 SUB-VAL
   USAGE IS INDEX.
```

In the first example, SUB-VAL is implicitly declared as an index-name associated with TABLE-ENTRY. In the second example, SUB-VAL is declared explicitly, as an index-data-item, but is not associated with a particular table.

Level 77 (Noncontiguous) Items

Some data-items and constants may not be part of a hierarchical relationship in the program. These items are not grouped into logical records, and they are not subdivided. Instead, they are given level number 77 and are classed as "noncontiguous elementary items." They are sometimes called "stand-alone items."

Level 77 entries follow the naming conventions and general format for standard data-description entries. The PICTURE clause is required.

Level 77 entries may be used only in the WORKING-STORAGE and LINKAGE SECTIONS.

Level 88 (Condition) Items

A level 88 condition-name entry specifies a value, list of values, or a range of values that an elementary item may assume. If the specified value matches the value of its associated elementary item, the condition is true; otherwise it is false. For example, the elementary item

```
02 PAYROLL-PERIOD PICTURE IS 9.
```

may be followed by the level 88 entries

```
88 WEEKLY VALUE IS 1.
88 SEMI-MONTHLY VALUE IS 2.
88 MONTHLY VALUE IS 3.
```

In this case, either of the following conditions may be applied:

```
IF MONTHLY
    PERFORM P100-DO-MONTHLY.

IF PAYROLL-PERIOD = 3
    PERFORM P100-DO-MONTHLY.
```

The elementary item associated with a level 88 entry is called the "conditional variable."

A level 88 entry must be preceded either by another level 88 entry (in the case of several condition-names pertaining to an elementary item) or by an elementary item (which may be FILLER). Index data-items should not be followed by level 88 items.

The general format for a level 88 entry is:

```
                  { VALUE IS    }  { literal-1 [literal-2 ...]                  }
88 condition-name;{ VALUES ARE  }  { literal-1 [ { THROUGH }  literal-2 ]       }
                                   {             { THRU    }                    }
```

For an edited elementary item, the values in a condition-name entry must be expressed as non-numeric (quoted) literals.

A VALUE clause may not contain both a series of literals and a range of literals.

The following rules apply to level 88 condition-names:

1. Every condition-name may be qualified by the name of its associated elementary item and that elementary item's qualifiers.
2. A condition-name may be used in the PROCEDURE DIVISION in place of a simple relational condition.
3. A condition-name may pertain to an elementary item that requires subscripts. In this case, the condition-name, when written in the PROCEDURE DIVISION, must be subscripted according to the same requirements as the associated elementary item.
4. The type of literal in a condition-name entry must be consistent with the data type of its conditional variable.

DATA DIVISION Limitations

There is a limitation on the number of items in the WORKING-STORAGE, LINKAGE, and FILE SECTIONS of the DATA DIVISION. The sum:

```
(W/4096) + F + L
```

must be less than or equal to 14, where W is the size of WORKING-STORAGE in bytes (W/4096 is rounded up), F is the number of files described in the FILE SECTION, and L is the number of level 01 or 77 entries in the LINKAGE SECTION.

In a run unit consisting of a main program linked together with an arbitrary number of subprograms, up to 14 files may be defined in each program or subprogram, subject to the limitations on WORKING-STORAGE and LINKAGE SECTION items described above. However, MS-COBOL attempts to close all opened files at program termination. In the case of certain severe errors, MS-COBOL can keep track of, at the most, 40 files. Therefore, to ensure that files will be closed if possible, a run unit should not define more than 40 files.

Sections

The DATA DIVISION of an MS-COBOL program contains four sections:

```
FILE SECTION
WORKING-STORAGE SECTION
LINKAGE SECTION
SCREEN SECTION
```

These sections are described in the following paragraphs.

FILE SECTION

In the FILE SECTION of the DATA DIVISION, an FD (file definition) entry is included for every file that was declared with a SELECT entry in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION.

Purpose Supplies logical and physical descriptions of the files used in the program.

Format The general format for the FILE SECTION includes the FD entries (and SD entries, if MS-SORT is used):

```
DATA DIVISION.

[FILE SECTION.

FD file-name

     [; BLOCK CONTAINS [integer-1 TO ] integer-2 {RECORDS   }]
                                                 {CHARACTERS}

     [; RECORD CONTAINS [integer-3 TO ] integer-4 CHARACTERS ]

      ; LABEL {RECORD IS   } {STANDARD}
              {RECORDS ARE } {OMITTED }

     [; VALUE OF FILE-ID IS {data-name-1}]
                            {literal-1  }

     [; DATA {RECORD IS  } data-name-2 [, data-name-3] ...]
             {RECORDS ARE}

     [; LINAGE IS {data-name-4} LINES [, WITH FOOTING AT {data-name-5}]
                  {integer-5  }                          {integer-6  }

          [, LINES AT TOP {data-name-6}] [, LINES AT BOTTOM {data-name-7}]
                          {integer- 7 }                     {integer-8  }

     [ ; CODE-SET IS alphabet-name ] .

[record-description-entry] ...] ...]

[SD file-name

     [ ; RECORD CONTAINS [integer-1 TO] integer-2 CHARACTERS]

     [; DATA {RECORD IS  } data-name-1 [, data-name-2] ... ]
             {RECORDS ARE}

     [; VALUE OF FILE-ID IS {data-name-1}] .
                            {literal-1  }

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

Remarks

The FILE SECTION header begins the FILE SECTION. It is followed by a period (.). Following the header, FD (file definition) entries are included for each file that was described in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION. FD entries specify the size of the logical and physical records, the presence or absence of label records, the value of implementor-defined label items, names of the data records which make up the file, and the number of lines to be included on a logical printer page. The FD entry is terminated by a period (.).

In the general format of the FILE SECTION, FD entries may be followed by SD (SORT definition) entries. SD entries are applicable only for implementations of MS-COBOL that have the
MS-SORT facility. For information on SD entries, see the *Microsoft SORT Sorting Facility Reference Manual.*

The following rules must be observed in the FILE SECTION:

1. The level indicator FD identifies the beginning of a file description. It must be followed by the filename.
2. The clauses included in the FD entry may be in any order.
3. One or more record-description entries must follow the file description entry. Record-description entries are discussed in the first part of this chapter.

Example

```
FILE SECTION.

FD  INVENTORY-MASTER-FILE
    LABEL RECORDS ARE STANDARD
    VALUE OF FILE-ID IS "MASTER.DAT".

01  MASTER-RECORD.
    05  MSTR-KEY             PIC X(10).
    05  MSTR-DESCRIPTION     PIC X(25).
    05  MSTR-AMT-ON-HAND     PIC S9(5).
    05  MSTR-WARNING-LEVEL   PIC S9(5).
```

WORKING-STORAGE SECTION

The WORKING-STORAGE SECTION describes the data that are developed and processed internally. These data will not be part of the external data files.

Purpose Describes internally developed and internally processed data.

Format The WORKING-STORAGE SECTION includes the WORKING-STORAGE header and record-description and level 77 entries. The general format is:

```
[WORKING-STORAGE SECTION.

 [ 77-level-description-entry ]
 [ record-description-entry   ]  ...]
```

Record-description entries are described in the first part of this chapter.

Remarks Record-description and data-description entries included in this section may use level numbers 01 through 49, and 77. Level 77 entries are discussed in the first part of this chapter.

VALUE clauses, which are prohibited in the FILE SECTION, are allowed in the WORKING-STORAGE SECTION.

Example

```
WORKING-STORAGE SECTION.

01  WORK-FIELDS.
    05  MASTER-STATUS       PIC XX
        VALUE SPACES.
    05  WARNING-STATUS      PIC XX
        VALUE SPACES.
    05  REC-COUNT           PIC S9(5)
        VALUE ZERO.
    05  WARNING-COUNT       PIC S9(5)
        VALUE ZERO.
    05  END-OF-FILE SW      PIC X
        VALUE "N".
        88 END-OF-FILE
           VALUE "Y".
```

LINKAGE SECTION

The LINKAGE SECTION in a program is needed only if the program has been called from another program, and the CALL statement in the calling program contains a USING phrase. The LINKAGE SECTION describes data that are defined in the calling program and are referenced by both the calling and the called programs.

Purpose To describe data that are referenced by a calling and called program. This section may contain record-description entries, level 77, and level 88 entries.

Format The LINKAGE SECTION begins with a header, followed by record-description entries, level 77, and level 88 entries. The general format is:

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

See individual listings in this chapter for details on individual parts of the LINKAGE SECTION.

In the LINKAGE SECTION, the VALUE IS clause may only be used in level 88 entries.

Remarks No space is allocated in the program for data-items described in the LINKAGE SECTION. Instead, PROCEDURE DIVISION references to these data are resolved by equating the descriptions with data whose addresses are passed to the called program by the CALL statement. Note that for index-items, no such correspondence is established; index-names in the calling and called programs always refer to separate indices.

Data-items that are defined in the LINKAGE SECTION can only be referenced in the PROCEDURE DIVISION of the program if they are specified in the USING phrase of the PROCEDURE DIVISION header or are subordinate to operands in that header.

Because names in the LINKAGE SECTION cannot be qualified, they must be unique within the called program.

See Chapter 7, "Inter-Program Communication," for more information on the LINKAGE SECTION.

Example

```
LINKAGE SECTION.

01 SHARED-LIST.
   05  MSTR-DESCRIPTION   PIC X(25).
   05  MSTR-AMT-ON-HAND   PIC S9(5).
```

SCREEN SECTION

The SCREEN SECTION defines terminal screen formats and the associated data-items. Data-items are entered as group or elementary items, with level numbers 01 through 49.

Purpose To define terminal format and to describe the data-items entered in the fields on the screen.

Format The SCREEN SECTION header begins the section, followed by a period (.). Items are entered as group or elementary items, numbered 01 through 49. Elementary screen items define individual display and/or data entry fields on the screen. Group items name any group of elementary screen items that are accepted or displayed with a single ACCEPT or DISPLAY statement.

The general format for an elementary screen item is:

```
[SCREEN SECTION.
 level-number [screen-name]
     [BLANK SCREEN]
     [LINE NUMBER IS [PLUS] integer-1]
     [COLUMN NUMBER IS [PLUS] integer]
     [BLANK LINE]
     [BELL]
     [ { UNDERLINE      } ]
     [ { REVERSE VIDEO  } ]
     [ { HIGHLIGHT      } ]
     [ { BLINK          } ]
     { [ [VALUE] IS literal-1 ]                                                               }
     { [ {PICTURE} IS character-string { [FROM {literal-2   } ] [TO identifier-2] } ]        }
     { [ {PIC    }                     {       {identifier-1}                     } ]        }
     {                                 { [USING identifier-3]                     }          }
       [BLANK WHEN ZERO]
     [ {JUSTIFIED} RIGHT ]
     [ {JUST     }       ]
       [AUTO]
       [SECURE]
       [REQUIRED]
       [FULL]
```

The general format for a group screen item is:

```
level-number [screen-name]
    [AUTO]
    [SECURE]
    [REQUIRED]
    [FULL]
```

Remarks

The clauses used in these formats are discussed under "Clauses" in this chapter. Clauses may be entered in any order. If the PICTURE clause is included, either USING or at least one of FROM and TO must be present. The AUTO and SECURE clauses may be used only if the PICTURE clause is also present.

The clauses that are specified with elementary data-items affect data input and display operations when ACCEPT and DISPLAY statements are executed at runtime. These effects are discussed under the description of the individual clauses.

With screen items, the following actions are always executed in the order shown below, regardless of the order in which they are specified:

```
BLANK SCREEN
LINE/COLUMN positioning
BLANK LINE
DISPLAY or ACCEPT data
```

Example

```
IDENTIFICATION DIVISION.

PROGRAM-ID. DOCTST.

ENVIRONMENT DIVISION.

DATA DIVISION.

WORKING-STORAGE SECTION.

01  WORK-FIELDS.

    05  WS-PASSWORD PIC X(10)
        VALUE "ABCDEFGHIJ".
    05  WS-PART-NO PIC S9(7)
        VALUE 1234567.
    05  WS-DESCRIPTION PIC X(25)
        VALUE "ASDSDKSDDDASABCDEHIJ".
    05  WS-UNIT-COST PIC S99V99
        VALUE 12.34.
    05  WS-QTY-ON-HAND PIC S999
        VALUE 987.

SCREEN SECTION.

01  BLANK-SCREEN.

    02  BLANK SCREEN.

01  INVENTORY-SCREEN.

    05  LINE 1 COLUMN 1
        VALUE "ENTER PASSWORD ".
    05  COLUMN PLUS 1 PIC X(10) SECURE
        USING WS-PASSWORD.
    05  LINE 2 COLUMN 1
        VALUE "PART NUMBER "
        HIGHLIGHT.
    05  SCR-PART-NO
        LINE 2 COLUMN 13 PIC S9(7)
        USING WS-PART-NO.
    05  LINE 3 COLUMN 1
        VALUE "DESCRIPTION ".
    05  COLUMN PLUS 1 PIC X(25)
        USING WS-DESCRIPTION.
```

```
    05  FOURTH-LINE AUTO.
        10  LINE 4 BLANK LINE.
        10  COLUMN 1 VALUE "UNIT-COST".
        10  COLUMN PLUS 1 PIC S999V99
            BLANK WHEN ZERO
            USING WS-UNIT-COST.
        0  COLUMN PLUS 4
            VALUE "QTY ON HAND".
        10  COLUMN PLUS 1 PIC S999
            BLINK
            USING WS-QTY-ON-HAND.

PROCEDURE DIVISION.

010-MAINLINE.
    DISPLAY BLANK-SCREEN.
    DISPLAY INVENTORY-SCREEN.
    ACCEPT INVENTORY-SCREEN.
    STOP RUN.
```

Clauses

The remainder of this chapter describes the clauses that may be used within the DATA DIVISION. The clauses are arranged here in alphabetical order. For information about how the clauses are used in the general format, see descriptions of the DATA DIVISON sections in the preceding section of this chapter.

The following rules apply to the use of clauses:

1. Clauses may appear in any order except that a REDEFINES clause, if used, must come first.
2. A clause included in a group item applies to all items within that group.
3. If a clause is entered at the group level, it may not be contradicted by a clause in an item that is subordinate to that group.

AUTO Clause

Purpose Specifies that when a field on a screen has been filled by user input, the cursor automatically skips to the next input field, rather than waiting for a terminating character. The ACCEPT statement is terminated when the last input field is accepted.

Format The AUTO clause appears as part of the SCREEN SECTION of the DATA DIVISION.

The general format is:

[AUTO]

Remarks AUTO is effective only when an ACCEPT statement is active during execution of the program.

Example

```
05 LINE 2 COLUMN 13 PIC S9(7)
   TO WS-QTY-ON-HAND
   AUTO.
```

BELL Clause

Purpose Sounds the terminal's audio alarm.

Format The BELL clause appears as part of the SCREEN SECTION of the DATA DIVISION.

The general format is:

```
[BELL]
```

Example

```
05 LINE 1 COLUMN 1
   VALUE "SOUND THE ALARM"
   BELL.
```

BLANK LINE Clause

Purpose Erases the screen from the current cursor position to the end of the current physical line.

Format The BLANK LINE clause appears in the SCREEN SECTION of the DATA DIVISION.

The general format is:

```
[BLANK LINE]
```

Remarks The area of the screen in which the specified line appears is cleared. No data are affected.

Example

```
05 LINE 10 BLANK LINE.
```

BLANK SCREEN Clause

Purpose Erases the entire screen and places the cursor at home position (line 1, column 1).

Format The BLANK SCREEN clause appears in the SCREEN SECTION of the DATA DIVISION.

The general format is:

```
[BLANK SCREEN]
```

Remarks Anything appearing on the screen is erased, but no data are affected.

Example

```
05 BLANK SCREEN.
```

BLANK WHEN ZERO Clause

Purpose Specifies that an item is displayed as spaces (i.e., is left blank) when its value is zero.

Format The BLANK WHEN ZERO clause may appear in any section with the DATA DIVISION.

The general format is:

```
[BLANK WHEN ZERO]
```

Example

```
05 UNIT-COST PIC S999V99 BLANK WHEN ZERO.
```

BLINK Clause

Purpose Specifies that an item is flashing and is shown in high-intensity (highlight) when displayed on the screen.

Format The BLINK clause appears as one of a choice of items in the SCREEN SECTION of the DATA DIVISION. The other choices are: UNDERLINE, REVERSE-VIDEO, and HIGHLIGHT.

The general format is:

```
[ { UNDERLINE      } ]
[ { REVERSE VIDEO  } ]
[ { HIGHLIGHT      } ]
[ { BLINK          } ]
```

Example

```
10 COLUMN PLUS 1 PIC S999 BLINK USING WS-QTY-ON-
HAND.
```

BLOCK Clause

Purpose

Specifies the size of the physical records in the file. Because this clause is normally used only for tape files, it is not functional in MS-COBOL. If it is present, however (e.g., if included for transferability), the syntax is checked.

Format

The BLOCK clause appears in an FD entry in the FILE SECTION.

The general format is:

```
[; BLOCK CONTAINS [integer-1 TO ] integer-2 { RECORDS    }]
                                            { CHARACTERS }
```

Remarks

If the BLOCK clause is specified, the following rules apply:

1. Files assigned to PRINTER must not have a BLOCK clause in the associated FD entry.
2. The size of a physical block should be stated in RECORDS, except when the records are variable in size or exceed the size of a physical block; in these cases the size should be expressed in CHARACTERS.

Example

```
FD MASTER-INV-FILE
   BLOCK CONTAINS 5 RECORDS.
```

CODE-SET Clause

Purpose

Specifies the character code set used to represent the data on the external media. In MS-COBOL, this clause is used for documentation only.

Format

The CODE-SET clause appears in the FD entry of the FILE SECTION. The specified code set for MS-COBOL is always ASCII.

The format is:

[; CODE-SET IS alphabet-name] .

Remarks

The CODE-SET clause should be specified only for non-mass-storage files.

When the CODE-SET clause is used, USAGE IS DISPLAY must also be specified. If the file contains signed numeric data, the SIGN IS SEPARATE clause must also be specified.

Example

```
FD INV-RECORD-FILE
   CODE-SET IS ASCII . . .
```

COLUMN Clause

Purpose Sets the cursor's column position on the screen.

Format The COLUMN clause appears in the SCREEN SECTION of the DATA DIVISION.

The general format is:

```
[COLUMN NUMBER IS [PLUS] integer]
```

Remarks The COLUMN and LINE clauses determine the screen location associated with an elementary screen item. As the SCREEN SECTION is processed at compile time, a current cursor position is maintained so that each elementary screen item can be identified with a particular region of the screen. When a level 01 screen item is encountered, the current screen position is reset to line 1, column 1. Then, as each item is processed, the current position is adjusted for the size of each definition encountered. By default, therefore, successively defined fields appear end to end on the screen.

The current column or line at the start of any elementary screen data-description may be changed with the COLUMN and LINE clauses. If neither clause is specified, the current screen position is not changed. If only COLUMN is specified, the line is not changed. If only LINE is specified, column 1 is assumed.

The COLUMN or LINE clause without PLUS causes the specified integer to be taken as the line or column at which the current screen item should start. When the PLUS phrase is specified, the specified integer is added to the current column or line, and the result is the column or line at which the current screen item starts. If the integer is not specified, COLUMN/LINE PLUS 1 is assumed.

See also "LINE Clause."

Example

```
05 COLUMN PLUS 4
   VALUE "QUANTITY ON HAND".
```

DATA RECORD(S) Clause

Purpose Names the records in a file. This clause is documentary only.

Format The DATA RECORD(S) clause appears in FD and SD entries in the FILE SECTION. Note that SD entries are used only with the MS-SORT facility.

The general format is:

```
[ ; DATA { RECORD IS    } data-name-1 [, data-name-2] ... ]
         { RECORDS ARE  }
```

Remarks Each record in the file is assigned a data-name (e.g., data-name-1, data-name-2, etc.). The records may be of different sizes, formats, etc. The data-names may be listed in any order.

Each data-name must have a corresponding 01 level number record-description entry, with the same data-name.

Example

```
FD RECORD-NAME
   DATA RECORDS ARE TOOLS-1, TOOLS-2
    .
    .
    .
   01 TOOLS-1.
      05 PART-NO PIC 9(8).
      05 DESCRIPTION PIC X(25).
      05 QTY PIC 999.

   01 TOOLS-2.
      05 PART-NO PIC 9(8).
      05 COST PIC 9(9)V99.
```

FROM/TO/USING Clause

Purpose

When a data-item is displayed on a screen, FROM or USING moves the contents of the data-item or a literal from storage to a temporary item that is defined by the PICTURE clause. This value is then displayed on the screen.

When an item is accepted, TO or USING implicitly moves the contents of the item to the data-item named in the TO or USING clause.

Format

The FROM/TO/USING clause appears in the SCREEN SECTION of the DATA DIVISION, and is part of the PICTURE clause for a data-item associated with a screen.

The general format is:

```
[ {PICTURE} IS character-string { [FROM {literal-2   } ] [TO identifier-2] } ]
[ {PIC    }                     {       {identifier-1}                    } ]
[                               { [USING identifier-3]                    } ]
```

Remarks

Identifiers may be qualified but not subscripted.

Note that the FROM and TO clauses are used together; USING, in effect, combines the two.

Examples

```
05 LINE 1 PIC S9(5)
   USING WS-PART-NO.

05 SCR-DESC PIC X(25)
   FROM LS-DESCRIPTION.

05 COLUMN PLUS 1 PIC X(10)
   TO FILE-IDENT.
```

The first example references data in the WORKING-STORAGE SECTION; the second example references data in the LINKAGE SECTION; and the third example references data in the FILE SECTION.

FULL Clause

Purpose When a data-item is accepted from a screen, FULL causes any terminator characters to be ignored.

Format The FULL clause is used in the SCREEN SECTION of the DATA DIVISION.

The general format is:

```
[FULL]
```

Example

```
05 LINE 3 PIC X(5)
   TO WS-IDENT-NO
   FULL.
```

HIGHLIGHT Clause

Purpose Specifies that an item is shown in high-intensity when displayed on the screen.

Format The HIGHLIGHT clause appears as one of a choice of items in the SCREEN SECTION of the DATA DIVISION. The other choices are: BLINK, UNDERLINE, and REVERSE-VIDEO.

The general format is:

```
[ { UNDERLINE     } ]
[ { REVERSE VIDEO } ]
[ { HIGHLIGHT     } ]
[ { BLINK         } ]
```

Example

```
05 LINE 2 VALUE "ENTER UNIT COST"
   HIGHLIGHT.
```

JUSTIFIED Clause

Purpose Specifies right-to-left alignment when the field is the receiving field for a MOVE statement.

Format The JUSTIFIED clause may appear in any section of the DATA DIVISION. The abbreviated form JUST is allowed.

The general format is:

```
[ { JUSTIFIED }  RIGHT ]
[ { JUST      }        ]
```

Remarks The JUSTIFIED clause applies only to unedited alphanumeric items. It can be used only for elementary items.

When data are moved to a longer receiving field, the data are aligned right-to-left, with space fill on the left. When the receiving field is shorter than the data, truncation occurs from the left.

Example

```
05 ALPHA-ITEM PIC X(20)
   JUSTIFIED RIGHT.
```

LABEL RECORD(S) Clause

Purpose Indicates whether a file contains labels.

Format The LABEL RECORD(S) clause appears in the FD entry of the FILE SECTION.

The general format is:

```
         { RECORD IS   }  { STANDARD }
; LABEL  { RECORDS ARE }  { OMITTED  }
```

OMITTED specifies that no labels exist for the file. OMITTED must be specified for files assigned to PRINTER.

STANDARD specifies that labels exist for the file and that they conform to system specifications. STANDARD must be specified for files assigned to DISK.

Remarks This clause is required in every FD entry.

Example

```
FD INVENTORY-WARNING-FILE
   LABEL RECORDS ARE STANDARD.
```

LINAGE Clause

Purpose Specifies the total number of lines assigned to a printed page, the number of lines allotted for top and bottom margins, and the line number at which the footing (information printed at the bottom of the page) begins.

Format The LINAGE clause appears in the FD entry of the FILE SECTION for a file assigned to the printer.

The general format is:

```
[; LINAGE IS {data-name-4} LINES [, WITH FOOTING AT {data-name-5}]
             {integer-5  }                         {integer-6  }

    [, LINES AT TOP {data-name-6}] [, LINES AT BOTTOM {data-name-7}]
                    {integer-7  }                     {integer-8  }
```

Remarks All data-names refer to unsigned numeric integer data-items. Integer-1 must be greater than zero, and integer-2 must not be greater than integer-1.

The total page size is the sum of the values in each phrase except for FOOTING. If TOP or BOTTOM margins are not specified, their size is assumed zero. The footing area is the part of the page between the line indicated by the FOOTING value and the last line of the page.

The values in each phrase at the time the file is opened (by the execution of an OPEN OUTPUT statement) specify the number of lines in each of the sections of the first logical page. Whenever a WRITE statement with the ADVANCING PAGE phrase is executed or a "page overflow" condition occurs (see Chapter 6, "The WRITE Statement"), the values in the phrases will be used for the next logical page.

A LINAGE-COUNTER is automatically created by the presence of a LINAGE clause. The value in the LINAGE-COUNTER at any given time represents the line number at which the printer is positioned within the current page. LINAGE-COUNTER may be referenced but may not be modified by PROCEDURE DIVISION statements. It is automatically modified during execution of a WRITE statement, according to the following rules:

1. When the "ADVANCING PAGE" phrase of the WRITE statement is specified or a "page overflow" condition occurs (see Chapter 6, "The WRITE Statement"), the LINAGE COUNTER is reset to one.
2. When the "ADVANCING identifier or integer" phrase is specified, LINAGE-COUNTER is incremented by the ADVANCING value.
3. When the ADVANCING phrase is not specified, LINAGE-COUNTER is incremented by one.

Example

```
FD INVENTORY-REPORT-FILE
    LABEL RECORDS ARE OMITTED
    LINAGE IS 56 LINES
    LINES AT TOP 3
    LINES AT BOTTOM 5.
```

LINE Clause

Purpose Sets the cursor's line position on the screen.

Format The LINE clause appears in the SCREEN SECTION of the DATA DIVISION.

The general format is:

[LINE NUMBER IS [PLUS] integer-1]

Remarks The COLUMN and LINE clauses determine the screen location associated with an elementary screen item. As the SCREEN SECTION is processed at compile time, a current cursor position is maintained so that each elementary screen item can be identified with a particular region of the screen. When a level 01 screen item is encountered, the current screen position is reset to line 1, column 1. Then, as each item is processed, the current position is adjusted for the size of each definition encountered. By default, therefore, successively defined fields appear end-to-end on the screen.

The current column or line at the start of any elementary screen data-description may be changed with the COLUMN and LINE clauses. If neither clause is specified, the current screen position is not changed. If only COLUMN is specified, the line is not changed. If only LINE is specified, column 1 is assumed.

The COLUMN or LINE clause without PLUS causes the specified integer to be taken as the line or column at which the current screen item should start. When the PLUS phrase is specified, the specified integer is added to the current column or line, and the result is the column or line at which the current screen item starts. If the integer is not specified, COLUMN/LINE PLUS 1 is assumed.

Example

```
05 LINE 1 PIC 999
   USING WS-QUANTITY.
```

OCCURS Clause

Purpose Specifies the number of times that a data-item is repeated in a record, i.e., within the associated level 01 group.

Format The OCCURS clause may not be used in any level number 01 or 77 entry. It may be used in the FILE, WORKING-STORAGE, or LINKAGE SECTIONS.

The general format is:

```
[ ; OCCURS integer-1 TIMES

    [ { ASCENDING  }  KEY IS data-name-4 [ , data-name-5]... ] ...
      { DESCENDING }

    [ INDEXED BY index-name-1 [, index-name-2] ...] ]
```

The KEY and INDEXED phrases specify key-names that can be used in SEARCH statements to access the items within a table. In the KEY phrase, ASCENDING/ DESCENDING specifies how the items are arranged according to their values. See Chapter 8, "Table Handling by the Indexing Method," for more information about the KEY and INDEXED phrases.

Remarks The OCCURS clause defines related sets of repeated data, such as tables, lists, and arrays. It specifies the number of times, up to a maximum of 1023, that a data-item with the same format is repeated in the record. The entire data-description entry applies to each repetition of the entry.

When the OCCURS clause is used in an entry, the data-name for the entry must be subscripted or indexed whenever it appears in the PROCEDURE DIVISION. If this data-name is the name of a group item, all data-names belonging to the group must be subscripted or indexed whenever they are used.

Subscripting enables the user to refer to a table or list of data-items that have not been assigned individual data-names. This is the case for items that have been specified by an OCCURS clause; therefore, any item that contains an OCCURS clause or belongs to a group containing an OCCURS clause must be subscripted or indexed whenever it is used. The one exception is in a SEARCH statement, where a table-name must be referenced without subscripts. See Chapter 8, "Table Handling by the Indexing Method," for an explanation of indexing.

A subscript is a positive, nonzero integer whose value indicates which element is referenced within a table or list. The subscript may be either a literal or a data-name whose value is an integer. A subscript must be a decimal or binary item (USAGE IS DISPLAY or COMPUTATIONAL-0; the latter is recommended for efficiency).

A subscript is always enclosed by parentheses. In the general format, it is given after the terminal space of the name of the element. Multiple subscripts are separated by a comma and a space (e.g., ITEM (3, 4)).

A data-name may not be subscripted if it is being used for:

1. a subscript
2. the defining name of a data-description entry
3. data-name-2 in a REDEFINES clause
4. a qualifier

Example

```
01 ARRAY.
    03 TABLE-VAL OCCURS 3 TIMES PIC 9(4).
```

In this example, storage would be allocated as follows:

```
    TABLE-VAL (1)
    TABLE-VAL (2)
    TABLE-VAL (3)
```

These three occurrences make up the ARRAY, which consists of 12 characters (each TABLE-VAL has 4 digits).

PICTURE Clause

Purpose Describes the contents of an elementary data-item. May also describe editing features of the item.

The abbreviation PIC is allowed.

Format The general format is:

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

In the SCREEN SECTION, the PICTURE clause must be followed by the USING clause, or one or both of the FROM and TO clauses.

Character-strings are discussed in the remarks which follow.

See the discussion of the FROM/TO/USING clause.

Remarks The character-string specification differs for alphanumeric, numeric, and report-edited items. These differences are described in the following paragraphs.

Alphanumeric Items

The PICTURE clause of an alphanumeric item may combine characters X, A, and 9. It may also contain the editing characters B, 0 and /.

An X indicates that the character position may contain any character from the computer's ASCII character set. A PICTURE clause that contains at least one of the combinations:

A and 9
X and 9
X and A

in any order, is considered as if every 9, A, or X character were X. The characters B, 0, and / may be used to insert blanks, zeros, or slashes in the item. The item is then called an alphanumeric-edited item.

If the string contains only A's and B's, it is considered alphabetic; if it has only 9's, it is numeric (see below). The NUMERIC and ALPHABETIC class tests may be used to determine whether an alphanumeric data-item is alphabetic or numeric.

Numeric Items

The PICTURE clause of a numeric item may combine the following characters:

9 Indicates that the actual or conceptual digit position contains a numeric character. The maximum number of 9's in a PICTURE clause is 18.

V Indicates the position of an assumed decimal point. This character is optional. Since a numeric item cannot contain an actual decimal point, the assumed decimal point provides the compiler with information about the scaling alignment of items involved in computations. Storage is never reserved for the character V. Only one V is permitted in any single PICTURE clause. V is redundant if it is the rightmost character.

S Indicates that the item has an operational sign. This character is optional. It must be the first character of the PICTURE clause. See also the "SIGN Clause."

P Indicates an assumed decimal scaling position. This character is optional. It specifies the location of an assumed decimal point when the point is not within the number that appears in the data item.

The scaling position character P is not counted in the size of the data item, and memory is not reserved for it. However, scaling position characters are counted in determining the maximum number of digit positions (18) in numeric edited items or in items that appear as operands in arithmetic statements.

If the clause contains more than one P, the P's must be continuous. The character P may appear only to the left or right of the other characters in the string, except that it may appear to the left of a leftmost string of P's. P implies an assumed decimal point to the left of the P's if the P's are leftmost, and to the right of the P's if the P's are rightmost. Therefore, the assumed decimal point symbol V is redundant as either the leftmost or rightmost character within a PICTURE clause that contains P's.

Report Items

A report item is a data item that can be used as an "edited" receiving field for a numeric value. The editing characters that may be combined to describe a report item are:

```
9 V . Z CR DB , $ + * B 0 - P /
```

The characters 9, P, and V have the same meaning as for a numeric item. The other editing characters are used as follows:

. The decimal point specifies that an actual decimal point is to be inserted in the indicated position and that the source item is to be aligned accordingly. Numeric character positions to the right of an actual decimal point in a PICTURE clause must consist of characters of one type. The decimal point must not be the last character in the PICTURE character-string. The decimal point and P may not be used in the same PICTURE clause.

Z
* The characters Z and * are replacement characters. Each one represents a digit position. During execution, leading zeros to be placed in positions defined by Z or * are suppressed, becoming blank or * respectively. Zero suppression ends when a decimal point (. or V) or a non-zero digit is encountered. All digit positions to be modified must be the same (either Z or *), and must be contiguous starting from the left. Z or * may appear to the right of an actual decimal point only if all digit positions are the same.

CR CR and DB are credit and debit symbols
DB respectively. They may appear only as the rightmost characters in a PICTURE clause. These symbols occupy two character positions. They indicate that the specified symbol is to appear in the indicated positions if the value of a source item is negative. If the value is positive or zero, spaces will appear instead. CR and DB and + and − are mutually exclusive.

, The comma specifies insertion of a comma between digits. Each comma is counted in the size of the data-item, but does not represent a digit position. The comma may also appear in conjunction with a floating string, as described below. It must not be the last character in the PICTURE character-string.

A floating string is a leading, continuous series of either dollar sign, plus sign or minus sign; or a string composed of one such character interrupted by one or more commas and/or decimal points. For example:

```
$$,$$$,$$$
++++
--,---,---
+(8).++
$$,$$$.$$
```

A floating string containing N + 1 occurrences of $ or + or − defines N digit positions. When a numeric value is placed in a report-item, the report-item will have one actual $ or + or − immediately to the left of the most significant nonzero digit, in one of the positions indicated by $ or + or − in the PICTURE clause. Blanks are placed in all character positions to the left of the float character. If the most significant digit appears in a position to the right of positions defined by the floating string, the report-item will contain $ or + or − in the rightmost position of the floating string, and non-significant zeros may follow. When a floating string contains an actual or implied decimal point, all digit positions to the right of the decimal point are treated as if they contain 9's.

In the following examples, B represents a blank in the developed items.

PICTURE Clause	Numeric Value	Report-Item
$$$999	14	BB$014
--,---,999	-456	BBBBBB-456
$$$$$$	14	BBB$14

A floating string need not constitute the entire PICTURE clause of a report-item, as shown in these examples. The characters that may follow a floating string are:

When a comma appears to the right of a floating string, the float character disregards the comma so that it may be as close to the leading digit as possible.

\+ − The plus sign (+) or minus sign (−) may appear in a PICTURE clause either singly or in a floating string. As a fixed-sign character, the + or − must appear as the last symbol in the PICTURE clause.

The plus sign indicates that the sign of the item is indicated by either a plus or minus placed in the character position, depending on the algebraic sign of the numeric value placed in the report field. The minus sign indicates that blank or minus is placed in the character position, depending on whether the algebraic sign of the numeric value placed in the report field is positive or negative, respectively.

B Each appearance of B in a PICTURE clause represents a blank in the final edited value.

/ Each slash in a PICTURE clause represents a slash in the final edited value.

0 Each appearance of zero in a PICTURE clause represents a position in the final edited value where the digit zero will appear.

Other rules for the PICTURE clause of a report-item are:

1. Only one type of floating string may be used in the item.

2. The item must have at least one digit position character.

3. The appearance of a floating sign string or fixed plus or minus character precludes the appearance of any other of the sign control insertion characters, namely, +, −, CR, and DB.

4. The characters from the immediate right of a decimal point to the end of the PICTURE clause (excluding the fixed insertion characters +, −, CR, and DB), are subject to the following restrictions:

   a. Only one type of digit position character may appear. That is, only one of Z * 9 and the floating-string digit position characters $ + − may be used.

   b. If one of the numeric character positions to the right of a decimal point is represented by + or − or $ or Z, then all the numeric character positions in the PICTURE clause must be represented by the same character.

5. The PICTURE character 9 can never appear to the left of a floating string or replacement character.

Additional notes on the PICTURE Clause:

1. A maximum of 30 character positions is allowed in a PICTURE character string. For example, PICTURE X(89) consists of the five PICTURE characters X, (, 8, 9, and).

2. A PICTURE must contain at least one of the characters A Z * X 9 or at least two consecutive appearances of + or − or $.

3. The characters ., S, V, CR, and DB can appear only once in a PICTURE clause.

4. When the DECIMAL-POINT IS COMMA clause is specified in the ENVIRONMENT DIVISION of the program, the explanations for period and comma apply to comma and period, respectively.

5. A PICTURE clause is used only with elementary items, not with group items.

The following examples illustrate how data are represented by the PICTURE clause. "Data Value" shows contents in storage; and "Edited Data" shows the value that is reported.

Source Area		Receiving Area	
PICTURE	Data Value	PICTURE	Edited Data
9(5)	12345	$$$,$$9.99	$12,345.00
9(5)	00123	$$$,$$9.99	$123.00
9(5)	00000	$$$,$$9.99	$0.00
9(4)V9	12345	$$$,$$9.99	$1,234.50
V9(5)	12345	$$$,$$9.99	$0.12
S9(5)	00123	-------.99	123.00
S9(5)	- 00001	-------.99	-1.00
S9(5)	00123	+++++++.99	+123.00
S9(5)	00001	-------.99	1.00
9(5)	00123	+++++++.99	+123.00
9(5)	00123	-------.99	123.00
S9(5)	12345	*******.99 CR	**12345.00
S999V99	02345	ZZZ.ZZ	23.45
S999V99	00004	ZZZ.ZZ	.04

RECORD Clause

Purpose

Specifies the number of characters each record in the file contains. It is documentary only, since the size of each data record is always defined by the data-description entries that make up the record (level 01) declaration.

Format

The RECORD clause appears in the FD (and SD) entry in the FILE SECTION.

The general format is:

[; RECORD CONTAINS [integer-1 TO] integer-2 CHARACTERS]

Integer-2 should be the size of the largest record in the file declaration. If the records are variable in size, integer-1 must be specified and equal to the size of the smallest record. The sizes are given as character positions required to store the logical records.

Remarks

This clause is always optional.

Example

```
FD INV-MSTR-FILE
   RECORD CONTAINS 80 CHARACTERS.
```

REDEFINES Clause

Purpose

Specifies that a storage area is to contain different data-items, or provides an alternative grouping or description of the same data.

Format

The REDEFINES clause is optional. If present, it must be the first clause in the data-description or record-description entry.

The general format is:

level-number { data-name-1 | FILLER }

[; REDEFINES data-name-2]

The level number and data-name are included here for clarity, but they are not actually part of the REDEFINES clause.

The data-description entry for data-name-2 should not contain a REDEFINES clause or an OCCURS clause.

Remarks

When an area is redefined, all descriptions of the area remain in effect. Thus, if B and C are two separate items that share the same storage area due to redefinition, the procedure statements MOVE X TO B or MOVE Y TO C could be executed at any point in the program. In the first case, B would assume the value of X and take the form specified by the description of B. In the second case, the same physical area would receive Y according to the description of C.

For purposes of discussing redefinition, data-name-1 is termed the subject, and data-name-2 is called the object. The levels of the subject and object are denoted by s and t, respectively. The following rules must be obeyed in order to establish a proper redefinition:

1. Level s must equal level t, but must not equal 88.
2. The object must be contained in the same record (01 group level item), unless s = t = 01.
3. The REDEFINES clause may not be used in level 01 entries in the FILE SECTION, because multiple 01 level items in the FILE SECTION are implicitly redefined.
4. Prior to definition of the subject and subsequent to definition of the object there can be no level numbers that are numerically less than s.
5. The length of data-name-1, multiplied by the number of occurrences of data-name-1, may not exceed the length of data-name-2, unless the level of data-name-1 is 01.
6. Data-name-1 and entries subordinate to data-name-1 must not contain any VALUE clauses, except in level 88.

REQUIRED Clause

Purpose When a data-item is accepted from a screen, REQUIRED causes terminator characters to be ignored until at least one non-terminator character is entered into a field.

The REQUIRED clause appears in the SCREEN SECTION of the DATA DIVISION.

Format The general format is:

[<u>REQUIRED</u>]

Example

```
05 LINE PIC X(5)
   TO WS-IDENT-NO
   REQUIRED.
```

SECURE Clause

Purpose Suppresses the echo of characters input at the terminal. Instead, an asterisk is displayed for each data character accepted.

Format The SECURE clause appears in the SCREEN SECTION of the DATA DIVISION.

The general format is:

[<u>SECURE</u>]

Remarks The SECURE clause is always optional.

Example

```
05 SCREEN-NAME PIC S9(5)
   USING WS-NAME
   SECURE.
```

SIGN Clause

Purpose Specifies that an operational sign be included as part of an external decimal item; also specifies one of four possible formats for placement of the sign.

Format The SIGN clause appears in the data-description entry for an external decimal item (USAGE IS DISPLAY).

The general format is:

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

where the possible forms of the clause are:

SIGN Clause	Sign Representation
TRAILING	Embedded in right-most byte
LEADING	Embedded in left-most byte
TRAILING SEPARATE	Stored in separate right-most byte
LEADING SEPARATE	Stored in separate left-most byte

The SEPARATE CHARACTER phrase increases the size of the data-item by one character.

Remarks

The following rules apply to the SIGN clause:

1. When an operational sign is specified, the PICTURE must begin with S. If no S is used, the item is not signed (and is capable of storing only absolute values), and the SIGN clause is prohibited. When S appears at the front of a PICTURE but no SIGN clause is included in an item's description, the "default" case SIGN IS TRAILING is assumed.
2. The SIGN clause may be written at the group level. In this case, the clause specifies the sign's format on any signed subordinate external decimal item.
3. The entries to which the SIGN clause applies must be implicitly or explicitly described as USAGE IS DISPLAY.
4. When the CODE-SET clause is specified for a file, all signed numeric data for that file must be described with the SIGN IS SEPARATE clause.

SYNCHRONIZED Clause

Purpose The SYNCHRONIZED clause was designed in order to efficiently allocate memory space for data. It specifies the alignment of an item on the computer's natural memory boundaries. However, in MS-COBOL, the SYNCHRONIZED clause is documentary only.

Format The SYNCHRONIZED clause is used in the standard data-description entry.

The general format is:

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

Remarks The SYNCHRONIZED clause may be used only with elementary data-items.

Although this clause is documentary only, the compiler checks the syntax.

Example

```
05 RECORD-ITEM PIC X(10)
   SYNCHRONIZED RIGHT.
```

TO Clause

See "FROM/TO/USING Clause."

USAGE Clause

Purpose Specifies the form in which numeric data are represented.

Format The USAGE clause appears in data-description or record-description entries in the FILE, WORKING-STORAGE, or LINKAGE SECTIONS.

The general format is:

```
[                { COMPUTATIONAL-0 } ]
[                { COMP-0          } ]
[                { COMPUTATIONAL   } ]
[ ; [USAGE IS]   { COMP            } ]
[                { COMPUTATIONAL-3 } ]
[                { COMP-3          } ]
[                { DISPLAY         } ]
[                { INDEX           } ]
```

COMP is an accepted abbreviation for COMPUTATIONAL.

A COMPUTATIONAL item is capable of representing a value to be used in computations. It must be numeric. If a group item is described as COMPUTATIONAL, the elementary items in the group are COMPUTATIONAL. The group item itself is not COMPUTATIONAL and cannot be used in computations.

COMPUTATIONAL-3, which may be abbreviated COMP-3, defines a packed (internal decimal) field. COMPUTATIONAL-0 (abbreviated COMP-0) defines a 16-bit binary field.

Note

In MS-COBOL for the Z80/8080 microprocessor, items with USAGE IS COMPUTATIONAL are treated as binary items, unless the /V (Validation) compiler switch is used. In that case, and in all other versions of MS-COBOL, items with USAGE IS COMPUTATIONAL are treated as external decimal items, as if defined with USAGE IS DISPLAY.

The USAGE IS DISPLAY clause indicates that the data are in standard ASCII data format.

USAGE IS INDEX indicates that the data-item will be used as an index data-item (see Chapter 8, "Table Handling by the Indexing Method" for more information on using tables). USAGE IS INDEX defines the data-item to be a binary item, in the same format as a COMPUTATIONAL-0 data-item. If USAGE IS INDEX is used, no PICTURE clause can be used.

Remarks

If a USAGE clause is given at a group level, it applies to each elementary item in the group. The USAGE clause for an elementary item must not contradict the USAGE clause of a group to which the item belongs.

The USAGE clause may be written at any level. If USAGE is not specified, the item is assumed to be USAGE IS DISPLAY.

Example

```
05 TOTAL-AMT-SALES PIC S9(5)V99
   USAGE IS COMP-3.
```

USING Clause

See "FROM/TO/USING Clause."

VALUE IS Clause

Purpose Specifies the initial value of data-items or conditions.

Format In MS-COBOL the VALUE IS clause appears only in the WORKING-STORAGE and SCREEN SECTIONS or in level 88 conditions. The format for a standard data-description entry is:

```
[ ; VALUE IS literal] .
```

The format for a level 88 condition-name is:

```
                   { VALUE IS   }  { literal-1 [literal-2 ...]                       }
88 condition-name; { VALUES ARE }  { literal-1 [ { THROUGH }  literal-2 ]            }
                                   {             { THRU    }                         }
```

THROUGH and THRU are equivalent.

Note that the VALUE IS clause is required for level 88 conditions.

Remarks The VALUE IS clause must not be written in a data-description entry that also has an OCCURS or REDEFINES clause, or in an entry that is subordinate to an entry containing an OCCURS or REDEFINES clause. Furthermore, it cannot be used in the FILE or LINKAGE SECTIONS, except in level 88 condition descriptions.

The size of the literal given in a VALUE IS clause must be less than or equal to the size of the item as given in the PICTURE clause. The positioning of the literal within a data area is the same as would result from specifying a MOVE of the literal to the data area, except that editing characters in the PICTURE have no effect on the initialization, nor do BLANK WHEN ZERO or JUSTIFIED clauses.

The type of literal written in a VALUE IS clause depends on the type of data-item, as described in Chapter 1, "Language Elements." For edited items, values must be specified as non-numeric literals, and must be presented in edited form. A figurative constant may be given as the literal.

When an initial value is not specified, no assumption should be made regarding the initial contents of an item in WORKING-STORAGE.

The VALUE IS clause may be specified at the group level, in the form of a correctly sized non-numeric literal, or a figurative constant. In these cases the VALUE IS clause cannot be stated at the subordinate levels within the group. However, the VALUE IS clause should not be written for a group containing items with descriptions that include JUSTIFIED, SYNCHRONIZED and USAGE clauses (other than USAGE IS DISPLAY).

See the description of the VALUE OF clause, which provides information for the label records associated with a disk file.

Examples

```
05 QTY PIC 99 VALUE IS 24.
   88 ON-HAND-QTY VALUE IS 1 THRU 3.
```

VALUE OF Clause

Purpose

Provides information for the label records associated with a file that is assigned to disk.

Format

The VALUE OF clause appears in any FD (or SD) entry for a disk-assigned file. (SD entries apply only to implementations that have MS-SORT.) The VALUE OF clause contains a file-ID expressed as a data-name or quoted literal.

The general format is:

```
[ ; VALUE OF FILE-ID IS { data-name-1 } ] .
                        { literal-1   }
```

Remarks

If a file is assigned to PRINTER, it is unlabelled and the VALUE OF clause must not be included in the associated FD or SD. If a file is assigned to DISK, it is necessary to include both LABEL RECORDS STANDARD and VALUE OF clauses in the associated FD or SD.

If a data-name is specified, it may contain as many characters as desired, but it must end with a space character.

See your MS-DOS manual for file-ID formats for specific operating systems.

Examples

```
(MS-DOS) VALUE OF FILE-ID "A:MASTER.ASM"
(DTC)      VALUE OF FILE-
ID IS "D0:X201A.L"
(ALTAIR)    VALUE OF FILE-
ID "F0:INVNT.LST"
```

Chapter 6

PROCEDURE DIVISION

This chapter describes the statements used in the PROCEDURE DIVISION of a program. This introduction provides an overview of the entire division. Descriptions of individual statements are arranged alphabetically under the "PROCEDURE DIVISION Statements" section.

Purpose

The PROCEDURE portion of a source program specifies the procedures needed to solve a given data processing problem. These steps (computations, logical decisions, etc.) are expressed in statements, similar to English, which employ the concept of verbs to denote actions, and statements and sentences to describe procedures.

Format

The PROCEDURE DIVISION must begin with the header:

```
PROCEDURE DIVISION.
```

which is followed by a period (.).

See Chapter 7, "Inter-Program Communication," for the general format for the PROCEDURE DIVISION of programs that use CALL or CHAIN statements.

The general format for the PROCEDURE DIVISION is:

```
PROCEDURE DIVISION [{USING   } data-name-1 [, data-name-2] ... ].
                    {CHAINING}

[DECLARATIVES.

{section-name SECTION [segment-number]. declarative-sentence

[paragraph-name. [sentence] ...] ...} ...

END DECLARATIVES. ]

{section-name SECTION [segment-number].

[paragraph-name. [sentence] ...] ...} ...
```

Remarks

The PROCEDURE DIVISION may be subdivided in three possible ways:

1. It may consist only of paragraphs.

2. It may consist of a number of paragraphs followed by a number of sections (each section subdivided into one or more paragraphs).

3. It may consist of a DECLARATIVES region and a series of sections (each section subdivided into one or more paragraphs).

   The DECLARATIVES region of the PROCEDURE DIVISION is optional; it provides a means of designating a procedure to be invoked in the event of an input-output error. If DECLARATIVES are used, only the organizational possibility described in the preceding paragraph may be used. See Chapter 12, "DECLARATIVES and the USE Sentence," for a discussion of DECLARATIVES.

Sections, paragraphs, sentences, and statements are defined in Chapter 2, "Structure of a COBOL Program."

Example

```
PROCEDURE DIVISION.

P000-MAINLINE.
    OPEN INPUT INVENTORY-MASTER-FILE,
         OUTPUT INVENTORY-WARNING-FILE,
         INVENTORY-REPORT FILE.
    WRITE REPORT-RECORD FROM PR-HEADER
         AFTER ADVANCING PAGE.
    PERFORM P100-WRITE-REPORT
         UNTIL END-OF-FILE.
    MOVE REC-COUNT TO PR-REC-COUNT.
    MOVE WARNING-COUNT TO PR-WARNING-COUNT.
    WRITE REPORT-RECORD
         FROM PR-TOTAL-RECORD
         AFTER ADVANCING 2 LINES.

    CLOSE INVENTORY-MASTER-FILE,
         INVENTORY-WARNING-FILE,
         INVENTORY-REPORT-FILE.

    STOP RUN.

P100-WRITE-REPORT.
    READ INVENTORY-MASTER-FILE
         AT END MOVE "Y" TO END-OF-FILE-SW.
    IF NOT END-OF-FILE
         PERFORM P200-PROCESS-RECORD.
```

```
P200-PROCESS-RECORD.
     MOVE MSTR-KEY TO PR-KEY.
     MOVE MSTR-DESCRIPTION
           TO PR-DESCRIPTION.
     MOVE MSTR-AMT-ON-HAND
           TO PR-AMT-ON-HAND.
     MOVE MSTR-WARNING-LEVEL
           TO PR-WARNING-LEVEL.
     PERFORM P300-WRITE-LINE.

     IF MSTR-AMT-ON-HAND <
        MSTR-WARNING-LEVEL
          MOVE MASTER-RECORD
               TO WARNING-RECORD
          ADD 1 TO WARNING-COUNT
          WRITE WARNING-RECORD.

P300-WRITE-LINE.
     WRITE REPORT-RECORD
           FROM PR-REPORT-RECORD
           AFTER ADVANCING 1 LINE.
```

Arithmetic Statements

There are five arithmetic statements: ADD, SUBTRACT, MULTIPLY, DIVIDE and COMPUTE. The following discussion applies to all arithmetic statements. The individual statements are described in the alphabetical listings later in this chapter. See Chapter 1, "Language Elements," for further discussion of arithmetic statements and for explanation of the evaluation order of precedence.

Any arithmetic statement may be either imperative or conditional. When an arithmetic statement includes an ON SIZE ERROR specification, the entire statement is termed conditional, because the size-error condition is data-dependent.

An example of a conditional arithmetic statement is:

```
ADD 1 TO RECORD-COUNT
    ON SIZE ERROR
        MOVE ZERO TO RECORD-COUNT
        DISPLAY "LIMIT 99 EXCEEDED".
```

Note that if a size error occurs (in this case, it is apparent that RECORD-COUNT has PICTURE 99, and cannot hold a value of 100), both the MOVE and DISPLAY statements are executed.

The following rules apply to arithmetic statements:

1. All data-names used in arithmetic statements must be elementary numeric data items that are defined in the DATA DIVISION of the program, except that operands of the GIVING option may be report (numeric edited) items. Index-names and index-items are not permissible in these arithmetic statements.

2. Decimal point alignment is supplied automatically throughout the computations.

3. Intermediate result fields generated for the evaluation of arithmetic expressions assure the accuracy of the result field, except where high-order truncation is necessary.

SIZE ERROR Option

If, after decimal-point alignment and any low-order rounding, the value of a calculated result exceeds the largest value that the receiving field is capable of holding, a size error condition exists.

The optional SIZE ERROR clause is written immediately after any arithmetic statement, as an extension of the statement. The format of the SIZE ERROR option is:

```
[; ON SIZE ERROR imperative-statement]
```

If the SIZE ERROR option is present, and a size error condition arises, the value of the resultant data-name is unaltered and the series of imperative statements specified for the condition is executed.

If the SIZE ERROR option has not been specified and a size error condition arises, no assumption should be made about the final result.

An arithmetic statement, if written with SIZE ERROR option, is not an imperative statement. Rather, it is a conditional statement and is prohibited in contexts where only imperative statements are allowed.

ROUNDED Option

If, after decimal-point alignment, the number of places in the fraction of the result is greater than the number of places in the fractional part of the data item that is to be set equal to the calculated result, truncation occurs unless the ROUNDED option has been specified.

When the ROUNDED option is specified, the least significant digit of the resultant data-name has its value increased by 1 whenever the most significant digit of the excess is greater than or equal to 5.

Rounding of a computed negative result is performed by rounding the absolute value of the computed result and then making the final result negative.

The following chart illustrates the relationship between a calculated result and the value stored in an item that is to receive the calculated result, with and without rounding.

Calculated Result	PICTURE	Value After Rounding	Value After Truncation
−12.36	S99V9	−12.4	−12.3
8.432	9V9	8.4	8.4
35.6	99V9	35.6	35.6
65.6	S99V	66	65
.0055	SV999	.006	.005

When the low-order integer positions in a resultant-identifier are represented by the character "P" in its picture, rounding or truncation occurs relative to the right-most integer position for which storage is allowed.

GIVING Option

If the GIVING option is written, the value of the data-name that follows the word GIVING is made equal to the calculated result of the arithmetic operation. The data-name that follows GIVING is not used in the computation and may be a report (numeric edited) item.

I-O Error Handling

If an I-O error occurs, the file's FILE STATUS item, if one exists, is set to the appropriate two-character code. Otherwise it assumes the value "00".

If an I-O error occurs and is of the type that is pertinent to an AT END or INVALID KEY clause, then the imperative statements in such a clause, if present on the statement that gave rise to the error, are executed. But, if there is not an appropriate clause (such clauses may not appear on OPEN or CLOSE, for example, and are optional for other I-O statements), then the logic of program flow is as follows:

1. If there is an associated DECLARATIVES error procedure, it is performed automatically; user-written logic must determine what action is taken because of the existence of the error. Upon return from the error procedure, normal program flow to the next sentence (following the I-O statement) is allowed.
2. If no DECLARATIVES error procedure is applicable but there is an associated FILE STATUS item, it is presumed that the user may base actions upon testing the status item, so normal flow to the next sentence is allowed.

Only if none of the above (INVALID KEY/AT END clause, DECLARATIVES error procedure, or testable FILE STATUS item) exists, then the runtime error handler receives control; the location of the error (source program line number) is noted, and the run is terminated "abnormally."

These remarks apply to processing of any file, whether organization is SEQUENTIAL, LINE SEQUENTIAL, INDEXED, or RELATIVE.

Dynamic Debugging Statements

The execution TRACE mode may be set or reset dynamically. When it is set, procedure-names are printed on the user's terminal in the order in which they are executed.

Execution of the READY TRACE statement sets the trace mode to cause printing of every section and paragraph name each time it is entered. The RESET TRACE statement inhibits such printing. A printed list of procedure-names in the order of their execution is invaluable in detection of a program malfunction, because it aids in detection of the point at which actual program flow departed from the expected program flow.

Another debugging feature may be required in order to reveal critical data values at specifically designated points in the procedure. The EXHIBIT statement provides this facility. EXHIBIT produces a printout of values of a specified literal, or data-items in the format data-name = value.

The EXHIBIT, READY TRACE, and RESET TRACE statements are extensions to ANSI 74 Standard COBOL. These statements are designed to provide a convenient aid to program debugging. For more information, see the discussions of the individual statements in Section 6.4, "PROCEDURE DIVISION Statements."

Note

It is often desirable to include such statements on source lines that contain D in column 7. In this case, the debugging statements are ignored by the compiler unless the WITH DEBUGGING MODE clause is included in the SOURCE-COMPUTER paragraph.

MS-COBOL also provides an interactive debug facility for dynamic program debugging. See the *MS-COBOL Compiler User's Guide* for information about this facility.

PROCEDURE DIVISION Statements

The remainder of this chapter discusses the individual PROCEDURE DIVISION statements. These statements are arranged alphabetically. For information about how the statements appear in the general format of the PROCEDURE DIVISION, see the introduction to this chapter.

ACCEPT Statement

Purpose The ACCEPT statement is used by a processing program to obtain low-volume input at runtime.

Format Four formats are available:

```
ACCEPT identifier
```

```
                          { DATE        }
ACCEPT identifier FROM    { DAY         }
                          { TIME        }
                          { LINE NUMBER }
                          { ESCAPE KEY  }
```

```
                                             { ZERO-FILL     }
ACCEPT (position-spec) identifier [WITH      { SPACE-FILL    } ... ]
                                             { LEFT-JUSTIFY  }
                                             { RIGHT-JUSTIFY }
                                             { TRAILING-SIGN }
                                             { PROMPT        }
                                             { UPDATE        }
                                             { LENGTH-CHECK  }
                                             { AUTO-SKIP     }
                                             { BEEP          }
                                             { NO-ECHO       }
                                             { EMPTY-CHECK   }
```

```
ACCEPT screen-name [ON ESCAPE imperative-statement]
```

Formats 3 and 4 are Microsoft COBOL extensions to ANSI 74 Standard COBOL.

Remarks

The function of each form of the ACCEPT statement is to acquire data from a source external to the program and place it in a specified receiving field or set of receiving fields. The forms differ primarily in the data source with which they are designed to interface. The format 1 ACCEPT obtains date or time information from the operating system clock. The next two formats of the ACCEPT statement receive data keyed in by an operator at the terminal. For Format 2, this device is assumed to be a teletype, a glass teletype, or a terminal in scrolling mode. For Format 3, it is assumed that the input device is a video terminal and that scrolling is not desired. The Format 4 ACCEPT receives an entire data entry form (as defined in the SCREEN SECTION) when it has been completed by the terminal operator. Note that an ordinary terminal is suitable as an input device for a format 2, 3, or 4 ACCEPT, although the effects on the appearance of the screen will differ. The effects of the various WITH phrase options of the Format 3 ACCEPT statement are summarized in "Format 3 ACCEPT Statement."

Format 1 ACCEPT Statement

Any of several standard values may be obtained at execution time by use of the Format 1 ACCEPT statement. The formats of the standard values are:

DATE

a six-digit value of the form YYMMDD (year, month, day). Example: July 4, 1976 is 760704

DAY

a five-digit "Julian date" of the form YYNNN where YY is the two low-order digits of year and NNN is the day-in-year number between 1 and 366.

TIME

an eight-digit value of the form HHMMSSFF where HH is from 00 to 23, MM is from 00 to 59, SS is from 0 to 59, and FF is from 00 to 99; HH is the hour, MM is the minutes, SS is the seconds, and FF represents hundredths of a second.

LINE NUMBER

the ACCEPT. . .FROM LINE NUMBER statement is provided for compatibility, but in MS-COBOL, the value of LINE NUMBER is always zero.

ESCAPE KEY

a two-digit code generated by the key that terminated the most recently executed Format 3 or Format 4 ACCEPT statement.

Identifier can be interrogated to determine exactly which key was typed. Input may be terminated by any of the following keys, which set the ESCAPE KEY value to:

Key Name	Value
Backtab (terminates only Format 3 ACCEPTs)	99
Escape	01
Field-terminator (of the last field if Format 4 ACCEPT is used)	00
Function key	02-nn

Refer to Appendix A of the *Microsoft COBOL Compiler User's Guide* for information on how key codes are defined for specific terminals. The identifier specified in the format should be an unsigned numeric integer whose length agrees with the content of the system-defined data-item. If not, the standard rules for a MOVE govern storage of the source value in the receiving item (identifier).

Format 2 ACCEPT Statement

Format 2 of the ACCEPT statement is used to accept a string of input characters from a scrolling device such as a teletype or a terminal in scrolling mode. When the ACCEPT statement is executed, input characters are read from the terminal until a carriage return is encountered, then a carriage return/line feed pair is sent back to the console. The input data string is considered to consist of all characters keyed prior to (but not including) the carriage return.

For a Format 2 ACCEPT with an alphanumeric receiving field, the input data string is transferred to the receiving field exactly as if it were being MOVEd from an alphanumeric field of length equal to the number of characters in the string. (That is, left justification, space filling, and right truncation occur by default, and right justification and left truncation occur if the receiving field is described as JUSTIFIED RIGHT.) If the receiving field is alphanumeric-edited, it is treated as an alphanumeric field of equal length (as if each character in its PICTURE were "X"), so that no insertion editing occurs.

For a Format 2 ACCEPT with a numeric or numeric-edited receiving field, the input data string is subjected to a validity test which depends on the PICTURE of the receiving field. (If the receiving field is described as COMP-0, its PICTURE is treated as "S9(5)" for purposes of this discussion.) The digits 0 through 9 are considered valid anywhere in the input data string.

The decimal point character is either a period (.) or a comma (,), depending on whether the DECIMAL POINT IS COMMA clause of the CONFIGURATION SECTION is used. In the following discussions, any reference to the decimal point character as a period should be interpreted as if the reference were to a comma if the DECIMAL POINT IS COMMA clause is active.

The decimal point character is considered valid if:

1. it occurs only once in the input data string,
2. the PICTURE of the receiving field contains a fractional digit position, that is, a 9, Z, *, or floating insertion character which appears to the right of either an assumed decimal point (V) or an actual decimal point (.).

The operational sign characters + and − are considered valid only as the first or last character of the input string and only if the PICTURE of the receiving field contains one of the sign indicators S, +, −, CR, or DB.

All other characters are considered invalid. If the input data string is invalid, the message "INVALID NUMERIC INPUT -- PLEASE RETYPE" is sent to the console, and another input data string is read.

When a valid input data string has been obtained, data are transferred to the receiving field exactly as if the instruction being executed were a MOVE to the receiving field from a hypothetical source field with the following characteristics:

1. a PICTURE of the form S9. . .9V9. . .9
2. USAGE DISPLAY
3. a total length equal to the number of digits in the input data string
4. as many digit positions to the right of the assumed decimal point as there are digits to the right of the explicit decimal point in the input data string (zero if there is no decimal point in the input data string)
5. current contents equal to the string of digits embedded in the input data string
6. a separate sign with a current negative status if the input data string contains the character "−", and a current positive status otherwise

Format 3 ACCEPT Statement

Format 3 of the ACCEPT statement is used to accept data into a field from a nonscrolling video terminal. The following syntax rules must be observed when the format 3 ACCEPT is used:

1. The identifier must reference a data item whose length is less than or equal to 1920 characters.
2. The options SPACE-FILL and ZERO-FILL may not both be specified in the same ACCEPT statement.
3. The options LEFT-JUSTIFY and RIGHT-JUSTIFY may not both be specified within the same ACCEPT statement.
4. If the identifier is described as a numeric-edited item, the UPDATE option must not be specified.
5. The TRAILING-SIGN option may be specified only if the identifier is described as an elementary numeric data-item. If the identifier is described as unsigned, the TRAILING-SIGN option is ignored.
6. For alphanumeric or alphanumeric-edited identifiers, the SPACE-FILL option is assumed if the ZERO-FILL option is not specified, and the LEFT-JUSTIFY option is assumed if the RIGHT-JUSTIFY option is not specified.
7. For numeric or numeric-edited identifiers, the ZERO-FILL option is assumed if the SPACE-FILL option is not specified.

Data Input Field

The position-spec and receiving field (identifier) specifications of the Format 3 ACCEPT statement are used to define the location and characteristics of a data input field on the screen of the terminal.

Location of the Data Input Field

The position-spec is of the form:

```
( [ LIN [{·} integer-1] ] , [ COL [{·} integer-3] ] )
  [       integer-2     ]   [       integer-4     ]
```

The opening and closing parentheses and the comma separating the two major bracketed groups are required. A space must follow the comma. The position-spec specifies the position on the terminal screen at which the data input field will begin. LIN and COL are MS-COBOL special registers. Each behaves like a numeric data item with USAGE IS COMP-0, but they may be referenced by every MS-COBOL program without being declared in the DATA DIVISION.

If LIN is specified, the data input field will begin on the screen row whose number is equal to the value of the LIN special register, incremented (or decremented) by integer-1 if "+ integer-1" (or "− integer-1") is specified. If integer-2 is specified, the data input field will begin on the row whose number is integer-2. If neither LIN nor integer-2 is specified, the data input field will begin on the screen row containing the current cursor position.

If COL is specified, the data input field will begin in the screen column whose number is equal to the value of the COL special register, incremented (or decremented) by integer-3 if "+ integer-3" (or "− integer-3") is specified. If integer-4 is specified, the data input field will begin in the screen column whose number is integer-4. If neither COL nor integer-4 is specified, the data input field will begin in the screen column containing the current cursor position.

Characteristics of the Data Input Field

The characteristics (other than position) of the data input field on the terminal screen are determined by the receiving field's PICTURE specification (which is treated as S9(5) in the case of an item whose USAGE is COMP-0). For alphanumeric or alphanumeric-edited identifier-3, the data input field is simply a string of data input character positions starting at the screen location specified by position-spec. The length of the data input field in character positions is equal to the length of the receiving field in memory.

For numeric or numeric-edited identifiers, the data input field may contain any or all of the following: integer digit positions, fractional digit positions, sign position, decimal point position. There will be one digit position for each 9, Z, *, P, or noninitial floating insertion symbol (a floating insertion symbol is a +, −, or $ which is not the last symbol in a PICTURE character string) in the PICTURE of the identifier.

Each digit position in the data input field is a fractional digit position if the corresponding PICTURE character is to the right of an assumed decimal point (V) or actual decimal point (.) in the PICTURE of the identifier. Otherwise it is an integer digit position. There will be one sign position if the identifier is described as signed, and no sign position otherwise. There will be one decimal point position if there is at least one fractional digit position, and no decimal point position otherwise.

The data input positions which are defined will occupy successive character positions on the terminal screen, beginning with the position specified by position-spec. If TRAILING-SIGN is specified in the ACCEPT statement, the data input positions will be in the following sequence: integer digit positions (if any), decimal point position (if any), fractional digit positions (if any), sign position (if any). If TRAILING-SIGN is not specified, the data input positions will be in the following sequence: sign position (if any), integer digit positions (if any), decimal point position (if any), fractional digit positions (if any).

Data Input and Data Transfer

A character entered into the data input field by the terminal operator may be treated either as an editing character, a terminator character, or as a data character. When a terminator key is typed, the ACCEPT is terminated and the ESCAPE KEY value is set as described in "Format 1 ACCEPT Statement." This value can be interrogated by using a Format 1 ACCEPT statement FROM ESCAPE KEY.

The editing characters are line-delete, forward-space, backspace, and rubout. See Appendix A in the *Microsoft COBOL Compiler User's Guide* to determine which keys perform these functions on your terminal. The action of the editing characters is described later in this section; for now, only data characters will be considered.

Alphanumeric Receiving Field

Consider first the execution of the Format 3 ACCEPT statement with an alphanumeric or alphanumeric-edited receiving field. An alphanumeric-edited receiving field is treated as an alphanumeric field of the same length (as if every character in its PICTURE were "X"). Specifically, no insertion editing will occur.

The initial appearance of the data input field depends on the specifications in the WITH phrase of the ACCEPT statement. If UPDATE is specified, the current contents of the identifier are displayed in the input field. In this case all data input positions will be treated as if they were keyed by the terminal operator. If UPDATE is not specified, but PROMPT is specified, a period (.) is displayed in each input data position. If neither UPDATE nor PROMPT is specified, the data input field is not changed. The cursor is placed in the first data input position, and characters are accepted as they are keyed by the operator until a terminator character (normally carriage return) is encountered.

If AUTO-SKIP is specified in the ACCEPT statement, the ACCEPT will also be terminated if the operator keys a character into the last (right-most) data input position.

As each input character is received, it is echoed to the terminal screen. If all positions of the data input field are filled, additional input is ignored until a terminator character or editing character is encountered. If RIGHT-JUSTIFY was specified in the ACCEPT statement, the operator-keyed characters are shifted to the right-most positions of the data input field when the ACCEPT is terminated. All unkeyed character positions are filled on termination; the fill character is either space (if SPACE-FILL is in effect) or zero (if ZERO-FILL was specified).

The contents of the receiving field will be the same set of characters which appears in the input field; however, the justification of operator-keyed characters will be controlled by the JUSTIFIED specification in the receiving field's data description, not by the RIGHT- or LEFT-JUSTIFY option of the ACCEPT statement. Excess positions of the receiving field will be filled with spaces or zeroes based on the SPACE-FILL or ZERO-FILL specification in the ACCEPT statement.

Numeric Receiving Field

Next, consider the execution of a Format 3 ACCEPT statement with a numeric or numeric-edited receiving field. As described above, the data input field on the terminal screen may contain integer digit positions, fractional digit positions, or both. First assume that both are present; the other cases will be treated as variations.

As with the alphanumeric ACCEPT, the data input field may be initialized in a way determined by the WITH options specified in the ACCEPT statement. If UPDATE is specified (not permitted for a numeric-edited receiving field), the integer and fractional parts of the data input field will be set to the integer and fractional parts of the decimal representation of the initial value of the receiving field, with leading and trailing zeroes included, if necessary, to fill all digit positions. Except for leading zeroes, these initialization characters are treated as operator-keyed data.

When a numeric field with UPDATE is accepted, any digit, sign, or decimal point entered will cause the entire field to be cleared and set to zero or the value of the entered digit. Such a numeric field can be accepted without change by entering a terminator key instead of a digit, sign, or decimal point.

If UPDATE is not specified, but PROMPT is specified, a zero will be displayed in each input digit position. In either of these cases (UPDATE or PROMPT) a decimal point will be displayed at the decimal point position.

If neither UPDATE nor PROMPT is specified, the input field on the screen will not be initialized, except for the sign position. The sign position is always initialized positive except when UPDATE is specified, in which case it is initialized according to the sign of the current contents of the receiving field. On most systems, a positive sign position is shown as a space, and a negative sign position is shown as a minus sign.

The cursor is initially placed in the right-most integer digit position, and characters are accepted one at a time as they are keyed by the operator. A received character may be treated in one of several ways: If the incoming character is a digit, previously keyed digits are shifted one position to the left in the input field and the new digit is displayed in the right-most integer digit position. If all integer digit positions have not been filled, the cursor remains on the right-most digit position and another character is accepted. If the entire integer part of the input field has been filled and AUTO-SKIP was specified, the integer part is terminated and the cursor is moved to the left-most fractional digit position. If the integer part has been filled and AUTO-SKIP was not specified, the cursor is moved to the decimal point position, and any further digits keyed are ignored until the integer part is terminated with a decimal point.

If the character entered is one of the sign characters + or −, the sign position is changed to a positive or negative status respectively. Cursor position is not affected.

If the character entered is a decimal point character, the integer part is terminated and the cursor is moved to the left-most fractional digit position.

If the character entered is a field terminator (normally carriage-return), the ACCEPT is terminated and the cursor is turned off. Any other character is ignored.

When the integer part is terminated, the cursor is placed in the left-most fractional digit position, and operator-keyed characters are again accepted. Digits are simply echoed to the terminal. The sign characters + and − are treated exactly as they were while integer part digits were being entered. The field terminator character terminates the ACCEPT. (If AUTO-SKIP is in effect, filling the entire fractional part also terminates the ACCEPT.) Other characters are ignored. After all digit positions of the fractional part have been filled, further digits are also ignored.

If no fractional digit positions are present, the decimal point is ignored as an input character, and entry of integer part digits may be terminated only by terminating the entire ACCEPT. If no integer digit positions are present, the cursor is initially placed in the left-most fractional digit position and entry of the fractional part digits proceeds as described above.

On termination of the Format 3 ACCEPT of a numeric or numeric-edited item, data are transferred to the receiving field. The exact form of the data in the receiving field after execution of the ACCEPT is as described in the last paragraph of the discussion of the Format 2 ACCEPT, where the role of the "input data string" mentioned in that paragraph is taken by the string of characters displayed in the data input field. After termination, if SPACE-FILL is in effect, leading zeroes in the integer part of the data input field (not in the receiving field) will be replaced by spaces, and the leading operational sign, if present, will be moved to the right-most space thus created.

Editing Characters

The editing characters (line-delete, forward-space, backspace, and rubout) may be used to change data which has already been keyed (or supplied by the MS-COBOL runtime system as a result of a WITH UPDATE specification). Entering the line-delete character will cause the ACCEPT to be restarted and all data keyed by the operator or initially present in the receiving field to be lost. The data input field on the terminal screen will be reinitialized if PROMPT is in effect. Otherwise, the data input field will be filled with spaces or zeroes according to the SPACE-FILL or ZERO-FILL specification.

Typing the forward-space or backspace characters will move the cursor forward or back one data input position in the case of an alphanumeric or alphanumeric-edited receiving field, or one digit position in the case of a numeric or numeric-edited receiving field. In no case, however, will the forward-space or backspace characters move the cursor outside the range of positions including

1. the positions already keyed by the operator (or filled by MS-COBOL runtime support when WITH UPDATE is specified), and
2. the right-most data input position which the cursor has occupied during the execution of this ACCEPT. If the cursor is moved to a position of this range other than the right-most, and a legal data character is entered, it is displayed at the current cursor position and the cursor is moved forward one data position (alphanumeric or alphanumeric-edited) or one digit position (numeric or numeric-edited).

Typing the rubout character effectively cancels the last data character entered. The cursor is moved back one data position (digit position if the receiving field is numeric or numeric-edited) and a fill character (space or zero) is displayed under the cursor (except when the cursor is to the left of the decimal point for a numeric ACCEPT. Then no fill character is displayed and the cursor is not moved, but the digit at the cursor position is deleted and all digits to the left of it are shifted one position to the right.)

Note

The rubout character has no effect unless the cursor is in position to accept a new data character; in other words, it has no effect if backspace character(s) have been used to move the cursor back over already keyed positions.

WITH Phrase

The following list summarizes the effects of the WITH phrase specifications for a Format 3 ACCEPT with an alphanumeric or alphanumeric-edited receiving field:

1. SPACE-FILL causes unkeyed character positions of the data input field and the receiving field to be space-filled when the ACCEPT is terminated.
2. ZERO-FILL causes unkeyed character positions of the data input field and the receiving field to be set to ASCII zeroes when the ACCEPT is terminated.
3. LEFT-JUSTIFY is treated by this compiler as commentary.
4. RIGHT-JUSTIFY causes operator-keyed characters to occupy the right-most positions of the data input field after the ACCEPT is terminated. Note that the justification of transferred data in the receiving field is controlled by the JUSTIFIED declaration or default of the receiving field's data description, not by the WITH RIGHT-JUSTIFY phrase.
5. PROMPT causes the data input field on the screen to be set to all periods (.) before input characters are accepted.
6. UPDATE causes the data input field to be initialized with the initial contents of the receiving field and the initial data to be treated as operator-keyed data.
7. LENGTH-CHECK causes a field terminator character to be ignored unless every data input position has been filled.
8. EMPTY-CHECK causes all field terminator characters to be ignored until at least one nonterminator character has been keyed.
9. AUTO-SKIP forces the ACCEPT to be terminated when all data input positions have been filled. A terminator character explicitly keyed has its usual effect.
10. BEEP causes an audible alarm to sound when the ACCEPT is initialized and the system is ready to accept operator input.

The following list summarizes the effects of the WITH phrase specifications for the Format 3 ACCEPT with a numeric or numeric-edited receiving field:

1. SPACE-FILL causes unkeyed digit positions of the data input field (not of the receiving field) to the left of the (possibly implied) decimal point to be space-filled when the ACCEPT is terminated and any leading operational sign to be displayed in the right-most space thus created.
2. ZERO-FILL causes all unkeyed digit positions of the data input field to be set to zero when the ACCEPT is terminated.
3. LEFT-JUSTIFY and RIGHT-JUSTIFY have no effect for a numeric or numeric-edited receiving field.
4. TRAILING-SIGN causes the operational sign to appear as the right-most position of the data input field. Ordinarily the sign is the left-most position of the field.
5. PROMPT causes the data input field positions to be initialized as follows before input characters are accepted: digit positions to zero, decimal point position (if any) to the decimal point character, and sign position (if any) to space.
6. UPDATE causes the data input field to be initialized to the current contents of the receiving field and this initial data to be treated like operator-keyed data.
7. LENGTH-CHECK causes a received decimal point character to be ignored unless all integer digit positions have been keyed, and a field terminator character to be ignored unless all digit positions have been keyed.
8. EMPTY-CHECK causes all field terminator characters to be ignored until at least one nonterminator character has been keyed.
9. AUTO-SKIP causes the integer part of the ACCEPT to be terminated when all integer digit positions have been keyed, and the entire ACCEPT to be terminated when all digit positions have been keyed.
10. BEEP causes an audible alarm to sound when the ACCEPT is initialized and the system is ready to accept operator input.

The following three examples use the Format 3 ACCEPT statement.

Example 1.

SET-UP PRIOR TO EXECUTING:

Receiving Field:

```
05 RS-DISCOUNT PIC X(8).
```

Initial Contents:

```
ABCDEFGH
```

ACCEPT Statement:

```
ACCEPT (1, 1) RS-DISCOUNT
WITH PROMPT.
```

EXECUTING THE ACCEPT:

At Start of ACCEPT:

```
. . . . . . . .
```

Operator Enters N:

```
N. . . . . . .
```

Operator Enters ONE:

```
NONE. . . .
```

Operator Enters Carriage Return:

```
NONEbbbb
```

RESULT:

Final Contents of Receiving Field:

```
NONEbbbb
```

Example 2.

SET-UP PRIOR TO EXECUTING:

Receiving Field:

```
10 VEND-NAME PIC X(12).
```

Initial Contents:

```
ACME WIDGETS
```

ACCEPT Statement:

```
ACCEPT (1, 1) VEND-NAME WITH
PROMPT UPDATE.
```

At Start of ACCEPT:

```
ACME WIDGETS
```

(If operator enters carriage return here, the receiving field will not be changed.)

EXECUTING THE ACCEPT:

Operator Enters Line-delete:

```
_. . . . . . . . . . . .
```

Operator Enters XYZ:

```
XYZ_. . . . . . . . .
```

Operator Enters Carriage Return:

```
XYZbbbbbbbbb
```

RESULT:

Final Contents of Receiving Field:

```
XYZbbbbbbbbb
```

Example 3.

SET-UP PRIOR TO EXECUTING:

Receiving Field:

```
05 CREDIT PIC S9(4)V99
```

Initial Contents:

```
     +
111111
```

ACCEPT Statement:

```
ACCEPT (LIN + 4, COL - 3) CREDIT
WITH PROMPT TRAILING-SIGN.
```

EXECUTING THE ACCEPT:

At Start of ACCEPT:

```
0000.00b
```

Operator Enters 8:

```
0008.00b
```

Operator Enters 7:

```
0087.00b
```

Operator Enters -:

```
0087.00-
```

Operator Enters 6:

```
0876.00-
```

Operator Enters N:

```
0876.00-
```

Operator Enters .:

```
0876.00-
```

Operator Enters 5:

```
0876.50-
```

Operator Enters Carriage Return:

```
0876.50-
```

RESULT:

Final Contents of Receiving Field:

```
0876.50
```

Format 4 ACCEPT Statement

Format 4 of the ACCEPT statement causes a transfer of information from the operator's terminal to all TO and/or USING fields specified in the SCREEN SECTION definition of screen-name or any screen item subordinate to screen-name. Screen items having only VALUE literals or FROM fields or literals have no effect on the operation of the ACCEPT statement. If you wish to have such fields displayed, perform the DISPLAY screen-name statement before the Format 4 ACCEPT statement.

Each such transfer of data consists of an implicit Format 3 ACCEPT of a field defined by the appropriate screen item's PICTURE followed by an implicit MOVE to the associated TO or USING field.

If an escape key is typed during data input, the entire ACCEPT is terminated without moving the current field to the associated TO or USING item, the ESCAPE KEY value is set to 01, and the ON ESCAPE statement is executed. If a function key is typed, the appropriate ESCAPE KEY value is set and the entire ACCEPT is terminated.

If a field-terminator key (carriage return, tab, etc.,) is typed, the ESCAPE KEY value is set to 00 and the cursor moves to the next input field defined under screen-name, if one exists. If the current field is the last field, the entire ACCEPT is terminated.

If the backtab key is typed, the current field is terminated and the cursor moves to the previous input field defined under screen-name. If the current field is the first field, the cursor does not move from that field.

When a field is terminated by a function key, field-terminator key, or backtab key, the contents of the current field are moved to the associated TO or USING item, except in the case where no data characters and no editing characters have been entered in that field. This allows the operator to tab forward or backward through the input fields without affecting the contents of the receiving items.

All the editing and validation features described in Chapter 6 for the Format 3 ACCEPT apply to the Format 4 ACCEPT as well. Several SCREEN SECTION specifications correspond to the Format 3 ACCEPT options:

AUTO	corresponds to AUTO-SKIP
BELL	corresponds to BEEP
JUSTIFIED	corresponds to RIGHT-JUSTIFY
SECURE	corresponds to NO-ECHO
REQUIRED	corresponds to EMPTY-CHECK
FULL	corresponds to LENGTH-CHECK

Furthermore, if an input field specifies the USING clause or both a FROM and TO clause, the ACCEPT will be executed with the UPDATE option. Format 4 ACCEPT statements always use the PROMPT and TRAILING-SIGN options when executing the individual Format 3 ACCEPTs.

If the screen item's PICTURE specifies a numeric-edited or alphanumeric-edited input field, the ACCEPT is executed as if the field were numeric or alphanumeric, respectively. When the field is terminated the data is edited according to the PICTURE and redisplayed in the specified screen position. In this case, the JUSTIFIED clause has no effect.

Moves from screen fields to receiving items follow the standard MS-COBOL rules for MOVE statements, except that moves from numeric-edited fields are allowed. In this case, the data is input as if the field were numeric and the MOVE uses only the sign, decimal point, and digit characters.

The Format 4 ACCEPT does not cause the display of any text or prompting label information. That is accomplished by using the "DISPLAY screen-name" statement before accepting the screen-name. See the discussion of "DISPLAY Statement" for more information on displaying text.

ADD Statement

Purpose Adds two or more numeric values and stores the resulting sum.

Format The general formats are:

```
ADD {identifier-1} , [identifier-2] ... TO identifier-m [ROUNDED]
    {literal-1   }   [literal-2   ]

    [; ON SIZE ERROR imperative-statement]

ADD {identifier-1} , {identifier-2} , [identifier-3] ...
    {literal-1   }   {literal-2   }   [literal-3   ]

    GIVING identifier-m [ROUNDED]

    [; ON SIZE ERROR imperative-statement]
```

Either the TO or the GIVING option must be specified.

Remarks When the TO option is used, the values of all the identifiers (including identifier-m) and literals in the statements are added, and the resulting sum replaces the value of identifier-m. When the GIVING option is used, at least two identifiers and/or numeric literals must be coded between ADD and GIVING. The sum of the values of these identifiers and literals (not including identifier-m) replaces the value of identifier-m.

Examples

```
ADD INTEREST,
    DEPOSIT TO BALANCE ROUNDED.

ADD REGULAR-TIME OVERTIME
    GIVING GROSS-PAY.
```

The first statement would result in the sum of INTEREST, DEPOSIT, and BALANCE being placed at BALANCE, while the second would result in the sum of REGULAR-TIME and OVERTIME earnings being placed in item GROSS-PAY.

ALTER Statement

Purpose Modifies a simple GO TO statement elsewhere in the PROCEDURE DIVISION, thus changing the sequence of execution of program statements.

Format The general format is:

ALTER procedure-name-1 TO [PROCEED TO] procedure-name-2

Remarks Paragraph (the first operand) must be an MS-COBOL paragraph that consists of only a simple GO TO statement; the ALTER statement in effect replaces the former operand of that GO TO by procedure-name.

Example

```
GATE.
    GO TO MF-OPEN.
MF-OPEN.
    OPEN INPUT MASTER-FILE
    ALTER GATE TO PROCEED TO NORMAL.
NORMAL.
    READ MASTER-FILE,
    AT END GO TO EOF-MASTER.
```

Examination of the above code reveals the technique of "shutting a gate," providing a one-time initializing program step.

CLOSE Statement

Purpose

Upon completion of the processing of a file, a CLOSE statement must be executed, causing the system to make the proper disposition of the file. Whenever a file is closed, or has never been opened, READ, REWRITE, or WRITE statements cannot be executed properly; a runtime error would occur, aborting the run.

Format

The general format for all file organizations is:

CLOSE file-name-1 [WITH LOCK] [, file-name-2 [WITH LOCK]] ...

Remarks

If the LOCK suffix is used, the file cannot be reopened during the current job. If LOCK is not specified immediately after a file-name, then that file may be reopened later in the program, if the program logic dictates the necessity.

An attempt to execute a CLOSE statement for a file that is not currently open is a runtime error, and causes execution to be discontinued.

Examples

```
CLOSE MASTER-FILE-IN WITH LOCK,
   WORK-FILE.
CLOSE PRINT-FILE,
TAX-RATE-FILE,
JOB-PARAMETERS WITH LOCK.
```

COMPUTE Statement

Purpose Evaluates an arithmetic expression and then stores the result in a designated numeric or report (numeric edited) item.

Format The general format is:

```
COMPUTE identifier-1 [ROUNDED]
    =arithmetic-expression [; ON SIZE ERROR imperative-statement]
```

Remarks Note that exponentiation to an integral power can be accomplished by using the COMPUTE statement.

Examples

```
COMPUTE GROSS-PAY ROUNDED = BASE-SALARY *
    (1 + 1.5 * (HOURS - 40) / 40).

COMPUTE AMT-CUBED ROUNDED = AMT ** 3.
```

DELETE Statement

See Chapters 9, 10, and 11 for discussion of use of the DELETE statement in SEQUENTIAL, INDEXED, and RELATIVE files.

DISPLAY Statement

Purpose Provides the capability of outputting low-volume data at runtime without the complexities of file definition.

Format The general formats are:

```
DISPLAY {identifier-1} [, identifier-2] ... [UPON mnemonic-name]
        {literal-1   } [, literal-2   ]

                                {identifier}
DISPLAY {(position-spec)        {literal   }} ... [UPON mnemonic-name]
                                {ERASE     }

DISPLAY screen-name
```

See the following remarks for information on individual parts of the format.

Remarks

The following rules must be observed:

1. All identifiers must reference data items whose lengths are less than or equal to 1920 characters.
2. Mnemonic-name must be defined in the PRINTER IS clause of the SPECIAL-NAMES paragraph of the CONFIGURATION SECTION.
3. Screen-name must be defined in the SCREEN SECTION of the DATA DIVISION.

The DISPLAY statement will cause output to be sent to the terminal unless UPON mnemonic-name is specified, in which case output will be sent to the printer. Each display-item (that is, each occurrence of identifier, literal, or ERASE) will be processed in turn as described in the paragraphs below; then, if no position-spec is coded in the entire DISPLAY statement, a carriage return/line-feed pair will be sent to the receiving device.

Position-Spec

For each display-item, if position-spec is specified, the cursor is positioned prior to the transfer of data for this item.

Position-spec is of the form:

```
( [ LIN [{+ } integer-1] ] , [ COL [{+ } integer-3] ] )
  [         integer-2    ]   [         integer-4    ]
```

The opening and closing parentheses and the comma separating the two major bracketed groups are required. A space must follow the comma. The position-spec specifies the position on the terminal screen at which the cursor will be placed. LIN and COL are MS-COBOL special registers. Each behaves like a numeric data-item with USAGE IS COMP-0, but they may be referenced by every MS-COBOL program without being declared in the DATA DIVISION.

If LIN is specified, the cursor will be placed on the screen row whose number is equal to the value of the LIN special register, incremented (or decremented) by integer-1 if "+ integer-1" (or "− integer-1") is specified. If integer-2 is specified, the cursor will be placed on the row whose number is integer-2. If neither LIN nor integer-2 is specified, the cursor will be placed on the screen row containing the current cursor position.

If COL is specified, the cursor will be placed in the screen column whose number is equal to the value of the COL special register, incremented (or decremented) by integer-3 if "+ integer-3" (or "− integer-3") is specified. If integer-4 is specified, the cursor will be placed in the screen column whose number is integer-4. If neither COL nor integer-4 is specified, the cursor will be placed in the screen column containing the current cursor position.

Identifier, Literal, and ERASE

If identifier or literal is specified for a given display-item, the contents of identifier or the value of literal are sent to the receiving device.

Note

Since the data transfer occurs without conversion or reformatting, it is recommended that numeric data be moved to numeric-edited fields for purposes of DISPLAY.

If ERASE is specified and if position-spec is coded for this or a previous display-item, the terminal screen will be cleared from the current cursor position to the end of the screen. The initial cursor position for the next display-item will be that specified by the position-spec coded in the ERASE display-item, if present, or the position in which the cursor was left by the previous display-item. If ERASE is specified and no position-spec has been encountered up to this point in the DISPLAY statement, no action will be taken.

Screen-name

The "DISPLAY screen-name" statement causes a transfer of information from screen-name (or each elementary screen item subordinate to screen-name) to the terminal screen. For each such screen item having a VALUE, FROM, or USING specification, the specified literal or field is the source of the displayed data. For a field having only a TO clause, the effect is as if FROM ALL "." had been specified. The source data is MOVEd implicitly to a temporary item defined by the appropriate screen item's PICTURE (or by the length of the data in the case of a VALUE literal). Then an implied identifier-type DISPLAY of the constructed temporary is executed as modified by the positioning and control clause coded in the definition of the appropriate screen item.

Examples

```
DISPLAY QTY-ON-HAND.
DISPLAY INPUT-SCREEN.
DISPLAY (10, 2) USER-NAMES.
DISPLAY (LIN, COL) ERASE.
DISPLAY USER-NAME UPON PRINTER.
```

DIVIDE Statement

Purpose Divides two numeric values and stores the quotient.

Format The general formats are:

```
DIVIDE {identifier-1} INTO identifier-2 [ROUNDED]
       {literal-1   }

    [; ON SIZE ERROR imperative-statement]

DIVIDE {identifier-1} INTO {identifier-2} GIVING identifier-3 [ROUNDED]
       {literal-1   }      {literal-2   }

    [; ON SIZE ERROR imperative-statement]

DIVIDE {identifier-1} BY {identifier-2} GIVING identifier-3 [ROUNDED]
       {literal-1   }    {literal-2   }

    [; ON SIZE ERROR imperative-statement]
```

Remarks The BY-form signifies that the first operand (identifier-1 or literal-1) is the dividend (numerator), and the second operand (identifier-2 or literal-2) is the divisor (denominator). If GIVING is not written in this case, then the first operand must be an identifier, in which the quotient is stored.

The INTO-form signifies that the first operand is the divisor and the second operand is the dividend. If GIVING is not written in this case, then the second operand must be an identifier, in which the quotient is stored.

Division by zero always causes a size-error condition.

Examples

```
DIVIDE QTY INTO TOTAL
     GIVING UNIT-COST.

DIVIDE WEIGHT BY 10.
```

EXHIBIT Statement

Purpose Reveals critical data values at specifically designated points in a procedure. This statement is used for debugging.

Format The general format is:

```
EXHIBIT NAMED { [position-spec] { identifier } } ... [UPON mnemonic-name]
                                { literal    }
                                { ERASE      }
```

Remarks EXHIBIT produces a printout of values of the specified literal, or identifiers in the format identifier = value; i.e., both the value of the identifier and its name are displayed.

The EXHIBIT, READY TRACE, and RESET TRACE statements are extensions to ANSI 74 Standard COBOL. These statements are designed to provide a convenient aid to program debugging.

Note

Since the data transfer occurs without conversion or reformatting, it is recommended that numeric data be moved to numeric-edited fields for purposes of display.

Also, it is often desirable to include such statements on source lines that contain D in column 7. In this case, the statements are ignored by the compiler unless the WITH DEBUGGING MODE clause is included in the SOURCE-COMPUTER paragraph.

For more information on debugging, see the "READY/ RESET TRACE Statements."

Examples

```
          EXHIBIT QTY-ON-HAND.
D         EXHIBIT DEBUG-VALUES.
```

In the second example, "D" is in column 7, and "EXHIBIT" begins in column 12.

EXIT Statement

Purpose Provides an end-point for a procedure.

Format The general format is:

EXIT.

Remarks EXIT must appear in the source program as a one-word paragraph preceded by a paragraph-name. An exit paragraph provides an end-point to which preceding statements may transfer control if operator decides to bypass some part of a section.

Example

```
P100-EXIT-POINT.
    EXIT.
```

EXIT PROGRAM Statement

Purpose Marks the logical end of a called program.

Format The general format is:

EXIT PROGRAM.

Remarks The EXIT PROGRAM statement must appear in a sentence by itself and must be the only sentence in the paragraph. It is used instead of STOP RUN to terminate a subprogram and to return control to the calling program.

If an EXIT PROGRAM statement is encountered in a program that was not called, the statement is treated as if it were an EXIT statement (see "EXIT Statement" above).

Example

```
EXIT PROGRAM.
```

GO TO Statement

Purpose Transfers control from one portion of a program to another.

Format The general formats are:

```
GO TO [procedure-name-1]

GO TO procedure-name-1 [, procedure-name-2] ... , procedure-name-n
    DEPENDING ON identifier
```

Remarks The simple form GO TO procedure-name changes the path of flow to a designated paragraph or section. If the GO statement is without a procedure-name, then that GO statement must be the only one in a paragraph, and must be altered prior to its execution.

The more general form designates N procedure-names as a choice of N paths to transfer to, if the value of identifier is 1 to N, respectively. Otherwise, there is no transfer of control and execution proceeds in the normal sequence. Identifier must be a numeric elementary item and have no positions to the right of the decimal point.

If a GO (non-DEPENDING) statement appears in a sequence of imperative statements, it must be the last statement in that sequence.

Examples

```
GO TO P200-PROCESS-RECORD.

GO TO DO-WEEKLY, DO-MONTHLY,
    DO-YEARLY
    DEPENDING ON MODE-OF-PAYMENT.
```

IF Statement

Purpose Permits the programmer to specify a series of procedural statements to be executed in the event a stated condition is true. Optionally, an alternative series of statements may be specified for execution if the condition is false.

Format The general format is:

```
IF condition; { statement-1     } { ; ELSE statement-2        }
              { NEXT SENTENCE   } { ; ELSE NEXT SENTENCE      }
```

Remarks The ELSE NEXT SENTENCE phrase may be omitted if it immediately precedes the terminal period of the sentence.

Examples

```
IF BALANCE = 0
   PERFORM NOT-FOUND.
IF T LESS THAN 5
   NEXT SENTENCE
ELSE
   PERFORM T-1-4.

IF ACCOUNT-FIELD = SPACES
OR NAME = SPACES
   ADD 1 TO SKIP-COUNT
ELSE
   PERFORM PROCESS-RECORD.
```

The first series of statements (all three IF parts) is executed only if the designated condition is true. The second series of statements (the two ELSE parts) is executed only if the designated condition is false.

The second series (the two ELSE parts) is terminated by a sentence-ending period unless it is ELSE NEXT SENTENCE, in which case more statements may be written before the period.

Regardless of whether the condition is true or false, the next sentence is executed after execution of the appropriate series of statements, unless a GO TO is contained in the imperatives that are executed, or unless the nominal flow of program steps is superseded because of an active PERFORM statement.

If there is no ELSE part to an IF statement, then the first series of statements must be terminated by a sentence-ending period. Refer to Appendix C for discussion of nested IF statements.

Conditions

A condition is either a simple condition or a compound condition. The four simple conditions are the relational, class, condition-name, and sign condition tests. A simple relational condition has the following structure:

```
operand-1 relation operand-2
```

where "operand" is a data-name, literal, or figurative-constant.

A compound condition may be formed by connecting two conditions, of any sort, by the logical operator AND or OR (e.g., A < B OR C = D). Refer to Appendix A for further permissible forms involving parentheses, NOT, or "abbreviation."

The simplest "simple relations" have three basic forms, expressed by the relational symbols equal to (=), less than (<), or greater than (>).

Another form of simple relation that may be used involves the reserved word NOT, preceding any of the three relational symbols. In summary, the six simple relations in conditions are:

Relation	**Meaning**
=	equal to
<	less than
>	greater than
NOT =	not equal to
NOT <	greater than or equal to
NOT >	less than or equal to

It is worthwhile to briefly discuss how relational conditions can be compounded. The reserved words AND or OR permit the specification of a series of relational tests, as follows:

1. Individual relations connected by AND specify a compound condition that is met (true) only if all the individual relationships are met.
2. Individual relations connected by OR specify a compound condition that is met (true) if any one of the individual relationships is met.

The following is an example of a compound relational condition containing both AND and OR connectors (refer to Appendix A for formal specification of evaluation rules):

```
IF X = Y AND FLAG = 'Z' OR SWITCH = 0
   PERFORM PROCESSING.
```

In the preceding example, execution will be as follows, depending on various data values.

Data X	Value Y	FLAG	SWITCH	Does Execution Go to PROCESSING?
10	10	'Z'	1	Yes
10	11	'Z'	1	No
10	11	'Z'	0	Yes
10	10	'P'	1	No
6	3	'P'	0	Yes
6	6	'P'	1	No

The reserved words EQUAL TO, LESS THAN, and GREATER THAN are accepted equivalents of =, < , and >, respectively. Any form of the relation may be preceded by the word IS, optionally.

Before discussing class-test, sign-test, and condition-name-test conditions, methods of performing comparisons will be discussed.

Numeric Comparisons

The data operands are compared after alignment of their decimal positions. The results are as defined mathematically, with any negative values being less than zero, which in turn is less than any positive value. An index-name or index-item may appear in a comparison. Comparison of any two numeric operands is permitted regardless of the formats specified in their respective USAGE clauses, and regardless of length.

Character Comparisons

Non-equal-length comparisons are permitted, with spaces being assumed to extend the length of the shorter item, if necessary. Relationships are defined in the ASCII code; in particular, the letters A-Z are in an ascending sequence, and digits are less than letters. Group items are treated simply as characters when compared. Refer to Appendix D for all ASCII character representations. If one operand is numeric and the other is not, the numeric operand must be an integer and have an implicit or explicit USAGE IS DISPLAY.

Returning to the discussion of simple conditions, there are three additional forms of a simple condition, in addition to the relational form, namely: class test, condition-name test (88), and sign test.

A class-test condition has the following syntactical format:

```
identifier IS [NOT] { NUMERIC    }
                    { ALPHABETIC }
```

This condition specifies an examination of the data-item content to determine whether all characters are proper digit representations regardless of any operational sign (when the test is for NUMERIC), or only alphabetic or blank space characters (when the test is for ALPHABETIC).

The NUMERIC test is valid only for a group, decimal, or character item (not having an alphabetic PICTURE). The ALPHABETIC test is valid only for a group or character item (alphanumeric PICTURE).

Example

```
IF NUM-VALUE IS NOT ALPHABETIC
   PERFORM NUMERIC-ROUTINE.
```

A sign-test condition has the following syntactical format:

```
                                  { POSITIVE }
arithmetic-expression IS [ NOT ]  { NEGATIVE }
                                  { ZERO     }
```

This test is equivalent to comparing an arithmetic expression to zero in order to determine the truth of the stated condition.

Example

```
IF RECORD-COUNT NOT ZERO
   NEXT SENTENCE
ELSE
   PERFORM INITIALIZE-ROUTINE.
```

In a condition-name-test condition, a conditional variable is tested to determine whether its value is equal to one of the values associated with the condition-name. A condition-name test is expressed by the following syntactical format:

```
condition-name
```

where condition-name is defined by a level 88 DATA DIVISION entry.

Example

```
IF END-OF-FILE
   PERFORM EOF-ROUTINE.
```

INSPECT Statement

Purpose Enables the programmer to examine a character-string item. Options permit various combinations of the following actions:

1. Counting appearances of a specified character
2. Replacing a specified character with another
3. Limiting the above actions by requiring the appearance of other specific characters

Format The general formats are:

```
INSPECT identifier-1 TALLYING

{ identifier-2 FOR { { {ALL    } {identifier-3} } [ {BEFORE} INITIAL {identifier-4} ] } ... } ...
                   { { {LEADING} {literal-1   } }   {AFTER }         {literal-2   }          }
                   { CHARACTERS                 }
```

```
INSPECT identifier-1 REPLACING

{ CHARACTERS BY {identifier-6} [ {BEFORE} INITIAL {identifier-7} ]                                    }
{               {literal-4   }   {AFTER }         {literal-5   }                                     }
{                                                                                                      }
{ { {ALL    } { {identifier-5} BY {identifier-6} [ {BEFORE} INITIAL {identifier-7} ] } ... } ...       }
{ { {LEADING} { {literal-3   }    {literal-4   }   {AFTER }         {literal-5   }          }          }
{ { {FIRST  }                                                                                          }
```

```
INSPECT identifier-1 TALLYING

{ identifier-2 FOR { { {ALL    } {identifier-3} } [ {BEFORE} INITIAL {identifier-4} ] } ... } ...
                   { { {LEADING} {literal-1   } }   {AFTER }         {literal-2   }          }
                   { CHARACTERS                 }

 REPLACING

{ CHARACTERS BY {identifier-6} [ {BEFORE} INITIAL {identifier-7} ]                                    }
{               {literal-4   }   {AFTER }         {literal-5   }                                     }
{                                                                                                      }
{ { {ALL    } { {identifier-5} BY {identifier-6} [ {BEFORE} INITIAL {identifier-7} ] } ... } ...       }
{ { {LEADING} { {literal-3   }    {literal-4   }   {AFTER }         {literal-5   }          }          }
{ { {FIRST  }                                                                                          }
```

Remarks

In the remarks that follow, operand-n refers to the braced pair which consists of identifier-n and its associated literal, e.g., operand-5 represents {Identifier-5|literal-3}

Because identifier-1 is to be treated as a string of characters by INSPECT, it must not be described by USAGE IS INDEX, COMP-0 or COMP-3. Identifier-2 must be a numeric data-item.

The TALLYING clause and REPLACING clause may not both be omitted; if both are present, the TALLYING clause must be first.

The TALLYING clause causes character-by-character comparison, from left to right, of identifier-1, incrementing identifier-2 by one each time a match is found. The matching is done under the following conditions:

1. When an AFTER INITIAL operand-4 subclause is present, the counting process begins only after detection of a character in identifier-1 matching operand-4.
2. If BEFORE INITIAL operand-4 is specified, the counting process terminates upon encountering a character in identifier-1 which matches operand-4. Also going from left to right, REPLACING clause causes replacement of characters under conditions specified by the REPLACING clause.
3. If BEFORE INITIAL operand-7 is present, replacement does not continue after detection of a character in identifier-1 matching operand-7.
4. If AFTER INITIAL operand-7 is present, replacement does not commence until detection of a character in identifier-1 matching operand-7.

With bounds on identifier-1 thus determined, TALLYING and REPLACING is done on characters as specified by the following:

1. CHARACTERS implies that every character in the bounded identifier-1 is to be TALLYed or REPLACEd.

2. ALL operand-n means that all characters in the bounded identifier-1 which match the operand-n character are to participate in TALLYING/ REPLACING.

3. LEADING operand specifies that only characters matching operand-n from the left-most portion of the bounded identifier-1 which are contiguous (such as leading zeros) are to participate in TALLYING or REPLACING.

4. FIRST operand-n specifies that only the first-encountered character matching operand-n is to participate in REPLACING. (This option is unavailable in TALLYING.)

When both TALLYING and REPLACING clauses are present, the two clauses behave as if two INSPECT statements were written, the first containing only a TALLYING clause and the second containing only a REPLACING clause.

In developing a TALLYING value, the final result in identifier-2 is equal to the tallied count *plus* the initial value of identifier-2. In the first example below, the item COUNTX is assumed to have been set to zero initially elsewhere in the program.

Examples

```
INSPECT ITEM TALLYING COUNTX
   FOR ALL "L" REPLACING LEADING "A"
   BY "E" AFTER INITIAL "L".

Original (ITEM):      SALAMI     ALABAMA
Result (ITEM):        SALEMI     ALEBAMA
Final (COUNTX):            1           1

INSPECT WORK-AREA REPLACING ALL DELIMITER
    BY TRANSFORMATION

Original (WORK-AREA):      NEW YORK N Y
                           (length 16)
Original (DELIMITER):      (space)
Original (TRANSFORMATION):. (period)
Result (WORK-AREA):       NEW.YORK..N.Y...
```

Note

If any identifier-1 or operand-n is described as signed numeric, it is treated as if it were unsigned.

MERGE Statement

The MERGE statement is available only with implementations that use the MS-SORT facility. See the *Microsoft SORT Sorting Facility Reference Manual* for discussion of MERGE.

MOVE Statement

Purpose Moves data from one area of main storage to another and performs conversions and/or editing on the data that is moved.

Format The general format is:

```
MOVE {identifier-1} TO identifier-2 [, identifier-3] ...
     {literal     }
```

The data represented by identifier-1 or the specified literal is moved to the area designated by identifier-2. Additional receiving fields may be specified (identifier-3, etc.). When a group item is a receiving field, characters are moved without regard to the level structure of the group involved and without editing.

Remarks Subscripting or indexing associated with identifier-2 is evaluated immediately before data is moved to the receiving field. The same is true for other receiving fields (identifier-3, etc.), if any. But for the source field, subscripting or indexing (associated with identifier-1) is evaluated only once, before any data is moved.

To illustrate, consider the statement

```
MOVE A (B) TO B, C (B),
```

which is equivalent to

```
MOVE A (B) TO temp
MOVE temp TO B
MOVE temp TO C (B)
```

where temp is an intermediate result field assigned automatically by the compiler.

The following considerations pertain to moving items:

1. Numeric (external or internal decimal, binary, numeric literal, or ZERO) or alphanumeric to numeric or report:

   a. The items are aligned by decimal points, with generation of zeros or truncation on either end, as required. If source is alphanumeric, it is treated as an unsigned integer and should not be longer than 31 characters.

   b. When the types of the source field and receiving field differ, conversion to the type of the receiving field takes place. Alphanumeric source items are treated as unsigned integers with usage display.

   c. The items may have special editing performed on them with suppression of zeros, insertion of a dollar sign, etc., and decimal point alignment, as specified by the receiving area.

   d. One should not move an item whose PICTURE declares it to be alphabetic or alphanumeric edited to a numeric or report item, nor is it possible to move a numeric item of any sort to an alphabetic item. Though numeric integers and numeric report items can be moved to alphanumeric items with or without editing, operational signs are not moved in this case even if SIGN IS SEPARATE has been specified.

2. Non-numeric source and destinations:

   a. The characters are placed in the receiving area from left to right, unless JUSTIFIED RIGHT applies.

   b. If the receiving field is not completely filled by the data being moved, the remaining positions are filled with spaces.

   c. If the source field is longer than the receiving field, the move is terminated as soon as the receiving field is filled.

3. When overlapping fields are involved, results are not predictable.

4. Appendix B shows, in tabular form, all permissible combinations of source and receiving field types.

5. An index-data-item or an index-name cannot appear as an operand of a MOVE statement. See "SET Statement."

The following examples show data movement (a lowercase "b" represents a blank).

Source Field		Receiving Field		
PICTURE	**Value**	**PICTURE**	**Before MOVE**	**After MOVE**
99V99	1234	S99V99	9876-	1234+
99V99	1234	S99V9	987	123
S9V9	12-	99V999	98765	01200
XXX	A2C	XXXXX	Y9X8W	A2Cbb
9V99	123	99.99	87.65	01.23

MULTIPLY Statement

Purpose Multiplies two numeric data items and stores the product.

Format The general formats are:

```
MULTIPLY {identifier-1} BY identifier-2 [ROUNDED]
         {literal-1   }

      [; ON SIZE ERROR imperative-statement]

MULTIPLY {identifier-1} BY {identifier-2} GIVING identifier-3 [ROUNDED]
         {literal-1   }    {literal-2   }

      [; ON SIZE ERROR imperative-statement]
```

Remarks When the GIVING option is omitted, the second operand must be an identifier; the product replaces the value of identifier-2. For example, a new BALANCE value is computed by the statement MULTIPLY 1.03 BY BALANCE. Since this order might seem somewhat unnatural, it is recommended that GIVING always be written.

Example

```
MULTIPLY UNIT-PRICE BY QTY
    GIVING TOT-PRICE.
```

OPEN Statement

Purpose Accesses a file. The OPEN statement must be executed prior to commencing file processing.

Format The general format for all file organizations is:

```
       { INPUT  file-name-1 [, file-name-2] ... }
       { OUTPUT file-name-3 [, file-name-4] ... }
OPEN   { I-O    file-name-5 [, file-name-6] ... } ...
       { EXTEND file-name-7 [, file-name-8] ... }
```

Remarks For a SEQUENTIAL INPUT file, opening initiates reading the file's first records into memory, so that subsequent READ statements may be executed without waiting.

For an OUTPUT file, opening makes available a record area for development of one record, which will be transmitted to the assigned output device upon the execution of a WRITE statement. An existing file which has the same name will be overwritten by the file created with OPEN OUTPUT.

An OPEN I-O statement is valid only for a DISK file; it permits use of the REWRITE statement to modify records which have been accessed by a READ statement. The WRITE statement may not be used in I-O mode for files with SEQUENTIAL organization. The file must exist on disk at OPEN time; it cannot be created by OPEN I-O.

When the EXTEND phrase is specified, the OPEN statement positions the file immediately following the last logical record of that file. Subsequent WRITE statements referencing the file will add records to the end of the file. Thus, processing proceeds as though the file had been opened with the OUTPUT phrase and positioned at its end. EXTEND can be used only for SEQUENTIAL or LINE SEQUENTIAL files.

Failure to precede (in terms of time sequence) file reading or writing by the execution of an OPEN statement is an execution-time error which will cause abnormal termination of a program run. (See the *Microsoft COBOL Compiler User's Guide* for information on error messages.) Furthermore, a file cannot be opened if it has been CLOSEd "WITH LOCK".

SEQUENTIAL files opened for INPUT or I-O access must have been written in the appropriate format described in the *Microsoft COBOL Compiler User's Guide* for such files.

Example

```
OPEN INPUT INV-MSTR-FILE,
   OUTPUT INV-REPORT-FILE.
```

PERFORM Statement

Purpose Permits the execution of a separate body of program steps.

Format The general formats are:

```
PERFORM procedure-name-1 [ { THROUGH } procedure-name-2 ]
                           { THRU    }

PERFORM procedure-name-1 [ { THROUGH } procedure-name-2 ] { identifier-1 } TIMES
                           { THRU    }                    { integer-1    }

PERFORM procedure-name-1 [ { THROUGH } procedure-name-2 ] UNTIL condition-1
                           { THRU    }

PERFORM procedure-name-1 [ { THROUGH } procedure-name-2 ]
                           { THRU    }

    VARYING { identifier-2 } FROM { identifier-3 }
            { index-name-1 }      { index-name-2 }
                                  { literal-1    }

    BY { identifier-4 } UNTIL condition-1
       { literal-3    }

    [ AFTER { identifier-5 } FROM { identifier-6 }
            { index-name-3 }      { index-name-4 }
                                  { literal-3    }

    BY { identifier-7 } UNTIL condition-2
       { literal-4    }

    [ AFTER { identifier-8 } FROM { identifier-9 }
            { index-name-5 }      { index-name-6 }
                                  { literal-5    }

    BY { identifier-10 } UNTIL condition-3 ] ]
       { literal-6     }
```

Remarks

In the discussion which follows, range is defined as a paragraph-name, a section-name, or the construct procedure-name-1 THRU procedure-name-2. (THRU is synonymous with THROUGH.) If only a paragraph-name is specified, the return is after the paragraph's last statement. If only a section-name is specified, the return is after the last statement of the last paragraph of the section. If a range is specified, control is returned after the appropriate last sentence of a paragraph or section. These return points are valid only when a PERFORM has been executed to set them up; in other cases, control will pass right through.

The designated range may be performed a fixed number of times, as determined by an integer or by the value of an integer data-item. If no TIMES, UNTIL, or VARYING phrase is given, the range is performed once. When any PERFORM has finished, execution proceeds to the next statement following the PERFORM.

The range may instead be performed a variable number of times, with {identifier-2|index-name-1} varying from an initial value of {identifier-3|index-name-2|literal-1} with increments of {indentifier-4|literal-3} until a specified condition is met, at which time execution proceeds to the next statement after the PERFORM.

The condition in a PERFORM using the UNTIL phrase is evaluated prior to each attempted execution of the range. Consequently, it is possible to not PERFORM the range, if the condition is met at the outset. Similarly, if the TIMES phrase is used, if {identifier-1|integer-1} ≤ 0, the range is not performed at all. At runtime, it is illegal to have concurrently active PERFORM ranges whose end points are the same.

Examples

```
PERFORM P000-MAINLINE
     THRU P100-WRITE-REPORT.

PERFORM P050-INITIALIZE
     TBL-LENGTH TIMES.

PERFORM P100-WRITE-REPORT
     UNTIL END-OF-FILE.

PERFORM P200-LOOP
     VARYING SUB1 FROM 1 BY 1
     UNTIL (SUB1 > 10
     OR TABLE-VAL (SUB1) = 0).
```

READ Statement

The description of the READ statement differs for the various types of file organization. See Chapters 9, 10, and 11 for discussion of READ statements for SEQUENTIAL, INDEXED, and RELATIVE files, respectively.

READY/RESET TRACE Statements

Purpose Execution of a READY TRACE statement sets trace mode to cause printing of every section and paragraph name each time it is encountered. The RESET TRACE statement inhibits such printing.

Format The general formats for these statements are:

READY TRACE

RESET TRACE

Remarks A printed list of procedure-names in the order of their execution is invaluable in detecting a program error, because it helps find the point at which actual program flow departed from the expected program flow.

The READY TRACE, RESET TRACE, and EXHIBIT statements are extensions to ANSI 74 Standard COBOL.

Note

It is often desirable to include such statements on source lines that contain a "D" in column 7. In this case, the statements are ignored by the compiler unless the WITH DEBUGGING MODE clause is included in the SOURCE-COMPUTER paragraph.

Examples

```
      READY TRACE.
D     RESET TRACE.
```

In the second example, the "D" is in column 7 and "RESET TRACE" begins in column 12.

RELEASE Statement

The RELEASE statement is available only with implementations that have the MS-SORT facility. See the *Microsoft SORT Sorting Facility Reference Manual* for information about this statement.

RESET TRACE Statement

For a description of the RESET TRACE statement, see "READY/RESET TRACE Statements."

RETURN Statement

The RETURN statement is available only with implementations that have the MS-SORT facility. See the *Microsoft SORT Sorting Facility Reference Manual* for information about this statement.

REWRITE Statement

The REWRITE statement differs for the various types of file organizations. See Chapter 9, 10, and 11 for discussion of the REWRITE statement in SEQUENTIAL, INDEXED, and RELATIVE files, respectively.

SEARCH Statement

The SEARCH statement is used for the indexing method of table handling. See Chapter 8, "Table Handling by the Indexing Method," for discussion of this statement.

SET Statement

The SET statement is used for the indexing method of table handling. See Chapter 8, "Table Handling by the Indexing Method," for discussion of this statement.

SORT Statement

The SORT statement is available only with implementations that have the MS-SORT facility. See the *Microsoft SORT Sorting Facility Reference Manual* for information about this statement.

START Statement

The START statement is used only with INDEXED and RELATIVE files. See Chapters 10 and 11 for discussion of this statement.

STOP Statement

Purpose Terminates or delays execution of the object program.

Format The general format is:

```
STOP { RUN     }
     { literal }
```

Remarks STOP RUN terminates execution of a program, returning control to the operating system. If used in a sequence of imperative statements, it must be the last statement in that sequence.

The form STOP literal displays the specified literal on the terminal and suspends execution.

Execution of the program is resumed only after operator intervention. Presumably, the operator performs a function suggested by the content of the literal prior to resuming program execution by pressing the carriage return key.

Examples

```
CLOSE INV-MSTR-FILE, INV-WARNING-FILE.
STOP RUN.

STOP "CHANGE DISKETTE,
    THEN PRESS RETURN".
```

STRING Statement

Purpose Allows joining together of multiple sending data-item values into a single receiving item.

Format The general format is:

```
STRING { identifier-1 } { , identifier-2 } ... DELIMITED BY { identifier-3 }
       { literal-1    } { , literal-2    }                  { literal-3    }
                                                            { SIZE         }

    [ , { identifier-4 } [ , identifier-5 ] ... DELIMITED BY { identifier-6 } ] ...
    [   { literal-4    } [ , literal-5    ]                  { literal-6    } ]
    [                                                        { SIZE         } ]

    INTO identifier-7 [WITH POINTER identifier-8]

    [; ON OVERFLOW imperative-statement]
```

Remarks In this format, identifier-7 is the receiving data-item name, which must be alphanumeric without editing symbols or the JUSTIFIED clause; identifier-8 is a counter and must be an elementary numeric integer data-item of sufficient size (plus 1) to point to positions within identifier-7.

If no POINTER phrase exists, the default value of the logical pointer is one. The logical pointer value designates the beginning position of the receiving field into which data placement begins. During movement to the receiving field, the criteria for termination of an individual source are controlled by the DELIMITED BY phrase.

DELIMITED BY SIZE

The entire source field is moved (unless the receiving field becomes full).

DELIMITED BY an identifier or literal.

The character string specified by the identifier or literal is a "key" which, if found to match a like-numbered succession of sending characters, terminates the function for the current sending field (and causes automatic switching to the next sending field, if any).

If at any point the logical pointer (which is automatically incremented by one for each character stored into identifier-7) is less than one or greater than the size of identifier-7, no further data movement occurs, and the imperative statement given in the OVERFLOW phrase (if any) is executed. If there is no OVERFLOW phrase, control is transferred to the next executable statement.

There is no automatic space fill into any position of identifier-7. That is, unaccessed positions are unchanged upon completion of the STRING statement.

Upon completion of the STRING statement, if there was a POINTER phrase, the resultant value of identifier-8 equals its original value plus the number of characters moved during execution of the STRING statement.

Example

```
STRING OLD-NAME DELIMITED BY SIZE
    INTO NEW-NAME
    WITH POINTER STRING-PTR.

STRING OLD-NAME
    DELIMITED BY STR-DELIM
    INTO NEW-NAME
    ON OVERFLOW
        PERFORM P500-OVERFLOW.

STRING OLD-NAME-1, OLD-NAME-2
    DELIMITED BY SIZE
    INTO NEW-NAME.
```

The following lists show how the values in the last example are affected by the STRING statement.

Variable	PICTURE	SIZE
OLD-NAME-1	PIC X(5)	5
NEW-NAME	PIC X(10)	10
OLD-NAME-2	PIC X(5)	5

For these same variables, the contents are affected as follows:

Variable	Before String	After String
OLD-NAME-1	ABCDE	unchanged
NEW-NAME	1234567	ABCDEFGHIJ
OLD-NAME-2	FGHIJ	unchanged

SUBTRACT Statement

Purpose Subtracts one or more numeric data items from a specified item and stores the difference.

Format The general format is:

```
SUBTRACT { identifier-1 } [ , identifier-2 ] ... FROM { identifier-m }
         { literal-1    } [ , literal-2    ]          { literal-m    }

    GIVING identifier-n [ROUNDED]

    [; ON SIZE ERROR imperative-statement]
```

Remarks The effect of the SUBTRACT statement is to sum the values of all the operands that precede FROM and subtract that sum from the value of the item following FROM.

The result (difference) is stored in identifier-n, if there is a GIVING option. Otherwise, the result is stored in identifier-m.

Example

```
SUBTRACT TOT-EXP,
    TOT-DEDUCTIONS FROM TOT-EARNINGS
    GIVING NET-INCOME.
```

UNSTRING Statement

Purpose Causes data in a single sending field to be separated into subfields that are placed into multiple receiving fields.

Format The general format is:

```
UNSTRING identifier-1

  [ DELIMITED BY [ALL] { identifier-2 } [ OR [ALL] { identifier-3 } ] ... ]
                       { literal-1    }            { literal-2    }

    INTO identifier-4 [, DELIMITER IN identifier-5] [, COUNT IN identifier-6]

    [, identifier-7 [, DELIMITER IN identifier-8] [, COUNT IN identifier-9] ] ...

    [WITH POINTER identifier-10] [TALLYING IN identifier-11]

    [; ON OVERFLOW imperative-statement]
```

Remarks The braced items {identifier-2|literal-1} and {identifier-3|literal-2} may be referred to in the following remarks as operand-i, where "i" refers to the number of the identifier being discussed.

Criteria for separation of subfields may be given in the DELIMITED BY phrase. Each time a succession of characters matches one of the non-numeric literals, one-character figurative constants, or data-item values named by operand-2, the current collection of sending characters is terminated and moved to the next receiving field specified by the INTO clause. When the ALL phrase is specified, more than one contiguous occurrence of operand-2 in identifier-1 is treated as one occurrence.

When two or more delimiters exist, an 'OR' condition exists. Each delimiter is compared to the sending field in the order specified in the UNSTRING statement.

Identifier-1 must be a group or character string (alphanumeric) item. When a data-item is employed as an operand, that operand must also be a group or character string item.

Receiving fields (identifiers 4,7,. . .) may be any of the following types of items:

1. an unedited alphabetic item
2. a character-string (alphanumeric) item
3. a group item
4. an external decimal item (numeric, usage DISPLAY) whose PICTURE does not contain any P character

When any examination encounters two contiguous delimiters, the current receiving area is either space or zero filled depending on its type. If there is a DELIMITED BY phrase in the UNSTRING statement, then there may be DELIMITER IN phrases following any receiving item (e.g., identifier-4) mentioned in the INTO clause. In this case, the character(s) that delimit the data moved into identifier-4 are themselves stored in identifier-5, which should be an alphanumeric item. Furthermore, if a COUNT IN phrase is present, the number of characters that were moved into identifier-4 is moved to identifier-6, which must be an elementary numeric integer item.

If there is a POINTER phrase, then identifier-10 must be an integer numeric item, and its initial value becomes the initial logical pointer value (otherwise, a logical pointer value of one is assumed). The examination of source characters begins at the position in identifier-1 specified by the logical pointer; upon completion of the UNSTRING statement, the final logical pointer value will be copied back into identifier-10.

If at any time the value of the logical pointer is less than one or exceeds the size of identifier-1, then overflow is said to occur and control passes over to the imperative statements given in the ON OVERFLOW clause, if any.

Overflow also occurs when all receiving fields have been filled prior to exhausting the source field.

During the course of source field scanning (looking for matching delimiter sequences), a variable length character string is developed which, when completed by recognition of a delimiter or by acquiring as many characters as the size of the current receiving field can hold, is then moved to the current receiving field in the standard MOVE fashion.

If there is a TALLYING IN phrase, identifier-11 must be an integer numeric item. The number of receiving fields acted upon, plus the initial value of identifier-11, will be produced in identifier-11 upon completion of the UNSTRING statement.

Any subscripting or indexing associated with identifier-1, 10, or 11 is evaluated only once at the beginning of the UNSTRING statement. Any subscripting associated with operand-i or identifier-4 through identifier-9 is evaluated immediately before access to the data-item.

Example

```
UNSTRING FIELD-A DELIMITED BY SPACES
    INTO FIELD-B.
```

USE Sentence

The USE sentence is found only in the DECLARATIVES region of the PROCEDURE DIVISION. See Chapter 12, "Declaratives and the USE Sentence," for discussion of this statement.

WRITE Statement

The WRITE statement differs for the various types of file organizations. See Chapters 9, 10, and 11 for discussion of the WRITE statement in SEQUENTIAL, INDEXED, and RELATIVE files, respectively.

Chapter 7

INTER-PROGRAM COMMUNICATION

Separately compiled MS-COBOL program modules may be combined into a single executable program. Inter-program communication is made possible through the use of the LINKAGE SECTION of the DATA DIVISION (which follows the WORKING-STORAGE SECTION) and by the CHAIN and CALL statements and the USING list appendage to the PROCEDURE DIVISION header of a subprogram module.

The LINKAGE SECTION describes data made available in memory from another program module. Record-description entries in the LINKAGE SECTION provide data-names by which data-areas reserved in memory by other programs may be referenced. Entries in the LINKAGE SECTION do not reserve memory areas because the data is assumed to be present elsewhere in memory, in a calling program.

Any record-description entry may be used to describe items in the LINKAGE SECTION as long as the VALUE clause is not specified for other than level 88 items.

CALL Statement

The CALL statement causes control to be transferred temporarily from one object program to another within the same run unit.

The format of the CALL statement is:

```
CALL literal-1 [USING data-name-1 [, data-name-2] ... ]
```

Literal-1 is a subprogram name defined as the PROGRAM-ID of a separately compiled program, and must be a non-numeric (quoted) literal. Data names in the USING list are made available to the called subprogram by passing addresses to the subprogram; these addresses are assigned to the LINKAGE SECTION items declared in the USING list of that subprogram. Therefore the number and order of data-names specified in matching CALL and PROCEDURE DIVISION USING lists must be identical. Information-passing conventions at the machine language level are described in the *Microsoft COBOL Compiler User's Guide.*

Note

Correspondence between caller and callee lists is by position, not by identical spelling of names.

EXIT PROGRAM Statement

The EXIT PROGRAM statement, appearing in a called subprogram, causes control to be returned to the next executable statement after CALL in the calling program. This statement must be a paragraph by itself.

CHAIN Statement

The CHAIN statement causes a specified program to be loaded into memory and executed.

The CHAIN statement is coded according to the following format:

```
CHAIN {literal     } [USING identifier-2 ...]
      {identifier-1}
```

Literal and identifier-1 must be alphanumeric, and identifier-1 must contain a terminating space. Each occurrence of identifier-2 must be defined in the WORKING-STORAGE or LINKAGE SECTION or in the record area of a file open at the time the CHAIN statement is executed.

When the CHAIN statement is executed, the value of literal or identifier-1, up to but not including the first space encountered (or the end of the literal), is interpreted as the name of an executable program file in the format of the appropriate operating system. The named program is loaded into memory and executed. All program and data structures of the CHAINing program are permanently destroyed except that the USING clause may be used to transfer parameters to the CHAINed program. See the following discussion on "PROCEDURE DIVISION Header With CALL and CHAIN."

The CHAINed program need not be an MS-COBOL program. If it is, it must be a main program.

PROCEDURE DIVISION Header With CALL and CHAIN

The PROCEDURE DIVISION header of a main program is coded as:

```
PROCEDURE DIVISION [{USING   } data-name-1 [, data-name-2] ...] .
                    {CHAINING}
```

where the PROCEDURE DIVISION header of the main program uses CHAINING, and the PROCEDURE DIVISION header of the subprogram uses USING.

The various forms of the PROCEDURE DIVISION header describe the linkage and parameter initialization requirements of a program. A main program must be linked by itself or with any number of subprograms. It may then be run independently or invoked by the execution of a CHAIN statement in another program. A subprogram must be linked with exactly one main program and, optionally, any number of other subprograms. It may only be executed by the action of a CALL statement. For a description of the linking process, see the *Microsoft COBOL Compiler User's Guide.*

Warning

A CHAINed or CALLed program should have a CHAINing list or nonempty USING list if and only if the invoking CHAIN or CALL statement has a USING list. Furthermore, the numbers of entries in the lists should be equal, and positionally corresponding entries in the two lists should reference data items of the same size and USAGE. Failure to conform to these rules will not be diagnosed and will cause unpredictable results at runtime.

The values of the data items named in the PROCEDURE DIVISION header are established at program initialization time by using the contents of positionally corresponding data items named in the invoking CALL or CHAIN statement. In the case of CALL, the identification is made by passing pointers. Therefore, if the value of a data item named in a PROCEDURE DIVISION USING clause is changed during subprogram execution, the corresponding data item in the CALLing program will reflect the change after control is returned from the subprogram.

For a description of the formats in which parameters are passed by the CALL and CHAIN statements, see the *Microsoft COBOL Compiler User's Guide.*

Chapter 8

TABLE HANDLING BY THE INDEXING METHOD

This chapter describes the indexing method of table handling.

Index-Names and Index-Items

An index-name is declared not by the usual method of level number, name, and data description clauses, but implicitly by appearance in the "INDEXED BY index-name" appendage to an OCCURS clause. An index-name must be unique.

An index-data-item is an item defined by the USAGE IS INDEX phrase. An index-data-item must not have a PICTURE clause. An index-name or index-data-item may only be referred to by a SET or SEARCH statement, a CALL statement's USING list or a PROCEDURE DIVISION header USING list; or used in a relational condition or as the variation item in a PERFORM VARYING statement, or in place of a subscript. In all cases the process is equivalent to dealing with a binary word integer subscript. Index-name must be initialized to some value before use via SET, SEARCH or PERFORM.

SET Statement

The SET statement permits the manipulation of index-names, index-items, or binary subscripts for table handling purposes. There are two formats:

```
                                                    { identifier-3  }
SET { identifier-1  } [, identifier-2]    ... } TO  { index-name-3  }
    { index-name-1  } [, index-name-2]    ... }     { integer-1     }

SET   index-name-4   [, index-name-5] ... { UP BY   } { identifier-4 }
                                          { DOWN BY } { integer-2    }
```

Format 1 is equivalent to moving the TO value (e.g., integer-2) to multiple receiving fields written immediately after the verb SET.

Format 2 is equivalent to reduction (DOWN) or increase (UP) applied to each of the quantities written immediately after the verb SET; the amount of the reduction or increase is specified by a name or value immediately following the word BY.

In any SET statement, identifiers are restricted to integer items.

Relative Indexing

A user reference to an item in a table controlled by an OCCURS clause is expressed with a proper number of subscripts (or indexes), separated by commas, and enclosed in matching parentheses. For example:

```
TAX-RATE   (BRACKET, DEPENDENTS)
XCODE  (I, 2)
```

where subscripts are ordinary integer decimal data-names, or integer constants, or binary integer (COMPUTATIONAL-0 or INDEX) items, or index-names. Subscripts may be qualified, but not subscripted. A subscript may be signed, but if so, it must be positive. The lowest acceptable value is 1, pointing to the first element of a table. The highest permissible value is the maximum number of occurrences of the item as specified in its OCCURS clause.

A further capability exists, called relative indexing. In this case, an index is expressed as

```
index-name + integer constant
```

where a space must be on either side of the plus or minus sign.

For example:

```
XCODE (I + 3, J - 1).
```

Format 1 SEARCH Statement

A linear search of a table may be done using the SEARCH statement. The general format is:

```
                        [            { identifier-2 } ]
SEARCH identifier-1     [ VARYING    { index-name-1 } ]   [; AT END imperative-statement-1]

        ; WHEN condition-1  { imperative-statement-2 }
                            { NEXT SENTENCE          }
```

Identifier-1 is the name of a data-item having an OCCURS clause that includes an INDEXED-BY list; identifier-1 must be written without subscripts or indexes because the nature of the SEARCH statement causes automatic variation of an index-name associated with a particular table.

There are four possible VARYING cases:

1. NO VARYING phrase

   The first-listed index-name for the table is varied.

2. VARYING index-name in a different table

   The first-listed index-name in the table's definition is varied, implicitly, and the index-name listed in the VARYING phrase is varied in like manner, simultaneously.

3. VARYING index-name defined for table

   This specific index-name is the only one varied.

4. VARYING integer data-item name

   Both this data-item and the first-listed index-name for table are varied, simultaneously.

The term "variation" has the following interpretation:

1. The initial value is assumed to have been established by an earlier statement such as SET.
2. If the initial value exceeds the maximum declared in the applicable OCCURS clause, the SEARCH operation terminates at once; and if an AT END phrase exists, the associated imperative statement is executed.

3. If the value of the index is within the range of valid indexes (1,2,. . . up to and including the maximum number of occurrences), then each WHEN condition is evaluated until one is true or all are found to be false. If one is true, its associated imperative statement is executed and the SEARCH operation terminates. If none is true, the index is incremented by one and the steps in this paragraph are repeated. Note that incrementation of index applies to whatever item and/or index is selected according to the four cases listed above.

If the table is subordinate to another table, an index-name must be associated with each dimension of the entire table via INDEXED BY phrases in all the OCCURS clauses. Only the index-name of the SEARCH table is varied (along with another "VARYING" index-name or data-item). To search an entire two- or three-dimensional table, a SEARCH must be executed several times with the other index-names set appropriately each time, probably with a PERFORM,VARYING statement.

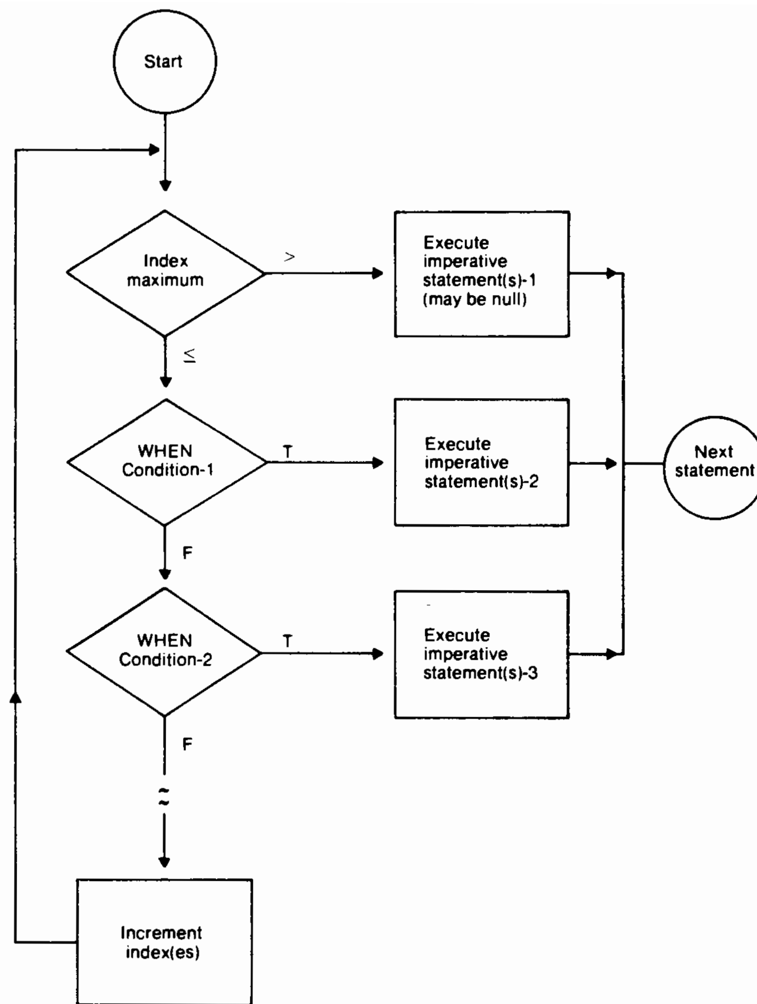

Figure 8.1. Logic Diagram for Format 1 SEARCH Statement

Format 2 SEARCH Statement

Format 2 SEARCH statements deal with tables of ordered data. The general format of such a SEARCH ALL statement is:

```
SEARCH ALL identifier-1 [; AT END imperative-statement-1]

          { data-name-1       } { IS EQUAL TO } { identifier-3            }
  ; WHEN  {                   } { IS =        } { literal-1               }
          { condition-name-1  }                 { arithmetic-expression-1 }

  [       data-name-2         { IS EQUAL TO } { identifier-4            } ]
  [ AND                       { IS =        } { literal-2               } ] ...
  [       condition-name-2                    { arithmetic-expression-2 } ]

  { imperative-statement-2 }
  { NEXT SENTENCE          }
```

Only one WHEN clause is permitted.

The following rules apply to the condition:

1. Only simple relational conditions or condition-names may be employed, and the subject must be properly indexed by the first index-name associated with identifier-1 (along with sufficient other indexes if multiple OCCURS clauses apply). Furthermore, each subject data-name (or the data-name associated with a condition-name) in the condition must be mentioned in the KEY clause of the table. The KEY clause is an appendage to the OCCURS clause having the following format:

   ```
   OCCURS integer-1 TIMES

   [ { ASCENDING  } KEY IS data-name-4 [, data-name-5]... ] ...
   [ { DESCENDING }                                      ]

   [INDEXED BY index-name-1 [, index-name-2] ...]]
   ```

   where the data-names are the names defined in this data-description entry (following level number) or one of the subordinate data-names. If more than one data-name is given, then all of them must be the names of entries subordinate to this group item.

   The KEY phrase indicates that the repeated data is arranged in ascending or descending order according to the data-names which are listed (in any given KEY phrase) in decreasing order of significance. More than one KEY phrase may be specified.

2. In a simple relational condition, only the equality test (using relation = or IS EQUAL TO) is permitted.

3. Any condition-name variable (level 88 items) must be defined as having only a single value.
4. The condition may be compounded by use of the logical connector AND, but not OR.
5. In a simple relational condition, the object (to the right of the equal sign) may be a literal or an identifier; the identifier must not be referenced in the KEY clause of the table or be indexed by the first index-name associated with the table. (The term identifier means data-name, including any qualifiers and/or subscripts or indexes.)

Warning

Failure to conform to the restrictions described in the preceding list may yield unpredictable results. Unpredictable results also occur if the table data is not ordered in conformance to the declared KEY clauses, or if the keys referenced in the WHEN-condition are not sufficient to identify a unique table element.

In a Format 2 SEARCH, a nonserial type of search operation may take place, relying upon the declared ordering of data. The initial setting of the index-name for table is ignored and its setting is varied automatically during the searching, always within the bounds of the maximum number of occurrences. If the condition (WHEN) cannot be satisfied for any valid index value, control is passed to imperative-statement-1, if the AT END clause is present, or to the next executable sentence in the case of no AT END clause.

If all the simple conditions in the single WHEN condition are satisfied, the resultant index value indicates an occurrence that allows those conditions to be satisfied, and control passes to imperative-statement-2. Otherwise the final setting is not predictable.

Chapter 9

SEQUENTIAL FILES

Definition of Sequential File Organization

Sequential file organization provides the capability of accessing the records of a file in an established sequence. Each record in the file except the first has a unique predecessor, and each record except the last has a unique successor. The order in which the records are stored is established by the order in which they were written when the file was created. This order does not change, except that records may be added to the end of the file.

In a sequential file, records are accessed in the order in which they were originally written.

There are two organizations of sequential files:

The SEQUENTIAL format consists of a 2-byte record length followed by the record itself, for as many records as exist in the file. This is the default format for files created by an MS-COBOL program.

The LINE SEQUENTIAL organization consists of records followed by carriage return and linefeed delimiters, for as many records as exist in the file. This type of file is often produced by non-COBOL programs, such as editors. No COMP-0 or COMP-3 information should be written into a LINE SEQUENTIAL file because these data-items may contain the same binary codes used for carriage return and line feed, and this would subsequently cause a problem when the file is read.

Note

If files in LINE SEQUENTIAL format are to be used as input to MS-COBOL programs, the ORGANIZATION IS LINE SEQUENTIAL phrase must be specified in the SELECT clause of the input file. If an attempt is made to read a LINE SEQUENTIAL file without this specification, a runtime error will result.

Syntax Considerations

Information about file organization is specified in the ENVIRONMENT DIVISION of a program. The general format for the FILE-CONTROL paragraph in the ENVIRONMENT DIVISION is:

```
FILE-CONTROL.

SELECT file-name

         ASSIGN TO { DISK    }
                   { PRINTER }

         [ ; RESERVE integer [ AREA  ] ]
                             [ AREAS ]

         [ ; ORGANIZATION IS [ LINE ] SEQUENTIAL ]

         [ ; ACCESS MODE IS SEQUENTIAL ]

         [ ; FILE STATUS IS data-name-1 ] .
```

For sequential organization, the SELECT clause must be specified first in the FILE-CONTROL paragraph. The clauses which follow the SELECT clause may appear in any order.

The SELECT clause must also specify ORGANIZATION IS SEQUENTIAL or ORGANIZATION IS LINE SEQUENTIAL. The ORGANIZATION clause is the only place where the distinction is made between the regular SEQUENTIAL and the LINE SEQUENTIAL organizations. In all other places, the description "sequential" refers to both the SEQUENTIAL and LINE SEQUENTIAL organizations.

Because sequential organization is the most common type of file organization, most optional clauses in MS-COBOL default to sequential organization. For example, the ORGANIZATION and ACCESS MODE clauses, if not specified, default to ORGANIZATION IS SEQUENTIAL and ACCESS MODE IS SEQUENTIAL, respectively. For LINE SEQUENTIAL files, the ACCESS MODE clause should be specified as ACCESS MODE IS SEQUENTIAL, or it should be omitted.

The general formats for the clauses used with sequential files are given in Chapter 4, "ENVIRONMENT DIVISION."

In the DATA DIVISION of the program, the FILE SECTION header begins the FILE SECTION. It is followed by a period (.). Following the header, FD (file definition) entries are included for each file that was described in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION. FD entries specify the size of the logical and physical records, the value of implementor-defined label items, names of the data records which make up the file, and the number of lines to be included on a logical printer page. The FD entry is ended by a period (.).

The general formats for FD entries and for record-description and data-description entries are given in Chapter 5, "DATA DIVISION."

File Status Reporting

If the FILE STATUS clause is specified in the FILE-CONTROL paragraph, the designated two-character data-item is set after any OPEN, CLOSE, READ, WRITE, or REWRITE statement and before any USE procedure is executed. The value of this data-item indicates to the program the status of the input-output operation. The possible settings are shown in the following table.

	Data Status Data Item RIGHT Character				
Status	*No Further Description (0)*	*Structure Error (1)*	*Duplicate Key (2)*	*No Record Found (3)*	*Disk Space Full (4)*
Successful Completion (0)	X				
At End(1)	X				
Permanent Error(3)	X				X
Special Cases(9)		X			

Table 9.1. SEQUENTIAL File Status Settings

In an OPEN INPUT or OPEN I-O statement, a file status of "30" means "File Not Found."

If file status "91" should occur in an OPEN EXTEND statement, it indicates that the end of the file could not be correctly determined.

PROCEDURE DIVISION Statements for Sequential Files

The statements that are used with sequential input and output are:

```
OPEN
CLOSE
READ
WRITE
REWRITE
DELETE
USE
```

The general formats for the OPEN and CLOSE statements are the same for SEQUENTIAL, INDEXED, and RELATIVE files. These statements are described in Chapter 6, "PROCEDURE DIVISION Statements." The USE statement applies only to SEQUENTIAL files, and it is used only in the DECLARATIVES region of the PROCEDURE DIVISION. See Chapter 12, "Declaratives and the USE Sentence," for discussion of USE.

The remaining statements differ for SEQUENTIAL, INDEXED, and RELATIVE file organizations. The formats and descriptions that apply to sequential files are given in the remainder of this chapter.

DELETE Statement

Purpose Removes a record from the file. The record that is deleted is the last one that was read.

Format The general format for a sequential file is:

DELETE file-name RECORD

Example DELETE INV-REC-FILE RECORD.

READ Statement

Purpose Makes available the next logical data record of the designated file from the assigned device, and updates the value of the FILE STATUS data item, if one was specified.

Format The general format for a sequential file is:

```
READ file-name RECORD [INTO identifier] [; AT END imperative-statement]
```

Since at some time the end-of-file will be encountered, the user should include the AT END clause.

Remarks The reserved word END is followed by any number of imperative statements, all of which are executed only if the end-of-file situation arises. The last statement in the AT END series must be followed by a period to indicate the end of the sentence. If end-of-file occurs but there is no AT END clause on the READ statement, an applicable DECLARATIVES procedure is performed. If neither AT END nor DECLARATIVE exists and no FILE STATUS item is specified for the file, a runtime I-O error is processed.

When a data record to be read exists, successful execution of the READ statement is immediately followed by execution of the next sentence.

When more than one level 01 item is subordinate to a file definition, these records share the same storage area. Therefore, the user must be able to distinguish between the types of records that are possible, in order to determine exactly which type is currently available. This is accomplished with a data comparison, using an IF statement to test a field which has a unique value for each type of record.

The INTO option permits the user to specify that a copy of the data record is to be placed into a designated data field in addition to the file's record area. The data-name must not be defined in the FILE SECTION.

Also, the INTO phrase should not be used when the file has records of various sizes, as indicated by their record descriptions. Any subscripting or indexing of data-name is evaluated after the data has been read but before it is moved to data-name. Afterward, the data is available in both the file record and data-name.

In the case of a blocked input file (such as disk files), not every READ statement performs a physical transmission of data from an external storage device; instead, READ may simply obtain the next logical record from an input buffer.

If the actual record is shorter than the file record area, the file record area is padded on the right with spaces.

Example

```
READ INV-MSTR-FILE
    INTO WS-MSTR-REC
    AT END MOVE "Y" TO END-OF-FILE-SW.
```

REWRITE Statement

Purpose Replaces a logical record on a SEQUENTIAL disk file.

Format The general format for a SEQUENTIAL file is:

REWRITE record-name [FROM identifier]

Record-name is the name of a logical record in the FILE SECTION of the DATA DIVISION and may be qualified. Record-name and identifier must refer to separate storage areas.

Remarks At the time of execution of this statement, the file to which record-name belongs must be open in the I-O mode.

If a FROM part is included in this statement, the effect is as if MOVE data-name TO record-name were executed just prior to the REWRITE.

Execution of REWRITE replaces the record that was accessed by the most recent READ statement; the READ must have been completed successfully. If the record which is rewriting the record in the file is longer than the file's record, only as many bytes as will fit are actually rewritten. On the other hand, if the record which is rewriting the record in the file is shorter than the file's record, unpredictable information will be written after the record, until the beginning of the next record in the file.

Example

```
REWRITE PR-REC FROM INV-COUNT.
```

WRITE Statement

Purpose Releases a logical record for an output or input-output file.

Format The general format for a SEQUENTIAL file is:

```
WRITE record-name [FROM identifier-1]

    [ {BEFORE} ADVANCING { {identifier-2} [LINE ] } ]
    [ {AFTER } ADVANCING { {integer     } [LINES] } ]
    [                    { PAGE                   } ]

    [ ; AT {END-OF-PAGE} imperative-statement ]
    [      {EOP        }                      ]
```

Remarks In MS-COBOL, file output is achieved by execution of the WRITE statement. Depending on the device assigned, "written" output may take the form of printed matter or magnetic recording on a floppy disk storage medium. Remember also that you READ filename, but you WRITE record-name. The associated file must be open in the OUTPUT mode at time of execution of a WRITE statement.

Record-name must be one of the level 01 records defined for an output file, and may be qualified by the filename.

If the data to be output has been developed in WORKING-STORAGE or in another area (for example, in an input file's record area), the FROM suffix permits the user to stipulate that the designated data (data-name-1) is to be copied into the record-name area and then output from there. Record-name and data-name-1 must refer to separate storage areas.

When an attempt is made to write beyond the externally defined boundaries of a SEQUENTIAL file, a DECLARATIVES procedure will be executed (if available) and the FILE STATUS (if available) will indicate a boundary violation. If neither is available, a runtime error occurs.

The ADVANCING option is restricted to line printer output files, and permits the programmer to control the line spacing on the paper in the printer.

{Identifier-1|integer} may have values from 0 to 120.

Integer	Carriage Control Action
0	No spacing
1	Normal single spacing
2	Double spacing
3	Triple spacing
.	.
.	.
.	.

Single spacing (i.e., "after advancing 1 line") is assumed if there is no BEFORE or AFTER option in the WRITE statement.

Use of the key word AFTER implies that the carriage control action precedes printing a line, whereas use of BEFORE implies that writing precedes the carriage control action. If PAGE is specified, the data is printed BEFORE or AFTER the printer is repositioned to the next physical page. However, if a LINAGE clause is associated with the file, the repositioning is to the first line that can be written on the next logical page as specified in the LINAGE clause.

If the END-OF-PAGE phrase is specified, the LINAGE clause must be specified in the file description entry for the associated file. EOP is equivalent to END-OF-PAGE.

An end-of-page condition is reached whenever a WRITE statement with the END-OF-PAGE phrase causes printing or spacing within the footing area of a page body. This occurs when such a WRITE statement causes the LINAGE-COUNTER to equal or exceed the value specified by the FOOTING value, if specified. In this case, after the WRITE statement is executed, the imperative statement in the END-OF-PAGE phrase is executed.

A "page overflow" condition is reached whenever a WRITE statement cannot be fully accommodated within the current page body. This occurs when a WRITE statement would cause the LINAGE-COUNTER to exceed the value specified as the size of the page body in the LINAGE clause. In this case, the record is printed before or after (depending on the phrase used) the printer is repositioned to the first line of the next logical page. The imperative statement in the END-OF-PAGE clause, if specified, is executed after the record is written and the printer has been repositioned.

Clearly, if no FOOTING value is specified in the LINAGE clause, or if the end-of-page and overflow conditions occur simultaneously, then only the overflow condition is effective.

Example

```
WRITE REPORT-REC FROM PR-HEADER
    AFTER ADVANCING PAGE.
```

Chapter 10

INDEXED FILES

Definition of INDEXED File Organization

An INDEXED file organization provides for recording and accessing records of a "data base" by keeping a directory (called the "control index") of pointers that enable direct location of records having particular unique key values. An INDEXED file must be assigned to DISK in its defining SELECT sentence.

Each INDEXED file declared in an MS-COBOL program will generate two disk files: a key file and a data file. The file specification in the VALUE OF FILE-ID clause specifies a file containing data only. The filename included in the file specification is joined with the extension .KEY to form the file specification of the key file.

Key File

The key file contains keys, pointers to keys, and pointers to data. A key file is divided into 256-byte units, called "granules." There are five possible granule types. A type indicator is located in the first byte of each granule. The granule type indicators have the following values:

Value	Type Indicator
1	Data Set Control Block
2	Key Set Control Block
3	Node
4	Leaf
5	Deleted granule

The key file will have only one Data Set Control Block in the first granule, one Key Set Control Block for the primary file key, and additional Key Set Control Blocks for alternate keys.

Each Data Set Control Block and Key Set Control Block contains, in the fourth byte, a "damaged" flag which notifies you when the last file use was not terminated properly. The runtime executor sets these flags to nonzero values when the file is opened for updating and restores them to zero when the file is closed.

Data File

The data file consists of data records. Each data record is preceded by a two-byte field and a one-byte "reference count" that indicates whether a record has been deleted. The data file is terminated by a control record with a length field containing a "2", followed by two bytes of high-values.

A file whose organization is INDEXED can be accessed either sequentially, dynamically or randomly.

Sequential access provides access to data records in ascending order of RECORD KEY values.

In the random access mode, the order of access to records is controlled by the programmer. Each record desired is accessed by placing the value of its key in a key data item prior to an access statement.

In the dynamic access mode, the programmer's logic may change from sequential access to random access, and vice versa, at will.

Syntax Considerations

In the ENVIRONMENT DIVISION, the SELECT entry must specify ORGANIZATION IS INDEXED. The general format for the SELECT entry is:

```
FILE-CONTROL.
SELECT file-name
      ASSIGN TO DISK
    [; RESERVE integer [AREA ]]
                       [AREAS]
     ; ORGANIZATION IS INDEXED
    [                  {SEQUENTIAL}]
    [; ACCESS MODE IS  {RANDOM    }]
    [                  {DYNAMIC   }]
     ; RECORD KEY IS data-name-1
    [; FILE STATUS IS data-name-3] .
```

ASSIGN, RESERVE, and FILE STATUS clause formats are identical to those specified in Chapter 4, "ENVIRONMENT DIVISION."

In the FD entry for an INDEXED file, both LABEL RECORDS STANDARD and a VALUE OF FILE-ID clause must appear. The formats of Chapter 5, "DATA DIVISION," apply, except that only the DISK-related forms are applicable.

RECORD KEY Clause

The general format of the RECORD KEY clause, which is required, is:

```
; RECORD KEY IS data-name-1
```

where data-name-1 is an item defined within the record descriptions of the associated file description, and is a group item or an elementary alphanumeric item. The maximum key length is 60 bytes and the key should never be made to contain all nulls.

If random access mode is specified, the value of data-name-1 designates the record to be accessed by the next DELETE, READ, REWRITE or WRITE statement. Each record must have a unique RECORD KEY value.

File Status Reporting

If a FILE STATUS clause appears in the ENVIRONMENT DIVISION for an INDEXED organization file, the designated two-character data-item is set after every I-O statement. The following table summarizes the possible settings:

	Data Status Data Item RIGHT Character				
Status	*No Further Description (0)*	*Structure Error (1)*	*Duplicate Key (2)*	*No Record Found (3)*	*Disk Space Full (4)*
Successful Completion (0)	X				
At End(1)	X				
Invalid Key(2)		X	X	X	X
Permanent Error(3)	X				X
Special Cases(9)		X			

Table 10.1. INDEXED File Status Settings

File status "21" arises if ACCESS MODE IS SEQUENTIAL when WRITEs do not occur in ascending sequence for an INDEXED file, or the key is altered prior to a rewrite. In an OPEN INPUT or OPEN I-O statement, a File status of "30" means "File Not Found." File status "91" occurs on an OPEN INPUT or OPEN I-O statement for a relative or indexed file whose structure has been destroyed (for example, by a system crash during output to the file). When this status is returned on an OPEN INPUT, the file is considered to be open, and READs may be executed. On an OPEN I-O, however, the file is not considered to be open, and all I-O operations fail.

Note that "Disk Space Full" occurs with INVALID KEY (2) for INDEXED and RELATIVE file handling, whereas it occurred with "Permanent Error" (3) for SEQUENTIAL files.

If an error occurs at execution time and no AT END or INVALID KEY statements are given, and no appropriate DECLARATIVES error section is supplied, and no FILE STATUS is specified, the error will be displayed on the console and the program will terminate.

PROCEDURE DIVISION Statements for INDEXED Files

The syntax of the SEQUENTIAL file OPEN statement also applies to INDEXED organized files, except that EXTEND is inapplicable.

The following table summarizes the available statement types and their permissibility in terms of ACCESS mode and OPEN option in effect.

ACCESS MODE IS	*Procedure Statement*	*OPEN Option in Effect*		
		Input	*Output*	*I-O*
SEQUENTIAL	READ	X		X
	WRITE		X	
	REWRITE			X
	START	X		X
	DELETE			X
RANDOM	READ	X		X
	WRITE		X	X
	REWRITE			X
	START			
	DELETE			
DYNAMIC	READ	X		X
	WRITE		X	X
	REWRITE			X
	START	X		X
	DELETE			X

Table 10.2. I-O Permitted With INDEXED Files

In the preceding table, an "X" indicates the statement is permissible. CLOSE is permissible under all conditions.

The statements described in the remainder of this chapter differ for SEQUENTIAL, INDEXED, and RELATIVE files.

DELETE Statement

Purpose Logically removes a record from the INDEXED file.

Format The general format for INDEXED files is:

```
DELETE file-name RECORD [; INVALID KEY imperative-statement]
```

Remarks For a file in the sequential access mode, the last input-output statement executed for file-name would have been a successful READ statement. The record that was read is deleted. Consequently, no INVALID KEY phrase should be specified for sequential-access mode files.

For a file having random or dynamic access mode, the record deleted is the one associated with the RECORD KEY; if there is no such matching record, the INVALID KEY condition exists, and control passes to the imperative statements in the INVALID KEY clause, or to an applicable DECLARATIVES error section if no INVALID KEY clause exists.

Example

```
DELETE INV-REC RECORD
    INVALID KEY DISPLAY "KEY NOT FOUND".
```

READ Statement

Purpose Makes available the next logical data record of the designated file from the assigned device, and updates the value of the FILE STATUS data item, if one was specified.

Format The general formats for INDEXED files are:

```
READ file-name [NEXT] RECORD [INTO identifier]
    [; AT END imperative-statement]

READ file-name RECORD [INTO identifier]
    [; KEY IS data-name]
    [; INVALID KEY imperative-statement]
```

Remarks Format 1 without NEXT must be used for all files having sequential access mode. Format 1 with the NEXT option is used for sequential reads of a dynamic access mode file. The AT END clause is executed when the logical end-of-file condition arises. If this clause is not written in the source statement, an appropriately assigned DECLARATIVES error section is given control at end-of-file time, if available.

Format 2 is used for files in random-access mode or for files in dynamic-access mode when records are to be retrieved randomly.

In Format 2, the INVALID KEY clause specifies action to be taken if the access key value does not refer to an existing key in the file. If the clause is not given, the appropriate DECLARATIVES error section, if supplied, is given control.

The optional KEY IS clause must designate the record key item declared in the file's SELECT entry. This clause serves as documentation only. The user must ensure that a valid key value is in the designated key field prior to execution of a random-access READ.

The rules for sequential files regarding the INTO phrase apply here as well.

Examples

```
READ INV-REC-FILE NEXT RECORD
    INTO REC-COUNT
    AT END PERFORM P300.

READ INV-REC-FILE RECORD INTO REC-COUNT
    KEY IS DATE-REC
    INVALID KEY DISPLAY "REC NOT FOUND".
```

REWRITE Statement

Purpose Logically replaces an existing record.

Format The general format for INDEXED files is:

<u>REWRITE</u> record-name [<u>FROM</u> identifier] [; <u>INVALID</u> KEY imperative-statement]

Remarks For a file in sequential-access mode, the last READ statement must have been successful in order for a REWRITE statement to be valid. If the value of the record key in record-name (or corresponding part of data-name, if FROM appears in the statement) does not equal the key value of the immediately previous READ, then the invalid key condition exists and the imperative statements are executed, if present; otherwise an applicable DECLARATIVES error section is executed, if available.

For a file in a random or dynamic access mode, the record to be replaced is specified by the record key; no previous READ is necessary. The INVALID KEY condition exists when the record key's value does not equal that of any record stored in the file.

Example

```
REWRITE PR-REC FROM INV-REC
    INVALID KEY PERFORM P300.
```

START Statement

Purpose Enables an INDEXED organization file to be positioned for reading at a specified key value. This is permitted for files open in either sequential or dynamic access modes.

Format The general format for INDEXED files is:

```
                    [       ( IS EQUAL TO        )           ]
                    [       ( IS =               )           ]
START file-name     [ KEY   { IS GREATER THAN    } data-name ]
                    [       ( IS >               )           ]
                    [       ( IS NOT LESS THAN   )           ]
                    [       ( IS NOT <           )           ]

    [; INVALID KEY imperative-statement]
```

Data-name must be the declared record key and the value to be matched by a record in the file must be prestored in the data-name.

Remarks When this statement is executed, the file must be open in the input or I-O mode.

If the KEY phrase is not present, equality between a record in the file and the record key value is sought. If key relation GREATER or NOT LESS is specified, the file is positioned for next access at the first record greater than, or greater than or equal to, the indicated key value.

If no matching record is found, the imperative statements in the INVALID KEY clause are executed, or an appropriate DECLARATIVES error section is executed.

Example

```
START INV-REC-FILE
    KEY IS EQUAL TO QTY-RECEIVED
    INVALID KEY DISPLAY "KEY NOT FOUND".
```

WRITE Statement

Purpose Releases a logical record for an output or input-output file.

Format The general format for INDEXED files is:

```
WRITE record-name [FROM identifier] [; INVALID KEY imperative-statement]
```

Remarks Just prior to executing the WRITE statement, a valid (unique) value must be in that portion of the record-name (or data-name-1 if FROM appears in the statement) which serves as RECORD KEY.

In the event of an improper key value, the imperative statements are executed if the INVALID KEY clause appears in the statement; otherwise an appropriate DECLARATIVES error section is invoked, if applicable. The INVALID KEY condition arises if:

1. for sequential access, key values are not ascending from one WRITE to the next WRITE
2. the key value is not unique
3. the allocated disk space is exceeded

Example

```
WRITE INV-REC FROM PR-REC
    INVALID KEY DISPLAY "REC NOT FOUND".
```

Chapter 11

RELATIVE FILES

Definition of RELATIVE File Organization

RELATIVE file organization is restricted to disk files. Records are differentiated on the basis of a relative record number, which ranges from 1 to 32,767, or to a lesser maximum for a smaller file. Unlike the case of an INDEXED file, where the identifying key field occupies a part of the data record, relative record numbers are conceptual and are not embedded in the data records. Relative file records are fixed length records whose length is that of the largest record in the file.

A RELATIVE file may be accessed either sequentially, dynamically, or randomly. In sequential access mode, records are accessed in the order of ascending record numbers.

In random access mode, the sequence of record access is controlled by the program, by placing a number in a relative key item. In dynamic access mode, the program may intermix random and sequential access at will.

Syntax Considerations

In the ENVIRONMENT DIVISION, the SELECT entry must specify ORGANIZATION IS RELATIVE. The general format for the SELECT clause of a RELATIVE file is:

```
FILE-CONTROL.

SELECT file-name

        ASSIGN TO DISK

       [                    [AREA  ]]
       [; RESERVE integer   [AREAS ]]

        ; ORGANIZATION IS RELATIVE

       [                  { SEQUENTIAL [, RELATIVE KEY IS data-name-1]  }]
       [; ACCESS MODE IS  {                                             }]
       [                  { { RANDOM  }                                 }]
       [                  { { DYNAMIC } , RELATIVE KEY IS data-name-1   }]

       [; FILE STATUS IS data-name-2 ] .
```

ASSIGN, RESERVE, and FILE STATUS clause formats are identical to those used for SEQUENTIAL or INDEXED files.

In the associated FD entry, STANDARD labels must be declared and a VALUE OF FILE-ID clause must be included.

The first byte of the record area associated with a RELATIVE file should not be set to binary zero by being described as part of a COMP-0 or COMP-3 item, nor set to LOW-VALUES by any record description for the file.

RELATIVE KEY Clause

In addition to the usual clauses in the SELECT entry, the "RELATIVE KEY IS data-name-1" clause is required for random or dynamic access mode. It is also required for sequential-access mode, if a START statement exists for such a file.

Data-name-1 must be described as an unsigned binary integer item not contained within any record description of the file itself. Its value must be positive and nonzero.

File Status Reporting

If a FILE STATUS clause appears in the ENVIRONMENT DIVISION for a RELATIVE file, the designated two-character data-item is set after every I-O statement. The following table summarizes the possible settings:

	Data Status Data Item RIGHT Character				
Status	*No Further Description (0)*	*Structure Error (1)*	*Duplicate Key (2)*	*No Record Found (3)*	*Disk Space Full (4)*
Successful Completion (0)	X				
At End(1)	X				
Invalid Key(2)		X	X	X	X
Permanent Error(3)	X				X
Special Cases(9)		X			

Table 11.1. RELATIVE File Status Settings

Note that the settings for RELATIVE files are the same as those for INDEXED files.

In an OPEN INPUT or OPEN I-O statement, a file status of "30" means "File Not Found."

File status "91" occurs on a OPEN INPUT or OPEN I-O statement for a relative or indexed file whose structure has been destroyed (for example, by a system crash during output to the file). When this status is returned on an OPEN INPUT, the file is considered to be open, and READs may be executed. On an OPEN I-O, however, the file is not considered to be open, and all I-O operations fail.

"Disk Space Full" occurs with INVALID KEY (2) for RELATIVE and INDEXED file handling, whereas it occurred with "Permanent Error" (3) for SEQUENTIAL files.

PROCEDURE DIVISION Statement for RELATIVE Files

Within the PROCEDURE DIVISION, the verbs OPEN, CLOSE, READ, WRITE, REWRITE, DELETE, and START are available, just as for files whose organization is INDEXED.

The formats for OPEN and CLOSE are the same as those described in Chapter 6, "PROCEDURE DIVISION," except that the EXTEND phrase is applicable to the OPEN statement for RELATIVE files.

The following table summarizes the available statement types and their permissibility in terms of ACCESS mode and OPEN option in effect.

ACCESS MODE IS	*Procedure Statement*	*OPEN Option in Effect*		
		Input	*Output*	*I-O*
SEQUENTIAL	READ	X		X
	WRITE		X	
	REWRITE			X
	START	X		X
	DELETE			X
RANDOM	READ	X		X
	WRITE		X	X
	REWRITE			X
	START			
	DELETE			
DYNAMIC	READ	X		X
	WRITE		X	X
	REWRITE			X
	START	X		X
	DELETE			X

Table 11.2. I-O Permitted With RELATIVE Files

In the preceding table, an "X" indicates the statement is permissible. CLOSE is permissible under all conditions.

The other input-output statements are described in the remainder of this chapter.

DELETE Statement

Purpose

Removes a record from the file. The record that is deleted is the last one that was read.

Format

The format of the DELETE statement is the same for a RELATIVE file as it is for an INDEXED file:

DELETE file-name RECORD [; INVALID KEY imperative-statement]

Remarks

For a file in a sequential access mode, the immediately previous action would have been a successful READ statement; the record thus previously made available is logically removed from the file. If the previous READ was unsuccessful, a runtime error will terminate execution. Therefore, an INVALID KEY phrase may not be specified for sequential-access mode files.

For a file with dynamic or random access mode declared, the removal action pertains to whatever record is designated by the value in the RELATIVE KEY item. If no such numbered record exists, the INVALID KEY condition arises.

Example

```
DELETE INV-REC RECORD
    INVALID KEY DISPLAY "KEY NOT FOUND".
```

READ Statement

Purpose

Makes available the next logical data record of the designated file from the assigned device, and updates the value of the FILE STATUS data item, if one was specified.

Format

The general formats for RELATIVE files are:

```
READ file-name [NEXT] RECORD [INTO identifier]
     [; AT END imperative-statement]
READ file-name RECORD [INTO identifier] [; INVALID KEY imperative-statement]
```

Remarks

Format 1 must be used for all files in sequential access mode. The NEXT phrase must be present to achieve sequential access if the file's declared mode of access is dynamic. The AT END clause, if given, is executed when the logical end-of-file condition exists, or, if not given, the appropriate DECLARATIVES error section is given control, if available.

Format 2 is used to achieve random access with declared mode of access either random or dynamic.

If a RELATIVE KEY is defined (in the file's SELECT entry), successful execution of a Format 1 READ statement updates the contents of the RELATIVE KEY item (data-name-1) so as to contain the record number of the record retrieved.

For a Format 2 READ, the record that is retrieved is the one whose relative record number is prestored in the RELATIVE KEY item. If no such record exists, however, the INVALID KEY condition arises, and is handled by

1. the imperative statements given in the INVALID KEY portion of the READ, or
2. an associated DECLARATIVES region.

The rules for SEQUENTIAL files regarding the INTO phrase apply here as well.

Examples

```
READ INV-REC-FILE NEXT RECORD
    INTO REC-COUNT
    AT END PERFORM P300.
READ INV-REC-FILE RECORD INTO REC-COUNT
    INVALID KEY DISPLAY "REC NOT FOUND".
```

REWRITE Statement

Purpose Replaces a logical record in the file.

Format The format of the REWRITE statement is the same for a RELATIVE file as it is for an INDEXED file:

REWRITE record-name [FROM identifier] [; INVALID KEY imperative-statement]

Remarks For a file in sequential access mode, the immediately previous action would have been a successful READ; the record thus previously made available is replaced in the file by executing REWRITE. If the previous READ was unsuccessful, a runtime error will terminate execution. Therefore, no INVALID KEY clause is allowed for sequential access.

For a file with dynamic or random access mode declared, the record that is replaced by executing REWRITE is the one whose ordinal number is preset in the RELATIVE KEY item. If no such item exists, the INVALID KEY condition arises.

Example

```
REWRITE PR-REC FROM INV-REC
    INVALID KEY PERFORM P300.
```

START Statement

Purpose

Enables a RELATIVE file to be positioned for reading at a specified key value. This statement is allowed only for files whose access mode is defined as sequential or dynamic.

Format

The format of the START statement is the same for a RELATIVE file as it is for an INDEXED file:

```
                 [       { IS EQUAL TO        }           ]
                 [       { IS =               }           ]
START file-name  [  KEY  { IS GREATER THAN    } data-name ]
                 [       { IS >               }           ]
                 [       { IS NOT LESS THAN   }           ]
                 [       { IS NOT <           }           ]

     [; INVALID KEY imperative-statement]
```

Remarks

Data-name may only be that of the previously declared RELATIVE KEY item, and the number of the relative record must be stored in it before START is executed. When this statement is executed, the associated file must be currently open in INPUT or I-O mode.

If the KEY phrase is not present, equality between a record in the file and the record key value is sought. If key relation GREATER or NOT LESS is specified, the file is positioned for next access at the first record greater than, or greater than or equal to, the indicated key value.

If no such relative record is found, the imperative statements in the INVALID KEY clause are executed, or an appropriate DECLARATIVES error section is executed.

Example

```
START INV-REC-FILE
    KEY IS EQUAL TO QTY-RECEIVED
    INVALID KEY DISPLAY "KEY NOT FOUND".
```

WRITE Statement

Purpose

Releases a logical record for an output or input-output file.

Format

The format of the WRITE statement is the same for a RELATIVE file as it is for an INDEXED file:

WRITE record-name [FROM identifier] [; INVALID KEY imperative-statement]

Remarks

If access mode is sequential, then completion of a WRITE statement causes the relative record number of the record just output to be placed in the RELATIVE KEY item.

If access mode is random or dynamic, then the user must preset the value of the RELATIVE KEY item in order to assign the record an ordinal (relative) number. The INVALID KEY condition arises if there already exists a record having the specified ordinal number, or if the disk space is exceeded.

Example

```
WRITE INV-REC FROM PR-REC
    INVALID KEY DISPLAY "REC NOT FOUND".
```

Chapter 12

DECLARATIVES AND THE USE SENTENCE

The DECLARATIVES region provides a method of including procedures that are executed not as part of the sequential coding written by the programmer, but rather when a condition that cannot normally be tested by the programmer occurs.

Although the system automatically handles checking and creation of standard labels and executes error recovery routines in the case of input-output errors, additional procedures may be specified by the COBOL programmer.

Since these procedures are executed only at the time an error in reading or writing occurs, they cannot appear in the regular sequence of procedural statements. They must be written at the beginning of the PROCEDURE DIVISION in a subdivision called DECLARATIVES. Related procedures are preceded by a USE sentence that specifies their function. A declarative section ends with the occurrence of another section-name with a USE sentence or with the key words END DECLARATIVES.

The key words DECLARATIVES and END DECLARATIVES must each begin in Area A and be followed by a period (.).

The general format is:

```
[DECLARATIVES.
{section-name SECTION [segment-number]. declarative-sentence
[paragraph-name. [sentence] ...] ...} ...
 END DECLARATIVES. ]
```

The USE sentence defines the applicability of the associated section of coding.

A USE sentence, when present, must immediately follow a section header in the DECLARATIVES region of the PROCEDURE DIVISION and must be followed by a period followed by a space. The remainder of the section must consist of zero, one, or more procedural paragraphs that define the procedures to be used. The USE sentence itself is never executed; rather, it defines the conditions for the execution of the USE procedure. The general format of the USE sentence is:

```
                                                       { file-name-1 [, file-name-2] ... }
                                                       { INPUT                           }
USE AFTER STANDARD { EXCEPTION } PROCEDURE ON          { OUTPUT                          }
                   { ERROR     }                       { I-O                             }
                                                       { EXTEND                          }
```

The words EXCEPTION and ERROR may be used interchangeably. The associated DECLARATIVES region is executed (by the PERFORM mechanism) after the standard I-O recovery procedures for the files designated, or after the INVALID KEY or AT END condition arises on a statement lacking the INVALID KEY or AT END clause. A given filename may not be associated with more than one DECLARATIVES region.

Within a DECLARATIVES region there must be no reference to any nondeclarative procedure. Conversely, in the nondeclarative portion there must be no reference to procedure-names that appear in the DECLARATIVES region, except that PERFORM statements may refer to a USE statement and its procedures; but in a range specification (see Chapter 6 "PERFORM Statement") if one procedure-name is in a DECLARATIVES region, then the other must be in the same DECLARATIVES region.

An exit from a DECLARATIVES region is inserted by the compiler following the last statement in the section. All logical program paths within the section must lead to the exit point.

Chapter 13

SEGMENTATION

The program segmentation facility is provided to enable the execution of Microsoft COBOL programs that are larger than physical memory. When segmentation is used (that is, when any section header in the program contains a segment number), the entire PROCEDURE DIVISION must be written in sections. Each section is assigned a segment number by a section header of the form:

[section-name SECTION [segment-number].

Segment-number must be an integer with a value in the range from 0 through 99. If the segment-number is omitted, it is assumed to be 0. DECLARATIVES regions must have segment-numbers less than 50. All sections which have the same segment number constitute a single program segment and must occur together in the source program. Furthermore, all segments with numbers less than 50 must occur together at the beginning of the PROCEDURE DIVISION.

Segments with numbers 0 through 49 are called fixed segments and are always resident in memory during execution.

Segments with numbers greater than 49 are called independent segments. Each independent segment is treated as a program overlay. An independent segment is in its initial state when control is passed to it for the first time during the execution of a program, and also when control is passed to that section (implicitly or explicitly) from another segment with a different segment number. Specifically, an independent segment is in its initial state when it is reached by "falling through" the end of a fixed or different independent segment.

Segmentation causes the following restrictions on the use of the ALTER and PERFORM statements:

1. A GO TO statement in an independent segment must not be referred to by an ALTER statement in any other segment.

2. A PERFORM statement in a fixed segment may have within its range only

   a. sections and/or paragraphs wholly contained within fixed segments, or

   b. sections and/or paragraphs wholly contained in a single independent segment

3. A PERFORM statement in an independent segment may have within its range only

   a. sections and/or paragraphs wholly contained within fixed segments, or

   b. sections and/or paragraphs wholly contained within the same independent segment as the PERFORM statement

Appendix A

ADVANCED FORMS OF CONDITIONS

Evaluation Rules for Compound Conditions

The following list presents rules for compound conditions:

1. Evaluation of individual simple conditions (relation, class, condition-name, and sign test) is done first.
2. AND-connected simple conditions are evaluated next as a single result.
3. OR and its adjacent conditions (or previously evaluated results) are then evaluated.

Examples:

1. `A < B OR C = D OR E NOT > F`

   The evaluation is equivalent to (A<B) OR (C=D) OR (E<F) and is true if any of the three individual parenthesized simple conditions is true.

2. `WEEKLY AND HOURS NOT = 0`

   The evaluation is equivalent, after expanding level 88 condition-name WEEKLY, to:

   ```
   (PAY-CODE = 'W') AND (HOURS NOT = 0)
   ```

   and is true only if both the simple conditions are true.

3. ```
   A = 1 AND B = 2 AND G = -3
             OR P NOT EQUAL TO "SPAIN"
   ```

   is evaluated as:

   ```
   [(A = 1) AND (B = 2) AND (G = -3)]
        OR (P NOT = "SPAIN")
   ```

   If P = "SPAIN", the compound condition can only be true if all three of the following are true:

   ```
   (c.1) A = 1
   (c.2) B = 2
   (c.3) G = -3
   ```

   However, if P is not equal to "SPAIN", the compound condition is true regardless of the values of A, B and G.
   ```

Parenthesized Conditions

Parentheses may be written within a compound condition or parts thereof in order to take precedence in the evaluation order.

Example:

```
IF A = B AND (A = 5 OR A = 1)
  PERFORM PROCEDURE-44.
```

In this case, PROCEDURE-44 is executed if A = 5 OR A = 1 while at the same time A = B. In this manner, compound conditions may be formed containing other compound conditions, not just simple conditions, with the use of parentheses.

Abbreviated Conditions

For the sake of brevity, the user may omit the "subject" when it is common to several successive relational tests. For example, the condition A = 5 OR A = 1 may be written A = 5 OR = 1. This may also be written A = 5 OR 1, where both subject and relation being implied are the same.

Another example:

```
IF A = B OR < C OR Y
```

is a shortened form of

```
IF A = B OR A < C OR A < Y
```

The interpretation applied to the use of the word NOT in an abbreviated condition is:

1. If the item immediately following NOT is a relational operator, then the NOT participates as part of the relational operator;
2. Otherwise, the beginning of a new, completely separate condition must follow NOT, not to be considered part of the abbreviated condition.

Warning

Abbreviations in which the subject and relation are implied are permissible only in relation tests; the subject of a sign-test or class-test cannot be omitted.

NOT, the Logical Negation Operator

In addition to its use as a part of a relation (e.g., IF A IS NOT = B), NOT may precede a condition. For example, the condition NOT (A = B OR C) is true when (A = B OR A = C) is false. The word NOT may precede a level 88 condition name, also.

Appendix B

TABLE OF PERMISSIBLE MOVE OPERANDS

Source Operand	Num. Int.	Num. Non-Int.	Num. Edit.	Alpha-Num. Edit.	Alpha-Num.	Group
Num. Int.	OK	OK	OK	OK(A)	OK(A)	OK(B)
Num. Non-Int.	OK	OK	OK			OK(B)
Num. Edit.				OK	OK	OK(B)
Alpha-Num. Edit.				OK	OK	OK(B)
Alpha-Num.	OK(C)	OK(C)	OK(C)	OK	OK	OK(B)
Group	OK(B)	OK(B)	OK(B)	OK(B)	OK(B)	OK(B)

The characters (A), (B), and (C) in the preceding table indicate:

(A) Source sign, if any, is ignored.
(B) If the source operand or the receiving operand is a group item, the move is considered to be a group move.
(C) Source is treated as an unsigned integer; source length may not exceed 31.

Note

No distinction is made in the compiler between alphabetic and alphanumeric. Therefore, numeric items should not be moved to alphabetic items, and vice versa.

Appendix C

NESTED IF STATEMENTS

A "nested IF" exists when the conjunction IF appears more than once in a single sentence.

Example:

```
IF X = Y
   IF A = B
      MOVE "*" TO SWITCH
   ELSE
      MOVE "A" TO SWITCH
ELSE
   MOVE SPACE TO SWITCH.
```

The flow of the preceding example may be represented by a tree structure. Such a structure is illustrated by the figure below.

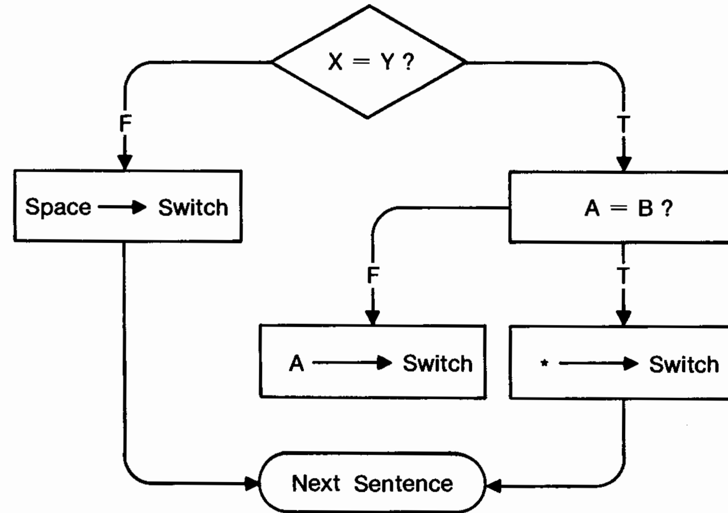

Figure C.1. Tree Structure of Nested IF Statements

Another useful way of viewing nested IF structures is based on numbering IF and ELSE verbs to show their priorities.

```
IF1    X = Y
  true-action1:
    IF2    A = B
      true-action2:
        MOVE "*" TO SWITCH
    ELSE2
      false-action2:
        MOVE "A" TO SWITCH
ELSE1
  false-action1:
    MOVE SPACE TO SWITCH.
```

The preceding illustration shows that IF2 is wholly nested within the true-action side of IF1.

The number of ELSEs in a sentence need not be the same as the number of IFs; there may be fewer ELSE branches.

Examples:

```
IF M = 1
     IF K = 0
        GO TO M1-K0
     ELSE
     GO TO M1-KNOT0.

IF AMOUNT IS NUMERIC
   IF AMOUNT IS ZERO
       GO TO CLOSE-OUT.
```

In the latter case, IF2 could equally well have been written as AND.

Appendix D

ASCII CHARACTER SET FOR ANSI 74 COBOL

Character	Octal Value	Character	Octal Value
A	101	0	60
B	102	1	61
C	103	2	62
D	104	3	63
E	105	4	64
F	106	5	65
G	107	6	66
H	110	7	67
I	111	8	70
J	112	9	71
K	113	SPACE	40
L	114	"	42
M	115	$	44
N	116	' non-ANSI	47
O	117	(	50
P	120	)	51
Q	121	*	52
R	122	+	53
S	123	,	54
T	124	-	55
U	125	.	56
V	126	/	57
W	127	;	73
X	130	<	74
Y	131	=	75
Z	132	>	76

Appendix E

RESERVED WORDS

A plus sign (+) before a reserved word indicates additional words required by Microsoft COBOL for interactive screens, debug extensions, and packed decimal format.

```
 ACCEPT              CHARACTER(S)
 ACCESS              CLOCK-UNITS
 ADD                 CLOSE
 ADVANCING           COBOL
 AFTER               CODE
 ALL                 CODE-SET
 ALPHABETIC         +COL
 ALSO                COLLATING
 ALTER               COLUMN
 ALTERNATE           COMMA
 AND                 COMMUNICATION
 ARE                 COMP
 AREA(S)             COMPUTATIONAL
 ASCENDING          +COMPUTATIONAL-0
+ASCII              +COMP-0
 ASSIGN             +COMPUTATIONAL-3
 AT                 +COMP-3
 AUTHOR              COMPUTE
+AUTO-SKIP           CONFIGURATION
                     CONTAINS
+BEEP                CONTROL(S)
 BEFORE              COPY
 BLANK               CORR(ESPONDING)
 BLOCK               COUNT
 BOTTOM              CURRENCY
 BY
                     DATA
 CALL                DATE
 CANCEL              DATE-COMPILED
 CD                  DATE-WRITTEN
 CF                  DAY
 CH                  DEBUGGING
```

DEBUG-CONTENTS
DEBUG-ITEM
DEBUG-LINE
DEBUG-NAME
DEBUG-SUB-1
DEBUG-SUB-2
DEBUG-SUB-3
DECIMAL-POINT
DECLARATIVES
DELETE
DELIMITED
DELIMITER
DEPENDING
DESCENDING
DESTINATION
DE(TAIL)
DISABLE
+DISK
DISPLAY
DIVIDE
DIVISION
DOWN
DUPLICATES
DYNAMIC

EGI
ELSE
EMI
ENABLE
END
END-OF-PAGE
ENTER
ENVIRONMENT
EOP
EQUAL
+ERASE
ERROR
ESI
EVERY
EXCEPTION
+EXHIBIT
EXIT
EXTEND

FD
FILE
FILE CONTROL
+FILE-ID
FILLER
FINAL
FIRST
FOOTING
FOR
FROM
GENERATE
GIVING
GO
GREATER
GROUP

HEADING
HIGH-VALUE(S)

IDENTIFICATION
IF
IN
INDEX
INDEXED
INITIAL
INITIATE
INPUT
INPUT-OUTPUT
INSPECT
INSTALLATION
INTO
INVALID
IS
I-O
I-O-CONTROL

JUST(IFIED)

KEY

LABEL
LAST
LEADING
LEFT
+LEFT-JUSTIFY
LENGTH
+LENGTH-CHECK
LESS
LIMIT(S)
+LIN
LINAGE
LINAGE-COUNTER
LINE(S)
LINE-COUNTER
LINKAGE
LOCK
LOW-VALUE(S)

MEMORY
MERGE

MESSAGE
MODE
MODULES
MOVE
MULTIPLE
MULTIPLY

NATIVE
NEGATIVE
NEXT
NO
NOT
NUMBER
NUMERIC

OBJECT-COMPUTER
OCCURS
OF
OFF
OMITTED
ON
OPEN
OPTIONAL
OR
ORGANIZATION
OUTPUT
OVERFLOW

PAGE
PAGE-COUNTER
PERFORM
PF
PH
PIC(TURE)
PLUS
POINTER
POSITION
POSITIVE
+PRINTER
PRINTING
PROCEDURE(S)
PROCEED
PROGRAM
PROGRAM-ID
+PROMPT

QUEUE
QUOTE(S)
RANDOM
RD
READ
+READY
RECEIVE
RECORD(S)
REDEFINES
REEL
REFERENCES
RELATIVE
RELEASE
REMAINDER
REMOVAL
RENAMES
REPLACING
REPORT(S)
REPORTING
RERUN
RESERVE
RESET
RETURN
REVERSED
REWIND
REWRITE
RF
RH
RIGHT
+RIGHT-JUSTIFY
ROUND
RUN

SAME
SD
SEARCH
SECTION
SECURITY
SEGMENT
SEGMENT-LIMIT
SELECT
SEND
SENTENCE
SEPARATE
SEQUENCE

SEQUENTIAL
SET
SIGN
SIZE
SORT
SORT-MERGE
SOURCE
SOURCE-COMPUTER
SPACE(S)
+SPACE-FILL
SPECIAL-NAMES
STANDARD
STANDARD-1
START
STATUS
STOP
STRING
SUB-QUEUE-1,2,3
SUBTRACT
SUM
SUPPRESS
SYMBOLIC
SYNC(HRONIZED)

TABLE
TALLYING
TAPE
TERMINAL
TERMINATE
TEXT
THAN
THROUGH
THRU
TIME
TIMES
TO
TOP
+TRACE
TRAILING
+TRAILING-SIGN
TYPE
UNIT
UNSTRING
UNTIL
UP
+UPDATE
UPON
USAGE
USE
USING

VALUE(S)
VARYING

WHEN
WITH
WORDS
WORKING-STORAGE
WRITE

ZERO((E)S)
+ZERO-FILL

+
-
*
/
**
<
>
=

```
 TABLE
 TALLYING         +
 TAPE             -
 TERMINAL         *
 TERMINATE        /
 TEXT             **
 THAN             <
 THROUGH          >
 THRU             =
 TIME
 TIMES
 TO
 TOP
+TRACE
```

Appendix F

PERFORM WITH VARYING AND AFTER CLAUSES

The general format for the PERFORM statement with VARYING and AFTER clauses is:

```
PERFORM procedure-name-1 [ { THROUGH } procedure-name-2 ]
                           { THRU    }

                                       { identifier-3 }
VARYING { identifier-2 }   FROM        { index-name-2 }
        { index-name-1 }               { literal-1    }

BY { identifier-4 } UNTIL  condition-1
   { literal-3    }

[        { identifier-5 }              { identifier-6 }
[ AFTER  { index-name-3 }  FROM        { index-name-4 }
[                                      { literal-3    }

BY { identifier-7 } UNTIL  condition-2
   { literal-4    }

[        { identifier-8 }              { identifier-9 }
[ AFTER  { index-name-5 }  FROM        { index-name-6 }
[                                      { literal-5    }

BY { identifier-10 } UNTIL  condition-3 ] ]
   { literal-6     }
```

The operation of this complex PERFORM statement is equivalent to the following MS-COBOL statements (example varying three items and using the identifier option for braced items):

```
START-PERFORM.
     MOVE identifier-3 TO identifier-2
     MOVE identifier-6 TO identifier-5
     MOVE identifier-9 TO identifier-8

TEST-CONDITION-1.
     IF condition-1 GO TO END-PERFORM.

TEST-CONDITION-2.
     IF condition-2
        MOVE identifier-6 to identifier-5
        ADD identifier-4 TO identifier-2
        GO TO TEST-CONDITION-1.

TEST-CONDITION-3.
     IF condition-3
        MOVE identifier-9 TO identifier-8
        ADD identifier-7 TO identifier-5
        GO TO TEST-CONDITION-2.

PERFORM range
        ADD identifier-10 TO identifier-8
        GO TO TEST-CONDITION-3.

END-PERFORM.   Next statement.
```

Note

If any identifier in the preceding example were an index-name, the associated MOVE would instead be a SET (TO form), and the associated ADD would be a SET (UP form).

INDEX

A

ACCEPT statement
- ANSI 74 extension Intro-8
- data input field 6-16
- described 6-10
- format 1 6-11
- format 2 6-13
- format 3 6-15
- format 4 6-27
- use as reference 1-7

ACCESS MODE clause 4-13, 10-3, 11-2
ADD STATEMENT 6-29
Advanced conditions A-1
ADVANCING option 9-9
ALL phrase
- figurative constant 1-11
- with UNSTRING 6-64

ALPHABET-NAME clause 4-16
Alphanumeric
- edited item 5-40
- item 5-8, 5-40
- receiving field 6-17

ALTER statement Intro-7, 6-30, 13-2
- MS-COBOL extensions Intro-8
- modules Intro-6

Arithmetic
- expressions 1-15
- operators 1-15
- statements Intro-7, 1-14, 6-5

ASSIGN clause 4-4, 4-7, 4-13
AT END clause 6-8, 9-6, 10-4, 10-6, 11-6
AUTHOR paragraph 3-2
AUTO clause 5-25
AUTO-SKIP option 6-18–6-20, 6-22, 6-23

B

BEEP option 6-22, 6-23
BELL clause 5-26
Binary item 5-9, 5-11
BLANK LINE clause 5-26
BLANK SCREEN clause 5-26
BLANK WHEN ZERO clause 5-27
BLINK clause 5-27
BLOCK clause Intro-7, 5-28

C

CALL statement 7-2, 7-4
CHAIN statement 7-3, 7-4
Character
- comparisons 6-43
- lowercase Intro-3
- set 1-3

Class test condition 6-43
Clause, definition 2-2
CLOSE statement Intro-8, 6-31
COBOL
- ANSI 74 Standard Intro-6–8
- Microsoft extensions Intro-9
- tutorials Intro-5

CODE-SET clause 5-29, 5-51
Coding rules 1-2
COLUMN clause 5-30
Comments 1-3
Compiler directing statements 1-13, 2-6
Compound condition 6-41

COMPUTATIONAL 5-53
COMPUTATIONAL-0 .... Intro-6-7, 5-53
COMPUTATIONAL-3 .... Intro-9, 5-53
COMPUTE statement 6-32
Condition-names 1-7, 6-41
Conditional
- items 5-13
- statements 1-13, 6-5

Conditions
- advanced A-1
- description 6-41

CONFIGURATION SECTION 4-5
Continued line 1-2, 1-10
Control index 10-1
COPY statement 1-13, 2-6
CORRESPONDING option Intro-7
COUNT IN phrase 6-65
CURRENCY SIGN clause 4-16

D

Damaged flags 10-2, 5-1, 5-15
DATA DIVISION 2-1
DATA RECORD(S) clause 5-31
Data
- file 10-2
- input field 6-15, 6-16
- set control block 10-1

Data-description entry 5-7
Data-item 2-4, 5-7
Data-name 1-6, 5-31, 5-38
DATE option 6-11
DATE-COMPILED paragraph 3-3
DATE-WRITTEN paragraph 3-3
DAY option 6-11
Debug
- dynamic Intro-9
- trace-style Intro-8

Debugging statements 6-9
Decimal point 5-42
DECIMAL-POINT IS COMMA clause 1-9, 4-17
DECLARATIVES 6-2, 12-1

DELETE statement 6-32, 9-5, 10-5, 11-5
Deleted granule 10-1
DELIMITED BY phrase 6-61
DISPLAY statement Intro-9, 1-7, 6-32
DIVIDE statement 6-36
DIVISION, definition 2-3
Division remainders Intro-7
Dynamic debugging Intro-9

E

Edited items
  alphanumeric 5-40
  numeric 5-9
Editing characters 6-18
Elementary item 2-4, 5-8
Ellipses Intro-4
Entry, definition 2-3
ENVIRONMENT DIVISION 2-1, 4-1, 4-6
ESCAPE KEY 6-12, 6-17, 6-27
ESCAPE KEY phrase 6-17
Evaluation order 1-15
EXHIBIT statement Intro-9, 6-9
EXIT PROGRAM statement 6-38, 7-3
EXIT statement 6-38
EXTEND phrase 6-54
Extensions to ANSI 74 Standard, COBOL Intro-6
External decimal item 5-9

F

FD entry 1-2, 2-4
Figurative constant
  as literal Intro-6
  described 1-11
FILE SECTION 5-16
FILE STATUS
  clause 4-13, 10-4, 11-3
  item 6-8
File-description entry
  See FD entry
File-ID 5-57
FILE-CONTROL paragraph 4-7
File-names 1-7

Files
  See INDEXED files
    SEQUENTIAL files
    RELATIVE files
FILLER as data-name 1-6
Fixed segments 13-1
Flags, damaged 10-2
Floating string 5-43
Footing 5-35
Formats
  see also specific reserved word
  DATA DIVISION 5-1
  ENVIRONMENT DIVISION 4-2
  IDENTIFICATION DIVISION 3-1
  PROCEDURE DIVISION 6-2
  syntax notation Intro-3
FROM clause 5-32
FULL clause 5-33

G

General format, See Formats
GIVING option 6-7
GO TO statement 1-7, 6-39
Granules 10-1
Group item 2-4, 5-8

H

Hierarchy of program 2-1
HIGH-VALUE figurative constant 1-11
HIGHLIGHT clause 5-33

I

I-O
  see also
  INDEXED files
  SEQUENTIAL files
  RELATIVE files
  error handling 6-8
  option of OPEN 6-53

I-O-CONTROL paragraph 4-9
IDENTIFICATION DIVISION 2-1, 3-1, 3-3
IF statement 6-40
Imperative-statement 1-13, 6-5
Independent segments 13-1
Index-data-item 1-8, 5-12, 8-1
Index-name 1-8, 5-12, 8-1
INDEXED
  file organization 10-1
  I-O Intro-8
Input file 6-53
INPUT-OUTPUT SECTION 4-8
INSPECT statement Intro-7, 6-45
INSTALLATION paragraph 3-4
Inter-program communication Intro-8, 7-1
Internal decimal item 5-10
INTO option 9-6
INVALID KEY clause 6-8, 10-4–10-9, 11-3, 11-5–11-9, 12-2

J

JUST clause 5-34
JUSTIFIED clause 5-34

K

KEY IS clause 10-7
KEY clause 8-6
Key
  file, INDEXED 10-1
  set control block 10-1

L

LABEL RECORD(S) clause 5-34
Leaf 10-1
LEFT-JUSTIFY 6-22, 6-23
LENGTH-CHECK 6-22, 6-23
Level
  77 items 5-13
  88 items 5-13
  numbers 1-2, 2-4

Library Intro-8
Limitations of DATA DIVISION 5-15
LINAGE clause 5-35
LINE SEQUENTIAL files 4-12, 4-14
LINE clause 5-37
LINE NUMBER option 1-2, 6-12
LINKAGE SECTION 5-19, 7-3
Literal
    constant 1-9
    figurative constants 1-11
    non-numeric 1-10
    numeric 1-9
    quoted 1-10
    size 5-55
LOCK suffix 6-31
LOW-VALUE figurative constant 1-11
Lowercase characters Intro-3

M

Main program 7-4
MEMORY SIZE clause 4-10
MERGE statement 6-48
Mnemonic-names 1-7
MOVE statement 6-49
MS-COBOL, See COBOL
MS-SORT
    ASSIGN clause 4-4
    FILE-CONTROL paragraph 4-7
    I-O CONTROL paragraph 4-9
    SELECT clause 4-12
MULTIPLE FILE TAPE CONTAINS clause Intro-7
MULTIPLY statement 6-52

N

Names 1-6
Nested IF statements C-1
Node 10-1
Non-numeric literals 1-10
Nucleus Intro-6

Numeric
comparisons 6-43
edited item 5-9
item 5-9, 5-41
literals 1-9
receiving field 6-18

O

OBJECT-COMPUTER paragraph 4-5, 4-10
OCCURS DEPENDING ON clause Intro-6
OCCURS clause 5-38
ON OVERFLOW clause 6-65
OPEN statement Intro-8, 6-53, 10-5
Operators 1-15
Option, definition 2-2
Order of evaluation 1-15
ORGANIZATION clause 4-14
Output file 6-53
Overlays 13-1

P

Packed decimal format 5-10
Paragraph, definition 2-3
PERFORM statement 1-7, 6-55, 13-2
Phrase, definition 2-2
PICTURE clause 5-40
POINTER phrase 6-61
PRINTER IS clause 4-17
PROCEDURE DIVISION 2-1, 6-1, 6-10, 7-4
Procedure-names 1-7
PROGRAM COLLATING SEQUENCE clause 4-10
PROGRAM-ID paragraph 3-4
Program
hierarchy 2-1
main 7-4
subprogram 7-4
PROMPT option 6-18, 6-22, 6-23
Punctuation, ... Intro-3, 1-4

Q

Qualification of names Intro-6, 1-8
QUOTE
  figurative 1-11
  constant 1-11
Quoted literals 1-10

R

Range in PERFORM 6-56
READ statement 6-57, 9-5, 10-6, 11-6
READY TRACE statement Intro-9, 6-9, 6-58
RECORD CONTAINS clause Intro-7
RECORD KEY clause 10-3
RECORD clause 5-47
Record-description entry 5-1
REDEFINES clause 2-5, 5-47
Region, definition 2-3
RELATIVE KEY clause 11-2, 11-6
RELATIVE files 11-1
Relative
  indexing 8-2
  I-O Intro-7
RELEASE statement 6-58
Remainders Intro-7
RENAMES phrase Intro-7
REPLACING clause 6-46
Report
  item 5-9, 5-42
  writer Intro-8
REQUIRED clause 5-49
RESERVE INTEGER AREA(S) clause Intro-7
RESERVE clause 4-14
Reserved words Intro-3, 1-5, E-1
RESET TRACE statement Intro-9, 6-9, 6-58, 6-59
RETURN statement 6-59
REWRITE statement 6-59, 9-7, 10-7, 11-7
RIGHT-JUSTIFY phrase 6-18, 6-22, 6-23
ROUNDED option 6-7

S

SAME AREA clause 4-5, 4-11
SCREEN SECTION 1-8, 5-21, 6-27
Screen-name 6-27
SEARCH ALL statement 8-6
SEARCH statement 6-59, 8-3
Section, definition 2-3
SECURE clause 5-49
SECURITY paragraph 3-4
Segmentation Intro-8, 13-1
SELECT OPTIONAL clause Intro-8
SELECT clause 4-8, 4-12, 10-3, 11-2
Sentence, definition 2-2
Separators Intro-3, 1-4
SEQUENTIAL files
    FILE-CONTROL 4-7
    organization 9-1
    PROCEDURE DIVISION statements 9-2
    SELECT 4-12
    status report 9-4
    syntax 9-3
SEQUENTIAL I-O Intro-7
SET statement 6-59, 8-2
SIGN clause 5-50
Sign test 6-44
Simple condition 6-41
SIZE ERROR option 6-6
SORT facility, See MS-SORT
SORT statement 6-59
SORT/MERGE verbs Intro-8
Source coding rules 1-2
SOURCE-COMPUTER paragraph 4-5, 4-15
SPACE, figurative constant 1-11
SPACE-FILL option 6-18, 6-22, 6-23
Special characters Intro-3
SPECIAL-NAMES paragraph Intro-6, 4-5, 4-16
START statement 6-60, 10-8, 11-8
Statements
    see also specific statement
    arithmetic 1-14
    compiler directing 1-13

conditional 1-13
definition 1-12, 2-2
imperative 1-12
STOP statement 6-60
STRING statement 6-61
Structural hierarchy 2-1
Subprogram 7-4
Subscript 5-39
SUBTRACT statement 6-63
Switch C-1
SWITCH-n clause 4-17
SYNCHRONIZED clause 5-52
Syntax notation in formats Intro-3

T

Tab stops 1-2
Table handling Intro-7
TALLYING clause 6-46
Tape handling Intro-7
THROUGH option 5-55
THRU option 5-55
TIME option 6-11
TO clause 5-32, 5-52, 6-28
TRACE mode 6-9
Trace-style debug Intro-8
TRAILING-SIGN 6-22
Tutorials, COBOL Intro-5

U

UNSTRING statement 6-64
UPDATE option 6-18, 6-22, 6-23
USAGE clause 5-8, 5-53
USE statement 1-13, 2-7, 6-66, 12-1
USING clause 5-32, 5-54, 6-28

V

VALUE IS clause 5-55
VALUE OF FILE-ID clause 10-1
VALUE OF clause 5-57
VARYING phrase 8-3

W

WHEN clause 8-6
WITH DEBUGGING MODE clause 1-2, 4-5, 4-15
WITH phrase 6-22
WORKING-STORAGE SECTION 5-18
WRITE statement 6-66, 9-8, 10-9, 11-9

Z

ZERO figurative constant 1-11
ZERO-FILL option 6-18, 6-22, 6-23