DataGeneral

Software Documentation

# Interactive COBOL Programmer's Reference

# Interactive COBOL Programmer's Reference

093-705013-02

*For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.*

# Notice



Interactive COBOL Programmer's Reference
Ordering Number 093-705013
Revision 02, August 1984

(Interactive COBOL, Rev. 1.30)

Original release: June 1982
First revision: April 1983

# Changes to Interactive COBOL (revision 1.30)

## Language Enhancements

Level 88 is implemented

The COLUMN and LINE NUMBER clauses accept identifiers as well as literals. This allows field positions to be specified at execution time.

The ACCEPT and DISPLAY *screen-name* statements allow dynamic definition of an entire screen position.

Identifiers can be used with the FROM, TO, and USING clauses.

The display size has been enlarged to 255 lines and 255 columns.

Abbreviated combined relation conditions have been implemented.

The CALL, CANCEL, and EXIT PROGRAM statements have been added. The CALL statement can be used to call an assembly language subroutine, and, under AOS and AOS/VS, to call the CLI or any .PR file.

The ADVANCING clause in the WRITE statement accepts an identifier.

The AFTER clause in the PERFORM statement has been implemented.

## Compiler Enhancements

The number of data, procedure, and file references has been increased from 764 to 2294.

Global /I switch allows ICOS cross-development.

Global /A switch implements ANSI 74 standard arithmetic for COMPUTATIONAL items.

Global /O switch suppresses copy files in the listing file.

Global /R switch does not round .PD file to a 2 KB boundary (AOS and AOS/VS); does round .PD file to a 2 KB boundary (RDOS, DG./RDOS).

The code revision of the .PD and .DD files is 6.

## Runtime Enhancements

#N system call to rename a file

Display size is determined at execution time with the CHARACTERISTICS command (AOS and AOS/VS), and the DEFINES utility (RDOS and DG/RDOS).

Internal computation registers now support 19 digits of accuracy.

In Interactive COBOL Revision 1.20, the runtime system was enhanced to handle support of 8-bit characters.

## Runtime Enhancements (RDOS and DG/RDOS)

#O system call runs a detached job.

#A system call terminates an ACCEPT statement.

Local /M switch assigns the master console to a QTY lines.

Global /C and /D switches allow CLI mode of execution.

Global /S switch enables spooling when runtime system is executing.

Global /P switch disables PASS.

## Runtime Enhancements (AOS and AOS/VS)

CTRL-C CTRL-A will terminate an ACCEPT statement.

## Runtime Enhancements (AOS/VS only)

The MINISAM lock server has been implemented in 32-bit code (in ring 6) of the runtime system process.

# Table of Contents

## Chapter 2 File Organization and Access



## Chapter 3 Interactive Screen Management

## Chapter 8 The Procedure Division

# Preface

## Document Set

Interactive COBOL is documented by a set of manuals that describe the language, its utilities, and the system-dependent features that affect its use. The *Interactive COBOL Programmer's Reference* defines the Interactive COBOL programming language. It is your primary reference regardless of the operating system.

The system-dependent *User's Guides* explain the features of your particular operating system as they relate to Interactive COBOL. Each manual describes such factors as the file system and gives specific instructions for invoking the runtime system, compiler, and debugger.

The set of Interactive COBOL utilities is essentially the same for each system and provides similar functions on each system. However, variations do exist for invoking and using the general utilities on each of the operating systems. Separate *Utilities* manuals provide instructions for using the utilities.

In addition to the general utilities, Interactive COBOL also includes two special COBOL source editors. *ICEDIT: Interactive COBOL Editor* describes an editor specifically designed for writing programs. *SCREEN: Screen Format Editor* describes the special-purpose editor for designing and automatically coding screen display formats.

The titles and order numbers of the Interactive COBOL documents are listed in "Related Documents" at the end of this manual.

## Scope

Chapters 1-4 of this manual are introductory; they explain COBOL concepts and present an overview of Interactive COBOL, file organization and access, screen management, and source entry. Chapters 5-8 are for reference; they discuss the Interactive COBOL syntax and list syntax formats.

## Organization

This manual is divided into eight chapters and five appendixes.

Chapter 1 presents an overview of language and data concepts and Procedure Division concepts.

Chapter 2 describes COBOL file organization and access.

Chapter 3 explains interactive screen management.

Chapter 4 discusses COBOL source entry formats: CRT and card.

Chapter 5 discusses the Identification Division.

Chapter 6 discusses the Environment Division.

Chapter 7 discusses the Data Division, including the Screen Section, an Interactive COBOL extension.

Chapter 8 lists Interactive COBOL verbs in alphabetical order, with formats, examples, and rules for use.

Appendix A gives a summary of the compiler command line.

Appendix B is a glossary of Interactive COBOL and standard terms.

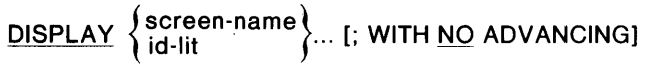
Appendix C lists the 7-bit and 8-bit ASCII character sets.

Appendix D summarizes Interactive COBOL syntax.

Appendix E is a listing of ANSI Standard and Interactive COBOL reserved words.

## Notational Conventions

The conventions described below are used in this manual to represent the various elements of COBOL language syntax. The following example contains most of the syntactical elements used:




**UPPERCASE** Indicates a COBOL reserved word. Underlined uppercase words are required. Nonunderlined uppercase words are optional and are used to improve readability. In either case, all uppercase words must be spelled as shown: no abbreviations are permitted.

**lowercase** Indicates a generic term representing words, literals, PICTURE character strings, comment entries, or a complete syntactical entry to be supplied by the programmer. For instance, where *screen-name* appears, the screen name that you have chosen should be used.

Throughout this manual, the abbreviations id, id-lit, and lit are used in the syntax in place of the common COBOL constructs identifier, identifier-literal, and literal.

**Hyphen** A hyphen appearing between uppercase words is required, as in PROGRAM-ID or SOURCE-COMPUTER. A hyphen between lowercase words indicates that your entry must not contain any spaces. In the example, the screen-name could be written as ACCTS-PAYABLE or ACCTSPAYABLE, but not ACCTS PAYABLE.

{ } Braces enclosing part of a format mean that you must select one of the options enclosed within the braces. Thus, the example indicates that either a screen-name or an id-lit must appear in the DISPLAY statement.

[ ] Brackets enclose optional portions of a format. In the example, the phrase WITH NO ADVANCING is optional.

... An ellipsis indicates that the item preceding it (defined by logically matching brackets or braces) may be repeated one or more times. In this example, you must enter a screen-name, an identifier, or a literal at least once; the ellipses indicate that you may repeat the entry.

**Format Punctuation**

The period is required when it is present in a format. The comma and semicolon are optional and interchangeable. They may be used only in certain positions; these positions are indicated by a semicolon. In the example above, a comma or a semicolon may precede the WITH NO ADVANCING phrase.

At least one space must follow a comma or semicolon used to separate statements.

**Special Characters**

When an arithmetic or logical operator (+, -, >, <, or =) appears in a format, it is required. These special characters are not underlined.

|

Vertical bars in the margin highlight technical changes made since the last revision of this document.

# Chapter 1
# Programming Concepts

This chapter outlines the basic structure of a COBOL program, including language and data concepts and Procedure Division concepts.

## Program Structure

A COBOL program has four divisions: Identification, Environment, Data, and Procedure. The divisions must appear in the program in the order stated.

The Identification Division provides the program name and, optionally, the program author, date of compilation, and other documentary information.

The Environment Division specifies the computer(s) used to compile and run the program, describes special conditions, designates the logical organization of files, and describes the relationships of data files with actual input/output devices.

The Data Division describes all the data that the program uses. This includes the data used in input/output operations, the data developed for internal processing, and screen data.

The Procedure Division contains the instructions for processing the data.

The information in a COBOL program is written in clauses (Identification, Environment, and Data divisions) and statements (Procedure Division).

A *clause* specifies an attribute of an entry. A series of clauses ending with a period is an *entry*. A *statement* specifies an action to be taken by the program. A series of statements ending with a period is a *sentence*.

Each clause or statement may be divided into smaller units called phrases. A *phrase* contains one or more words that are considered a syntactical unit. When the programmer must choose between two or more phrases in a format, the phrase is called an *option*.

Within a division, entries and sentences can be combined into larger logical units called paragraphs or sections. A *paragraph* consists of a paragraph name followed by a period and space, and zero or more entries or sentences. In the Environment and Data divisions, paragraph names are reserved words (e.g., FILE-CONTROL, I-O-CONTROL), and paragraphs contain entries. In the Procedure Division, paragraph names are programmer defined and are composed of sentences.

In the Environment and Data divisions, a *section* consists of a section header and zero or more entries. The section header contains a reserved section name followed by the word SECTION. A section header must be followed by a period and a space. In the Procedure Division, a section consists of a programmer-defined section name, followed by a period and space, and zero or more paragraphs.

# Language Structure

COBOL is an Englishlike language using sentences composed of characters, words, and statements. Interactive COBOL includes the language components described in the following sections.

## Character Set

There are three character sets, listed below:

- The COBOL character set. This is a set of the 51 characters that are legal in COBOL. The characters include the uppercase letters A-Z, the digits 0-9, and special characters used as separators, logical operators, etc.
- The 96-character ASCII set. Characters in this set are used to form data-items and nonnumeric literals.
- The DG international character set. This set, which can be used on terminals equipped to handle 8-bit ASCII values, includes 69 displayable characters in addition to the 96-character ASCII set.

## Separators

In a COBOL source program, words and sentences are delimited by separators. The separators are:

- Space character. When a format indicates a space, more than one may be used.
- Comma, semicolon, and period. These characters must be followed by a space.
- Left and right parentheses. They may be used only in balanced pairs to denote subscripts, indexes, arithmetic expressions, or conditions.
- Quotation marks. Quotation marks are used in balanced pairs to delimit nonnumeric literals. An opening quotation mark must be preceded by a space or a left parenthesis; a closing quotation mark must be followed by one of the following separators: space, comma, semicolon, period, or right parenthesis.

## COBOL Words

Words can be up to 30 characters long. There are two classes of COBOL words: programmer-created words and reserved words.

Programmer-created words are supplied by the programmer to satisfy the requirements of a specific format. A programmer-created word can consist of the characters A-Z, 0-9, and hyphen. It must not contain any spaces, and the hyphen may not be the first or last character. All programmer-created words—except paragraph names, section names, level numbers, and segment numbers—must contain at least one alphabetic character.

All programmer-created words must be unique within a program. A word is unique if no other word has the identical spelling or punctuation, or if uniqueness can be ensured by qualification (see "Uniqueness of Reference" later in this chapter).

Reserved words are assigned a special meaning within the COBOL language; they can be used only as indicated in the syntax formats. Appendix E contains a list of ANSI Standard and Interactive COBOL reserved words.

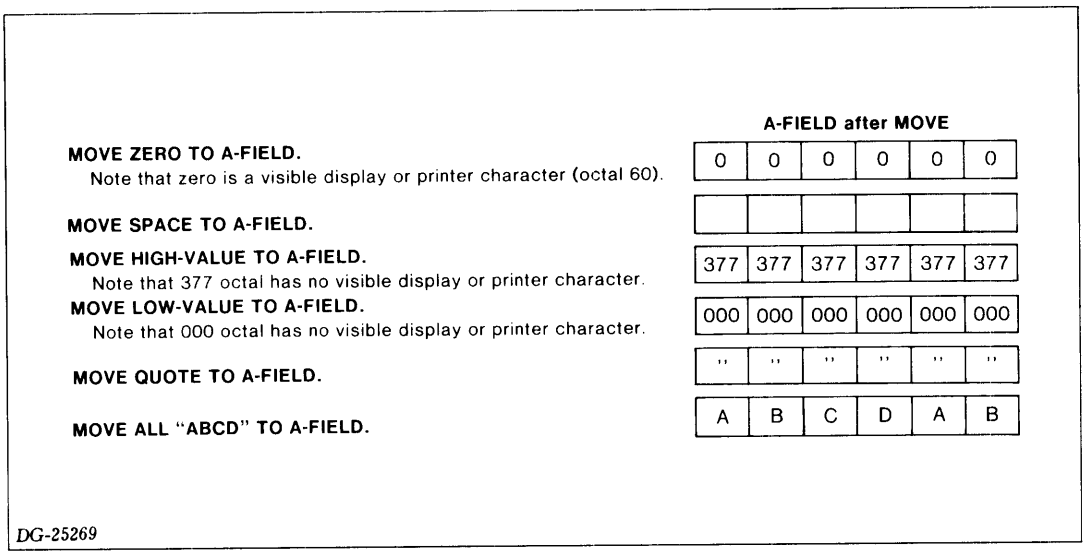



***Figure 1-1*** *Use of Figurative Constants*

## Figurative Constants

COBOL has several reserved words that represent specific constant values. These figurative constants are used in a program to avoid repetitive coding. The singular and plural forms may be used interchangeably. Figurative constants must not be bounded by quotation marks. The figurative constants and their values are:

| | |
|---|---|
| ZERO, ZEROS, ZEROES | The digit 0 |
| SPACE, SPACES | The space character |
| HIGH-VALUE, HIGH-VALUES | Octal value 377 |
| LOW-VALUE, LOW-VALUES | Octal value 000 |
| QUOTE, QUOTES | The quotation mark character |
| ALL literal | The specified literal. The literal must be either a nonnumeric literal or a figurative constant. However, the ALL figurative-constant form is redundant. |

Given a data-item initially defined as A-FIELD PIC X(6) VALUE "123456", the examples in Figure 1-1 illustrate the use of figurative constants in MOVE statements.

## Nonnumeric Literals

A nonnumeric literal is a character string that may contain any elements from the ASCII character sets. The literal must be enclosed in quotation marks and may be up to 132 characters long, including the quotation marks. The value of the literal is the string of characters between the bounding quotation marks. Embedded spaces are counted as part of the literal. Quotation marks within a literal are represented by two consecutive quotation marks; these represent only one occurrence of the character in the literal. For example:

```
MOVE "THE DATE IS" TO HEAD-LINE.
DISPLAY "INPUT ERROR, PLEASE RE-ENTER THE DATA!".
DISPLAY "* * Please type your name!".
MOVE "-The current time is:" TO TIME-LINE.
DISPLAY "The title is " "Learning to Program" ".".
```

## Numeric Literals

A numeric literal is a character string selected from the set 0-9, plus, minus, and decimal point. It may contain up to 18 digits. An unsigned literal is positive. If a sign is used, it must be the leftmost character. A numeric literal may not contain more than one sign character. If a decimal point is used, it may not be the rightmost character. A numeric literal may not contain more than one decimal point.

The following are legal numeric literals:

```
ADD 122 TO CATEGORY.
MULTIPLY 3.1416 BY DIAM GIVING CIRCUM.
MOVE - 167 TO TEMP.
```

The following are illegal numeric literals:

```
ADD 1543. TO CATEGORY. (The decimal is on the right.)
MULTIPLY 3.14+ BY DIAM GIVING CIRCUM. (The sign is not on the left.)
MOVE 1234567890987654321 TO TEMP. (The literal contains more than 18 digits.)
```

# Record Concepts

A COBOL program deals with data grouped into files. Within a file, the data is logically organized by records. A COBOL *record* consists of items that are uniquely identifiable and are treated as a unit. A record is the most inclusive COBOL data-item; it may be divided into logical subdivisions, which may in turn be further subdivided. A record or record division that has no subdivisions is called an *elementary item.* An item that is subdivided is called a *group item,* and the entire record structure with all its subdivisions is referred to as a *hierarchy.* The programmer must assign a level number and a unique name to each elementary and group item within the hierarchy.

## Level Numbers

Level numbers in a data description show the organization of elementary and group items, and identify special-purpose data-items. Because records are the most inclusive data-items, they are assigned 01 level numbers. Subdivisions of the record are given successively higher level numbers. The level numbers 02 through 49 identify elementary and group items within a record.

There are two special level numbers: 77 and 88. Level 77 identifies independent items in the Working-Storage Section. An independent item is not in a hierarchy; it is not part of a record. Level 88 gives condition-names to a value that a data-item may contain.

## The PICTURE Clause

The PICTURE clause occurs in the Data Division of a COBOL program. It specifies the length and data type of an elementary data-item. Every elementary data-item except those described as USAGE IS INDEX must have a PICTURE clause.

The format of the PICTURE clause is:

{ PICTURE / PIC } IS picture-string

The words PIC and PICTURE are synonomous. The number of characters in the picture-string determines the length of the data-item, and the specific characters in the picture-string determine the type of data-item. A data-item can be alphabetic, numeric, alphanumeric, alphanumeric edited, or numeric edited.

The following sections explain the types of data and how to specify them in the PICTURE clause. Table 1-1 summarizes the symbols in the PICTURE clause and their meanings.

| Symbol | Function |
|---|---|
| X | Alphanumeric character |
| A | Alphabetic character |
| 9 | Numeric character |
| V | Assumed decimal point |
| S | Arithmetic sign |
| P | Assumed decimal scaling position |
| B | Space insertion |
| / | Slash insertion |
| 0 | Zero insertion |
| . | Decimal point insertion |
| , | Comma insertion |
| + | Plus sign insertion |
| - | Minus sign (if negative value) or blank insertion |
| CR | CR (credit sign) insertion |
| DB | DB (debit sign) insertion |
| $ | Currency symbol insertion |
| Z | Zero suppression by space |
| * | Zero suppression by asterisk |

**Table 1-1 Summary of PICTURE Clause Symbols**

## Alphabetic Data-Items

Alphabetic items are described using only the symbols A and B. An alphabetic data-item may contain only uppercase letters of the alphabet or spaces.

The following examples show alphabetic data-items:

```
05 STATE-ABBREV PIC AA.
```

The data-item STATE-ABBREV is alphabetic and two characters long. Each character may be an uppercase letter of the alphabet or a space.

```
05 STOCK-REF PIC A(5)BAA.
```

The data-item STOCK-REF is alphabetic and 8 characters long, with a space after the first 5 characters.

## Numeric Data-Items

Numeric items are described by the symbols 9, V, S, and P. A numeric data-item can contain only the characters 0 through 9; if the S symbol is included, the item can contain a plus or a minus sign. Data-items used in arithmetic computations must be described as numeric.

A numeric data-item may be 1 to 18 characters long, 19 if the SIGN IS SEPARATE clause is present.

Each numeric picture string must contain at least one 9. The 9 represents a numeric digit position that can contain a value from 0 to 9.

The V symbol represents an assumed decimal point. It specifies a location but not an actual position of storage; therefore it is not counted in the length of a data-item. Only one V is allowed in a picture-string; if it is the rightmost character, it is redundant.

The S symbol represents a signed data-item. It is not counted in determining the length of the data-item. Only one S is allowed in a picture-string, and it must be the leftmost character. You must include the S symbol if you specify the SIGN IS SEPARATE clause.

The P symbol indicates an assumed decimal scaling position when there is no decimal point in the data-item. The P specifies leading or trailing digits and scales the number by powers of 10. Thus the data-item is assumed to contain a zero in each position held by a P, but no zeros are actually stored.

The P is not counted in determining the size of the data-item. However, it is counted when determining the number of digit positions in numeric and numeric-edited items. Thus if the data-item evaluates to a number with more than 18 digits, you cannot perform arithmetic operations with it.

P can appear only in a sequence in the rightmost or leftmost portion of the format. The implicit decimal point is assumed to be on the right if the string of Ps is to the right; the decimal point is assumed to be on the left if the Ps are to the left. PICTURE 999P(6) represents a number in the millions, storing only the three leftmost digits. PICTURE PPP999 represents a decimal fraction in the millionths, storing only the three rightmost digits.

If a data-item has P in its picture-string, any value moved to it must reflect the scaled form. Similarly, if the data-item has a VALUE clause, the value specified must be in the scaled form. However, if you display a scaled data-item, the unscaled version appears on your screen.

The following examples show numeric data-items:

```
15 ACCT-NUM    PIC 9(10).
```

The data-item ACCT-NUM is unsigned numeric with 10 digits.

```
10 COST    PIC 99999V99.
```

The data-item COST is unsigned numeric with 5 digits to the left of the assumed decimal point and 2 digits to the right. The total number of characters is 7; the V is not counted in the total.

```
10 TOTAL-PAY    PIC S9(5)V99.
```

The data-item TOTAL-PAY is signed numeric with 5 digits to the left of the assumed decimal point and 2 digits to the right. The total number of characters is 7; neither the V nor the S (unless the SIGN IS SEPARATE clause is present) is counted in the total.

```
10 MEGA-VALUE    PIC 9999PPPPPP.
```

This data-item has four digits and 6 scaling positions. If the value stored were 241, the effective, or scaled, value would be 241,000,000.

```
10 CENTI-VAL    PIC PP99999.
```

This data-item has 5 digits and 2 scaling positions. If the value stored were 1000, the effective, or scaled, value would be .0001.

## Alphanumeric Data-Items

The symbols A, X, and 9 describe alphanumeric data-items. The picture-string of an alphanumeric data-item must conform to one of the following rules:

- It must contain at least one A and at least one 9.
- It must contain at least one X.

An alphanumeric data-item is treated as though its picture-string contained all X's. The data-item can contain any character in the COBOL data character set.

The following examples describe alphanumeric data-items:

```
10 SS-NUM    PIC X(11).
```

The data-item SS-NUM is alphanumeric and 11 characters long.

```
10 MIXED-UP-PIC    PIC AA99.
```

This is a four-character alphanumeric data-item. The picture-string is equivalent to X(4).

## Edited Data-Items

A COBOL program may edit numeric and alphanumeric data-items. There are five types of editing: simple insertion, special insertion, zero suppression and replacement, fixed insertion, and floating insertion. You may use simple insertion editing on both numeric and alphanumeric data-items; the other types of editing are valid only on numeric data-items.

**Simple Insertion.** Numeric and alphanumeric data-items can be edited with simple insertion editing. The insertion characters are comma, B (space), 0, and /. They represent the position in the item where the character will be inserted. Insertion characters are counted in the size of an item. Table 1-2 gives examples of simple insertion editing.

| PICTURE | Data Value | Result |
|---|---|---|
| XXXBXXBX(4) | JAN111984 | JAN 11 1984 |
| 99,999 | 36895 | 36,895 |
| A(5)BA(5) | ABCDEFGHIJ | ABCDE FGHIJ |
| XX/XX/XX | 121584 | 12/15/84 |
| 99,000,000 | 17 | 17,000,000 |

**Table 1-2 Simple Insertion Editing**

**Special Insertion.** The decimal point is the special insertion symbol; it controls how data-items are aligned. You cannot use an actual decimal point and the implied decimal point (V) in the same picture-string. When you move data from an item having an assumed decimal point to an item having an actual decimal point, the data is placed in the receiving item with the actual decimal point aligned with the implied decimal point (see Table 1-3). The decimal point cannot be the last character in the character string.

*Note:* If you use the DECIMAL-POINT IS COMMA clause in the SPECIAL-NAMES paragraph, the rules applying to periods and commas in the PICTURE clause are reversed.

| Sending PIC | Receiving PIC | Data Value | Result |
|---|---|---|---|
| 9(5) | 999.99 | 12345 | 345.00 |
| 999V99 | 999.99 | 12345 | 123.45 |
| 99V999 | 999.99 | 12345 | 012.34 |
| 9(4)V9 | 999.99 | 12345 | 234.50 |

**Table 1-3 Special Insertion Editing**

**Fixed Insertion.** There are five symbols used for fixed insertion editing: +, –, CR, DB, and the currency symbol (usually a $). A picture-string can contain only one currency symbol and only one other editing symbol (+, –, CR, or DB).

The + or – must occupy either the left or rightmost position. If you specify the plus in the picture-string, Interactive COBOL inserts a + or – character in the data-item, depending on whether the value being stored is positive or negative. If you specify a minus, Interactive COBOL inserts a minus character if the value is negative and a space if it is positive.

The symbols CR and DB count as two character positions in determining the size of the item; they must always appear in the rightmost position. The sign-control symbols produce the results shown in Table 1-4, depending on the value of the data-item.

| | Value of Data-Items | |
|---|---|---|
| **Editing Symbol** | **Positive or Zero** | **Negative** |
| + | + | - |
| - | space | - |
| CR | 2 spaces | CR |
| DB | 2 spaces | DB |

**Table 1-4 Results of Sign-Control Editing**

The currency symbol must be the leftmost position, except that either a plus or a minus may precede it. The currency symbol is usually the dollar sign ($). However, if you have specified the CURRENCY SIGN IS clause, use the symbol specified in that clause instead of the dollar sign.

Table 1-5 shows some examples of fixed insertion editing.

**Floating Insertion.** In floating insertion editing, leading zeros are suppressed and the the editing symbol is placed immediately before the first significant digit. Three of the fixed insertion symbols also serve as floating insertion symbols: the plus (+), the minus (–), and the currency symbol. The floating symbols are mutually exclusive, although a fixed plus or minus may appear to the left of a floating currency symbol.

Floating insertion occurs when one of the floating insertion symbols is specified two or more times within the picture-string. However, the symbol appears only once in the output. The string of floating insertion characters may contain the B, zero, comma, slash, or decimal point; they are treated as part of the floating string.

There are two ways to represent floating insertion editing in a picture-string:

1. Represent any or all of the leading numeric character positions to the left of the decimal point by the insertion character (e.g., $$$9.99 or $$$.99).
2. Represent all numeric character positions in the picture-string by the insertion character (e.g., $$$.$$).

| PICTURE | Data Value | Result |
|---|---|---|
| $ 9999.99 | +1234.50 | $ 1234.50 |
| | - 1234.50 | $ 1234.50 |
| | 0 | $ 0000.00 |
| 9999.99CR | +1234.50 | 1234.50 |
| | - 1234.50 | 1234.50CR |
| | 0 | 0000.00 |
| 9999.99DB | +1234.50 | 1234.50 |
| | - 1234.50 | 1234.50DB |
| | 0 | 0000.00 |
| +9999.99 | +1234.50 | +1234.50 |
| | - 1234.50 | - 1234.50 |
| | 0 | +0000.00 |
| 9999.99+ | +1234.50 | 1234.50+ |
| | - 1234.50 | 1234.50- |
| | 0 | 0000.00+ |
| - 9999.99 | +1234.50 | □1234.50 |
| | - 1234.50 | - 1234.50 |
| | 0 | □0000.00 |
| 9999.99- | +1234.50 | 1234.50 |
| | - 1234.50 | 1234.50- |
| | 0 | 0000.00 |

**Table 1-5 Fixed Insertion Editing**

For example, if the PICTURE $,$$$,999.99 is specified for a data-item and that item has the value 12345, the edited result is □□□□□$123.45. Note that both the first comma, which is imbedded in the floating $ string, and the second comma, which appears to the right of the floating $ string, are suppressed in the edited result.

The leftmost character of the floating string represents the leftmost postion of the floating symbol in the data-item. The rightmost character of the floating string represents the rightmost position.

The second leftmost character represents the leftmost limit at which numeric data can appear in the data-item. Nonzero numeric data may replace all characters at or to the right of this limit. For example, with the PICTURE $$$$$.99, a maximum of four digits may appear to the left of the decimal point (e.g., $1234.00).

When floating insertion characters are only to the left of the decimal point in the picture-string, a single floating insertion character is placed either in the position immediately to the left of the first nonfloating position, the first nonzero digit in the data, or the decimal point. The positions to the left of the inserted character are filled with spaces (see Table 1-6 for examples).

If all of the character positions are represented by a floating insertion symbol and the data value is zero, the entire data-item contains spaces. If the value is not zero, the result is the same as when the insertion character is only to the left of the decimal point in the picture-string.

**Zero Suppression.** The Z and asterisk (*) control the suppression of leading zeros in a data-item. With Z the replacement character is the space; with an asterisk the replacement character is an asterisk. The Z and * cannot appear together in a picture-string.

Any of the simple insertion characters (B, 0, /, and comma) may appear within or immediately to the right of the suppression string; they are counted as part of the suppression string.

Suppression symbols may be used to represent either all of the digit positions in the data-item (e.g., ZZZ.ZZ), or any of the leading digit positions to the left of the decimal point (e.g., ZZZ9.99).

| PICTURE | Data Value | Result |
|---|---|---|
| $$9.99 | .23 | □$0.23 |
| $$$.99 | 1.23 | □$1.23 |
| $$$.99 | .23 | □□$.23 |
| $$$9.99 | .12 | □□$0.12 |
| $$$$,$$$.99 | 000123 | □□□□$123.00 |
| -------.00 | 123456 | □123456.00 |
| $$,$$$,$$$.99CR | -1234567 | $1,234,567.00CR |
| ++,+++,+++ | 0 | □□□□□□□□□□ |
| ++,+++ | 123 | □□+123 |

**Table 1-6 Floating Insertion Editing**

If the suppression symbols appear only to the left of the decimal point, any leading zero in the data that corresponds to a suppression symbol in the string is replaced by a space or asterisk. Suppression terminates at the first nonzero digit or at the decimal point, whichever is encountered first.

If all numeric positions in the picture-string are represented by suppression symbols and the value of the data is not zero, the result is the same as if the suppression characters were only to the left of the decimal point. If the value is zero and the suppression symbol is Z, the entire data-item is spaces. If the value is zero and the suppression symbol is asterisk, the data-item is all asterisks, except for the actual decimal point.

A PICTURE clause using the asterisk as the zero suppression symbol may not appear in the same entry with the BLANK WHEN ZERO clause. Table 1-7 gives examples of zero suppression and replacement editing.

| PICTURE | Data Value | Result |
|---|---|---|
| Z,ZZZ.99 | 1234.56 | 1,234.56 |
| ZZZZ.ZZ | 1234 | 1234.00 |
| $Z,ZZZ.99 | 123.45 | $□□123.45 |
| ZZZZZ.ZZ | .01 | □□□□□.01 |
| *****.** | 0 | *****.** |
| *****.99 | 0 | *****.00 |
| ZZZZZ.ZZ | 0 | □□□□□□□□ |

**Table 1-7 Zero Suppression and Replacement Editing**

## Alignment of Data-Items

The standard rules for positioning data within an elementary item depend on the category of the receiving item. The alignment rules for numeric receiving items are:

- If no decimal point is specified, it is assumed to be immediately to the right of the field.
- The data is aligned on the decimal point and, if necessary, truncated or padded with zeros at either end.

The alignment rule for numeric edited receiving items is:

- The data is aligned on the decimal point and, if necessary, truncated or padded with zeros at either end, except where editing causes replacement of leading zeros.

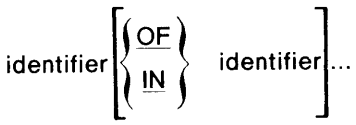
The alignment rules for alphabetic, alphanumeric, and alphanumeric edited receiving items are:

- The data is aligned at the leftmost character position and, if necessary, truncated or padded with spaces at the right.
- If the JUSTIFIED clause is specified for a receiving item, the above rule is modified as described in the JUSTIFIED clause.

# Uniqueness of Reference

Programmer-created names must be unique, either through different spellings or through qualification. Duplicate names can appear in separate hierarchies, but the names must be made unique by references to higher-level names, or *qualifiers,* within each hierarchy. Data Division names and paragraph-names may be qualified. File-names, record-names, section-names, index-names, condition-names, and 77-level Working-Storage data-names cannot be qualified.

Specify qualifying names in ascending order within the hierarchy until the name is uniquely qualified. Enough qualifiers must be mentioned to make the name unique; however, mentioning every level of the hierarchy may not be necessary. The names must be separated by OF or IN. OF and IN are interchangeable; use the one that is most readable. The format for a data-name qualifier is:




## Qualifying Data Division Names

In Data Division references, all data-names used for qualification must be associated with a file-name or level number. In the hierarchy of qualification, a file-name is the most significant name. The names associated with level number 01 are the next most significant, followed by the names associated with level numbers 02, 03, and so on. For a data-item in the File Section, the highest qualification is a file-name. For a data-item in the Working-Storage Section, the highest qualification is an 01 item. The following example illustrates qualified references to duplicate data-names:

```
DATA DIVISION.
...
FD  I-TAX-FILE...                FD  O-TAX-FILE...
01  SINGLE-REC...                01  TAX-REC...
    03 EMPLOYEE-NAME...              03 EMPLOYEE-NAME...
       05 FIRST-NAME...                 05 FIRST-NAME...
       05 LAST-NAME...                  05 LAST-NAME...
    03 ADDR...                       03 ADDR...
    03 GROS...                       03 GROS...
    03 INET...                       03 INET...
    03 TAXS...                       03 TAXS...

PROCEDURE DIVISION.
...
MOVE LAST-NAME IN EMPLOYEE-NAME IN SINGLE-REC TO
    LAST-NAME IN EMPLOYEE-NAME IN TAX-REC.
MULTIPLY INET OF I-TAX-FILE BY .024 GIVING TAXS IN TAX-REC.
```

## Qualifying Procedure-Names

Two or more paragraphs in the Procedure Division may have identical names if they appear in different sections. A paragraph-name can be qualified only by a section name. The keyword SECTION cannot appear in the entry. An example of paragraph-name qualification is:

```
PERFORM PARAGRAPH-1 OF QUALIFYING-SEC.
```

*Note:* A paragraph-name need not be qualified when referred to within the section in which it appears.

# Tables, Subscripts, Indexes

Subscripts and indexes are used to refer to elements in a table. A subscripted data-item has the form:

```
data-name (subscript-1 [, subscript-2 [, subscript-3]])
```

The subscript is an integer or an identifier that evaluates to an integer.

An indexed data-item has the form:

```
data-name (index-1 [, index-2 [, index-3]])
```

The index may be in any of the following forms: identifier, identifier + integer, or identifier – integer.

Interactive COBOL treats an index in much the same manner as a subscript. For both subscripts and indexes:

- The data-name must be the name of a table element, and it may be qualified.
- The parentheses immediately follow the data-name or its last qualifier. One or more spaces may precede the opening parenthesis.
- The lowest possible subscript or index value is 1; the highest possible value is the maximum number of elements in the table, which is specified in the OCCURS clause.
- Three levels of subscripts and indexes are permitted.

Identifier subscripts are defined in the File, Working-Storage, and Linkage sections. Change a subscript value the same way as any other data-item—move values to it, add values to it, or subtract values from it.

Declare indexes in the Working-Storage Section with the INDEXED BY phrase. The SET and PERFORM verbs change index values, which allows the data-name to point to different items in the table.

## Loading a Table

Your COBOL program must insert values into, or load, the table. There are two ways to load a table: hard-coding the values into the program, and entering the table values at execution time.

Use the VALUE clause to hard-code data-values into the table. The following example shows a subscript and a table that includes salary grades and the corresponding hourly pay.

```
WORKING-STORAGE SECTION.
...
77  PAY-SUB     PIC 99.

01  PAY-VALUES.
    05  FILLER          PIC X(6)    VALUE "020500".
    05  FILLER          PIC X(6)    VALUE "040550".
    05  FILLER          PIC X(6)    VALUE "060600".
    05  FILLER          PIC X(6)    VALUE "080650".
    05  FILLER          PIC X(6)    VALUE "100700".

01  PAY-TABLE REDEFINES PAY-VALUES.
    05  PAY-TABLE-ENTRY OCCURS 5 TIMES.
        10  ITEM-PAY-CLASS  PIC 99.
        10  ITEM-PAY-RATE   PIC 99V99.
```

Under COBOL rules, a data-item cannot have both a VALUE clause and an OCCURS clause. Therefore, enter the numeric values with the VALUE clause and redefine them with the OCCURS clause.

Use the same format for indexes, except

- Do not define a subscript (PAY-SUB, in this example)
- Include the INDEXED BY phrase of the OCCURS clause. In this example, the coding could be

```
05 PAY-TABLE-ENTRY OCCURS 5 TIMES INDEXED BY ITEM-INDEX.
```

If a table will contain volatile data, you can enter it during program execution. There are two ways to do this. The Interactive COBOL program can read the table items from a data file, or you can enter the table items interactively.

To set up a program that reads table items from a data file, define the data file in your program with the SELECT and FD statements, and process it with OPEN, READ, and CLOSE statements. Since you will not insert the table item values with the VALUE clause, you do not need to redefine the table.

Using the same pay scale and pay rate table as above, define and load the table as follows:

```
FILE-CONTROL.
    SELECT PAY-FILE ASSIGN TO DISK, "PAY$FILE".
FILE SECTION

FD  PAY-FILE
    RECORD CONTAINS 6 CHARACTERS
    LABEL RECORDS ARE STANDARD.
01  PAY-RECORD.
    05  PAY-CLASS   PIC 99.
    05  PAY-RATE    PIC 99V99.
...
WORKING-STORAGE SECTION.
...
```

```
77  SUB         PIC 99.
...
01  PAY-TABLE.
    05  PAY-TABLE-ENTRY OCCURS 5 TIMES.
        10  ITEM-PAY-CLASS  PIC 99.
        10  ITEM-PAY-RATE   PIC 99V99.

PROCEDURE DIVISION.
...
    PERFORM LOAD-TABLE VARYING SUB FROM 1 BY 1
        UNTIL SUB > 5.

LOAD-TABLE.
    READ PAY-FILE.
    MOVE PAY-CLASS TO ITEM-PAY-CLASS (SUB).
    MOVE PAY-RATE TO ITEM-PAY-RATE (SUB).
```

In this example, the disk file PAY$FILE contains the data for the table.

You can also load the table interactively, using Interactive COBOL's screen management facility. Screen management is discussed in detail in chapter 3; however, the following example shows how to load the pay grade and rate table by entering data at the terminal:

```
WORKING-STORAGE SECTION.
01  PAY-TABLE.
    05  PAY-TABLE-ENTRY OCCURS 5 TIMES
                        INDEXED BY ITEM-INDEX.
        10  ITEM-PAY-CLASS  PIC 99.
        10  ITEM-PAY-RATE   PIC 99V99.

SCREEN SECTION.
01  PAY-SCREEN.
    05  LINE 2 COL 20 VALUE "EMPLOYEE PAY GRADES AND RATES".
    05  LINE 5 COL 10 VALUE "Pay Grades".
    05  LINE 5 COL +10 VALUE "Pay Rates".
    05  LINE PLUS 2 COL 13 PIC 99 TO ITEM-PAY-CLASS (ITEM-INDEX).
    05  LINE 7 COL +10 PIC 99V99 TO ITEM-PAY-RATE (ITEM-INDEX).
    05  LINE PLUS 1 COL 13 PIC 99 TO ITEM-PAY-CLASS (ITEM-INDEX + 1).
    05  LINE 8 COL +10 PIC 99V99 TO ITEM-PAY-RATE (ITEM-INDEX + 1).
    05  LINE PLUS 1 COL 13 PIC 99 TO ITEM-PAY-CLASS (ITEM-INDEX + 2).
    05  LINE 9 COL +10 PIC 99V99 TO ITEM-PAY-RATE (ITEM-INDEX + 2).
    05  LINE PLUS 1 COL 13 PIC 99 TO ITEM-PAY-CLASS (ITEM-INDEX + 3).
    05  LINE 10 COL +10 PIC 99V99 TO ITEM-PAY-RATE (ITEM-INDEX + 3).
    05  LINE PLUS 1 COL 13 PIC 99 TO ITEM-PAY-CLASS (ITEM-INDEX + 4).
    05  LINE 11 COL +10 PIC 99V99 TO ITEM-PAY-RATE (ITEM-INDEX + 4).

PROCEDURE DIVISION.
    ...
    SET ITEM-INDEX TO 1.
    DISPLAY PAY-SCREEN.
    ACCEPT PAY-SCREEN.
```

## Referencing a Table

Tables are referenced in the Procedure Division. To refer to a specific item in a table, specify its data-name and subscript (or index). For two- or three-dimensional tables, specify a subscript for each dimension. The compiler associates the rightmost subscript in a list with the innermost array. Therefore, list subscript levels in order of successively smaller inclusion and separate them with spaces.

You reference a table in different ways, depending on whether it is subscripted with an integer or data-item, or whether it is indexed. In the following examples, the third value in PAY-TABLE is moved to the data-name TEMP.

```
MOVE ITEM-PAY-CLASS (3) TO TEMP.

MOVE 3 TO SUB.
MOVE ITEM-PAY-CLASS (SUB) TO TEMP.

SET ITEM-INDEX TO 3.
MOVE ITEM-PAY-CLASS (ITEM-INDEX) TO TEMP.
```

You can also change subscript and index values by putting the table-manipulation code in a PERFORM VARYING loop, varying the subscript or index as required.

# Procedure Division Structure

The Procedure Division is the last division in a COBOL program. It contains the actual logic of the program. The program logic is written in statements that begin with a COBOL verb. The COBOL verb expresses an action to be taken by the program. The words and symbols that make up the rest of the statement specify what the COBOL verb is to act upon.

Chapter 8 lists the format of the Procedure Division. The Procedure Division begins with the header, which, like other division headers, is a required entry. Under the header, in decreasing order, are the Procedure Division sections, paragraphs, sentences and statements, and verbs.

## Interactive COBOL Verbs

The categories of Interactive COBOL verbs are listed in Table 1-8. Each of these verbs is discussed in detail in chapter 8.

## Sentences and Statements

A sentence is simply a statement followed by a period and one or more spaces. Although this difference may seem trivial, it has a profound effect on your program logic.

For example, the following code is one sentence. Both COMPUTE statements are executed if BYPASS-REC-SW has a value of N. However, if you add a period to the end of the second line, thus making two sentences, the third line—which is now a separate sentence—is performed unconditionally.

```
IF BYPASS-RES-SW = "N"
   COMPUTE NET = GROSS - TOT-TAXES
   COMPUTE YTD-NET = NET + OLD-NET.
```

| | |
|---|---|
| **Arithmetic Operations** | |
| ADD | Sums two or more operands and stores the result. |
| ADD CORRESPONDING | Adds items from one group to items in another group that have the same name. |
| COMPUTE | Evaluates an arithmetic expression. |
| DIVIDE | Divides one operand into another. |
| MULTIPLY | Multiplies one operand by another. |
| SUBTRACT | Subtracts the (sum of) operand(s) from another operand. |
| SUBTRACT CORRESPONDING | Subtracts items in one group from items in another group that have the same name. |
| **Compiler Directing** | |
| COPY | Instructs the compiler to copy source code from another file and merge into the current source during compilation. |
| USE | Defines procedures for I/O error handling. |
| **Data Manipulation** | |
| INSPECT | Counts and/or replaces specific single characters in data-items. |
| MOVE | Transfers data, in accordance with the rules of editing, to one or more data-items. |
| MOVE CORRESPONDING | Moves items that have the same name from one group to another. |
| SET | Loads a specific value into index items associated with table elements. |
| **Input/Output** | |
| ACCEPT | Transfers data from the screen (or from a system-name) to a data-item. |
| CLOSE | Terminates the processing of files. |
| DELETE | Logically deletes a record from an indexed or relative file, or deletes a file from the disk. |
| DISPLAY | Displays a literal, data-item, or a group of these on the screen. |
| OPEN | Initializes files for processing. |
| READ | Makes a record available from a file. |
| REWRITE | Replaces an existing record in a file. |
| START | Provides logical positioning within a relative or indexed file for subsequent retrieval of records. |
| UNDELETE | Restores a logically deleted record in an indexed or relative file. |
| UNLOCK | Releases records locked by previous READ statements. |
| WRITE | Releases a logical record to an output file. |
| **Transfer of Control** | |
| CALL | Transfers control to specified subprogram. |
| CALL PROGRAM | Chains to another program or invokes a COBOL runtime utility. |
| CANCEL | Restores a called subprogram to its initial state. |
| EXIT | Documents the end of a PERFORM subroutine. |
| EXIT PROGRAM | Returns control to the calling program. |
| GO TO | Transfers control within the Procedure Division. |
| IF | Evaluates a condition to determine the path of execution. |
| PERFORM | Executes one or more procedures and returns control to the next statement. |
| STOP | Terminates or temporarily suspends program execution. |

**Table 1-8 Procedure Division Verbs**

On the other hand, because the code in SCREEN-PARA is performed unconditionally, lines 2 and 3 can be sentences (periods) or statements (no periods). The only constraint is that the last statement in the paragraph end with a period.

```
SCREEN-PARA.
  SET ITEM-INDEX TO 1.
  DISPLAY PAY-SCREEN.
  ACCEPT PAY-SCREEN.
```

When you execute a program, the computer executes each sentence, in order, until it encounters a branching sentence; for example, a sentence instructing the system to PERFORM another routine, CALL another program, and so on.

## Sections and Paragraphs

A COBOL program is structured with sections and paragraphs or with paragraphs only. The format for a section is:

```
section-name SECTION [segment-number].
[paragraph-name. [sentence]...]...
```

The format of a paragraph is:

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

The above examples indicate that the Procedure Division can contain, at the minimum, one section-name and the word SECTION or one paragraph-name. Although this would be a valid syntactical construction, it would perform nothing. Sentences and statements perform the logic of the program. Statements must be contained in sentences, and sentences, in turn, must be contained in paragraphs. Sections are not required in COBOL.

Sections and paragraphs are referenced by the PERFORM and GO TO verbs. Segment-number is used for documentation purposes only.

# Arithmetic Expressions

The verbs used to form arithmetic statements are ADD, DIVIDE, MULTIPLY, SUBTRACT, and COMPUTE. They have several common features:

- The PICTUREs of the items must be numeric (9, S, V, P). However, they need not be identical; any necessary conversion and decimal point alignment are supplied throughout the calculation.
- The maximum size of each expression is 18 digits.
- The composite must not exceed 18 digits. The composite is a hypothetical data-item resulting from superimposing all expressions in a statement, aligned on their decimal points.

Two optional phrases may appear in all of the arithmetic statements: the ROUNDED phrase and the SIZE ERROR phrase. These phrases, which are described below, allow more specific control over computational results.

## Arithmetic Operators

The five binary arithmetic operators and two unary arithmetic operators may be used in arithmetic expressions. These operators must be preceded and followed by a space. Table 1-9 defines the operators.

## Evaluation Order

An arithmetic expression is a combination of identifiers, literals, and arithmetic operators that evaluates to a single numeric value. The evaluation order within an expression is:

1. Unary plus and minus
2. Exponentiation in left-to-right order

| Binary | Meaning |
|---|---|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| / | Division |
| ** | Exponentiation |
| **Unary** | |
| + | The effect of multiplication by +1 |
| - | The effect of multiplication by -1 |

**Table 1-9 Arithmetic Operators**

3. Multiplication and division in left-to-right order
4. Addition and subtraction in left-to-right order

Parentheses change the order of evaluation. Elements within parentheses are evaluated first. With nested parentheses, the innermost expression is evaluated first. Each left parenthesis must have a corresponding right parenthesis.

## Overlapping Operands

When a sending and a receiving item in an arithmetic statement or an INSPECT, MOVE, or SET statement share a part of their storage areas, the result of the execution of such a statement is undefined. The following example illustrates an *incorrect* move:

```
01 A.
   02 B.
      03 C PIC X.
      03 D.
         04 E PIC X.
         04 F PIC X.

   MOVE B TO D.
```

## Optional Phrases

Interactive COBOL provides two optional phrases: the ROUNDED phrase and the SIZE ERROR phrase. These phrases can be used in all arithmetic statements.

### The ROUNDED Phrase

When the result of an arithmetic operation has more digits to the right of the decimal point than the result-identifier provides for, low-order truncation occurs. The number of truncated digits is determined by the PICTURE clause of the result-identifier. When the ROUNDED phrase is specified and the first digit of the result's truncated portion is greater than 4, the value of the low-order digit in the result-identifier is increased by one. Conversely, when the first digit of the truncated portion is 4 or less, the digits are simply dropped. In either case, only the value of the first digit of the truncated portion is considered.

If the low-order integer positions in a result-identifier are represented by P in its PICTURE clause, the rightmost integer position for which storage is allocated is affected by the rounding or truncation. For example, if the value 6853 is stored in an item with PICTURE 99PP without rounding, the value 6800 is stored in the result-identifier; if rounding is specified, the value stored is 6900. The examples in Table 1-10 illustrates the logical (edited) results when using the ROUNDED phrase. Note that the actual results do not have decimal points or zeros that correspond with the P's.

| Picture | Amount | Result |
|---|---|---|
| 9999 | 1234.49 | 1234 |
| 9999 | 1234.50 | 1235 |
| 999V99 | 123.995 | 124.00 |
| 9V9 | 9.375 | 9.4 |
| 99V9 | 50.11 | 50.1 |
| 999PP | 12345 | 2300 |
| 999PP | 67891 | 67900 |

**Table 1-10 Results with the ROUNDED Phrase**

### The SIZE ERROR Phrase

A size error condition exists when, after decimal point alignment, the absolute value of a result is greater than the largest value that can be contained in the associated result-identifier. The size error condition applies both to the intermediate and to the final results of all arithmetic operations. If the ROUNDED phrase is specified, rounding takes place before checking for size error. Division by zero always causes a size error condition.

If a size error condition occurs during the performance of an arithmetic statement, the data-item affected by the statement retains its original value. If you specify a SIZE ERROR *imperative-statement*, the imperative statement is executed after the current statement is completed. Without a SIZE ERROR phrase, the arithmetic statement is, in effect, bypassed.

### The CORRESPONDING Phrase

The CORRESPONDING phrase processes all the subordinate data-items having the same names within two group items. For full details, see ADD CORRESPONDING, SUBTRACT CORRESPONDING, and MOVE CORRESPONDING in chapter 8.

## Conditional Expressions

The IF statement specifies conditional expressions. The program takes different actions depending on whether the condition is true or false. There are two categories of conditional expressions: simple and complex.

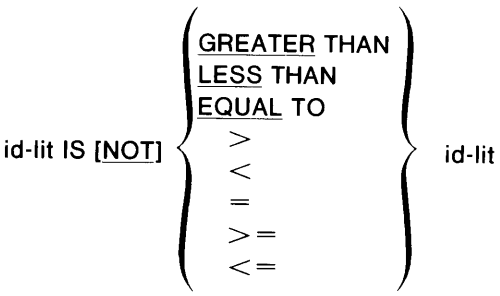
### Simple Conditions

The simple conditions are the relation, condition-name, class, switch-status, and sign conditions. A simple condition has a value of true or false. Examples of simple conditions are:

| | |
|---|---|
| Relation | `PERFORM CHECK UNTIL A IS LESS THAN B` |
| Condition-name | `IF END-OF-FILE` |
| Class | `IF NAME IS NOT ALPHABETIC` |
| Switch-status | `IF SWITCH-IS-ON` |
| Sign | `IF COUNT IS NEGATIVE` |

## Relation Conditions

A relation condition compares two items. Each item may be an identifier, a literal, or the value of an arithmetic expression. The general format for a relation condition is:




There are two types of comparisons: numeric and nonnumeric. If both items are numeric (that is, their PICTUREs contain only 9, S, V, P), the comparison is numeric. If one or more items are nonnumeric, the comparison is nonnumeric.

**Numeric comparisons.** Numeric comparisons are made according to algebraic value. The number of digits in an item does not affect the comparison; for example, 009 and 9 are equal. When numeric items are compared, the following rules apply:

- The items may have different lengths and USAGE designations.
- Unsigned numeric items are considered positive.
- Zero is considered a unique value regardless of sign.

**Nonnumeric comparisons.** If either item is a group item or is not numeric, a nonnumeric comparison takes place. A nonnumeric comparison is made according to the character collating sequence. Characters in corresponding positions are compared, starting from the high-order end. When the first unequal characters are encountered, the item that contains the character higher in the collating sequence is considered greater. Both items are equal if all comparisons are equal. The following rules apply to nonnumeric comparisions:

- The items being compared must be defined as USAGE IS DISPLAY.
- If the items have unequal lengths, comparison is made as if the shorter item were extended to the right with enough spaces to make both equal in length.
- When a figurative constant (except zero) is compared to a numeric identifier, the identifier must be defined as USAGE IS DISPLAY.

If one item is numeric:

- It must be an integer.
- It is treated as if it were moved to an elementary or group alphanumeric data-item of the same size as the nonnumeric data-item (depending on whether the nonnumeric data-item is an elementary or group item), and the contents of the alphanumeric item are used in the comparison.

**Example.** The following example illustrates a relation comparison:

```
WORKING-STORAGE SECTION.
01  NONNUMERIC-GROUP.
    02 SUBGROUP-1.
       03  ITEM-1   PIC XX VALUE "aa".
       03  ITEM-2   PIC XX VALUE "aa".
       03  ITEM-3   PIC XX VALUE "aa".
          02 SUBGROUP-2   PIC XXX VALUE "aaa".

01  NUMERIC-ITEM  PIC 9(9) VALUE 111111111.

PROCEDURE DIVISION.

MAIN-PARAGRAPH.
    IF NONNUMERIC-GROUP IS >= NUMERIC-ITEM PERFORM SHOW-RESULTS.
    STOP RUN.

SHOW-RESULTS.
    DISPLAY "The nonnumeric group is >= numeric item.".
    DISPLAY "NONNUMERIC GROUP = ", NONNUMERIC-GROUP.
    DISPLAY "     NUMERIC ITEM = ", NUMERIC-ITEM.
```

When the program is run, the following appears on the screen:

```
The nonnumeric group is >= numeric item.

NONNUMERIC GROUP = aaaaaaaaa

    NUMERIC ITEM = 111111111
```

## Condition-name Condition

A condition-name is a special way of writing a relation condition; it is used to improve program readability. A condition-name must have a level number of 88 and it must be followed by a VALUE clause, which specifies the literal or literals that apply to that condition.

The format of a level 88 entry is as follows:

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

    lit-3 [ { THROUGH } lit-4 ] ] ... .
            { THRU    }       ]
```

Within a range, *lit-1* must be less than *lit-2* and *lit-3* less than *lit-4*. If the Working-Storage Section contains the following code:

```
05  HOURS-WORKED              PIC 99.
    88 REG-HOURS                  VALUES ARE 1 THRU 40.
    88 OVERTIME-HOURS             VALUES ARE 41 THRU 99.
```

Then the Procedure Division could contain the following sentences:

```
IF REG-HOURS PERFORM REG-HOURS-ROUTINE.
IF OVERTIME-HOURS PERFORM OVERTIME-HOURS-ROUTINE.
```

The first sentence evaluates to true if the contents of REG-HOURS are from 1 to 40. If the sentence is true, REG-HOURS-ROUTINE is performed. Similarly, the second sentence evaluates to true if the the contents of OVERTIME-HOURS are from 41 to 99. If the sentence is true, OVERTIME-HOURS-ROUTINE is performed.

## Class Condition

The class condition determines whether the item being tested is alphabetic or numeric. An item is considered alphabetic if it consists entirely of the uppercase letters A-Z and spaces.

If an item's PICTURE does not contain an S, the item tests as numeric only if it contains digits 0 through 9.

If an item's PICTURE contains an S, the item is numeric only if the contents are numeric (0-9) and a valid sign is present in the correct position. Valid signs for data-items described with the SIGN IS SEPARATE clause are the plus and the minus signs. Valid signs for data-items not described with the SIGN IS SEPARATE clause are specified in the "SIGN Clause" section, chapter 7.

The general format for the class condition is:

id IS [NOT] { NUMERIC / ALPHABETIC }

The following rules apply to class conditions:

- The item being tested must be described as USAGE IS DISPLAY.
- The ALPHABETIC test cannot be used with a numeric item.
- The NUMERIC test cannot be used with an alphabetic item or a group item that contains a signed elementary item (PIC S).

## Switch-Status Condition

A switch-status condition determines the on or off status of a switch defined in the Environment Division. The switch-name and the condition-name associated with the switch must be named in the SPECIAL-NAMES paragraph. The general format for the switch-status condition is:

IF condition-name imper-stmt

where *condition-name* is the literal defined in an ON or OFF clause in the SPECIAL-NAMES paragraph of the Environment Division. Note that condition-name does not have to be defined elsewhere, i.e., there is no entry in Working-Storage. For example:

```
CONFIGURATION SECTION.
...
SPECIAL-NAMES.
    SWITCH "A", ON STATUS IS SW0.
   ...
PROCEDURE DIVISION.
   ...
    IF SW0 GO TO NEW-FILE-RTN.
    (where SW0 is the condition-name being tested.)
```

The result of the test is true if the program is executed or chained to with the same local switch declared in the program; in this case, /A. To set a switch, simply append the switch to the program-name when it is called. For example, in a calling program, the statement CALL PROGRAM "TESTPROG/A" sets the A switch on. You can set up to 26 switches at runtime.

### Sign Condition

The sign condition determines whether the algebraic value of a numeric item is less than, greater than, or equal to zero. An item is positive if its value is greater than zero, negative if its value is less than zero, and zero if its value is equal to zero. The general format for a sign condition is:

id IS [NOT] { POSITIVE / NEGATIVE / ZERO }

## Complex Conditions

Complex conditions result from combining conditions with the Boolean operators NOT, AND, and OR, and using optional parentheses to indicate the order of evaluation. Table 1-11 lists the logical operators and their meanings.

| Operator | Meaning |
|---|---|
| AND | Logical conjunction. The value is true if both conditions are true, and false if one or both conditions are false. |
| OR | Logical inclusive OR. The value is true if one or both of the included conditions are true, and false if both conditions are false. |
| NOT | Logical negation or reversal of truth value. The value is true if the condition is false, and false if the condition is true. |

**Table 1-11 Logical Operators**

Complex conditions occur in two forms: the negated simple condition and the combined condition. The general form for a negated simple condition is:

NOT simple-condition

The general format for a combined condition is:

condition [ { AND / OR } condition ] ...

where *condition* may be a simple condition, negated simple condition, combined condition, or negated combined condition.

The following example illustrates a negated simple condition:

```
NOT TOT IS LESS THAN MAX
```

The following examples illustrate combined conditions:

```
TOT IS LESS THAN MAX OR PART IS LESS THAN WHOLE
TOT IS LESS THAN 0 OR GREATER THAN 100
```

The truth value of a complex condition is determined by first evaluating the simple conditions and then evaluating the conditions formed by the truth values and the logical operators. Each logical operator must be preceded and followed by a space.

### Abbreviated Combined Relation Conditions

If a condition is combined, any condition except the first may be abbreviated by omitting either the subject of the relation condition or the subject and relational operator of the relation condition.

The format for an abbreviated combined relation condition is:

relation-condition $\left\{\begin{matrix}\underline{\text{AND}}\\ \underline{\text{OR}}\end{matrix}\right\}$ [NOT] [relational-operator] object ...

The following conditions are equivalent:

```
PAYROLL IS LESS THAN 0 OR PAYROLL IS LESS THAN TOTAL
PAYROLL IS LESS THAN 0 OR LESS THAN TOTAL
PAYROLL IS LESS THAN 0 OR TOTAL
```

In an abbreviated combined relation condition, if the word following NOT is GREATER, >, LESS, <, EQUAL, =, >=, or <=, then NOT participates as part of the relational operator. Otherwise, NOT is interpreted as a logical operator and, therefore, the implied insertion of subject or relational operator results in a negated relation condition. Some examples of abbreviated combined and negated combined relation conditions follow.

| Condition | Expanded Equivalent |
|---|---|
| a > b AND NOT < c OR d | ((a > b) AND (a NOT < c)) OR (a NOT < d) |
| a NOT EQUAL b OR c | (a NOT EQUAL b) OR (a NOT EQUAL c) |
| NOT a = b OR c | (NOT (a = b)) OR (a = c) |
| NOT (a GREATER b OR < c) | NOT ((a GREATER b) OR (a < c)) |

## Evaluation Order of Conditions

When parentheses are not used, or parentheses are not nested, conditions are evaluated in the following order:

1. Truth values are established for arithmetic expressions
2. Truth values are established for:
   - Simple conditions, in the following order: relation, class, switch-status, sign
   - Negated simple conditions
   - Combined conditions with the logical operator AND
   - Combined conditions with the logical operator OR
   - Negated combined conditions

Use parentheses to change the order of evaluation; conditions within parentheses are evaluated first. Within nested parentheses, evaluation proceeds from the innermost pair of parentheses to the outermost pair. When the sequence of evaluation is not completely specified by parentheses, the order of evaluation is from left to right in the sequence given above.

Parentheses are unnecessary when either AND or OR is used exclusively in one combined condition. However, you may need parentheses to determine a final truth value when the condition includes a combination of AND, OR, and NOT.

# File I/O Operations

I/O operations transfer data to and from files and devices. The following Interactive COBOL verbs are used for I/O: ACCEPT, CLOSE, DELETE, DISPLAY, OPEN, READ, REWRITE, START, UNDELETE, UNLOCK, and WRITE.

## Current Record Pointer

The current record pointer is a conceptual item indicating a logical position in an indexed or relative file in dynamic access mode. It is used to select the next record to be read. The current record pointer is changed by the OPEN, START, and READ statements. See these statements in chapter 8 for more information about the current record pointer.

## File Status Codes

After an I/O operation is executed, the system generates a File Status code indicating the outcome of the operation. If you specify the FILE STATUS clause in a file control entry, a value indicating the status of the I/O operation is placed into the specified two-character data-item. Your program can then test this item to determine the condition that terminated the I/O operation.

The File Status codes can be examined anywhere within the Procedure Division. However, special rules apply for examining File Status codes generated by an unsuccessful OPEN or CLOSE.

If an OPEN or CLOSE statement on your file is unsuccessful, your interactive COBOL program terminates immediately; the statement examining File Status codes is not executed. To prevent program termination, include a Declaratives section with a USE procedure for which this file qualifies.

Appendix A of each *User's Guide* lists the File Status codes.

## Exception Conditions

During I/O processing, a number of exception conditions might arise. Some examples of these are:

- Specifying an invalid key on a random access to a file
- Reaching the end of a file during sequential processing
- Opening a file with an illegal filename
- Encountering device or data errors

Interactive COBOL provides two methods of handling these exceptions. You can specify the AT END and INVALID KEY options in the appropriate I/O processing statements for a file. You can also specify the Declaratives section at the beginning of the Procedure Division to process all exceptions for any files or sets of files.

### The INVALID KEY Condition

The INVALID KEY condition occurs as the result of an unsuccessful execution of a DELETE, READ, REWRITE, START, UNDELETE, or WRITE statement. Two examples of invalid key errors are:

- A READ, REWRITE, or DELETE statement attempts to access a record with a primary key that does not exist in the file.
- A WRITE statement is executed for an existing primary key.

When the INVALID KEY condition is recognized, the system takes the following actions in the order given:

1. A non-zero value is placed into the FILE STATUS data-item, if one is specified for this file, to indicate an INVALID KEY condition.
2. If the INVALID KEY phrase is present, control is passed to the associated imperative statement; any USE procedure specified for the file is not executed.
3. If the INVALID KEY phrase is not present, but a USE procedure is specified for the file, that procedure is executed.

If the INVALID KEY condition occurs, the I/O statement that recognizes the condition fails, and the file is not affected.

### The AT END Condition

The AT END condition occurs when a sequential READ, READ NEXT, or READ PREVIOUS statement attempts to process past the end of a file or before the beginning of a file. When the AT END condition is recognized, the system takes the following actions:

1. A non-zero value is placed in the FILE STATUS data-item, if specified for this file, to indicate an AT END condition.
2. If the AT END phrase is present, control passes to the specified imperative statement; any USE procedure designated for the file is not executed.
3. If the AT END phrase is not present, but a USE procedure is specified for the file, that procedure is executed.

If the AT END condition occurs, the I/O statement that causes the condition fails. The contents of the associated record area and the status of the current record pointer are undefined.

For more information, see the READ statement in chapter 8.

## The Declaratives Section

The Declaratives section contains procedures to process exceptional I/O conditions that arise during execution of the file handling statements READ, WRITE, UNLOCK, OPEN, CLOSE, START, REWRITE, DELETE, and UNDELETE. The Declaratives section is written at the beginning of the Procedure Division. Each Declaratives section contains a USE statement identifying the conditions under which the associated procedures are executed.

Procedures specified in the Declaratives section are executed when the AT END or INVALID KEY condition exists and the appropriate phrase has not been specified in the I/O statement, or when some other I/O error occurs. After execution of a USE procedure, control returns to the statement following the statement that caused the error.

For more information on the Declaratives section, see the "Declaratives Format" and the USE statement in chapter 8.

# Chapter 2
# File Organization and Access

Interactive COBOL supports sequential, indexed, and relative file organizations. Files are automatically created by the system when a file that does not exist is opened for OUTPUT by a COBOL program, or when a relative or indexed file is opened for I-O. Table 2-1 lists file organizations and their associated record and key lengths.

| File Organization | Record Length | Key Length |
|---|---|---|
| Sequential | | |
| fixed or variable (assigned to DISK) | limited only by program size | n/a |
| line (assigned to PRINTER, KEYBOARD, or DISPLAY) | 132-byte max. | n/a |
| Indexed | 4096-byte max. | 100-byte max. for each key |
| Relative | 4096-byte max. | 2-byte binary |

**Table 2-1 File Organizations, Record Lengths, and Key Lengths**

The FILESTATS utility may be used to determine the amount of space in blocks required for an indexed or relative file. The utility uses the number of records, key length, and record length as input, and provides index size, data size, and total file size as output. (For details of FILESTATS, see the *Interactive COBOL Utilities* manual for your operating environment.)

## Sequential Files

The records in a sequential file are arranged in the order in which they were written; you can retrieve the records only in that order. You can READ a sequential file only from the beginning of the file. If you add a record, it is appended to the end of the file.

No keys are associated with records in a sequential file. A record in a sequential file cannot be deleted; a new file must be created to accomplish this.

There are three types of sequential files:

- Fixed sequential
- Variable sequential
- Line sequential

Sequential files are specified as ORGANIZATION IS SEQUENTIAL in the file-control entry in the Environment Division. This entry may be omitted, as sequential organization is the default.

In fixed-length sequential files, each record is the same length. This length is determined at the time the file is created, and it cannot change. The record length is set with the RECORD CONTAINS clause in the Data Division. The RECORDING MODE IS FIXED clause specifies the file type as fixed sequential; however, if this clause is omitted, fixed sequential is assumed.

In variable-length sequential files, each record contains a 2-byte header that stores the record length. However, you must give a range of possible record sizes with the RECORD CONTAINS clause in the Data Division. The RECORDING MODE IS VARIABLE clause specifies the file type as variable sequential.

In fixed and variable sequential files, storage for the record is reserved in the program file. Therefore, the only length restriction on a record is that the resulting program file not be too large to run under the runtime system. For information on maximum program sizes, see the *Interactive COBOL User's Guide* for your operating system.

Line sequential files written by Interactive COBOL programs contain variable-length records terminated by CR (on RDOS and DG/RDOS) or NEW LINE (on AOS and AOS/VS). A line sequential file is required when data is output to a sequential device such as a line printer. Line sequential files do not use the RECORDING MODE clause. They can have a RECORD CONTAINS clause to specify maximum record size, but this is not required. There is a system limit, however, on maximum record size: each record cannot exceed 132 characters (plus the terminator).

You can create a line sequential file with the ASSIGN clause. Whereas fixed and variable sequential files must be assigned to DISK, a line sequential file must be assigned to PRINTER (for output files), KEYBOARD (for input files), or DISPLAY (to send data directly to the screen without using the Screen Section).

The usual way of entering data through the keyboard is by using the ACCEPT verb and the Interactive COBOL Screen Section. An alternative to this is to assign a line sequential file to KEYBOARD with no external file-name and open that file for input. A record "read" from the keyboard is terminated when you press CR or NEW LINE.

Similarly, data is usually output to the terminal via DISPLAY and the Interactive COBOL Screen Section. An alternative to this is to assign a line sequential file to DISPLAY with no external file-name, open that file for OUTPUT, and write to it.

In a file with line sequential records, the READ statement inputs characters up to the first CR (octal 015), null (octal 000), form feed (octal 014), or NEW LINE (octal 012). (NEW LINE is not a line terminator on RDOS or DG/RDOS.)

When a READ statement is executed, the system overwrites the contents of the record area up to and including the terminator. Therefore, after execution of a READ statement for a line sequential file, the contents of the associated record area contain the record, the terminator, and any characters after the terminator that were present before execution of the READ. To avoid processing these characters, clear the record area before each read; for example, by moving spaces or zeros to the record area.

You can append records to the end of a sequential file with the OPEN EXTEND statement. Records can be read, modified, and rewritten consecutively with the OPEN I-O statement.

open I-O on seq - file has to exist or 91 error

open output - If file exists - it will be overwritten.

# Indexed Files

An indexed file consists of records identified by key values, rather than by physical or logical position. The key allows direct access to a particular record using a single program statement. Indexed file organization is permitted only for files assigned to DISK.

An indexed file must have at least one key, and it may have up to one primary key and four alternate keys. The data-item named in the RECORD KEY clause of the SELECT clause for an indexed file is the primary record key for that file. The ALTERNATE RECORD KEY clause specifies the optional alternate keys. Both primary and alternate keys must be defined in a record description in the File Section of the Data Division. The definition consists of a level number, a data name (the same as used in the SELECT clause), and a PICTURE clause. For the purposes of inserting, updating, deleting, and undeleting records in a file, each record is identified solely by the value of its primary key, which must be unique for each record. The key value of HIGH-VALUES is reserved for system use. Your program should not have any key values equal to HIGH-VALUES.

An indexed file is implemented as two external files at the operating system level. One is the *index portion,* containing keys and pointers; the other is the *data portion,* containing records. An Interactive COBOL program references the file by a single filename. The runtime system appends the extensions .NX and .XD to the filename to identify the index portion and the data portion, respectively.

## The Index Portion

The blocks in an index portion are logically organized in a tree structure, with a maximum of six levels. The root level consists of one block, which contains key entries in ascending collating sequence. Each key entry in the root level has the highest ASCII value in a block at the second level. Pointers connect each key entry in the root level to the appropriate block at the next level. Succeeding levels follow the same pattern until a terminal level is reached. Blocks in the terminal level contain key entries for all existing records. Associated with each key entry is a pointer to the data record in the data portion. Figure 2-1 shows the tree structure of an indexed file.

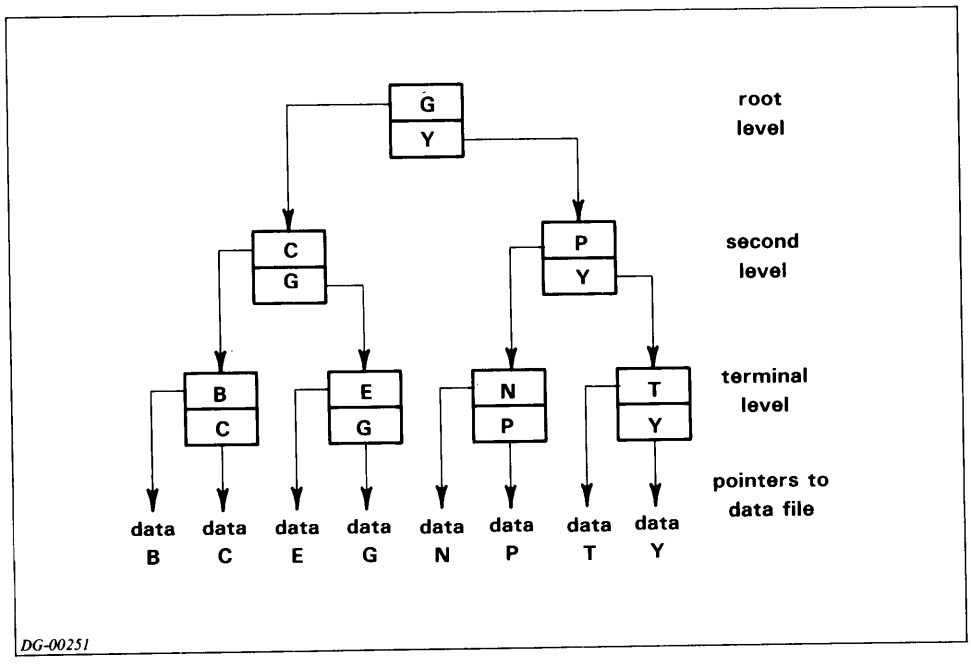



***Figure 2-1*** *Indexed File Structures*

## The Data Portion

The data portion of an indexed file contains the data records, including the keys, in the order in which records are written; they are not necessarily in ascending collating sequence. This implementation makes file expansion simpler since file space on the disk is allocated as needed. Consequently, records need no overflow areas.

**Sequential Access.** The data portion of an indexed file is structured so that sequential access is efficient. Data records are linked within the data portion; thus, sequential access is independent of the index portion. In sequential access mode, records are written and retrieved in ascending order of record key values.

**Random Access.** In random access, the search algorithm compares a given key to the key entries in a block at one level of the structure and follows a pointer to a block at the next level. The index structure and the algorithm make the search efficient. Suppose the record with the key E (refer to Figure 2-1) is to be located. First E is compared to the first entry at the root level. Since E is less than G (i.e., it precedes it in the collating sequence), the search proceeds to the second level by following the pointer from G. The pointer leads to the block containing C and G. Since E is greater than C and less than G, the search continues to the third level by following the pointer from G. In this example, the third level is the terminal level, and the pointer to the data record is reached.

**Dynamic Access.** Dynamic access enables you to use one OPEN statement to access records in a relative or indexed file sequentially and randomly. The ACCESS MODE clause, described in chapter 6, is used to access files dynamically.

## File Efficiency

As shown by the example of random access, the search through the index portion for a given key requires a "read and compare" at each level of the tree structure. Thus, one objective in designing a file is to minimize the number of levels in the index structure. Keys may be up to 100 bytes long, but shorter keys are generally more efficient. Longer keys mean fewer key entries per block, hence a greater number of levels in the tree structure. The number of levels also increases as the number of records increases.

Use the FILESTATS utility at the file design stage to predict the size of the index portion of a file with a given key length and a given number of records. The output of FILESTATS includes the number of key entries per index block and the number of levels in the structure, as well as the calculated sizes of the index portion and the data portion.

## Record Deletion

The COBOL verb DELETE deletes records logically. The record in the data portion is flagged as deleted and, for the user, no longer exists. However, the data record is not physically removed from the file. Interactive COBOL provides the UNDELETE verb, which restores a logically deleted record to active use. DELETE and UNDELETE require the use of the primary key for the record.

The ANALYZE utility reports the number of records that have been logically deleted from a file. If this number becomes large in relation to the total number of records, disk space is being wasted; and the index portion contains more levels than necessary. The COLLAPSE and REORG utilities physically delete logically deleted records. For more information on these utilities, see the *Interactive COBOL Utilities* manual for your operating system.

## Primary and Alternate Keys

An indexed file may be designed so that a record can be accessed by more than one path. In addition to a primary key, you can specify up to four alternate keys. The primary key must uniquely identify a record. An alternate key can have duplicates; several records may have the same value for an alternate key. For example, a file containing sales force records might have the salesperson's name as the primary key and the salesperson's employee number and sales region as alternate keys. The sales region key can be used to retrieve the names of all the sales personnel in a given region.

When duplicate alternate keys are created, access paths to the records are built in the order of entry to the file. Therefore, when an alternate key with duplicates is used to retrieve a record, the first duplicate written is the first to be read.

An alternate key can also serve as another unique record identifier. However, if an alternate key is to be a unique reference, the applications program must check the uniqueness of the key, because COBOL will not. The INVALID KEY phrase controls checking for duplicates of the primary key only.

Alternate keys are named in ALTERNATE RECORD KEY clauses in the SELECT clause and defined along with the primary key in a record description in the Data Division. The definition consists of a level number, a data name (the same as used in the SELECT clause), and a PICTURE clause. Each key may be up to 100 bytes long, with a maximum total of 500 bytes for a primary key and four alternates.

At the operating system level, the index portion of an indexed file with alternate keys contains index structures for the primary key and for each alternate key. All index structures are the same as the structure described above (see Figure 2-1). Each index structure points to a common data portion. At the index's terminal level for an alternate key, each key entry points to the first data record that contains that alternate key value. Once the first record is located, additional records with that value for the alternate key can be accessed sequentially.

## File Updating

A READ or START statement may refer to an alternate key, but a WRITE or REWRITE statement must refer to a primary key. When a WRITE statement is executed, the primary key and alternate keys are added to their respective index structures.

A REWRITE statement can change an alternate key, but not a primary key. The record update takes longer for a file with alternate keys. When an alternate key is changed, the old index entry is flagged for deletion and a new index entry is written. The data portion is merely updated in place. Therefore, if ANALYZE is run on a file after a large number of alternate keys have been deleted, ANALYZE may report a number for deleted keys that is greater than the total number of records (including deletions) currently in the file. (See the *Interactive COBOL Utilities* manual for a description of ANALYZE.)

# Relative Files

A relative file consists of records and a 2-byte binary key that uniquely identifies the logical ordinal position of each record in the file. The key is not a part of the data record. The representation of relative files is similar to that for indexed files. A relative file is also implemented as two files, an index portion and a data portion, named with the extensions .NX and .XD. The index portion has the tree structure described for indexed files.

The advantage of using a relative file is that the 2-byte key, described as PIC 9(4) COMP to PIC 9(18) COMP, can identify up to 65,534 records. Five bytes per key would be required for this many records if the key were in decimal format in an indexed file. The reduced key length means that relative files have fewer index levels than indexed files; hence access time is improved. Relative files also provide all the advantages of indexed access when keys can be expressed as numbers from 0 to 65,534. The data portion is the same as that for indexed files, except that the key is not part of the record.

The following restrictions apply to relative key descriptions:

- If you use the /I compiler switch, the relative key must be described as PIC 9(4) COMP.
- If you use the /A compiler switch, PIC 9(4) COMP allows you to access relative keys with values from 0 to 9999 only. To access keys with values from 10,000 to 65,534, define the relative key as PIC 9(5) COMP.
- If you use neither the /A nor /I compiler switches, defining the relative key as PIC 9(4) COMP allows you to access all records (keys with values from 0 to 65,534).

The FILESTATS utility helps design relative files. The ANALYZE utility outputs statistics on existing relative files.

# Chapter 3
# Interactive Screen Management

With Interactive COBOL's screen management facility, an Interactive COBOL program can:

- Specify exact display locations for input, output, and update fields
- Display literal text in predefined display positions
- Accept data typed in designated positions on the display
- Update the contents of fields on the display
- Control such console features as blinking or sounding a tone

A single program may contain as many different screens as desired. Screen entries are defined in the Screen Section of the Data Division and are referenced by the DISPLAY and ACCEPT verbs.

In addition, Interactive COBOL provides the SCREEN utility for development systems. SCREEN lets you compose input and output fields on the display terminal and to produce a file with lines of source code that describe the image just created. The source code may then be inserted directly into a program or used in a COPY file. For more information see *SCREEN: Screen Format Editor*.

## Screen Data Description

Each field that appears on the display must be described in the Screen Section. Each screen entry consists of a level number, an optional screen-name, and optional clauses that specify the positions of display fields and other display terminal functions, such as blinking or sounding a tone.

Screen entries can be literal, data-item, or group items. Literal format is used to display constant information, such as a heading or a menu. Data-item format is used to pass data between File, Working-Storage, and Linkage sections and screen storage. Literal and data-item entries can be combined to form a group entry, which allows you to display and accept an entire screen by executing one DISPLAY and ACCEPT sequence.

The following rules apply to the Screen Section:

- The Screen Section must follow the Working-Storage and Linkage sections.
- The level number must be the first element in each data description. Level numbers may range from 01 to 49. Entries at the 01 level must begin in area A; other levels may be indented as desired.
- A screen-name is required with 01 level numbers but not with other level numbers.
- The REDEFINES, OCCURS, and FILLER clauses are not permitted in the Screen Section.

## Literal Format

A literal entry is typically a line of text, although it may be a single character or simply positioning information. The syntax for a literal entry is:

```
level-number [screen-name]
    [; BLANK SCREEN]
    [; BLANK LINE]
    [; BELL]
    [; BLINK]
    [; LINE NUMBER IS {integer-1 | PLUS integer-2 | identifier-1}]
    [; {COLUMN | COL} NUMBER IS {integer-1 | PLUS integer-2 | identifier-1}] [; VALUE IS literal].
```

A literal entry consists of a level number, an optional screen-name, and at least one other clause. The clause can be a screen positioning clause (LINE and COLUMN) or a literal to be displayed. If a literal is displayed, it must be a quoted string. The string can contain any text, such as a report heading, a prompt requesting information, or an error message. The following example shows a literal entry. Note that the VALUE IS clause is not required when specifying a literal.

```
01 NAME-PROMPT LINE 5 COL 5 "ENTER CUSTOMER NAME:".

PROCEDURE DIVISION.
    ...
    DISPLAY NAME-PROMPT.
```

When the DISPLAY statement is executed, the prompt appears on line 5, column 5. However, the operator cannot respond to the prompt, because literal format lets you display constant information only. You cannot execute an ACCEPT statement on an entry in literal format. To let the operator enter information, you must use data-item format.

## Data-Item Format

The syntax for a data-item entry is:

```
level-number [screen-name]
    [; BLANK SCREEN]
    [; BLANK LINE]
    [; BELL]
    [; BLINK]
    [; LINE NUMBER IS {integer-1 | PLUS integer-2 | identifier-1}]
    [; {COLUMN | COL} NUMBER IS {integer-1 | PLUS integer-2 | identifier-1}]
    ; {PICTURE | PIC} IS picture-string {; FROM id-lit | ; TO id | ; USING id}
```

[; BLANK WHEN ZERO]

; { JUSTIFIED / JUST } RIGHT

[; AUTO]
[; SECURE]
[; REQUIRED]
[; FULL].

An entry in data-item format must contain a PICTURE clause, which describes how the system edits the contents of the data-item when it is displayed or accepted, and a FROM, TO, or USING clause, which specifies a corresponding File, Working-Storage, or Linkage Section data-item.

The FROM clause specifies an output data-item. During execution of a DISPLAY statement, the contents of the data-item are displayed.

The TO clause specifies an input data-item. During execution of a DISPLAY statement, underscores are displayed according to the data-item's PICTURE clause. The underscores indicate the number of character positions in the display field. During execution of an ACCEPT statement, the contents of the data-item can be altered.

The USING clause specifies an update data-item. During execution of a DISPLAY statement, the current contents of the data-item are displayed. During execution of an ACCEPT statement, the operator can alter that information.

The other clauses define the position of the data-item on the display and any special terminal functions; they are discussed below.

The following example illustrates the pertinent statements for a simple screen I/O operation. In the example, NAME-PROMPT is a literal entry and SCR-NAME is a data-item entry. Execution of the first DISPLAY statement displays the contents of NAME-PROMPT, the execution of the second DISPLAY displays 25 underscores. The ACCEPT statement allows the operator to enter data in the entry SCR-NAME; when the ACCEPT is complete, the contents of screen storage are transferred to the data-item CUST-NAME in the Working-Storage Section.

```
WORKING-STORAGE SECTION.
    ...
    05  CUST-NAME   PIC X(25).
    ...
SCREEN SECTION.

01  NAME-PROMPT LINE 5 COL 5 "ENTER CUSTOMER NAME:".

01  SCR-NAME LINE 5 COL 27 PIC X(25) TO CUST-NAME.
    ...
PROCEDURE DIVISION.

    DISPLAY NAME-PROMPT.
    DISPLAY SCR-NAME.
    ACCEPT SCR-NAME.
```

## Group Format

A group entry consists of one or more data-item or literal entries, or a combination of both. The syntax for a group entry is:

```
level-number screen-name
    [; AUTO]
    [; SECURE]
    [; REQUIRED]
    [; FULL]
    {data-item or literal entry ...}
```

With a group entry, you can display or accept any number of screen entries by executing a single DISPLAY or ACCEPT statement. Although you cannot execute an ACCEPT on a literal entry or a data-item entry that includes a FROM clause only, you can execute an ACCEPT on a group entry that contains these types of entries, as long as it also contains one or more data-item entries with the TO or USING clause.

A group entry is completed after the operator presses a terminator at the last input or update field and the data is validated. Until that time, the operator can use the arrow keys to move to any input or update field on the display and then modify it.

A group entry is constructed according to the usual COBOL rules. It can contain a hierarchy of other group and elementary items, with a maximum of 49 levels. Level numbers can range from 01 to 49. The following example illustrates group format:

```
DATA DIVISION.
    ...
WORKING-STORAGE SECTION.
77  CUST-NAME      PIC X(25).
77  CUST-NUMBER    PIC 9(8).
77  CHARGE-AMOUNT  PIC 999V99.
    ...
SCREEN SECTION.

01  CUST-DATA.
    02  LINE 5  COL 5   "ENTER CUSTOMER NAME:".
    02  LINE 5  COL 33  PIC X(25)  USING CUST-NAME.
    02  LINE 7  COL 5   "ENTER ACCOUNT NUMBER:".
    02  LINE 7  COL 33  PIC 9(8) USING CUST-NUMBER.
    02  LINE 9  COL 5   "ENTER AMOUNT OF CHARGE:".
    02  LINE 9  COL 33  PIC $999.99 USING CHARGE-AMOUNT.
    ...
PROCEDURE DIVISION.
    ...
    DISPLAY CUST-DATA.
    ACCEPT CUST-DATA.
```

A group entry must have a name; however, it is permissible to omit names for items within the group, as shown above. Omitting a name is equivalent to a FILLER entry (the word FILLER is illegal in the Screen Section). The individual entry cannot be referenced except as part of a higher level group.

| Group Format | Literal Format | Data-Item Format |
|---|---|---|
| AUTO[1] | BELL | AUTO[1] |
| FULL[1] | BLANK LINE | BELL |
| REQUIRED[1] | BLANK SCREEN | BLANK LINE |
| SECURE[1] | BLINK | BLANK SCREEN |
| | COLUMN | BLANK WHEN ZERO |
| | LINE | BLINK |
| | VALUE IS literal | COLUMN |
| | | FROM[2] |
| | | FULL[1] |
| | | JUSTIFIED |
| | | LINE |
| | | PICTURE IS picture-string |
| | | REQUIRED[1] |
| | | SECURE[1] |
| | | TO[1] |
| | | USING |

**Table 3-1 Screen Clauses**

[1]AUTO, FULL, REQUIRED, and SECURE cannot be used for an entry designated as output only (FROM clause).

[2]An entry may contain a FROM clause and a TO clause; however, if it includes both clauses it cannot also specify the USING clause.

# Screen Section Clauses

Interactive COBOL provides an extended set of clauses for the Screen Section. More than one clause may be associated with each screen entry. Clauses can be in any order within an entry; however, certain clauses may not be used together in a screen definition. Table 3-1 lists the clauses that can be used with each format. Each clause is described below.

## VALUE Clause

The VALUE clause specifies literal information to be displayed. It can be used only in a literal entry. The VALUE literal must be nonnumeric and must be enclosed in quotation marks. It cannot be a figurative constant. The words VALUE IS do not have to precede the literal.

## PICTURE Clause

The PICTURE clause functions in much the same way as in the other sections of the Data Division. It describes the category, size, and appearance of the display field's contents.

The picture-string may include any standard editing characters, such as those for comma insertion, zero suppression, or floating currency sign. The picture-string characters P, V, CR, and DB can be used only with output fields (FROM clause).

The PICTURE clause controls the appearance of data displayed. During execution of a DISPLAY statement, data is displayed according to the PICTURE associated with the screen data-item. All editing characters are present and any special clauses are in effect.

The PICTURE clause also checks the category of data that is accepted in a field. When an operator terminates an input or update field, the runtime system checks

whether the entered data conforms to the PICTURE of the screen data-item. For example, if the operator enters an alphabetic character in a numeric field, the system displays an error message, positions the cursor to the first incorrect character, and waits for the operator to reenter the data.

The PICTURE of the File, Working-Storage, or Linkage data-item named in the FROM, TO, or USING clause does not have to correspond exactly to the PICTURE clause in the screen data-item. During execution of a DISPLAY/ACCEPT sequence, data is transferred between the File, Working-Storage, or Linkage data-item and the screen data-item according to the usual rules for MOVE, with one exception: data from a numeric edited data-item in the Screen Section can be moved to a numeric data-item in another section. In this case, all characters of the numeric edited data-item except digits, sign, and decimal point are ignored. The decimal point is used for alignment purposes only.

## FROM, TO, and USING Clauses

Every data-item entry must include a FROM, TO, or USING clause. The identifier following the FROM, TO, or USING clause must be defined in the File, Working-Storage, or Linkage sections of the Data Division. The identifier may be qualified, subscripted, or indexed.

### FROM Clause

A FROM clause identifies an output field. When a DISPLAY statement is executed, the contents of the associated File, Working-Storage, or Linkage item are moved into the screen data-item and the contents of the screen data-item are displayed. Fields with a FROM clause appear at half intensity.

### TO Clause

A TO clause identifies an input field. The TO identifier references a data-item in the File, Working-Storage, or Linkage sections. When a DISPLAY statement is executed on a screen data-item with the TO clause, underscores are displayed. The number of underscores displayed corresponds to the number of characters in the picture-string. When the operator enters a character, the underscore acts as a terminator; characters to the right of the first underscore are ignored. After the ACCEPT statement is completed, the contents of the data-item in screen storage are moved to the associated File, Working-Storage, or Linkage data-item. Fields with a TO clause appear at full intensity.

If the operator terminates an input field without entering any data and no previous value is retained, or the operator does not visit an input field, a default value is transferred from the screen data-item to the associated File, Working-Storage, or Linkage data-item. This default value is zero for numeric data-items and spaces for other data-items.

### USING Clause

A USING clause identifies an update field; that is, a field that is both input and output. When a DISPLAY statement is executed, the current contents of the screen data-item are displayed.

When an ACCEPT statement is executed, the operator can enter new data in the update field. The contents of the display field are left-justified and editing characters are removed. The remainder of the field is filled with underscores.

You cannot combine the USING clause with the FROM or TO clause in one entry. Fields with a USING clause appear at full intensity.

A screen entry can specify that data is to be displayed from one identifier or literal and accepted into another identifier. This operation requires both a FROM and TO clause, rather than a USING clause. Also, data is transferred between the File, Working-Storage, or Linkage data-item and the screen data-item according to the usual rules for MOVE, with one exception: data from a numeric edited screen data-item can be moved to a numeric data-item.

## LINE and COLUMN Clauses

The LINE and COLUMN clauses specify the position of the field on the display. When a screen item is accepted or displayed, the field position starts at the location designated in the LINE and COLUMN clauses. Any existing characters in that location on the display are overwritten.

Under RDOS and DG/RDOS, the DEFLINES utility sets the display size. Under AOS and AOS/VS, the CHARACTERISTICS command sets the display size. Most terminals have a display size of 80 columns wide and 24 lines long. System error messages are displayed on line 24. Consequently, you should keep this line free when designing displays.

If the COLUMN and LINE values are larger than the display size, the fields will wrap onto the same line or column. If your display is 80 columns wide, for example, a field that begins at column 82 will appear in column 2. A display field must start at column 127 or less, although it can continue up to column 255.

A display position may be specified absolutely or relatively. Relative positions are always set at compile time. Absolute positions are set at compile time, if LINE or COLUMN are specified by an integer, or at runtime, if LINE or COLUMN are specified by an identifier. The following sections discuss absolute positioning at execution time, as well as relative positioning.

### Absolute Positioning Using an Integer

An absolute position indicates a specific line and column number. The line or column number can be specified with an unsigned integer or an identifier (discussed below). The first positioning clause in a group must always be absolute, for example:

```
01  HEADER-SCREEN.
    02  LINE 3 COL 5 VALUE "MONTHLY UPDATE REPORT".
```

### Relative Positioning

The PLUS phrase specifies relative positioning. The positioning is relative to the last position of the cursor. This eliminates the need to count lines or columns. An example of relative positioning is:

```
01  HEADER-SCREEN.
    05  LINE 3 COL 5 VALUE "MONTHLY UPDATE REPORT".
    05  LINE PLUS 2 COL 3 VALUE "CUSTOMER NAME".
    05  LINE PLUS 1 COL 3 VALUE "-------- ----".
```

Relative display positions are translated to absolute positions by the compiler. All relative positions are absolute at compile time. Consequently, the order in which relatively positioned data-items are defined determines their actual display positions, not the order in which the fields are displayed or accepted. This is illustrated by the following example:

```
SCREEN SECTION.

01  SCREEN-PROMPTS.
    05  SCR-EMP-PROMPT LINE 5 COL 5 PIC X(20) FROM EMP-PROMPT.
    05  SCR-EMP-NUM LINE 5 COL 28 PIC 9(9) TO EMP-NUM.
    05  SCR-PAY-PROMPT LINE PLUS 2 COL 5 PIC X(20) FROM PAY-PROMPT.
    05  SCR-PAY-RATE COL 28 PIC 99.99 TO PAY-RATE.
    05  SCR-HRS-PROMPT LINE PLUS 2 COL 5 PIC X(20) FROM HRS-PROMPT.
    05  SCR-HRS COL 28 PIC 99.99 USING HRS.
    ...
    DISPLAY SCR-PAY-PROMPT.
    DISPLAY SCR-PAY-RATE.
    ACCEPT SCR-PAY-RATE.
    IF PAY-RATE < 50.00 PERFORM PAY-RATE-ERROR-PARA.
    ...
    DISPLAY SCR-HRS-PROMPT.
    DISPLAY SCR-HRS.
    ACCEPT SCR-HRS.
    IF HRS < 40.00 PERFORM OVERTIME-PARA.
```

The DISPLAY and ACCEPT sequence for the pay rate and the hours worked can appear in any order; they do not affect the final appearance of the display. Of course, they do affect the order in which the operator responds to the prompts.

Each clause has four possibilities:

- No LINE or COLUMN clause
- Only the LINE or COLUMN clause with no number
- The LINE or COLUMN clause with a number: LINE *m* or COLUMN *n*
- The LINE or COLUMN clause with the PLUS phrase: LINE PLUS *m* or COLUMN PLUS *n*.

Furthermore, using one form of one clause may affect the other clause. Table 3-2 shows the resultant field position for each combination.

| LINE Clause | COLUMN Clause | Field Position |
|---|---|---|
| No clause | No clause | No change |
| | COL | Same line, column plus 1 |
| | COL *n* | Same line, column *n* |
| | COL PLUS *n* | Same line, column plus *n* |
| LINE | No clause | Line plus 1, column 1 |
| | COL | Line plus 1, column plus 1 |
| | COL *n* | Line plus 1, column *n* |
| | COL PLUS *n* | Line plus 1, column plus *n* |
| LINE *m* | No clause | Line *m*, column 1 |
| | COL | Line *m*, column plus 1 |
| | COL *n* | Line *m* , column *n* |
| | COL PLUS *n* | Line *m*, column PLUS *n* |
| LINE PLUS *m* | No clause | Line plus *m*, column 1 |
| | COL | Line plus *m*, column plus 1 |
| | COL *n* | Line plus *m*, column *n* |
| | COL PLUS *n* | Line plus *m*, column plus *n* |

**Table 3-2 COBOL Line and Column Positioning**

### Absolute Positioning Using an Identifier

Lines and columns can also be specified by an identifier, which allows positioning to be defined at execution time. The identifier must be an elementary unsigned item and can be subscripted or indexed. The following example shows a screen entry with dynamic LINE and COLUMN setting:

```
05 LINE LINE-NUM COL COL-NUM "ENTER CUSTOMER NAME: "
```

If a line or column number is specified dynamically, any following lines or columns in the screen entry cannot be specified relatively; that is, the PLUS phrase cannot occur after an identifier is used to specify LINE or COLUMN.

### Positioning Sequence

Line and column positioning is performed in the following order:

- Absolute position specified with integers (LINE *n*, COL *n*). These line numbers are set during compile time.
- Relative position (LINE PLUS *n*, COL PLUS *n*). These line numbers are set during compile time.
- Absolute position specified with identifiers (LINE id, COLUMN id). These line numbers are set during runtime.
- Variable origin for an entire display (DISPLAY *screen-name* AT, ACCEPT *screen-name* AT). The runtime system offsets all absolute and relative positions according to this origin.

## BLANK SCREEN and BLANK LINE Clauses

The BLANK SCREEN clause erases the entire display and positions the cursor to line 1, column 1. BLANK LINE erases the current line from the cursor position to the end of the line without changing the cursor position. Since BLANK LINE is executed after the positioning commands LINE and COLUMN, you can use it to clear a previously displayed item (such as an error message) before displaying a new item at the same position.

## BELL Clause

The BELL clause sounds the terminal tone. It is used to attract the operator's attention. Use the BELL clause with literal or data-item formats. If the BELL clause is part of a group entry, the tone is sounded when the group is displayed.

## Input Control Clauses

The input control clauses affect input fields during data entry. They restrict the way in which data is entered, the way data is echoed on the display, and the way fields are terminated. These clauses are AUTO, FULL, REQUIRED, and SECURE.

Input control clauses in a group entry are associated only with the input or update items in the group (items having the TO or USING clause).

The AUTO terminates the input or update field automatically when it is full. The operator does not have to type a terminator to position the cursor to the next field. If only one entry is to be input, or if the entry is the last one in a group, execution of the ACCEPT statement is completed.

The FULL clause requires that, if the operator enters data, he or she must enter a character or space in each position before terminating the input or update field. The operator can press a terminator to bypass the entire field.

The REQUIRED clause specifies that the operator cannot bypass the input or update field. He or she must enter at least one character or space in the field.

The SECURE clause echoes asterisks on the display rather than the actual data. This protects the privacy of such data as authorization codes. When SECURE is specified, the JUSTIFIED and BLANK WHEN ZERO clauses have no effect.

### Display Clauses

The display clauses define the way in which a literal or data-item field is displayed, and the way in which entered data is redisplayed after a data-item field is accepted. These clauses are BLANK WHEN ZERO, BLINK, and JUSTIFIED.

BLANK WHEN ZERO fills the field with spaces when the data-item has a zero value. This clause can be used only with a numeric or numeric-edited screen entry. An entry is regarded as numeric edited when this clause is specified.

The BLINK clause causes a literal or data-item field to blink on the display. BLINK has no effect on input fields.

JUSTIFIED specifies right justification of data displayed in a field. This clause can be used only with an unedited alphabetic or alphanumeric data-item.

## Order of Execution

When a screen I/O statement is executed, terminal functions for individual entries are performed in the following order, regardless of the order in which they appear in a screen description:

```
BLANK SCREEN
LINE and COLUMN positioning
BLANK LINE
BELL
DISPLAY or ACCEPT the literal or data-item.
```

## Data Movement with DISPLAY and ACCEPT

The data in the Screen Section is referenced by the DISPLAY and ACCEPT verbs. The DISPLAY verb moves contents of data-items in the File, Working-Storage, or Linkage sections to screen storage. The ACCEPT verb moves contents of data-items in screen storage to the associated data-items in the File, Working-Storage, or Linkage sections. An ACCEPT does not perform an implicit DISPLAY; therefore, always execute a DISPLAY statement before you execute an ACCEPT.

Interactive COBOL performs the following steps when executing a DISPLAY sequence.

1. Contents are moved from the File, Working-Storage, or Linkage sections to screen storage:
   a. If a Screen Section data-item has a FROM or USING clause, the contents of the associated data-item are moved to screen storage.
   b. If a Screen Section data-item has a TO clause, underscores are moved to screen storage.

c. If a Screen Section item is a literal, its contents are moved to screen storage.

2. After the moves to screen storage are complete, the entire screen is displayed. Literals and data-items with a FROM clause are displayed at half intensity; Data-items with a USING clause are displayed at full intensity. Underscores are displayed at input fields (TO clause).
3. After a DISPLAY statement is executed, the cursor remains after the last displayed field.

When the ACCEPT statement is executed:

1. The cursor moves to the first input or update field (described in the Screen Section with a TO or USING clause) and remains there until the operator completes the field by:
   a. pressing NEW LINE or CR
   b. filling the field (if you have specified AUTO)
   c. pressing ESC or a function key
2. If the operator presses NEW LINE, CR, a function key, or fills the field (AUTO), the system validates the entered data against the PICTURE of the associated Screen Section data-item:
   a. If the format of the data does not match the PICTURE clause, the system displays an error message; the operator must correct the field before he or she can move to the next input field.
   b. If the data in the input or update field is valid, its contents are moved to screen storage. The underscore acts as a terminator; the underscore and everything to the right of it are ignored. The data is redisplayed.
3. If the operator has pressed NEW LINE or CR, the cursor moves to the next input or update field, and the process is repeated.
4. If the operator has a pressed a function key, the cursor does not visit the remaining input or update fields.
5. If the operator has pressed ESC, the data in the display field is not validated or moved to screen storage. The cursor does not visit the remaining input or update fields.
6. After the operator presses ESC, a function key, or completes the last input or update field, contents of all data-items in the screen referenced by the ACCEPT are moved from screen storage to the corresponding TO or USING items in the File, Working-Storage, or Linkage sections. Note that if a field is terminated by ESC, the data in the display field is not moved to screen storage but the contents in screen storage are still moved to the data-item in the File, Working-Storage, or Linkage sections.
7. If the operator has pressed ESC or a function key, ON ESCAPE *imper-stmt,* if present, is executed.

# Example of Interactive Screen Management

This example shows the Working-Storage and Screen sections of a program that displays a personnel record. Figure 3-1 shows the screen as it might appear on the terminal. The discussion following Figure 3-1 describes the action of the screen and the steps taken by the operator.

```
WORKING-STORAGE SECTION.

01  PERSONNEL-REC.
    02  EMP-NAME       PIC X(30).
    02  ADDRESS.
        03 STREET      PIC X(30).
        03 APT-NO      PIC X(5).
        03 CITY        PIC X(24)  VALUE SPACES.
        03 STATE       PIC X(10).
        03 ZIP         PIC 9(5)   VALUE ZEROS.
    02  PHONES.
        03 HOME-PH     PIC X(12).
        03 BUS-PH      PIC X(12).
    02  DEPARTMENT     PIC X(30).
    02  SUPER          PIC X(30).
    02  BADGE          PIC 9(10).

77  DEFAULT-STATE      PIC X(10)  VALUE "N.C.".

SCREEN SECTION.

01  PERSONNEL-SCREEN.
    02  BLANK SCREEN LINE 1 COLUMN 25 VALUE "PERSONNEL RECORD".
    02  LINE 3   VALUE "NAME: ".
    02            PIC X(30) TO EMP-NAME REQUIRED.
    02  LINE 5   VALUE "ADDRESS".
    02  LINE 7   COLUMN 5  VALUE "STREET: ".
    02            PIC X(30) TO STREET.
    02            COLUMN PLUS 2  VALUE "APT NO.: ".
    02            PIC X(5) TO APT-NO.
    02  LINE 9   COLUMN 5 VALUE "CITY: ".
    02            PIC X(24) USING CITY.
    02            COLUMN 35      VALUE "STATE: ".
    02            PIC X(10) FROM DEFAULT-STATE TO STATE.
    02            COLUMN PLUS 2  VALUE "ZIP: ".
    02            PIC 9(5) USING ZIP.
    02  LINE 11  COLUMN 5 VALUE "HOME PHONE: ".
    02            PIC X(12) TO HOME-PH.
    02            COLUMN 31 VALUE "BUSINESS PHONE: ".
    02            PIC X(12) TO BUS-PH.
    02  LINE 13  VALUE "DEPARTMENT: ".
    02            PIC X(30) TO DEPARTMENT REQUIRED.
    02  LINE 15  VALUE "SUPERVISOR: ".
    02            PIC X(30) TO SUPER.
    02  LINE 17  VALUE "BADGE NUMBER: ".
    02            PIC 9(10) TO BADGE REQUIRED FULL.
```

The execution of the statements DISPLAY PERSONNEL-SCREEN and ACCEPT PERSONNEL-SCREEN has the following effects:

1. The program blanks the display and displays the literal entries, the current contents of USING and FROM data-items, and underscores, which define the input entries. All characters, except the CITY and ZIP fields, are displayed at half intensity.

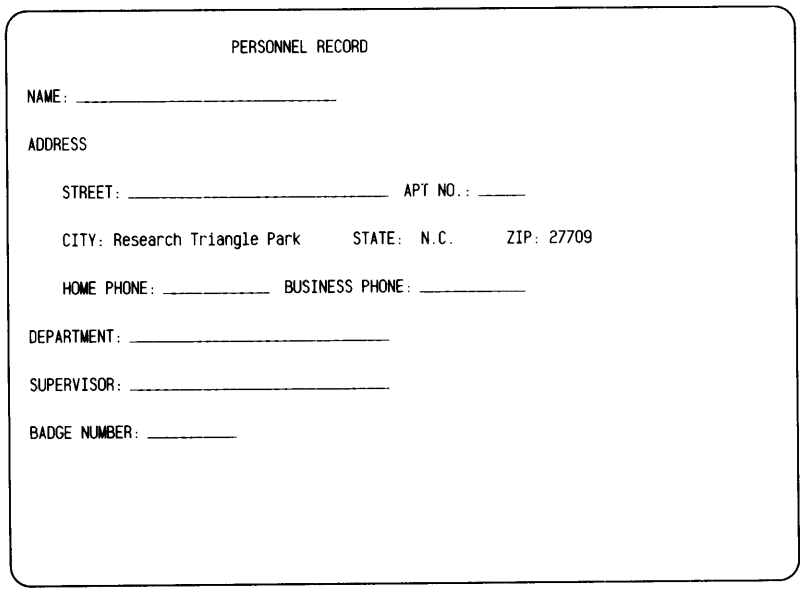



***Figure 3-1** Sample Screen Definition*

2. The cursor moves to the first position in the field following EMP-NAME. Because this field is described as input (a TO clause is in the PICTURE description), it is delimited by underscores. As data is entered, the characters are displayed at full intensity. Since EMP-NAME includes the REQUIRED clause, you must type at least one character before terminating the entry. When you press a terminator (except ESC), the data is verified against the PICTURE clause and redisplayed if valid. If you have entered invalid data, the cursor moves to the first incorrect character. When the data is correct, press a terminator to move to the next field.
3. The fields following STREET and APT NO. may receive data or remain blank.
4. The fields following CITY and ZIP are update fields (defined with a USING clause in the screen description) and, as such, retain their current contents. In Figure 3-1, the contents of CITY and ZIP are displayed. Pressing a terminator reassigns the displayed contents to screen storage and moves the cursor to the next field. Displayed data can be modified by typing in new data.
5. The field following STATE is described in the Screen Section with both a FROM and a TO clause. This means that N.C., the contents of DEFAULT-STATE, is reassigned to the field each time a DISPLAY is executed. As with CITY and ZIP, pressing a terminator assigns the current contents of DEFAULT-STATE to screen storage. Change the displayed data by typing in new data.
6. The fields following HOME PHONE and BUSINESS PHONE may receive data or remain blank.
7. You must enter at least one character for DEPARTMENT. After you enter the data and press a terminator, the field is redisplayed.
8. The field following SUPERVISOR may receive data or remain blank.
9. The field following BADGE NUMBER must be filled. If you enter too few characters, an error message is displayed, and the cursor moves to the end of the entered data and waits for it to be filled.
10. When the field is full and you press a terminator, contents of all data-items in the Screen Section referenced by the ACCEPT statement are moved to the corresponding TO or USING items in the File, Working Storage, and Linkage sections.
11. Execution of the ACCEPT statement is complete. Control passes to the next executable statement in the program.

# Chapter 4
# COBOL Source Entry

You may use any text editor for writing COBOL source text. ICEDIT, which is provided with Interactive COBOL, automatically maintains source files in card format with indexed organization. ICEDIT can optionally create sequentially organized source files in card and CRT format. The /C switch in the compiler command line indicates that the source file is in card format, and the local /I switch indicates an indexed source file (see appendix A).

## CRT Format

In CRT format, the only restrictions imposed are the contents of areas A and B and the use of indicator characters. The compiler reads characters until it finds a line terminator or until it has read 132 characters, whichever comes first.

A line of Interactive COBOL source code in CRT format has the following layout:

| Indicator area (optional) | Area A | Area B |
|---|---|---|

**Indicator (optional)** Column 1. Only an indicator character (*, –, or /) is permitted in this optional 1-character field.

If the line has an indicator character (–, /, or *), enter it in column 1. Area A then begins in column 2, and area B begins in column 6. If you do not include an indicator character, area A begins in column 1 and area B begins in column 5.

**Area A** Columns 2-5 with an indicator character, and columns 1-4 without. The following COBOL elements must begin in area A: division headers, section-names, paragraph-names, FD entries, data descriptions at the 01 and 77 levels, and the phrases DECLARATIVES and END DECLARATIVES.

A tab in area A terminates the area, with area B beginning in the next column.

**Area B** Columns 6-132 with an indicator, and columns 5-132 without. Continuation lines, sentences, and statements must begin in area B. A tab in area B is equivalent to a space.

Comment lines and level numbers 02-49 and 88 may begin in area A or B. Figure 4-1 gives an example of COBOL source code in CRT format.

```
SCREEN SECTION.

    COPY "SCREEN10".

01  MASTER-MENU.
    02  LINE 2 BLANK SCREEN COLUMN 20 VALUE "THE FUNCTIONS       "
-       " AVAILABLE ARE: ".
    02  LINE PLUS 2 COLUMN 20 VALUE "1) ADD RECORD".
    02  LINE PLUS 1 COLUMN 20 VALUE "2) CHANGE RECORD".
    02  LINE PLUS 1 COLUMN 20 VALUE "3) EXAMINE RECORD".
    02  LINE PLUS 1 COLUMN 20 VALUE "4) DELETE RECORD".
    02  LINE PLUS 2 COLUMN 10 BELL VALUE "FUNCTION? ".
    02  COLUMN PLUS 1 PIC 9 TO CHOICE.

    COPY "SCREEN15".
```

***Figure 4-1** COBOL Source Code in CRT Format*

# Card Format

A line of Interactive COBOL source code in card format has the following layout:

| Sequence number area | Indicator area | Area A | Area B | Comment area |
|---|---|---|---|---|

**Sequence number area** — A six-character field (columns 1-6) that typically contains a line number. This field must be present in all source lines in the main program file and in all COPY files.

After checking for the presence of this field, the compiler does not check it further. Therefore, it may contain letters, nonsequential numbers, tabs, spaces, and so forth.

If you do not want numbers, letters, or spaces, in columns 1-6, enter a tab. A tab in the line number area terminates the area.

**Indicator area** — Column 7. If the sequence number area contains a tab, the system inspects the next character. If it is an indicator character (/, *, -), column 7 is considered an indicator area. If any other character is in column 7, it is considered the beginning of area A.

If the sequence number area does not contain a tab, column 7 must contain a space or an indicator character. Area A begins in column 8.

**Area A** — Columns 8-11 (with indicator), or 7-10 (without indicator). The following COBOL elements must begin in area A: division headers, section-names, paragraph-names, FD entries, data descriptions at the 01 and 77 levels, and the key phrases DECLARATIVES and END DECLARATIVES. A tab in area A terminates the area, with area B beginning in the next column.

**Area B** — Columns 13-72. Continuation lines, sentences, and statements must begin in area B. A tab in area B is equivalent to a space.

**Comment Area** — Columns 73-80. Characters in the comment area are placed in the compilation listing but are not further processed by the compiler.

Comment lines and level numbers 02-49 and 88 may begin in area A or B. Figure 4-2 gives an example of COBOL source code in card format.

```
001000 SCREEN SECTION.
001010
001020     COPY "SCREEN10".
001030
001040 01  MASTER-MENU.
001050     02 LINE 2 BLANK SCREEN COLUMN 20 VALUE "THE    "
001060-        " FUNCTIONS AVAILABLE ARE: ".
001070     02 LINE PLUS 2 COLUMN 20 VALUE "1) ADD RECORD".
001080     02 LINE PLUS 1 COLUMN 20 VALUE "2) CHANGE RECORD".
001090     02 LINE PLUS 1 COLUMN 20 VALUE "3) EXAMINE RECORD".
002000     02 LINE PLUS 1 COLUMN 20 VALUE "4) DELETE RECORD".
002010     02 LINE PLUS 2 COLUMN 10 BELL VALUE "FUNCTION? ".
002020     02 COLUMN PLUS 1 PIC 9 TO CHOICE.
002030
002040     COPY "SCREEN15".
```

***Figure 4-2*** *COBOL Source Code in Card Format*

# Continuation Lines

The continuation indicator (-) breaks a word or literal between successive lines. When the continuation indicator is present, the compiler ignores the terminator of the previous line and interprets the text on the current line as a continuation. The following rules apply to continued lines:

- The text on the continued line must begin in area B.
- When a COBOL word is broken between two lines, the first nonblank character of the continuation line follows the last nonterminator character of the previous line. For example:

  line 1: <TAB>DIVIDE GROSS-SALES BY TOT-NUM-SALES GIV<NL>
  line 2: -<TAB>ING AVERAGE-SALES; ON SIZE ERROR PERFORM

  The compiler interprets the continued word as GIVING.
- When a continued statement, phrase, or entry is broken between two words, a continuation indicator is not necessary. If a hyphen is not present in the indicator area of a line, the compiler assumes that the last character on the previous line is followed by a space. For example:

  line 1: 004210 DIVIDE GROSS-AMT BY TOTAL-SALES<NL>
  line 2: 004220      GIVING AVERAGE-SALES.

  The compiler interprets the continued line as:

  ```
  DIVIDE GROSS-AMT BY TOTAL-SALES GIVING AVERAGE-SALES.
  ```

- When a nonnumeric literal is broken between two lines, the continued line must begin with a quotation mark ("). In this case, the space following the word REPORT appears between the words REPORT and OF when the literal is output:

  line 1:  DISPLAY  "THIS IS THE ANNUAL REPORT <NL>
  line 2: -      "OF THE CENTRAL HARDWARE CORP.".

  When the statement is executed, the runtime system displays:

  ```
  THIS IS THE ANNUAL REPORT OF THE CENTRAL HARDWARE CORP.
  ```

# Comment Lines

You may enter comments anywhere in the source code, and a program may contain any number of comment lines. Indicate a comment line by typing an asterisk (*) or slash (/) in the indicator area. When the asterisk is used, the compiler advances one line before writing to the output listing; when the slash is used, the compiler advances to a new page. The text typed on a comment line is produced in the listing but has no effect on compilation or execution.

# Blank Lines

Blank lines may be inserted in the source code at any point, except immediately after a line with a continuation indicator.

# Chapter 5
# The Identification Division

The Identification Division must be the first division of a source program. It consists of the PROGRAM-ID paragraph, which is required, and several optional paragraphs that should be included for documentation purposes.

The general format of the Identification Division is as follows:

```
IDENTIFICATION DIVISION.

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

The PROGRAM-ID paragraph must be the first paragraph in the Identification Division. The program-name is required and may be any valid COBOL data-name, but it is used only for documentation. It is not the name by which the program is known to the system. The external file-name of the file containing the source code is the name used to compile and execute the program.

The optional paragraphs, if used, must be in the order shown.

A comment entry may be one or more lines long and can appear in area A or area B. The DATE-COMPILED paragraph does not affect compilation. The system date is placed in the compilation listing on the line following the DATE-COMPILED entry.

The following is an example of the Identification Division.

```
IDENTIFICATION DIVISION.

PROGRAM-ID.         INVENTORY-01.
AUTHOR.             HEADQUARTERS PROGRAMMING GROUP.
INSTALLATION.       CENTRAL HARDWARE DISTRIBUTORS INC.
DATE-WRITTEN.       JANUARY, 1984.
DATE-COMPILED.      JANUARY, 1984.
SECURITY.           NOT APPLICABLE.
```

# Chapter 6
# The Environment Division

The Environment Division provides a standard way of expressing program values that are dependent on the physical characteristics of a specific computer. It must follow the Identification Division and consists of two sections: the Configuration Section and the Input-Output Section.

The general format of the Environment Division is as follows:

```
[ENVIRONMENT DIVISION.

[CONFIGURATION SECTION.

[SOURCE-COMPUTER. computer-name.]
[OBJECT-COMPUTER. computer-name.]

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

        [; PROGRAM COLLATING SEQUENCE IS alphabet-name].
[SPECIAL-NAMES. [SWITCH literal
      [; IS mnemonic-name]
      [; ON STATUS IS condition-name]
      [; OFF STATUS IS condition-name]]...
      [; alphabet-name IS { STANDARD } ]
                          { NATIVE   }
      [; CURRENCY SIGN IS lit]
      [; DECIMAL-POINT IS COMMA].]]

[INPUT-OUTPUT SECTION.

FILE-CONTROL. file-control-entry...
[I-O-CONTROL. i-o-control-entry...].]]
```

The following is an example of the Environment Division.

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. CS-SERIES-200.
OBJECT-COMPUTER. CS-SERIES-200.
SPECIAL-NAMES.
    SWITCH "A", ON STATUS IS SW0, OFF STATUS IS SW1
    SWITCH "B", OFF STATUS IS SW2, ON STATUS IS SW3
    CURRENCY SIGN IS "M"
    DECIMAL-POINT IS COMMA.
INPUT-OUTPUT SECTION.
```

```
FILE-CONTROL.
    SELECT AFILE ASSIGN TO DISK.
    SELECT BFILE ASSIGN TO DISK.
I-O-CONTROL.
    SAME RECORD AREA FOR AFILE, BFILE.
```

# Configuration Section

The Configuration Section names the computer(s) that compile and execute the program. It also defines switches and other special conditions.

SOURCE-COMPUTER names the computer used to compile the program, and OBJECT-COMPUTER names the computer on which the program is run. Both paragraphs are for documentation purposes only. The computer name may be any valid COBOL data-name.

MEMORY SIZE may be any integer value. This clause and the PROGRAM COLLATING SEQUENCE clause are used for documentation purposes only.

In the SPECIAL-NAMES paragraph, the SWITCH literal must be an uppercase letter from A to Z. This literal names a logical switch that may be tested for an ON or OFF condition under program control. The ON and OFF clauses may appear in either order. At least one of the clauses must be present when a switch is named. A switch value is set at runtime. For example, if NEWPROG declares switches A and B, these switches are set to ON by executing the program using the local A and B switches (that is, NEWPROG/A/B). The value of a switch remains unchanged while the program is running. Note that the condition-name describing the ON or OFF status need not be defined elsewhere; no Working-Storage entry is needed. The SWITCH *literal* IS *mnemonic-name* clause is for documentation purposes only.

The *alphabet-name* IS clause is for documentation purposes only.

The literal in the CURRENCY SIGN clause represents the currency symbol in the PICTURE clause. The literal is a single character, but it *cannot* be one of the following:

Digits 0-9
Letters A B C D L P R S V X Z space
Special characters * + - ( ) / = , ;

If you omit this clause, the currency sign is the dollar sign ($).

The DECIMAL-POINT IS COMMA clause exchanges the functions of the period and comma in numeric literals and in the PICTURE clause character string. All occurrences of commas in numeric literals are changed when this clause is used. If subsequent source code contains an index or subscript entry—for example, (4, 2)—the comma in this literal is changed to a period, resulting in errors.

# Input-Output Section

The Input-Output Section consists of two paragraphs that supply information needed to control the transmission and manipulation of data between the object program and external devices. These two paragraphs are called FILE-CONTROL and I-O-CONTROL.

# The FILE-CONTROL Paragraph

The entries in the FILE-CONTROL paragraph specify the name of each file to be used by the program, associate the file with a particular hardware device, and designate other file characteristics.

## The SELECT Clause

The SELECT clause names internal program files and associates each one with a given hardware device and external file-name. It may be used to specify logical file organization, access method, a File-Status item, keys, and the size of the file.

The SELECT formats for sequential, relative, and indexed file organizations are as follows:

**Sequential SELECT**

```
SELECT file-name ASSIGN TO { PRINTER   } [; id-lit]
                           { PRINTER-1 }
                           { DISPLAY   }
                           { KEYBOARD  }
                           { DISK      }

[; ORGANIZATION IS SEQUENTIAL]
[; ACCESS MODE IS SEQUENTIAL]
[; FILE STATUS IS data-name]
[; DATA SIZE IS integer].
```

**Relative SELECT**

```
SELECT file-name ASSIGN TO DISK [; id-lit]

  ; ORGANIZATION IS RELATIVE

[ ; ACCESS MODE IS { { RANDOM  } ; RELATIVE KEY IS data-name                  } ]
                   { { DYNAMIC }                                              }
                   { SEQUENTIAL  [; RELATIVE KEY IS data-name]                }

[; FILE STATUS IS data-name]
[; INDEX SIZE IS integer]
[; DATA SIZE IS integer].
```

**Indexed SELECT**

```
SELECT file-name ASSIGN TO DISK [; id-lit]

  ; ORGANIZATION IS INDEXED

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

  ; RECORD KEY IS data-name
[; ALTERNATE RECORD KEY IS data-name [WITH DUPLICATES] ] ...
[; FILE STATUS IS data-name]
[; INDEX SIZE IS integer]
[; DATA SIZE IS integer].
```

The following is an example of a SELECT clause for a single-keyed indexed file:

```
SELECT CFILE ASSIGN TO DISK;
    ORGANIZATION IS INDEXED;
    ACCESS MODE IS DYNAMIC;
    RECORD KEY IS C-KEY;
    FILE STATUS IS CFSTAT.
```

A sequential file may be assigned to PRINTER, PRINTER-1, DISPLAY, KEYBOARD, or DISK. The records can only be accessed sequentially, or in the order that they were written.

Relative and indexed files can be assigned only to disk. If the access mode is not specified, sequential is assumed. In relative and indexed files, sequential access is in ascending order of the RELATIVE KEY or RECORD KEY specified. In random access, the value of the RELATIVE KEY or RECORD KEY determines the record to be accessed. If the dynamic access mode is specified, access can be sequential or random.

Table 6-1 specifies the access mode and assignment statement of the various file types. The access mode and assignment statements are the same for fixed and variable sequential files; the RECORDING MODE IS clause in the Data Division specifies FIXED or VARIABLE (see "FD Entry" in chapter 7). A line sequential file assigned to PRINTER or DISPLAY must be OPENed for OUTPUT or EXTEND; one assigned to KEYBOARD must be OPENed for INPUT.

| | Access Mode Is | | | Assign to | | | |
|---|---|---|---|---|---|---|---|
| **File Type** | **Sequential** | **Random** | **Dynamic** | **Printer** | **Display** | **Keyboard** | **Disk** |
| Fixed Seq. | yes | no | no | no | no | no | yes |
| Variable Seq. | yes | no | no | no | no | no | yes |
| Line Seq. | yes | no | no | yes | yes | yes | no |
| Relative | yes | yes | yes | no | no | no | yes |
| Indexed | yes | yes | yes | no | no | no | yes |

**Table 6-1 Interactive COBOL File Handling**

Assign a line sequential file to DISPLAY to bypass Interactive COBOL's Screen Section and send output directly to the display terminal screen. To read a line sequential disk file, assign to KEYBOARD. On RDOS and DG/RDOS, the READ statement for files assigned to KEYBOARD is terminated by CR (octal 015), form feed (octal 014), or null (octal 000). On AOS and AOS/VS, NEW LINE (octal 012) is also a terminator.

To create a line sequential file on disk, assign to DISPLAY or PRINTER, and specify an external file-name. For example:

```
SELECT CUST-RPT ASSIGN TO PRINTER, "CUST$RPT".
```

The id-lit following the file device type in the SELECT clauses is an Interactive COBOL extension. It allows a program to associate an internal file-name with an external file-name. (For details, see "External File-Names" in chapter 1 of the *Interactive COBOL User's Guide*.)

The ORGANIZATION clause specifies the logical structure of a file. The file organization is established when a file is created and can be changed only by the REORG utility. Interactive COBOL provides three types of file organizations: sequential, indexed, and relative.

The ACCESS MODE clause designates the method to be used in obtaining records from a file or placing records into a file. The three access modes are sequential, random, and dynamic.

The data-item named in the RECORD KEY clause is the primary record key for the file. This key provides an access path to records in an indexed file. The primary record key values must be unique among records of the file.

Each data-item name in the ALTERNATE RECORD KEY clause is an alternate record key for the file. An alternate key provides an additional access path to records in an indexed file. Up to four ALTERNATE RECORD KEY clauses can be specified.

The RECORD KEY and ALTERNATE RECORD KEY data-names must be defined as alphanumeric items within a record description entry associated with the file-name. Each record key can be a maximum of 100 characters long. Record key data-names can be qualified.

When a data description refers to an existing file, the descriptions of the RECORD KEY and the ALTERNATE RECORD KEY data-names, as well as their relative locations within the data record, must be the same as when the file was created.

The DUPLICATES phrase specifies that the value of the associated alternate record key may be duplicated within any of the records in the file. This phrase is used only for documentation; Interactive COBOL allows duplicates even if the DUPLICATES phrase is not specified. However, it is good programming practice to document the presence of duplicate keys by specifying the phrase.

The RELATIVE KEY phrase specifies a key whose contents identify a logical record in a relative file. Relative keys must be defined as PIC 9(4) to 9(18) COMPUTATIONAL.

The FILE STATUS data-item holds a value that indicates the outcome of an I/O operation. It must be described as a two-character alphanumeric item. See appendix A of the appropriate *User's Guide* for a table of File Status codes.

The INDEX SIZE and DATA SIZE clauses are Interactive COBOL extensions used on RDOS and DG/RDOS for contiguous files. On AOS and AOS/VS, files are dynamically allocated as they grow; the user does not need to preallocate contiguous file space. These clauses are compiled but are ignored by the AOS and AOS/VS runtime systems.

The INDEX SIZE clause specifies the number of 512-byte blocks of contiguous disk storage space to be reserved for the index portion of an indexed or relative file when the file is created.

The DATA SIZE clause specifies the number of 512-byte blocks of contiguous disk storage space to be reserved for the data portion of a sequential, indexed, or relative file when the file is created.

## The I-O-CONTROL Paragraph

The format for an I-O-CONTROL entry is:

SAME [RECORD] AREA FOR file-name, {file-name}...

The following is an example:

```
I-O-CONTROL.
    SAME RECORD AREA FOR GOOD-CUST-FILE, CREDIT-RISK-FILE
    SAME RECORD AREA FOR PARTS-1, PARTS-2, PARTS-3.
```

The SAME AREA and SAME RECORD AREA clauses are treated identically; in both cases, the specified files share a common record area. More than one of the files may be opened at any given time. Since the two clauses are treated in the same manner, it is illegal for a file-name to appear in more than one of the clauses.

# Chapter 7
# The Data Division

The Data Division describes the data that the object program creates, manipulates, uses as input, or produces as output. It must follow the Environment Division in the source program.

The Data Division consists of four sections: the File Section, the Working-Storage Section, the Linkage Section, and the Screen Section, which is an Interactive COBOL feature. At least one of these sections must be present in the Data Division.

The general format of the Data Division is as follows:

```
[DATA DIVISION.

[FILE SECTION. [FD-entry
    [record-description-entry]...]...]

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

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

[SCREEN SECTION.
    [data-description-entry]...]]
```

The following example shows the Data Division of an Interactive COBOL program. Table 7-1 lists the clauses that are legal in the Data Division sections.

```
DATA DIVISION.


FILE SECTION.
FD  F-CUSTOMER LABEL RECORD IS OMITTED
        DATA RECORD IS F-CUST-REC.
01  F-CUST-REC.
    03 FAREA PIC 99.
    ...
    03 FADDRESS.
       05 FSTREET PIC X(20).
    ...
WORKING-STORAGE SECTION.
01  FUNCTION PIC 99 VALUE 00.
77  WORDAMT PIC 9(5)V99 VALUE ZERO.
01  SECURITY-CODE.
    03 W-CATEGORY PIC X VALUE " ".
    ...
```

```
SCREEN SECTION.
01  ACCEPT-SECURITY.
    03 LINE 1 BLANK SCREEN.
    03 LINE PLUS 1 COLUMN 10 VALUE "PLEASE ENTER AUTHORIZATION".
    03 S-CATEGORY COLUMN 50 PIC X TO W-CATEGORY SECURE AUTO.
```

# The File Section

The File Section consists of FD entries. These entries contain record descriptions that Interactive COBOL uses to transfer data to and from files.

## FD Entry

The FD entry describes the physical attributes of a file. It continues the file declaration that the SELECT clause began. You must specify one FD entry for each file declared in a SELECT clause. You must follow each FD file entry with one or more 01-level record description entries. The general format for the FD entry is:

```
FD file-name
[; BLOCK CONTAINS integer  {RECORDS    }  ]
                           {CHARACTERS }
[; RECORD CONTAINS integer [TO integer] CHARACTERS]
[; RECORDING MODE IS  {VARIABLE}  ]
                      {FIXED   }
; LABEL  {RECORD IS  }   {STANDARD}
        {RECORDS ARE}   {OMITTED }
[; DATA  {RECORD IS  }  data-name ...]
         {RECORDS ARE}
[; CODE-SET IS alphabet-name].
```

The BLOCK CONTAINS clause is used for documentation purposes only.

The RECORD CONTAINS clause defines the record size or the record size limits of the file. The maximum record length for files assigned to PRINTER, DISPLAY, or KEYBOARD is 132 bytes. The maximum record length for indexed or relative files is 4096 bytes. The only length restriction on a sequential file assigned to DISK is that the resulting program file must not be too large to run under the runtime system.

The RECORDING MODE clause, which is an extension to ANSII COBOL, is permitted only for fixed and variable sequential files. It states whether the file contains fixed or variable length records. If this clause is omitted, fixed length is assumed.

The LABEL clause is required, even though it is used for documentation purposes only.

The DATA RECORD and CODE-SET clauses are used for documentation purposes only.

| Clause or Entry | File | W-S | Linkage | Screen |
|---|---|---|---|---|
| FD entry | Y | N | N | N |
| BLOCK CONTAINS | Y | N | N | N |
| RECORD CONTAINS | Y | N | N | N |
| RECORDING MODE | Y | N | N | N |
| LABEL RECORDS | Y | N | N | N |
| DATA RECORDS | Y | N | N | N |
| CODE-SET | Y | N | N | N |
| FILLER | Y | Y | Y | N |
| REDEFINES | Y | Y | Y | N |
| PICTURE | Y | Y | Y | Y |
| USAGE | Y | Y | Y | N |
| SIGN | Y | Y | Y | N |
| OCCURS | Y | Y | Y | N |
| SYNCHRONIZED | Y | Y | Y | N |
| JUSTIFIED | Y | Y | Y | Y |
| BLANK WHEN ZERO | Y | Y | Y | Y |
| VALUE | N | Y | N | Y |
| Level 77 | N | Y | Y | N |
| Level 88 | Y | Y | Y | N |
| AUTO | N | N | N | Y |
| SECURE | N | N | N | Y |
| REQUIRED | N | N | N | Y |
| FULL | N | N | N | Y |
| FROM | N | N | N | Y |
| TO | N | N | N | Y |
| USING | N | N | N | Y |
| BELL | N | N | N | Y |
| BLINK | N | N | N | Y |
| BLANK SCREEN | N | N | N | Y |
| LINE | N | N | N | Y |
| COLUMN | N | N | N | Y |
| BLANK LINE | N | N | N | Y |

**Table 7-1 Legal Entries in the Four Data Division Sections**

# Working-Storage Section

The Working-Storage Section must follow the File Section. It describes records and elementary data-items that are not part of external files, but are developed and processed internally. These elements function as temporary data storage items during program execution.

# Linkage Section

The Linkage Section is found only in a program that is called by another program. The data descriptions in the Linkage Section enable a called program to reference data from the calling program.

Data-items to be passed must be defined in either the File or Working-Storage sections of the calling program. Corresponding data-items that receive the passed information must be defined in the Linkage Section of the called program.

The Linkage Section must immediately follow the Working-Storage Section. Its structure is the same as the Working-Storage Section.

Linkage Section data descriptions may use the same descriptor clauses as appear in the Working-Storage Section with one exception: the VALUE clause may not appear in a Linkage Section data entry.

Each sending item defined in a calling program and each receiving item defined in the Linkage Section of a called program must have a level number of 01, 77, or 88.

Corresponding sending and receiving data-items must have the same total length, but need not have the same identifier or the same internal structure. A list of sending data-items must appear in a CALL USING or CALL PROGRAM USING statement of the calling program (see CALL and CALL PROGRAM in chapter 8). A list of the corresponding receiving items must appear, in the same order, in the Procedure Division Using header of the called program.

An identifier may be used more than once in a CALL PROGRAM USING statement. An identifier may not be used more than once in a Procedure Division Using header.

You may define any number of items in the Linkage Section. However, only the items named in the receiving list, and items subordinate to them, may be referenced by the called program.

*Note:* Up to 12 KB of disk space per terminal is allocated for the Linkage Section for Interactive COBOL on RDOS or DG/RDOS. On AOS or AOS/VS, the Linkage Section has no size restriction other than the limits of program size.

The following example shows the relevant entries in a calling program and in the called program.

In the calling program:

```
DATA DIVISION.
...
WORKING-STORAGE SECTION.
...
01  SENDING-RECORD  PIC X(25).
01  PHONE-NUMBER.
    03 AREA-CODE    PIC 999.
    03 EXCHANGE     PIC 999.
    03 NUMBER       PIC 9(4).
77  TODAYS-DATE     PIC 9(6).
```

In the called program:

```
DATA DIVISION.
...
LINKAGE SECTION.
77  RECEIVING-RECORD  PIC A(25).
01  PHONE-NUMBER.
    03 AREA-CODE PIC 999.
    03 EXCHANGE PIC 999.
    03 NUMBER PIC 9(4).
01  TODAYS-DATE.
    03 MONTH PIC 99.
    03 DAY PIC 99.
    03 YEAR PIC 99.
```

When the calling program executes the following statement, the values of the listed data-items are preserved:

```
CALL PROGRAM "GET" USING SENDING-RECORD,
    PHONE-NUMBER, TODAYS-DATE.
```

Then, when the called program (GET) begins execution, the procedure header

```
PROCEDURE DIVISION USING RECEIVING-RECORD,
    PHONE-NUMBER, TODAYS-DATE.
```

causes the listed items to be initialized to the values of the preserved items from the calling program.

# Screen Section

The Screen Section defines the attributes of screens to be used in interactive screen I/O. The Screen Section data descriptions are generally in the same form as in the other sections of the Data Division. However, additional screen clauses allow you to specify the position of items on the console screen and other console controlling functions. A Screen Section entry is referenced in the Procedure Division with the DISPLAY and ACCEPT verbs.

## Screen Data Description Entry

A Screen Section may describe a literal or a data-item entry. A literal entry displays constant information such as a title or a menu; a data-item entry is considered a "window" through which data is transferred from the screen to program storage, from program storage to the screen, or in both directions. Each data-item entry must contain a PICTURE clause to describe it, and a FROM, TO or USING clause to identify an associated File, Working-Storage, or Linkage Section item.

You may combine literal and data-item entries to form a group entry. A group entry is defined with level numbers in a format similar to group items in the File and Working-Storage sections. The syntax for describing screen entries is as follows:

### Literal Format

```
level-number [screen-name]
    [; BLANK SCREEN]
    [; BLANK LINE]
    [; BELL]
    [; BLINK]
    [; LINE NUMBER IS { integer-1 | PLUS integer-2 | identifier-1 }]
    [; { COLUMN | COL } NUMBER IS { integer-1 | PLUS integer-2 | identifier-1 }]
    [; VALUE IS literal].
```

### Data-Item Format

```
level-number [screen-name]
    [; BLANK SCREEN]
    [; BLANK LINE]
    [; BELL]
    [; BLINK]
    [; LINE NUMBER IS {integer-1 | PLUS integer-2 | identifier-1}]
    [; {COLUMN | COL} NUMBER IS {integer-1 | PLUS integer-2 | identifier-1}]
    ; {PICTURE | PIC} IS picture-string {; FROM id-lit | ; TO id | ; USING id}
    [; BLANK WHEN ZERO]
    ; {JUSTIFIED | JUST} RIGHT
    [; AUTO]
    [; SECURE]
    [; REQUIRED]
    [; FULL].
```

### Group Format

```
level-number screen-name
    [; AUTO]
    [; SECURE]
    [; REQUIRED]
    [; FULL].
    {data-item or literal entry}...
```

For examples of complete screen descriptions, see the sample program in chapter 3.

## Level Number

The data description for a Screen Section entry must begin with a level number. Level 01 entries must begin in Area A; other levels may be indented as desired. Level numbers in the Screen Section may range from 01 to 49.

## Screen-Name

The screen-name must immediately follow the level number. Screen-names must conform to the standard rules for programmer-created words. Omitting the screen-name from an entry is equivalent to a FILLER entry. Unnamed items in a group entry may not be referenced separately. Screen-names are required at the 01 level.

## Screen Clauses

More than one clause can be specified for each entry; they can be in any order. Regardless of their order, however, the clauses are executed in the following order:

BLANK SCREEN
LINE and COLUMN positioning
BLANK LINE
BELL
DISPLAY or ACCEPT the literal or data-item

The clauses are summarized in Table 7-2 (see chapter 3 for a more detailed discussion).

| Clause | Meaning |
|---|---|
| AUTO | Causes automatic termination when a screen data-item is full. Used with input or update fields. |
| BELL | Sounds the console's tone. |
| BLANK SCREEN | Erases the screen and positions the cursor to line 1, column 1. |
| BLANK LINE | Erases a line from the current cursor position to the end of the line and leaves the cursor position unchanged. |
| BLANK WHEN ZERO | Functions in the same manner as in other sections of the Data Division. |
| BLINK | Causes a field to blink. |
| COLUMN | Specifies horizontal positioning. |
| FROM | Identifies an output field. |
| FULL | A character or space must be entered in each position. Used with input or update fields. |
| JUSTIFIED | Functions in the same manner as in other sections of the Data Division. |
| LINE<br>PICTURE | Specifies vertical positioning. Describes the type, size, and appearance of a data-item entry and controls the way in which data is accepted or displayed. |
| REQUIRED | At least one character must be entered. Used with input or update fields. |
| SECURE | Causes asterisks to be echoed on the display during data entry. Used with input or update fields. |
| TO | Identifies an input field. |
| USING | Identifies an update field. |
| VALUE | Specifies literal information to be displayed. |

**Table 7-2 Screen Section Clauses**

# Data Description Entries

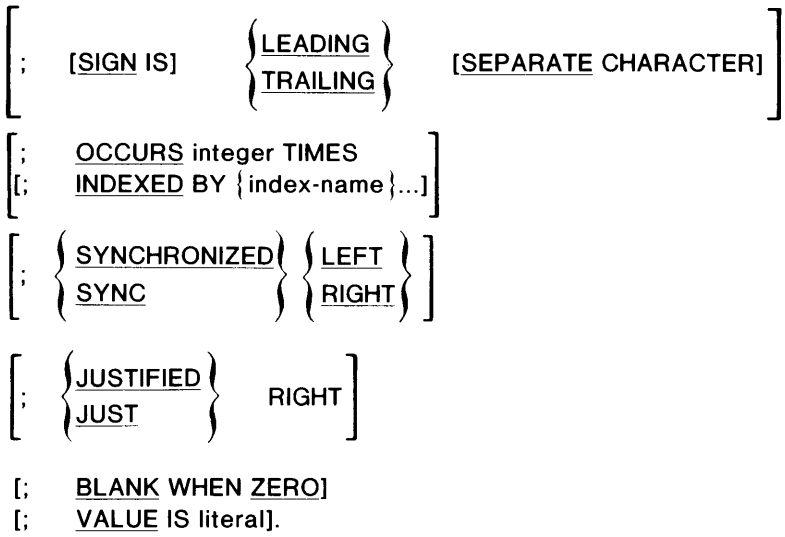
The general format for a Data Description entry is listed below. The clauses, except for the level number, the data-name or FILLER clause, and the REDEFINES clause, may be written in any order.

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

    [;   REDEFINES data-name]

    ;  { PICTURE }
       { PIC     }  IS picture-string

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




## Level Numbers

The level number must always be the first element in an entry. Level numbers may be 01, 02-49, 77, and 88.

### Level 01

The level number 01 identifies the record as a whole. Level 01 can be used in any section in the Data Division and must begin in area A.

### Levels 02-49

Level numbers 02-49 identify the hierarchy of data within the record. They must begin in area B. Multiple 01-level entries subordinated to an FD entry implicitly redefine the same memory area. Data description entries subordinate to an FD entry must have level numbers 01-49 or 88.

### Level 77

Level number 77 identifies elementary data-items and is assigned to certain entries where there is no real concept of level. Level 77 items may not be subdivisions of other items, and they may not be subdivided. Level 77 must begin in area A and can be used only in the Working-Storage and Linkage sections.

### Level 88

Level number 88 is used to define condition-names. It can begin in area A or B and may occur only in the File, Working-Storage and Linkage sections. The condition-name must be followed by a VALUE clause specifying the literal or literals that apply to that condition. The condition-name can then be used in place of a relation condition.

## Data-name/FILLER Clause

A data-name entry specifies the name of the data-item being described. A FILLER entry specifies an elementary item in a logical record that cannot be referred to explicitly. The data-name or FILLER clause must immediately follow the level number. FILLER cannot be used in the Screen Section.

## REDEFINES Clause

The REDEFINES clause allows the program to describe the same memory area in different data description entries. For example:

```
01  TAX-GROUPS.
    03 GROUP-A.
       05 A-ITEM OCCURS 6 TIMES PIC 9999.
    03 GROUP-B REDEFINES GROUP-A.
       05 B-ITEM OCCURS 12 TIMES PIC XX.
```

The REDEFINES clause, when used, must immediately follow the data-name. It cannot be used with level 01 entries in the File Section.

The level numbers of both data-names must be identical, and no entry having a level number numerically lower than this level number may occur between the data description entries for the two names. The following example illustrates an *illegal* description:

```
    05 A PIC 999.
03 B PIC AAA.
    05 C REDEFINES A.
```

The description of the data-name being redefined cannot contain a REDEFINES or OCCURS clause; however, it may be subordinate to an entry that contains either clause.

The data types of the original item and the redefined area do not have to be the same. However, for items not at the record level (i.e., those that have a level number greater than 01), the redefinition must specify the same number of character positions as the original definition. For items at the record level, there is no such constraint.

You may redefine the same character positions more than once. The entries giving the new descriptions of the positions must immediately follow the entries defining the area being redefined, without intervening entries that define new positions. Multiple redefinitions of the same positions must all use the data-name of the entry that originally defined the area. Thus the following structure is legal:

```
05  A                PIC 9999.
05  B REDEFINES A    PIC 9V999.
05  C REDEFINES A    PIC 99V99.
```

However, the following structure is *illegal:*

```
05  A                PIC 9999.
05  B REDEFINES A    PIC 9V999.
05  C REDEFINES B    PIC 99V99.
```

In an FD entry, multiple 01-level entries implicitly redefine the same record area; a REDEFINES clause is not permitted. For example:

```
FD  CUSTOMER-ACCT-FILE ...
01  PARTS-CUST-REC.
    03  CUST-NAME        PIC X(25).
    03  CUST-ADDRESS     PIC X(30).
    ...
01  ASSEMBLY-CUST-REC.
    03  CUST-ADDRESS     PIC X(20).
    03  CUST-NAME        PIC X(35).
```

# PICTURE Clause

The PICTURE clause describes the general characteristics and editing requirements of an elementary data-item. A PICTURE clause can be specified only at the elementary item level, and it must be specified for every elementary data-item, except for those declared as USAGE IS INDEX. A picture-string may contain a maximum of 30 characters.

The format of the PICTURE clause is as follows:


PIC is an abbreviation for PICTURE. A picture-string consists of combinations of characters from the COBOL character set. The picture-string specifies:

- The type of data in the data-item (alphabetic, alphanumeric, numeric, alphanumeric edited, or numeric edited)
- The size of the data-item
- The location of the decimal point (if the data-item is numeric)
- Whether a numeric data-item contains an arithmetic sign
- Any editing to be performed on the data-item.

## Symbols Used in the PICTURE Clause

The function of each PICTURE clause symbol is given in the following list.

A Represents a position that may contain only an uppercase letter or a space.

B Represents a position where a space is to be inserted.

P Indicates an assumed decimal scaling position. P can appear only in a sequence in the rightmost or leftmost portion of the format. The symbol P is not counted in the size of the data-item; it is counted in determining the maximum number of digit positions (18) in numeric or numeric edited items, or items that appear as arithmetic operands.

S Indicates the presence of an operational sign; it may appear only once in the character string and must be the leftmost character in the PICTURE. The S is not counted in determining the length of an elementary item unless you specify the SEPARATE CHARACTER option.

V Indicates the location of an assumed decimal point and may only appear once in a character string. When the assumed decimal point is to the right of the rightmost symbol in the string, the V is redundant. The V is not counted in determining the length of the elementary item.

X Represents a position that may contain any character from the data character set.

Z Represents a leading numeric position that is replaced by a space when the contents of that position are zero.

9 Represents a position that may contain a digit from 0 through 9.

0 Represents a position where a zero is to be inserted.

/ Represents a position where a slash is to be inserted.

, Represents a position where a comma is to be inserted. A comma cannot be the last character in the PICTURE character string.

In an input data-item, represents a decimal point used for alignment purposes. In an output data-item, represents a postion where a period is to be inserted. The period cannot be the last PICTURE character in an output data-item.

*Note:* You may exchange the functions of the period and the comma in a PICTURE character string by specifying the DECIMAL-POINT IS COMMA clause in the SPECIAL-NAMES paragraph.

+, -, CR, DB
These editing sign-control symbols are mutually exclusive in any one character string. The plus and minus symbols may be used on the left or right side of the value; the CR and DB symbols may be used only on the right side.

* Represents a leading numeric position where an asterisk is to be placed when the content of that position is zero.

CS The currency symbol CS represents a position where a currency symbol is to be placed. The currency symbol is either the dollar sign ($) or the single character specified in the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph.

## USAGE Clause

The USAGE clause specifies the way in which a data-item is stored in memory. The USAGE clause can be written at any level. When written at the group level, the USAGE clause applies to all items in the group. If USAGE is specified for any of the items in that group, it must not conflict with the USAGE specified at the group level.

Its format is as follows:

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

DISPLAY indicates that the data is stored as ASCII characters in bytes. If a USAGE clause is not specified for an elementary item, the USAGE is DISPLAY by default.

The PICTURE of a COMPUTATIONAL data-item can contain only 9s, the operational sign symbol S, the implied decimal point symbol V, and one or more Ps. COMP is an abbreviation for COMPUTATIONAL.

When a group is described as COMPUTATIONAL, the elementary items in the group are COMPUTATIONAL. The group item itself is treated as DISPLAY and cannot be used in arithmetic operations.

COMPUTATIONAL items are represented internally in twos-complement binary notation. Storage for COMPUTATIONAL items is more efficient than for DISPLAY items. Table 7-3 lists the bytes required to store COMPUTATIONAL items.

USAGE IS INDEX indicates that the item contains a binary value that is used to reference a particular element of a table.

Items that are INDEX are represented internally as 2-byte unsigned items. The SYNCHRONIZED, JUSTIFIED, PICTURE, VALUE IS, and BLANK WHEN ZERO clauses cannot be used with an item described as USAGE IS INDEX.

| No. of Decimal Digits | | |
|---|---|---|
| **Unsigned** | **Signed** | **Bytes Required** |
| 1-2 | 1-2 | 1 |
| 3-4 | 3-4 | 2 |
| 5-7 | 5-6 | 3 |
| 8-9 | 7-9 | 4 |
| 10-12 | 10-11 | 5 |
| 13-14 | 12-14 | 6 |
| 15-16 | 15-16 | 7 |
| 17-18 | 17-18 | 8 |

**Table 7-3 COMPUTATIONAL Item Storage**

## SIGN Clause

Use the SIGN clause to explicitly state the manner of sign representation in numeric data-items. The format of the SIGN clause is as follows:

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

Specify the SIGN clause only for an elementary numeric data-item with a PICTURE containing an S, or for a group item containing at least one such numeric data-item. A data-item may have only one SIGN clause and must be described, implicitly or explicitly, as USAGE IS DISPLAY.

The terms used in this clause and their meanings are:

| Sign Clause | Sign Type |
|---|---|
| SIGN LEADING | Decimal with leading sign overpunch |
| SIGN TRAILING | Decimal with trailing sign overpunch (default) |
| SIGN LEADING SEPARATE | Decimal with leading separate sign |
| SIGN TRAILING SEPARATE | Decimal with trailing separate sign |

A data-item may be from 1 to 18 characters long. If you include the SEPARATE CHARACTER phrase, the S in the PICTURE string is counted in determining the length of the data-item. If you omit the SEPARATE CHARACTER phrase, the S is not counted.

In decimal data-items with trailing sign overpunch, the sign is in the rightmost digit; with leading sign overpunch, the sign is in the leftmost digit (see Table 7-4).

| Digit | Positive | Negative |
|---|---|---|
| 0 | { <173> | } <175> |
| 1 | A <101> | J <112> |
| 2 | B <102> | K <113> |
| 3 | C <103> | L <114> |
| 4 | D <104> | M <115> |
| 5 | E <105> | N <116> |
| 6 | F <106> | O <117> |
| 7 | G <107> | P <120> |
| 8 | H <110> | Q <121> |
| 9 | I <111> | R <122> |

**Table 7-4 Sign Overpunch Characters**

The following program illustrates how the SIGN clause affects data-item display:

```
PROGRAM-ID. TESTPROG.
...
WORKING-STORAGE SECTION.
77  DATA-1  PIC S999 SIGN IS LEADING SEPARATE
                    VALUE +470.
77  DATA-2  PIC S999 SIGN IS LEADING
                    VALUE +470.
77  DATA-3  PIC S999 SIGN IS TRAILING SEPARATE
                    VALUE +470.
77  DATA-4  PIC S999 SIGN IS TRAILING
                    VALUE +470.

PROCEDURE DIVISION.
DISPLAY-PARA.
    DISPLAY DATA-1.
    DISPLAY DATA-2.
    DISPLAY DATA-3.
    DISPLAY DATA-4.
    STOP RUN.
```

When TESTPROG is executed, the following appears on the screen:

```
+470
D70
470+
47{
```

## OCCURS Clause

The OCCURS clause defines tables and other sets of repeated data-items. Within a table description, the OCCURS clause specifies the number of times that the named item is to be repeated. All data description clauses associated with an item with a description containing an OCCURS clause apply to each occurrence of the item.

The format of an OCCURS clause is:

```
OCCURS integer TIMES
[; INDEXED BY {index-name}...]
```

Do not use the VALUE clause in a data description entry that contains, or is subordinate to, an OCCURS clause. Do not specify the OCCURS clause at the 01 or 77 level number.

The data-name that is the subject of an OCCURS clause must either be subscripted or indexed when referred to in a Procedure Division statement. In addition, if the data-name of the entry is the name of a group item, all data-names belonging to the group must be subscripted or indexed when they are referred to, except as the object of a REDEFINES clause (see above). When the data-name is used as the object of a REDEFINES clause, it represents every occurrence of that name and therefore cannot be subscripted or indexed.

The INDEXED BY phrase is required when the data-name in an OCCURS clause or its subordinate items are to be referenced by indexing. Each index-name must be a unique name within the program. Index-names are not explicitly defined elsewhere. Specifying a name in the INDEXED BY phrase implicitly defines the index-name as a two-byte unsigned binary item.

Use nested OCCURS clauses to define multidimensional tables. An OCCURS clause is nested when an item containing an OCCURS clause is subordinated to another item containing an OCCURS clause. Three levels of nesting are permitted. Thus, a table may have one, two, or three dimensions.

## SYNCHRONIZED Clause

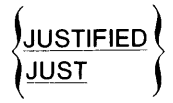
The SYNCHRONIZED clause is ignored and has no effect on compilation or execution of the object program. SYNC is an abbreviation for SYNCHRONIZED.

Its format is:


## JUSTIFIED Clause

The JUSTIFIED clause specifies nonstandard positioning of data within a receiving data-item. JUST is an abbreviation for JUSTIFIED.

The format for the JUSTIFIED clause is:




This clause can be specified only at the elementary item level. It cannot be specified for a numeric item or for an item whose PICTURE specifies editing.

When you describe a receiving item with the JUSTIFIED clause and the the sending item is larger, the leftmost characters are truncated. When the receiving item is larger, the data is aligned at the right and the leftmost positions are filled with spaces. Figure 7-1 contrasts how data is moved with and without the JUSTIFIED clause.

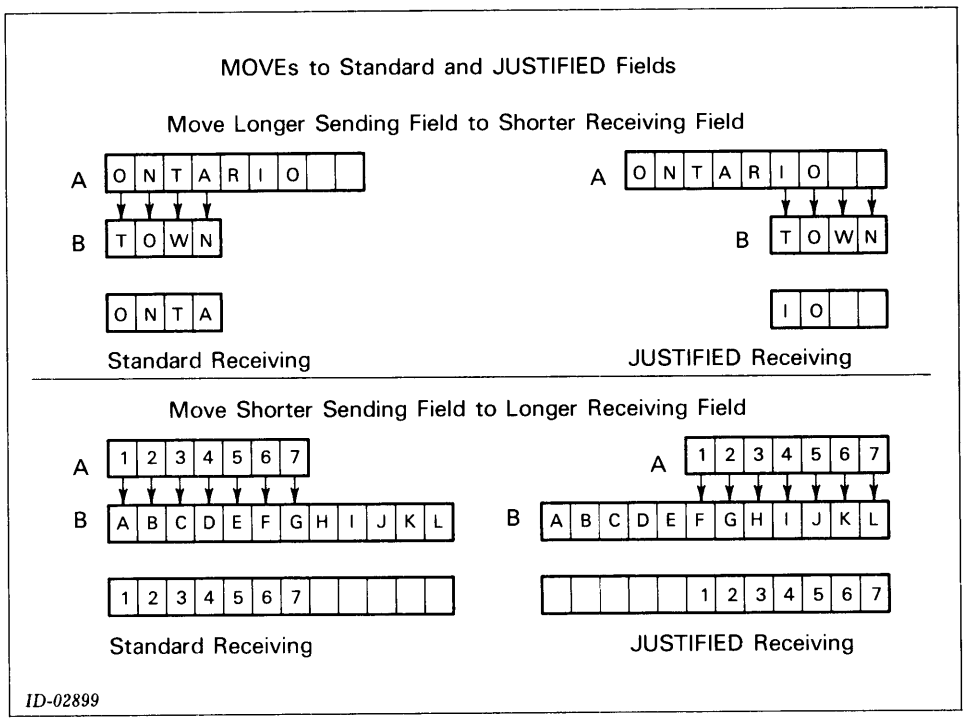



***Figure 7-1*** *Moving Data to Standard and Justified Fields*

## BLANK WHEN ZERO Clause

The BLANK WHEN ZERO clause substitutes spaces for a numeric or numeric edited item when its value is zero.

The format of BLANK WHEN ZERO is

BLANK WHEN ZERO

A numeric item is considered to be numeric edited when its description contains the BLANK WHEN ZERO clause.

## VALUE Clause

The VALUE clause defines the values of constants and the initial values of Working-Storage and Screen Section data-items. It cannot be used in the File or Linkage sections. The format is:

VALUE IS literal

If the VALUE clause occurs in Working-Storage Section, a figurative constant can be substituted for the literal. Figurative constants cannot be used in the Screen Section.

Since the initial value of any data-item is unpredictable, you should assign it an initial value with the VALUE clause or with a procedure statement before using it. The initial value of an index item cannot be defined with the VALUE clause; it must be set with the SET statement.

The VALUE clause must not be stated in a data description entry that contains, or is subordinate to, an entry containing a REDEFINES clause or an OCCURS clause.

The VALUE clause must not conflict with other clauses in the data description of the item. If the category of the item is numeric, the VALUE literal must be numeric. The value of a numeric literal must be within the range of values indicated in the associated PICTURE clause, and it must not have a value that requires truncating nonzero digits. The literal may be signed only if the PICTURE is signed.

If a data-item is alphabetic, alphanumeric, alphanumeric edited, or numeric edited, the VALUE literal must be nonnumeric, with one exception: an item with a PICTURE containing only Zs and 9s may have a numeric value. For example, an item could be described as PIC ZZZ9 VALUE 1234. The value of the literal must not exceed the size indicated by the PICTURE clause.

Editing characters in the PICTURE clause are included in determining the size of the data-item but do not affect initialization. When a VALUE clause is specified with a JUSTIFIED or BLANK WHEN ZERO clause, the VALUE literal is neither justified nor blank when zero. These clauses do not take effect until a value is moved to the data-item.

If the VALUE clause is used at the group level, the literal must be a figurative constant or a nonnumeric literal. The group area is initialized without consideration for the individual elementary or group items contained within this group. The VALUE clause cannot be stated at the subordinate levels within such a group.

# Chapter 8
# The Procedure Division

The Procedure Division is required in every COBOL source program. It contains the operational statements that govern program input and output, data movement, logic or sequence control, and arithmetic.

The Procedure Division may be in either declarative or nondeclarative format. In declarative format, you specify procedures that are to be used in the event of I/O errors. For more information on declaratives, see the USE statement later in this chapter.

## Declarative Format

```
PROCEDURE DIVISION [USING {id}... ].

[DECLARATIVES.

{ section-name SECTION [segment-number].      }
{     USE sentence                            } ...
{ [paragraph-name. [sentence]...]...          }

END DECLARATIVES.]

{ section-name SECTION [segment-number].      }
{ [paragraph-name. [sentence]...]...          } ...
```

## Nondeclarative Format

```
PROCEDURE DIVISION [USING {id}... ].

{ section-name SECTION [segment-number].      }
{ [paragraph-name. [sentence]...]...          } ...
```

*Note:* Segment-number is used for documentation purposes only.

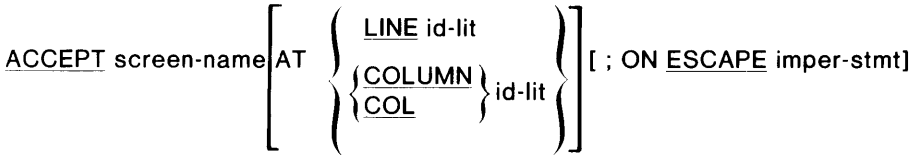
## Procedure Division Statements

This chapter describes the Interactive COBOL procedure statements, arranged alphabetically by the verb that begins the statement. The purpose and format of each statement is presented, along with a discussion of execution features. Whenever *id* or *id-lit* appears, an index-name may be used.

# ACCEPT

## Transfers data from the screen to the File, Working-Storage, or Linkage sections, or transfers system data into a specified data-item.

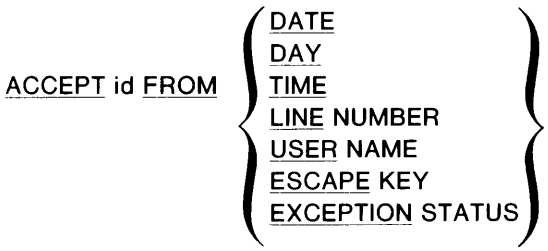
### Screen Format




### Identifier Format

ACCEPT id [;ON ESCAPE imper-stmt]

### System Format




### Examples

```
ACCEPT INFO-SCREEN AT LINE LINE-NUM COL COL-NUM;
    ON ESCAPE GO TO FUNCTION-ROUTINE.
ACCEPT TEMP-AMT.
ACCEPT DATE-TODAY FROM DATE.
ACCEPT FUNCTION-CODE FROM ESCAPE KEY.
ACCEPT EXCEP-CODE FROM EXCEPTION STATUS.
```

### Rules for Screen ACCEPT

Screen names may not be subscripted.

The ACCEPT *screen-name* statement transfers information entered on the screen to the data-items associated with the screen-name. The program must execute a DISPLAY before an ACCEPT to display any associated prompts. ACCEPT *screen-name* is equivalent to ACCEPT *screen-name* LINE 1 COLUMN 1.

The LINE and COLUMN clauses enable you to accept a screen name with a specified origin. The LINE and COLUMN identifiers or literals must be elementary integers. If you execute an ACCEPT with an optional LINE or COLUMN value that is larger than your screen, the screen will wrap; that is, if a display is 24 lines X 80 columns, LINE and COLUMN values of 25 and 81 are equivalent to 1 and 1.

If you have displayed a screen with the LINE and COLUMN options, you must accept it using the same coordinates.

The words COLUMN and COL are interchangeable.

The section "Data Movement with DISPLAY and ACCEPT" in chapter 3 lists details of the execution of an ACCEPT *screen-name* statement.

Moves from the Screen data-item to the File, Working-Storage, or Linkage data-item are treated in the normal COBOL manner with one exception: a move from a numeric edited to a numeric data-item is allowed. In this case, all characters of the numeric edited data-item are ignored except digits, sign, and decimal point. The decimal point is used for alignment purposes only.

If you press ESC at any time during an ACCEPT statement, the data in the current field is not validated or transferred to screen storage. However, the contents of screen storage are still moved to the associated data-items in the File, Working-Storage, or Linkage sections. The ACCEPT terminates and the ON ESCAPE *imper-stmt*, if present, is executed.

## Rules for Identifier ACCEPT

Data is transferred according to the rules of the MOVE verb.

The ACCEPT *identifier* statement transfers information entered on the screen to the specified data-item. It is usually preceded by a DISPLAY statement that prompts the user for the required information. For example:

```
DISPLAY "Enter the purchase price:".
ACCEPT PURCHASE-PRICE.
```

You must input data valid for the identifier. If the data is invalid because of sign, length, decimal point, or characters that do not agree with the item's PICTURE, an error message is displayed on line 24 of the display screen and the input may be corrected.

## Rules for System ACCEPT

DATE, DAY, TIME, LINE NUMBER, USER NAME, ESCAPE KEY, and EXCEPTION STATUS are system-defined data-items treated as elementary items. These items are not explicitly defined in COBOL programs. The identifier used with a system ACCEPT must have a PICTURE that agrees with the content of the system-defined item, according to the rules for MOVE.

DATE is composed of year, month, and day. Its PICTURE is 9(6). For example, July 1, 1984, is expressed as 840701.

DAY is composed of year and day of year in Julian format. Its PICTURE is 9(5). For example, July 1, 1984, is expressed as 84183.

TIME is composed of hours, minutes, seconds, and hundredths of a second, and is based on elapsed time after midnight on a 24-hour clock. Its PICTURE is 9(8). Thus 2:41 P.M. is expressed as 14410000. The seventh and eighth digits, representing hundredths of a second, are always 00.

LINE NUMBER contains the three-digit number of the terminal line on which the program is running. Its PICTURE is 9(3). On RDOS and DG/RDOS, the master terminal is line number 000; the other terminals have line numbers 001 through *nnn*. On AOS and AOS/VS the line number is the console number.

Under AOS and AOS/VS, USER NAME allows the program to retrieve the user name of the current process. Its PICTURE is X(15). Under RDOS and DG/RDOS, ACCEPT id FROM USER NAME causes *id* to be filled with blanks.

The ESCAPE KEY item contains the two-digit code generated by a termination key (see "Termination Codes" below). Its PICTURE is 9(2).

The EXCEPTION STATUS item contains a three-character code that identifies the type of exception condition that has occurred during the execution of a CALL or CALL PROGRAM statement. Its PICTURE is 9(3). A table of Exception Status codes appears in appendix A of each *User's Guide.*

If EXCEPTION STATUS is to be examined, it should be done immediately after the CALL or CALL PROGRAM—before execution of any other statements. File I/O operations will alter the Exception Status value, making the value undefined.

## Terminating Screen Input

Depending on the operating system and the keyboard being used, input may be terminated by the NEW LINE key, the CR key, the down-arrow, the TAB key, the ESC key, the enter and enter-minus keys, and the function keys.

In order for the program to use the ESC or function keys for terminating input, you must code a standard ACCEPT before coding an ACCEPT...FROM ESCAPE KEY. Code the ACCEPT statement in the following manner:

```
WORKING-STORAGE SECTION.
...
77  DECISION-KEY PIC 99.
...
PROCEDURE DIVISION.
...
    ACCEPT DECISION-KEY-ENTRY.
    ACCEPT DECISION-KEY FROM ESCAPE KEY.
    IF DECISION-KEY = 02 PERFORM F1-ROUTINE.
```

## Termination Codes

When you press a terminator key, a two-digit value is moved to the ESCAPE KEY identifier. The item can then be interrogated by the program to determine which key was struck. The codes generated are listed in Table 8-1 (on terminals without CTRL keys, the CMD key is equivalent).

*Note:* Different terminals have differing numbers of available function keys, and the CR, NEW LINE, CTRL, and CMD keys also function differently depending on the keyboard and the operating system. For information on the number, location, and use of the various keys, see the appropriate terminal operating manual.

| | Key Alone | Key+SHIFT | Key+CTRL | Key+SHIFT+CTRL |
|---|---|---|---|---|
| CR | 00 | 00 | 00 | 00 |
| NEW LINE | 00 | 00 | 00 | 00 |
| Down-arrow | 00 | 00 | 00 | 00 |
| TAB | 00 | 00 | 00 | 00 |
| ESC | 01 | 01 | 01 | 01 |
| F1 | 02 | 10 | 18 | 26 |
| F2 | 03 | 11 | 19 | 27 |
| F3 | 04 | 12 | 20 | 28 |
| F4 | 05 | 13 | 21 | 29 |
| F5 | 06 | 14 | 22 | 30 |
| F6 | 07 | 15 | 23 | 31 |
| F7 | 08 | 16 | 24 | 32 |
| F8 | 09 | 17 | 25 | 33 |
| F9 | (-) | 41 | 48 | 55 |
| F10 | (ENTER -) | 42 | 49 | 56 |
| F11 | (ENTER) | 43 | 50 | 57 |
| F12 | 37 | 44 | 51 | 58 |
| F13 | 38 | 45 | 52 | 59 |
| F14 | 39 | 46 | 53 | 60 |
| F15 | 40 | 47 | 54 | 61 |

**Table 8-1 Termination Codes**

Note: Function key F9 enters a value of minus. It is not a terminator. The F10 and F11 keys are terminators that return the same values as the keys labeled ENTER and ENTER -.

# ADD

## Sums two or more numeric data-items and stores the result.

### Cumulative ADD Format

ADD {id-lit}... TO id [ROUNDED] [; ON SIZE ERROR imper-stmt]

### Totaling ADD Format

ADD id-lit {, id-lit}... GIVING id [ROUNDED] [; ON SIZE ERROR imper-stmt]

### Examples

```
ADD REG-HRS, OT-HRS TO TOT-HRS; ON SIZE ERROR GO TO HRS-ERR-RTN.
ADD NEW-STOCK, OLD-STOCK GIVING TOT-STOCK.
```

### Rules

The identifiers in the cumulative or totaling ADD must refer to elementary numeric items. However, the identifier following the word GIVING may refer to an elementary numeric or numeric edited item.

Literals must be numeric.

A SIZE ERROR may occur during the execution of an ADD statement if any intermediate result exceeds 18 digits. A size error also occurs if the sum overflows the receiving item, after decimal point alignment.

With cumulative ADD, the values of the data-items before the word TO are added together. This sum is then added to the current value of the identifier following the word TO, and the result is stored as the new value of that identifier. For example, in the sentence

```
ADD A, B TO C.
```

the sum of A+B is added to the value in C, and this result is stored in C.

With totaling ADD, the values of the data-items preceding the word GIVING are added together, and the sum is then stored as the new value of the identifier following the word GIVING. For example, in the sentence

```
ADD A, B GIVING C.
```

the value in A is added to the value in B, and the result is stored in C.

# ADD CORRESPONDING

## Adds identically named items in two groups.

### Format

```
ADD { CORRESPONDING } id TO id [ROUNDED] [; ON SIZE ERROR imper-stmt]
    { CORR          }
```

### Examples

```
ADD CORRESPONDING LIST1 TO LIST2.
ADD CORR NEW-INVEN TO CURR-INVEN; ON SIZE ERROR GO TO ROUTINE3.
```

### Rules

Both id's must refer to group items. The elements to be added must all be elementary items. No subgroups are added together. None of the items may be designated as USAGE IS INDEX.

The groups need not contain exactly the same list of id's. ADD CORRESPONDING compares the elements that make up each group. Those that have the same name are added together, and the result is stored in the second operand. All others are ignored.

If a SIZE ERROR occurs on any one of the additions, all of the remaining additions are performed before the imperative statement is executed.

The following examples illustrate the use of ADD CORRESPONDING. Two items are described as follows:

```
01  LIST1.                      01  LIST2.
    02  APPLES    PIC 99.          02  LEMONS    PIC 99.
    02  ORANGES   PIC 99.          02  APPLES    PIC 99.
    02  PEARS     PIC 99.          02  ORANGES   PIC 99.
```

The initial values are:

```
LIST1                       LIST2
    APPLES    5                 LEMONS    2
    ORANGES   10                APPLES    6
    PEARS     8                 ORANGES   3
```

After execution of ADD CORRESPONDING LIST1 TO LIST2 the values are:

```
LIST1                       LIST2
    APPLES    5                 LEMONS    2
    ORANGES   10                APPLES    11
    PEARS     8                 ORANGES   13
```

The CORRESPONDING phrase does not require that the level numbers be identical; however, the record structures must be the same. Given the following structure

```
01  LIST1.                    01  LIST2.
    02  APPLES     PIC 99.        02  SUBLIST2.
    02  ORANGES    PIC 99.            03  LEMONS     PIC 99.
    02  PEARS      PIC 99.            03  APPLES     PIC 99.
                                       03  ORANGES    PIC 99.
```

The statement `ADD CORRESPONDING LIST1 TO LIST2` is illegal. The statement `ADD CORRESPONDING LIST1 TO SUBLIST2` is legal.

# CALL

## Transfers control to another program.

### Format

CALL id-lit [USING {data-name}...][ON {EXCEPTION / OVERFLOW} imper-stmt]

### Examples

Calling program:

```
WORKING-STORAGE SECTION.
01  DATA.
    05  DATA-1 PIC  XX.
    05  DATA-2  PIC XXX.
PROCEDURE DIVISION.
    MOVE "ADSYS1" TO PROG-NAME.
    CALL PROG-NAME USING DATA-1, DATA-2
        ON EXCEPTION PERFORM OOPS.
    DISPLAY DATA-1, DATA-2.
```

Called program:

```
PROGRAM-ID.  ADSYS1.
...
DATA DIVISION.
...
LINKAGE SECTION.
01  DATA-ONE    PIC 99.
01  DATA-TWO    PIC 999.
...
PROCEDURE DIVISION USING DATA-ONE, DATA-TWO.
...
EXIT-PARA.
   EXIT PROGRAM.
```

### Rules

The words OVERFLOW and EXCEPTION have the same meaning.

Id-lit must be an alphanumeric identifier or nonnumeric literal; its value must be a valid external filename.

Upon encountering a CALL statement, the runtime system first determines whether the called program is executable. If it is, the runtime system transfers control to the new program. All programs in a run unit except the one that is running are saved in a virtual memory (.VM) file. A run unit can contain a maximum of 16 programs (one main program and 15 subprograms).

The USING phrase passes data to called programs; you can pass up to 32 identifiers with this phrase. CALL...USING can be specified only if there is a Procedure Division Using header in the called program; otherwise the CALL fails with an Exception Status of 209. The CALL...USING and the Procedure Division Using header must contain the same number of identifiers, as a one-to-one correspondence is set up between them. The corresponding identifiers must be the same size, although their PICTUREs do not have to be exactly the same.

The CALL statement cannot pass switches to a called program. Switches are inherited from the calling program automatically.

In the example above, a correspondence is set up between the two sets of identifiers so that DATA-1 and DATA-ONE refer to the same data-item, and DATA-2 and DATA-TWO refer to the same data item. Therefore, if the called program modifies DATA-ONE and DATA-TWO, the results are available in the calling program through DATA-1 and DATA-2.

The Linkage Section must appear in the called program and must contain the data descriptions of the identifiers named in the Procedure Division Using header.

If you call a program more than once, the data-items in the called program remain as they were when the program last exited, unless the program has been cancelled. Recursive calls are not allowed. If PROG1 calls PROG2, PROG2 cannot call PROG1. However, PROG2 can call a third program, PROG3. PROG3 cannot, in turn, call PROG1 or PROG2. If you attempt to do this, the CALL fails with an Exception Status of 207.

The EXIT PROGRAM statement returns control to the calling program. Control is passed to the next statement in the calling program (the DISPLAY statement, in the example above).

*Note:* Interactive COBOL implements its CALL statement as a call by value result, not as a call by reference. Both methods have the same results, except in the presence of side effects.

## Calls to Programs in Other Directories

On RDOS and DG/RDOS, if the called program is not in the current directory, the pathname to the program must be supplied or a link must be established.

On AOS and AOS/VS, if the called program or a link to it is not in the current directory or the pathname is not supplied, the runtime system follows the searchlist to locate the executable file. (See chapter 1 of the *Interactive COBOL User's Guide* for an explanation of searchlists.)

In addition, on AOS and AOS/VS the access control list (ACL) is examined during a CALL statement. For the called program to execute, the user must have (1) execute access to the current directory, (2) execute access to the directory containing the called program, and (3) read access to both the .PD and .DD files of the called program. Without proper access, the CALL fails with an Exception Status of 206.

## CLI, .PR, and Assembly Language Calls

Under RDOS, DG/RDOS, AOS, and AOS/VS, an Interactive COBOL program can call an assembly language subroutine. The *Interactive COBOL User's Guide* for your system gives a detailed explanation.

Under AOS and AOS/VS, an Interactive COBOL program can call the CLI or any .PR file. The format for calling the CLI or a .PR file is basically the same as calling an Interactive COBOL subprogram. The *Interactive COBOL User's Guide (AOS, AOS/VS)* provides a detailed explanation.

# CALL PROGRAM

## Chains to another COBOL program or performs a system function.

### Format

CALL PROGRAM id-lit [USING {data-name}]... [; ON EXCEPTION imper-stmt]

### Examples

```
CALL PROGRAM "ADSYSO/A".
CALL PROGRAM PROG-NAME.
CALL PROGRAM "ADSYS1" USING PASS-DATA-1, PASS-DATA-2.
CALL PROGRAM "ADSYS1" ON EXCEPTION PERFORM CANT-FIND-IT.
CALL PROGRAM "#W".
```

### Rules

Id-lit must be an alphanumeric identifier or nonnumeric literal; its value must be a valid external filename. Id-lit can include from 1 to 26 switches. For example:

```
CALL PROGRAM "ADSYSO/A".
```

See "Switch-Status Condition" in chapter 1 for more information on setting and testing the status of switches within an Interactive COBOL program.

The successful transfer of control to a called program is equivalent to the execution of a STOP RUN statement within the calling program followed by the start of the called program. Thus you cannot return to the original (calling) program, except when performing system calls.

Upon encountering a CALL PROGRAM statement, the runtime system first determines whether the called program is executable. If it is, the runtime system closes all currently OPEN files and passes control to the new program.

*Note:* On RDOS and DG/RDOS, CALL PROGRAM "logically" closes files, flushes buffers, and, if necessary, updates files; but the operating system channels that the files use remain open. The #C system call must be used to close the channels. See chapter 2 of the *User's Guide* for full details.

If the call is unsuccessful, control is passed to the ON EXCEPTION imperative statement, if one is specified. If an ACCEPT...FROM EXCEPTION STATUS clause follows the CALL PROGRAM statement, a value is placed in the EXCEPTION STATUS data-item. (See the table of Exception Status codes in the related *User's Guide.*) If the ON EXCEPTION clause is omitted, control passes to the statement following the CALL PROGRAM statement. (In certain cases, the calling program may no longer be present, resulting in a return to Logon or the CLI.) It is good programming practice to specify an ON EXCEPTION clause.

## The USING Phrase

Data is passed to the called program with the USING phrase. Sending data names must be defined as 01 or 77-level entries in the File Section or Working-Storage Section of the calling program and listed in its CALL PROGRAM USING statement. Corresponding receiving data-names must be defined in the Linkage Section and listed, in the same order, in the Procedure Division Using header of the called program. Sending-receiving pairs need not have the same names or internal structures, but they must have the same total length. Note in the example below that ITEM-TABLE and REVISED-TABLE both have a total length of 144.

The USING data-name may not be subscripted. The same data-name may appear more than once in a CALL PROGRAM USING statement. The same data-name may not appear more than once in the Procedure Division Using header.

The following example illustrates the CALL PROGRAM USING statement.

```
PROGRAM-ID. MAINPROG.
...
DATA DIVISION.
...
WORKING-STORAGE SECTION.
01  EMP-NAME          PIC X(20).
01  ITEM-TABLE.
    02  A-ITEM OCCURS 12 TIMES.
        03  ALPHA     PIC X(10).
        03  BETA      PIC 99.
01  AMOUNT            PIC 9(5).
...
PROCEDURE DIVISION.
...
    CALL PROGRAM CALLEDPROG USING
        EMP-NAME, ITEM-TABLE, AMOUNT
       ON EXCEPTION PERFORM
           CANT-FIND-IT.
```

```
PROGRAM-ID. CALLEDPROG.
...
DATA DIVISION.
...
LINKAGE SECTION.
01  CUSTOMER     PIC X(20).
01  REVISED-TABLE.
    02 A-ITEM OCCURS 8 TIMES.
       03  ALPHA PIC X(15).
       03  BETA  PIC 999.
01  CHARGE       PIC 9(5).
...
PROCEDURE DIVISION USING
    CUSTOMER,
    REVISED-TABLE,
    CHARGE.
...
STOP RUN.
```

## Calls to Programs in Other Directories

On RDOS and DG/RDOS, if the called program is not in the current directory, the pathname to the program must be supplied or a link must be established.

On AOS and AOS/VS, if the called program or a link to it is not in the current directory or the pathname is not supplied, the runtime system follows the searchlist to locate the executable file.

In addition, on AOS and AOS/VS the access control list (ACL) is examined during a CALL PROGRAM statement. For the called program to execute, the user must have (1) execute access to the current directory, (2) execute access to the directory containing the called program, and (3) read access to both the .PD and .DD files of the called program. Without proper access, the CALL PROGRAM fails with an Exception Status of 206.

## System Calls

The USING phrase is not valid when executing system calls.

Table 8-2 lists all system calls. Unless otherwise noted, the calls are available on all systems. However, even those that are not implemented on certain systems may be included in any program. If a particular call is not implemented and there is an ON EXCEPTION *imper-stmt*, that statement is executed. If the ON EXCEPTION clause is missing, execution continues with the next statement. In either case, EXCEPTION STATUS has the value of 203, "COBOL program not found."

For an extended discussion of system calls, see chapter 2 of the *Interactive COBOL User's Guide* for your operating environment.

| Call | Action |
|---|---|
| #A | Abort (RDOS and DG/RDOS only) |
| #C | Physically close operating system file channels (RDOS and DG/RDOS only) |
| #D | Debug |
| #F | Fully initialize a device (RDOS and DG/RDOS only) |
| #H | Hang up |
| #I | Initialize a directory (RDOS and DG/RDOS only) |
| #L | Logon |
| #M | Message (RDOS and DG/RDOS only) |
| #N | Rename a file |
| #O | Run a program detached from a console (RDOS and DG/RDOS only) |
| #P | Call Printer Access Scheduler System (RDOS and DG/RDOS only) |
| #R | Release a directory (RDOS and DG/RDOS only) |
| #S | Shut down |
| #T | Report terminal status (RDOS and DG/RDOS only) |
| #W | Wait |

**Table 8-2 System Calls**

# CANCEL

## Returns a program executed with a CALL to its initial state.

### Format

CANCEL {id-lit}...

### Example

```
MOVE "PROG1" TO IDEN.
CANCEL IDEN.

CANCEL "PROG1".
```

### Rules

If an identifier is specified, its value must be a program-name. If a literal is specified, it must be nonnumeric.

The CANCEL statement closes any files that are open in a called program. When a CANCEL statement is executed, a subsequent CALL to that program will find it in its initial state.

If a CANCEL statement specifies a program that has not been called or one that has already been called and cancelled, no action is taken and control passes to the next statement after the CANCEL. Any program can issue the CANCEL statement. However, a program cannot CANCEL itself or a program that calls it directly. For example, if PROG1 calls PROG2, PROG2 cannot CANCEL PROG1.

# CLOSE

## Terminates the processing of files.

### Format

CLOSE {filename [WITH LOCK]}...

### Examples

```
CLOSE CUSTOMER-FILE WITH LOCK.
CLOSE CUSTOMER-FILE, PRINTING-FILE.
```

### Rules

The files referenced in the CLOSE statement need not all have the same organization or access.

A CLOSE statement may be executed only for a file that is open (see the OPEN statement). Following successful execution of a CLOSE statement, the record area associated with the filename contains no valid data. If the CLOSE is unsuccessful, a USE procedure, if specified, is executed.

The LOCK option prevents the file from being reopened within the program that issued the LOCK.

After a file is closed with the CLOSE statement, no statement except DELETE FILE can access that file until it is opened with another OPEN statement.

On RDOS and DG/RDOS, the CLOSE statement "logically" closes files, but the operating system channels that are reserved for files remain open. In this case, files may not be accessed from the concurrent-operations console. Executing a CALL PROGRAM "#C" statement closes the channels and makes all files available. Executing a CALL PROGRAM "#C*filename*" statement makes available the specified file. The system automatically closes any open files when a STOP RUN statement is executed.

# COMPUTE

## Assigns the value of an arithmetic expression to a single data-item.

### Format

COMPUTE id [ROUNDED] = arith-expr [; ON SIZE ERROR imper-stmt]

### Examples

```
COMPUTE WAGERATE = REG * OVER + BONUS.
COMPUTE PARTS-ON-HD ROUNDED = MIN-QTY - SALES-RPT;
    ON SIZE ERROR GO TO WS.
COMPUTE MAILIST = 100 ** 26 + 4.
```

### Rules

The identifier that appears to the left of the equal sign must refer to an elementary numeric item or an elementary numeric edited item. All other identifiers and literals must be elementary numeric items.

The arithmetic expression can be a single identifier or literal. The COMPUTE statement can then be used to assign the value of a single identifier or literal to a single identifier. Use this assignment instead of MOVE when the ROUNDED or SIZE ERROR options are desired.

The arithmetic operators and the order of evaluation are explained in the section on arithmetic expressions in chapter 1.

COMPUTE is the only COBOL statement that can be used for exponentiation, as shown in the third example above. The value of an exponent must be a single literal or identifier. Exponents may not contain a sign or a fraction.

Data descriptions of operands need not be the same; any necessary conversion and decimal point alignment are supplied throughout the calculation.

The maximum size of each operand or composite of operands is 18 digits.

*Note:* Interactive COBOL truncates intermediate results of a COMPUTE operation to the precision of the final result. This could cause a problem, as in the following example:

```
01  OUT-REC     PIC 9(6).99.
...
01  A           PIC S9(6)V99.
01  C           PIC 99V99 VALUE 0.50.
01  D           PIC S9(6) VALUE 13519.
```

The statement COMPUTE A ROUNDED = D * C / 100 is performed as follows:

```
13519 * 0.50 / 100
= 13519 * .005
```

However, with the truncation of the intermediate result to the precision of the final result (S9(6)V99), the calculation performed is:

```
13519 * .00
= 0
```

If such truncation poses a problem, substitute ADD, SUBTRACT, MULTIPLY, and DIVIDE statements for the COMPUTE statement.

# COPY

## Inserts a copy of another file into the source file.

### Format

COPY [INDEXED] lit

### Examples

```
COPY INDEXED "INFOSCREEN".
COPY "DP2:SECURITY.SR".
IF NUM > TOO-LARGE
    COPY "TOO$LARGE".
    ELSE NEXT SENTENCE.
```

### Rules

The COPY literal must be nonnumeric and a valid filename under your operating system. It must be preceded by a separator, and it must end with a period. Any characters in the line after the period are ignored.

The keyword INDEXED indicates that the COPY file organization is indexed. The Interactive COBOL compiler treats each record of an indexed COPY file as one source line. The maximum record length accepted by the compiler is 132.

COPY statements that reference sequential and indexed files may be mixed in a source program. However, all COPY files must be in the same source entry format as the source program (CARD format or CRT format). If the COPY files are not in the same format as the source program, they are not compiled correctly.

The COPY statement may appear anywhere in the source program, with three exceptions:

- It cannot begin in area A. If COPY begins in area A, ambiguous error messages appear in the copied lines.
- It cannot appear where a comment entry is expected.
- It cannot appear where a PICTURE string is expected.

A source program may contain any number of COPY statements; however, a COPY file cannot contain a COPY statement.

When the compiler encounters a COPY statement, it reads the entire file into the source program. Reading is then resumed at the next line after the COPY statement in the original source file.

If you compile your program with the global /O switch, copy files are not included in the listing file. Otherwise, the copied text is numbered beginning with line 1, and line numbers have a "C" suffix.

On RDOS and DG/RDOS, the file to be copied must be in the current directory, the pathname to the file must be supplied, or a link must be established.

On AOS and AOS/VS, if the file (or a link to it) is not located in the current directory and you do not supply a pathname, the COPY statement follows your searchlist to find the file.

To compile a program on AOS and AOS/VS, you must have write and execute access to the current directory, execute access to directories containing source or copy files, and read access to the files. Lack of the proper access causes an error message.

# DELETE

## Logically removes a record from an indexed or relative disk file or physically removes a file from the disk.

### Record DELETE Format

DELETE filename RECORD [; INVALID KEY imper-stmt]

### File DELETE Format

DELETE FILE {filename}...

### Examples

```
DELETE CUSTOMER-FILE RECORD; INVALID KEY GO TO DEL-ERR-RTN.
DELETE FILE TEMP-TRANS-FILE, TEMP-INVENTORY-FILE.
```

### Rules for Record DELETE

The DELETE RECORD statement can be used only for files with indexed or relative organization.

The execution of a DELETE statement affects neither the contents of the record area in memory associated with filename nor the current record pointer.

For sequential access, a DELETE statement must be preceded by a successfully executed READ statement for the same file. However, the program may contain non-I/O statements between the READ and DELETE statements. DELETE logically removes from the file the record retrieved by the previous READ statement. The next sequential READ retrieves the record following the deleted record.

The INVALID KEY phrase may not be specified for a DELETE statement that references a file in sequential access mode.

For random or dynamic access, an INVALID KEY condition exists if the file does not contain the record specified by the key. Either the INVALID KEY phrase or a USE procedure must be specified to handle error conditions.

When a record DELETE is executed for random or dynamic access, the system logically removes from the file the record identified by the contents of the RELATIVE KEY or primary RECORD KEY data-item associated with the filename.

A logically deleted record cannot be further accessed, although the record may be restored by executing the UNDELETE statement.

Execution of a DELETE statement updates the value of the FILE STATUS item.

The key value of HIGH-VALUES is reserved for system use. Your program should not attempt to delete a record with a key value equal to HIGH-VALUES.

## Rules for File DELETE

The DELETE FILE statement physically removes the specified files from the physical devices on which they reside. The files must be closed when this statement is executed.

If the file specified is a link file, under AOS and AOS/VS, the link file is deleted. Under RDOS and DG/RDOS, the resolution file is deleted.

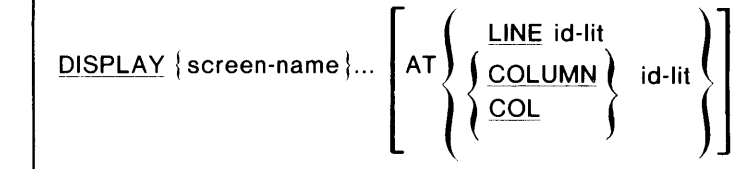
Under AOS and AOS/VS, for the DELETE FILE statement to execute:

- You must have execute access to all directories in the file's pathname, including the directory containing the file. (The DELETE FILE statement does not follow the AOS or AOS/VS searchlist.)
- You must have either write access to the directory containing the file, or owner or write access to the file being deleted.
- The PERMANENCE attribute of the file to be deleted must be OFF. To do this, issue the CLI command `PERMANENCE filename OFF`.

# DISPLAY

## Transfers a group or elementary item to the screen. Performs screen positioning and special screen functions.

### Screen Format




### Id-Lit Format

DISPLAY {id-lit}... [WITH NO ADVANCING]

### Examples

```
DISPLAY INFO-SCREEN  AT LINE LINE-NUM COL COL-NUM.
DISPLAY DATE-TODAY, "REPORT NUMBER", RPT-NO.
DISPLAY "INCORRECT INPUT, PLEASE RE-ENTER" WITH NO ADVANCING.
```

### Rules for Screen DISPLAY

Screen-names may not be subscripted.

DISPLAY *screen-name* is equivalent to DISPLAY *screen-name* LINE 1 COLUMN 1. The LINE and COLUMN clauses enable you to display a screen name with a variable origin. The LINE and COLUMN identifiers must be elementary integers; and their values may be from 1 to 255. If you DISPLAY a LINE or COLUMN value that is larger than your screen, the screen will wrap; that is, if a display is 24 lines × 80 columns, LINE and COLUMN values of 25 and 81 are equivalent to 1 and 1. However, no fields in a screen can begin beyond column 127.

Inclusion of the word AT indicates LINE or COLUMN or both are specified. The words COLUMN and COL are interchangeable.

After a DISPLAY *screen-name* statement is executed, the program must execute an ACCEPT statement to permit data entry. See chapter 3, "Data Movement with DISPLAY and ACCEPT," for details. If you have displayed a variable origin screen, you must accept it using the same coordinates.

When a screen DISPLAY is executed, data-items following a FROM or USING clause are moved to screen storage. If data is to be entered (i.e., the screen data-item has a TO clause but no FROM clause), underscores are moved to screen storage. All moves are executed in the COBOL manner. Once the moves are made, the screen is displayed.

Literal fields are displayed exactly as they appear in the VALUE clause in the Screen Section. The appearance of other fields is determined by the PICTURE clause, which may specify editing symbols and whether a data-item is for input, output, or both. Table 8-3 lists the differences in the appearance of the displayed fields.

After a DISPLAY statement is executed, the cursor is positioned after the last displayed field.

| Type | Clause | Screen Appearance |
|---|---|---|
| Input | TO (without FROM) | Underscores defining its width (from the size of the screen data-item's PICTURE). |
| Output | FROM | Current value of FROM literal or data-item, edited according to the screen data-item's PICTURE. |
| Input/ Output | USING | Current value of USING data-item, edited according to the screen data-item's PICTURE. The displayed value is available for modification when the screen data-item is accepted. |

**Table 8-3 Screen Appearance after Execution of a DISPLAY Statement**

## Rules for Id-lit DISPLAY

With this format, the program cannot control placement of the values on the screen. DISPLAY places the specified literals or data values on the screen in one continuous string at the current cursor position. The cursor then moves to the next line. When NO ADVANCING is specified, the cursor remains at the end of the displayed text.

Do not use the scaling character P in a PICTURE clause, since character positions marked by P are truncated in the display data-item.

When a DISPLAY statement contains more than one operand, the length of the sending item is the sum of the lengths of the operands. DISPLAY moves the operands in the sequence in which they are encountered.

You can display any figurative constant; however you cannot use the figurative constant ALL with a DISPLAY statement. When a figurative constant is used with DISPLAY, a single occurrence is displayed.

# DIVIDE

## Divides one numeric data-item into another and sets the value of data-items equal to the quotient and remainder.

### Overlaying DIVIDE INTO Format

DIVIDE id-lit INTO id [ROUNDED] [;ON SIZE ERROR imper-stmt]

### Nonoverlaying DIVIDE INTO Format

DIVIDE id-lit INTO id-lit GIVING id [ROUNDED] [REMAINDER id]
    [; ON SIZE ERROR imper-stmt]

### Nonoverlaying DIVIDE BY Format

DIVIDE id-lit BY id-lit GIVING id [ROUNDED] [REMAINDER id]
    [; ON SIZE ERROR imper-stmt]

### Examples

```
DIVIDE QUANTITY INTO TOTAL ROUNDED; ON SIZE ERROR GO TO ERR-9.
DIVIDE QUANTITY INTO TOTAL GIVING AVERAGE ROUNDED REMAINDER AVG-R;
   ON SIZE ERROR GO TO ERR-6.
DIVIDE TOTAL BY QUANTITY GIVING AVERAGE ROUNDED.
```

### Rules

Each identifier must refer to an elementary numeric item. However, an identifier associated with the GIVING or REMAINDER phrase must refer to either an elementary numeric or numeric edited item.

A literal must be a numeric literal.

When the overlaying DIVIDE INTO is used, the value of id-lit is divided into the value of id. The resulting quotient replaces the value of the dividend (id). For example, in the sentence

```
DIVIDE A INTO B.
```

the value in B is divided by the value in A, and the result is stored in B.

With the nonoverlaying DIVIDE INTO, the value of the first id-lit is divided into the value of the second id-lit, and the resulting quotient is placed in the GIVING id. For example, in the sentence

```
DIVIDE A INTO B GIVING C.
```

the value in B is divided by the value in A, and the result is placed in C. The values in A and B do not change.

With the nonoverlaying DIVIDE BY, the value of the first id-lit is divided by the value of the second id-lit, and the resulting quotient is placed in the GIVING id. For example, in the sentence

```
DIVIDE A BY B GIVING C.
```

the value in A is divided by the value in B, and the result is placed in C. Again, the values in A and B do not change.

The REMAINDER is the product of the quotient and the divisor subtracted from the dividend. If the quotient id is defined as a numeric edited item, the quotient used to calculate the remainder is an intermediate field that contains the unedited quotient. If ROUNDED is also specified, the remainder is calculated on an intermediate value that contains the truncated, rather than rounded, value of the quotient.

The accuracy of the REMAINDER item is defined by the calculation described above. Appropriate decimal alignment and truncation, but not rounding, are performed for the content of the REMAINDER data-item as needed.

The ON SIZE ERROR phrase specifies a statement for execution in the event of a SIZE ERROR. The SIZE ERROR conditions are described in chapter 1.

When the ON SIZE ERROR phrase is used in the formats specifying REMAINDER, the following rules apply:

- If the size error occurs on the quotient, the remainder is not calculated. Thus, the contents of both the quotient id and the remainder id are unchanged.
- If the size error occurs on the remainder, the contents of the remainder id are unchanged.

See "Arithmetic Expressions" in chapter 1 for an explanation of the ROUNDED phrase.

# EXIT

## Documents the end of a series of procedures.

### Format

EXIT.

### Example

```
RTN-EXIT-PARA.
    EXIT.
```

### Rules

The EXIT statement must appear in a sentence by itself, and it must be the only statement in the paragraph.

The EXIT statement lets you assign a procedure-name to a given point in a program. You may then use this reference point to define the limits of a PERFORM. An EXIT statement has no other effect on the compilation or execution of the program.

# EXIT PROGRAM

## Marks the logical end of a called program.

### Format

EXIT PROGRAM.

### Example

```
PROCEDURE DIVISION.
    ...
EXIT-PARA.
    EXIT PROGRAM.
```

### Rules

EXIT PROGRAM does not close files.

The EXIT PROGRAM statement must be the only statement of a single-statement paragraph.

The EXIT PROGRAM statement occurs at the logical end of a called program. It causes control to be returned to the first statement after the CALL statement of the calling program. The called program remains in the state it was in when the EXIT PROGRAM was executed.

If the EXIT PROGRAM statement occurs in a program that is not called:

- If the paragraph was not performed, control passes to the first sentence in the paragraph after EXIT PROGRAM.
- If the paragraph was performed, the EXIT PROGRAM statement functions the same way as EXIT.

A called program must execute an EXIT PROGRAM statement before being cancelled. Otherwise, results are undefined.

# GO TO

## Transfers control from one part of the Procedure Division to another.

### Simple GO TO Format

GO TO procedure-name

### Conditional GO TO Format

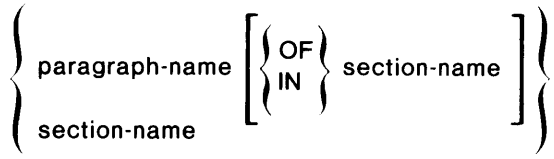
GO TO procedure-name {; procedure-name}... DEPENDING ON id

### Examples

```
GO TO BALANCE-RTN.
GO TO CHECK-1, CHECK-2, CHECK-3 DEPENDING ON CHECK-VALUE.
```

### Rules

*Procedure-name* has the format:




A simple GO TO statement transfers control to the specified procedure-name. When a simple GO TO is used in a sentence containing a sequence of imperative statements, it must be the last statement in the sequence.

In the conditional GO TO format, the identifier is the name of an elementary integer data-item. A GO TO DEPENDING statement transfers control to the procedure-name whose position in the procedure-name list corresponds to the current value of the identifier. If the current value of the identifier is less than 1 or greater than the number of procedure-names in the list, no transfer occurs and control passes to the next statement. For example:

```
GO TO PARA-10, PARA-20, PARA-30  DEPENDING ON PARA-CHECK.
NEXT-PARA.
```

If PARA-CHECK has a value of 1, 2, or 3, control passes to procedure-name PARA-10, PARA-20, or PARA-30, respectively. If PARA-CHECK contains any other value, control passes to the next executable statement in paragraph NEXT-PARA.

With either format, it is *illegal* to transfer control to:

- A procedure-name in a Declaratives section from a nondeclaratives section
- A procedure-name in a nondeclaratives section from a Declaratives section
- A Declaratives section from another Declaratives section

# IF

## Evaluates a condition and takes an action depending on whether the condition is true or false.

### General Format

```
IF condition; { statement-1       } { ; ELSE statement-2       }
              { NEXT SENTENCE     } { ; ELSE NEXT SENTENCE     }
```

### Condition Formats

```
                                  { GREATER THAN }
                                  { LESS THAN    }
                                  { EQUAL TO     }
id-lit IS [NOT]                   { >            }  id-lit
                                  { <            }
                                  { =            }
                                  { >=           }
                                  { <=           }

                                  { NUMERIC      }
                                  { ALPHABETIC   }
id IS [NOT]                       { POSITIVE     }
                                  { NEGATIVE     }
                                  { ZERO         }
```

### Combined Condition Format

```
condition [ { AND } condition ] ...
          { OR  }
```

### Abbreviated Combined Condition Format

```
condition { AND } [NOT] [relation-operator] object ...
          { OR  }
```

### Examples

```
IF LINE-COUNT = 60 GO TO NEW-PAGE-RTN.
IF CHOICE = 1 OR CHOICE = 2 OR CHOICE = 3 NEXT SENTENCE
    ELSE GO TO CHOICE-RTN.
IF CHOICE = 1 OR 2 OR 3 NEXT SENTENCE
    ELSE GO TO CHOICE-RTN.
IF OP-AREA EQUAL CUST-AREA PERFORM EXAMINE-RECORD
    ELSE DISPLAY "Incorrect data".
```

## Rules

If the ELSE NEXT SENTENCE phrase immediately precedes the terminal period of the sentence, it can be omitted.

When the condition stated in an IF statement is true:

- Statement-1 is executed. If it contains a branching statement, control is explicitly transferred in accordance with the rules of that statement; otherwise, statement-2 is ignored and control passes to the next executable sentence.
- If the NEXT SENTENCE phrase is used in place of statement-1, statement-2 is ignored and control passes to the next executable sentence.

When the condition stated in an IF statement is false:

- Statement-1 is ignored and statement-2 is executed. If statement-2 contains a branching statement, control is explicitly transferred in accordance with the rules of that statement. Otherwise, control passes to the next executable sentence. If statement-2 is not specified, statement-1 is ignored and control passes to the next executable sentence.
- If the ELSE clause is not specified, control passes to the next executable statement.
- If the ELSE NEXT SENTENCE phrase is specified, control passes to the next executable sentence.

When the condition is in combined form, any condition except the first may be abbreviated by omitting either:

- The subject of the relation condition
- The subject and relational operator of the relation condition

## Nested IF Statements

Either or both statements may contain IF statements, creating a nested IF statement. IF statements contained within IF statements are considered as paired IF and ELSE combinations. Proceeding from left to right, each ELSE encountered applies to the immediately preceding IF that is not already paired with an ELSE. The following is an example of a nested IF statement:

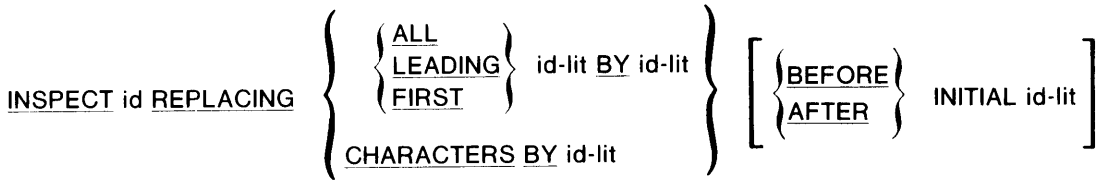
```
IF INVALID-KEY-SW = "Y"
    IF TRANS-CODE = "A"
        PERFORM ADD-ROUTINE
    ELSE
         MOVE "NONMATCHING MASTER RECORD" TO ERROR-MESS
         PERFORM ERROR-ROUTINE
ELSE
    IF TRANS-CODE = "A"
        MOVE "DUPLICATE MASTER RECORD" TO ERROR-MESS
        PERFORM ERROR ROUTINE
    ELSE
        IF TRANS-CODE = "C"
            PERFORM CHANGE-ROUTINE
        ELSE
            IF TRANS-CODE = "D"
                PERFORM DELETE-RTN.
```

# INSPECT

## Counts and/or replaces specific single characters in a data-item.

### Tally INSPECT Format


### Replace INSPECT Format




### Tally and Replace INSPECT Format

INSPECT id TALLYING tally-clause REPLACING replacing-clause

### Rules

The INSPECT statement must specify either the TALLYING or REPLACING options or both. The TALLYING identifier must be an integer.

The identifier immediately following the INSPECT verb is the item that is to be inspected. It may be an elementary or group item. It must be described as USAGE IS DISPLAY.

All id-lits must be one character long and may not be signed numeric items. When a figurative constant is used, it refers to an implicit one-character data-item.

If the CHARACTERS option is specified, an implied one-character comparison is used; the comparison is always considered a match.

If the ALL option is specified, all characters in the comparison area of the specified data-item are considered for the comparison. The LEADING option matches only contiguous occurrences of the specified character string that begin in the leftmost position of the comparison area in the data-item.

The INSPECT comparison cycle normally begins at the leftmost character of the INSPECT identifier and proceeds to the rightmost character. During comparison of the INSPECT identifier, each matched occurrence of the string is tallied and/or replaced as specified in the REPLACING clause.

The BEFORE and AFTER phrases modify the boundaries of the INSPECT comparison. Both phrases always refer to the initial (leftmost) occurrence of the id-lit specified in the phrase.

When the BEFORE phrase is used, comparison continues up to, but does not include, the first occurrence of the id-lit specified in the BEFORE phrase.

When the AFTER phrase is used, comparison begins immediately after the first occurrence of the specified id-lit.

## Tally INSPECT

With the ALL phrase, the TALLYING identifier increases by one for each occurrence of id-lit within the INSPECT identifier.

With the LEADING phrase, the TALLYING identifier increases by one for each contiguous occurrence of id-lit within the INSPECT identifier, provided the leftmost id-lit is at the point where comparison begins.

When the CHARACTERS phrase is used, the TALLYING identifier increases by one for each character in the INSPECT identifier.

## Replace INSPECT

The statement must specify one of the options: ALL, LEADING, FIRST, or CHARACTERS.

- When the ALL phrase is used, each occurrence of the id-lit preceding the BY phrase is replaced by the id-lit following the BY phrase.
- With the LEADING phrase, each contiguous occurrence of the id-lit preceding the BY phrase is replaced by the id-lit following the BY phrase, provided the leftmost id-lit is at the point where comparison begins.
- When the FIRST phrase is used, the leftmost id-lit preceding the BY phrase is replaced by the id-lit following the BY phrase.
- When the CHARACTERS phrase is used, each character in the INSPECT identifier is replaced by the character specified in the REPLACING id-lit.

## Tally and Replace INSPECT

In this format, INSPECT functions as two separate statements, INSPECT TALLYING and INSPECT REPLACING. The INSPECT TALLYING portion is executed first.

## Sample INSPECT Statements

In the following examples the *count* data-item has value 0 before the INSPECT statement is executed.

```
INSPECT word TALLYING count FOR LEADING "L" BEFORE INITIAL "A".
```

Where word = LARGE, count = 1.
Where word = ANALYST, count = 0.

```
INSPECT word TALLYING count FOR ALL "L" REPLACING LEADING
    "A" BY "E" AFTER INITIAL "L".
```

Where word = CALLAR, count = 2, word = CALLAR.
Where word = LATTER, count = 1, word = LETTER.

```
INSPECT word TALLYING count FOR CHARACTERS AFTER INITIAL
    "J" REPLACING ALL "A" BY "B".
```

Where word = ADJECTIVE, count = 6, word = BDJECTIVE.
Where word = JACK, count = 3, word = JBCK.

```
INSPECT word REPLACING ALL "X" BY "Y" AFTER INITIAL "R".
```

Where word = REXMALL, word = REYMALL.
Where word = TEXARKANA, word = TEXARKANA.

```
INSPECT word REPLACING CHARACTERS BY "B" BEFORE INITIAL "A".
```

Where word = 12XZABCD, word = BBBBABCD.

```
INSPECT word REPLACING ALL "A" BY "G" BEFORE INTITIAL "X".
```

Where word = ARXAS, word = GRXAS.
Where word = HANDAX, word = HGNDGX.

# MOVE

## Transfers data from one data-item to one or more data-items.

### Format

MOVE id-lit TO {id}...

### Examples

```
MOVE BALANCE TO LINE-3.
MOVE "THE DATE IS" TO HEAD-2.
MOVE 0 TO SUB-1, SUB-2, SUB-3.
```

### Rules

Id-lit represents the sending area; id represents the receiving area.

The rules for moves apply equally to all receiving areas.

Any subscripting or indexing associated with the sending item is evaluated only once, immediately before the data is moved to the first of the receiving operands.

Any subscripting or indexing associated with a receiving item is evaluated immediately before the data is moved to the respective item. Consider the following statement, where B = 1 and A(1) = 2:

```
MOVE A(B) TO B, C(B).
```

This statement is equivalent to:

```
MOVE 2 TO B.
MOVE 2 TO C(2).
```

Note that A(B) is evaluated at the start only, and C(B) is evaluated immediately before moving data into that item.

### Moves Between Data Types

An elementary move occurs when both the sending and receiving items are elementary items. Every elementary item belongs to one of the following categories: numeric, alphabetic, alphanumeric, numeric edited, or alphanumeric edited. The editing symbols used in the PICTURE clause specify the category for an item. The data categories for literals and figurative constants are:

- Numeric literals are numeric.
- Nonnumeric literals are alphanumeric.
- All figurative constants except ZERO and SPACE are alphanumeric.
- The figurative constant ZERO is numeric.
- The figurative constant SPACE is alphabetic.

| Sending Data-Item | Category of Receiving Data-Item: Alphabetic | Alphanumeric Edited Alphanumeric | Numeric Integer Numeric Noninteger Numeric Edited |
|---|---|---|---|
| Alphabetic | Yes | Yes | No |
| Alphanumeric | Yes | Yes | Yes |
| Alphanumeric Edited | Yes | Yes | No |
| Numeric Integer | No | Yes | Yes |
| Numeric Noninteger | No | No | Yes |
| Numeric Edited | No | Yes | No |

**Table 8-4 Legal Interactive COBOL Move Combinations**

All numeric receiving fields are decimal point aligned, right justified, and padded with zeros. All numeric edited receiving fields are decimal point aligned and padded with the appropriate editing character(s). All other receiving fields are left justified and padded with spaces.

The following restrictions apply to elementary moves between data categories:

- The figurative constant SPACE must not be moved to a numeric or numeric edited item.
- A numeric edited, alphanumeric edited, or alphabetic data-item must not be moved to a numeric or numeric edited item.
- A numeric literal, the figurative constant ZERO, a numeric item, or a numeric edited item must not be moved to an alphabetic item.
- A noninteger numeric literal or a noninteger numeric item must not be moved to an alphanumeric or alphanumeric edited item.

All other elementary moves are legal. Any necessary conversion of data from one form of internal representation to another occurs during the move, along with any specified editing in the receiving item. Table 8-4 summarizes the legality of the various types of MOVE statements.

# MOVE CORRESPONDING

## Moves items with the same name from one group item to another group item.

### Format

```
MOVE  {CORRESPONDING}  id-1 TO id-2
      {CORR         }
```

### Examples

```
MOVE CORRESPONDING TEMP-LIST TO PERM-LIST.
MOVE CORR CURR-SALES TO BACKUP SALES.
```

### Rules

Id-1 and id-2 must be group items.

For two items within each group to correspond they must have the same name. One of the corresponding items must be elementary. The other may be group or elementary.

The MOVE CORRESPONDING verb looks inside each of the group items named as operands. The names of the items subordinate to these groups are compared. Items within id-1 are moved to items within id-2 that have the same name.

The data is combined into one alphanumeric string when the sending item is a group. When the receiving item is a group, the sending data is allocated on a position-by-position basis, with truncation and space filling.

Consult the description of the verb MOVE to see how elementary moves are handled. The following example illustrates the use of MOVE CORRESPONDING. Two items are described as follows:

```
01  LIST1.                    01  LIST2.
    02 APPLES    PIC 99.          02 LEMONS    PIC 99.
    02 ORANGES   PIC 99.          02 APPLES    PIC 99.
    02 PEARS     PIC 99.          02 ORANGES   PIC 99.
```

The initial values are the following:

```
LIST1                      LIST2
   APPLES    4                 LEMONS    8
   ORANGES   9                 APPLES    13
   PEARS     2                 ORANGES   5
```

After execution of MOVE CORRESPONDING LIST1 TO LIST2 the values are:

```
LIST1                    LIST2
   APPLES    4              LEMONS    8
   ORANGES   9              APPLES    4
   PEARS     2              ORANGES   9
```

The CORRESPONDING phrase does not require that the level numbers be identical; however, the record structures must be the same. Given the following structure

```
01  LIST1.                  01  LIST2.
    02  APPLES    PIC 99.       02  SUBLIST2.
    02  ORANGES   PIC 99.           03  LEMONS    PIC 99.
    02  PEARS     PIC 99.           03  APPLES    PIC 99.
                                     03  ORANGES   PIC 99.
```

The statement **MOVE CORRESPONDING LIST1 TO LIST2** is illegal. The statement **MOVE CORRESPONDING LIST1 TO SUBLIST2** is legal.

# MULTIPLY

## Multiplies two numeric data-items and sets the value of a data-item equal to their products.

### Overlaying MULTIPLY Format

MULTIPLY id-lit BY id [ROUNDED] [;ON SIZE ERROR imper-stmt]

### Nonoverlaying MULTIPLY Format

MULTIPLY id-lit BY id-lit GIVING id [ROUNDED] [; ON SIZE ERROR imper-stmt]

### Examples

```
MULTIPLY HOURS BY RATE; ON SIZE ERROR GO TO ERR-RTN-6.
MULTIPLY 3.1416 BY DIAMETER ROUNDED;
    ON SIZE ERROR GO TO CHECK-IT.
MULTIPLY HOURS BY RATE GIVING GROSS.
MULTIPLY DIAM BY 3.1416 GIVING CIRC ROUNDED.
```

### Rules

Each id must refer to an elementary numeric item, except that in the nonoverlaying format the id following the word GIVING must refer to either an elementary numeric or numeric edited item.

A literal must be a numeric literal.

When the overlaying format is used, the value of id-lit is multiplied by the value of id, and the result of the operation replaces the value of id. For example, in the statement

```
MULTIPLY A BY B
```

the value in A is multiplied by the value in B, and the result is stored in B.

When the nonoverlaying format is used, the value of the first id-lit is multiplied by the id-lit following the word BY, and the result is stored in the id following the word GIVING.

See "Arithmetic Expressions" in chapter 1 for details on the ROUNDED and SIZE ERROR phrases.

# OPEN

## Initiates processing of a file.

### Format

```
OPEN [EXCLUSIVE]  { INPUT  {filename}... }
                  { OUTPUT {filename}... }  ...
                  { I-O    {filename}... }
                  { EXTEND {filename}... }
```

### Examples

```
OPEN INPUT TRANS-FILE, CUSTOMER-FILE.
OPEN EXCLUSIVE I-O MASTER-FILE-1.
OPEN INPUT TRANS-FILE OUTPUT PRINT-RPT I-O MASTER-FILE.
```

### Rules

The OPEN statement makes the associated record area available to the program. It does not obtain or release the first data record. A READ obtains a record, and a WRITE releases a record. Before a successful OPEN statement for a given file, no statement can be executed that refers to that file, with the exception of the DELETE FILE statement.

The execution of an OPEN statement updates the value of the FILE STATUS item. When the OPEN statement is unsuccessful, the program executes the USE procedure, if specified, for the file.

Each OPEN statement must specify one of the OPEN modes: INPUT, OUTPUT, I-O, or EXTEND. The file remains in that mode until the program executes a CLOSE statement for it, the run unit terminates, or the program is cancelled. Table 8-5 lists permissible I/O statements with various file organizations, access modes and OPEN modes. Table 8-6 lists the results of an OPEN on existing and nonexisting files.

A file can be opened with one OPEN mode and then reopened with a different OPEN mode in the same program, provided that the file is closed (without the LOCK option) prior to a subsequent OPEN. After successful execution of an OPEN INPUT statement, the file's records can be read whether they are locked or unlocked.

When a program opens an existing indexed or relative file for INPUT or I-O, the program must describe the file exactly as it was described when it was created (i.e., record size, number of keys, etc.)

When any file is opened for INPUT or I-O, the OPEN statement sets the current record pointer to the first record in the file. For an indexed file, this is the record corresponding to the lowest primary key. If no record exists in the file, the current record pointer is set so that the first READ statement executed for the file results in an AT END condition.

| File Access Modes | Statement | Input | Open Mode | | |
|---|---|---|---|---|---|
| | | | Output | I-O | Extend |
| Sequential | READ | S,R,I | | S,R,I | |
| | WRITE | | S,R,I | | S |
| | REWRITE | | | S,R,I | |
| | START | R,I | | R,I | |
| | DELETE/UNDELETE | | | R,I | |
| Random | READ | R,I | | R,I | |
| | WRITE | | R,I | R,I | |
| | REWRITE | | | R,I | |
| | START | | | | |
| | DELETE/UNDELETE | | | R,I | |
| Dynamic | READ | R,I | | R,I | |
| | WRITE | | R,I | R,I | |
| | REWRITE | | | R,I | |
| | START | R,I | | R,I | |
| | DELETE/UNDELETE | | | R,I | |

**Table 8-5 Permissible I/O Statements with OPEN**

Key
S: Statement accesses for files with sequential organization.
R: Statement accesses files with relative organization.
I: Statement accesses files with indexed organization.

The OPEN OUTPUT statement opens a file for output operations. For sequential files, it creates a new file with the designated filename. An existing file with that filename is automatically deleted. For indexed or relative files, the OUTPUT mode should be used only if the file does not exist; in this case the file is created. If the file already exists and output operations are desired, the file should be opened in I-O mode.

The OPEN I-O statement permits the program to open a file for both input and output operations. I-O mode can be specified only for disk files. For sequential files, OPEN I-O may be specified only for an existing file. For relative or indexed files, the OPEN I-O statement creates the file if it does not already exist.

The EXTEND phrase positions the record pointer immediately after the last logical record of a sequential file, allowing a program to write additional records to that file. It may be used only for sequential files. Subsequent WRITE statements referencing the file add records as though the file had been opened in OUTPUT mode. If the file does not exist, it is created.

## The EXCLUSIVE Phrase

The EXCLUSIVE phrase is an Interactive COBOL feature that prevents any other user from opening the file. An attempt to open a file already opened with EXCLUSIVE results in an I/O error.

EXCLUSIVE is redundant for files opened in EXTEND mode, as the system does not allow multiple users of a file opened for EXTEND. EXCLUSIVE is not implied for files assigned directly to PRINTER or PRINTER-1, whether they are opened for OUTPUT or EXTEND.

| Option | File Is Available | File Is Unavailable |
|---|---|---|
| INPUT | Normal open. If the file is empty, the first READ causes the AT END or INVALID KEY condition. | Open is unsuccessful. |
| I-O | Normal open. If the file is empty, the first READ causes the AT END or INVALID KEY condition. | Sequential: Open is unsuccessful. Indexed or relative: Creates the file. |
| OUTPUT | Sequential: Deletes old file and creates new one. Indexed or relative: Open is unsuccessful. | Creates the file. |
| EXTEND | Sequential: Positions to last record. Indexed or relative: Open is unsuccessful. | Sequential: Creates the file. Indexed or relative: Open is unsuccessful. |

**Table 8-6 Permissible Options with OPEN**

## OPEN Rules for RDOS and DG/RDOS

A CLOSE or CALL PROGRAM "logically" closes a file; that is, the buffer is flushed and, if necessary, the file is updated. However, the file remains physically open; that is, the operating system channels used by the file remain open. These channels remain open until one of the following occurs:

1. The runtime system is terminated.
2. All available channels are used, and a request is made for another file.
3. A #C system call is issued.

If the file to be opened is not in the current directory, a pathname must be supplied or a link must be established.

## OPEN Rules for AOS and AOS/VS

If a pathname is not supplied and the file (or a link to it) is not in the current directory, the OPEN statement follows the user's searchlist to find the file.

The access control list (ACL) is examined before a file is opened. The access types required for the OPEN statement depend on the open mode, as follows:

- INPUT. Read access required
- OUTPUT, EXTEND, I-O. Read access required. Write access required for the directory if a file is being created.

If you do not have the proper access type(s) when executing an OPEN statement, the OPEN fails with a File Status code of 91.

## Open File Limits

Files opened by one or more COBOL programs use operating system input/output channels. A sequential file uses a single channel for each program that opens it. An indexed or relative file uses two channels, no matter how many programs have opened it. Therefore, the number of simultaneously open files depends on the type of file organization and the number of channels available on the operating system. If any program attempts to open a file when channels are not available, the OPEN fails with a File Status code of 91.

The number of channels varies among systems. For information about the number of channels available on your system, see the *Interactive COBOL User's Guide* for your operating environment.

# PERFORM

## Transfers control to and executes one or more procedures.

### Unconditional PERFORM Format

```
PERFORM procedure-name-1 { THROUGH } procedure-name-2
                         { THRU    }
```

### Iterative PERFORM Format

```
PERFORM procedure-name-1 { THROUGH } procedure-name-2 { integer } TIMES
                         { THRU    }                  { id      }
```

### Conditional PERFORM Format

```
PERFORM procedure-name-1 { THROUGH } procedure-name-2 UNTIL condition
                         { THRU    }
```

### Variable PERFORM Format

```
PERFORM procedure-name-1 { THROUGH } procedure-name-2
                         { THRU    }
VARYING id-1 FROM id-lit-1 BY id-lit-2 UNTIL condition-1
[AFTER id-2 FROM id-lit-3 BY id-lit-4 UNTIL condition-2]
[AFTER id-3 FROM id-lit-5 BY id-lit-6 UNTIL condition-3]
```

### Examples

```
PERFORM SET-UP THROUGH CALLING-SEQUENCE-EXIT.
PERFORM INCREMENT-RTN 10 TIMES.
PERFORM INCREMENT-RTN UNTIL CHECK-VALUE = QUANTITY.
PERFORM OUTPUT-RTN VARYING SUB-1 FROM CHECK-VALUE BY 2
    UNTIL SUB-1 = 60.
```

### Rules

An *active* PERFORM is one that has started executing its first statement. The PERFORM remains active until its end limit is executed. Interactive COBOL allows 30 active PERFORM statements. An attempt to activate more than 30 results in a runtime error. A message is displayed and the program terminates.

The words THROUGH and THRU are equivalent.

When a PERFORM statement is executed, control passes to the first statement of the procedure named in procedure-name-1. Transfer of control to the next executable statement following the PERFORM statement is established as follows:

- If procedure-name-1 is a paragraph-name and procedure-name-2 is not specified, the return is after the last statement of the paragraph.
- If procedure-name-1 is a section-name and procedure-name-2 is not specified, the return is after the last statement of the last paragraph in the section.
- If procedure-name-2 is specified and is a paragraph-name the return is after the last statement of the paragraph.
- If procedure-name-2 is specified and is a section-name, the return is after the last statement of the last paragraph in the section.
- If control passes to these procedures by means other than a PERFORM statement, control passes through the last statement of the procedure to the next executable statement as if no PERFORM statement mentioned these procedures.

## Unconditional PERFORM

The unconditional format is the basic PERFORM statement. A procedure referenced by this type of PERFORM statement is executed once; control then passes to the next executable statement following the PERFORM statement.

## Iterative PERFORM

The iterative format uses the TIMES option. The initial value of the identifier specifies the number of times the PERFORM is executed. If you change this value during execution of the PERFORM, the procedure is still executed the original number of times. After the procedure is executed the specified number of times, control passes to the next executable statement.

The identifier or integer must be positive with a value less than 32,768. If the value is zero or negative when execution of the PERFORM begins, control passes immediately to the next executable statement. If the value of the identifier or integer is greater than 32,767, an error message is displayed and the program terminates.

## Conditional PERFORM

The conditional format uses the UNTIL option. The procedures are performed until the condition specified by the UNTIL phrase is true. When the condition is true, control passes to the next executable statement after the PERFORM statement. If the condition is true when the PERFORM statement is encountered, the PERFORM is not executed; control passes immediately to the next executable statement.

## Variable PERFORM

The variable format uses the VARYING option. A variable perform terminates only when condition-1 is true. Condition-2 and condition-3 affect only the number of times the PERFORM is executed. At the end of a PERFORM having two or three conditions, id-2 and id-3 have the current values of id-lit-3 and id-lit-5, respectively.

When one item is varied, the following occurs:

1. Id-1 is set to the value specified by id-lit-1.
2. The condition of the UNTIL phrase is evaluated.
3. If the condition is true, control passes to the statement following the PERFORM.
4. If the condition is false, the specified procedures are executed once.
5. The value of id-1 is incremented or decremented by id-lit-2.
6. The condition in the UNTIL phrase is tested again; the cycle continues until the condition is true.

When two items are varied, the following occurs:

1. Id-1 and id-2 are set to their initial values as specified by id-lit-1 and id-lit-3.

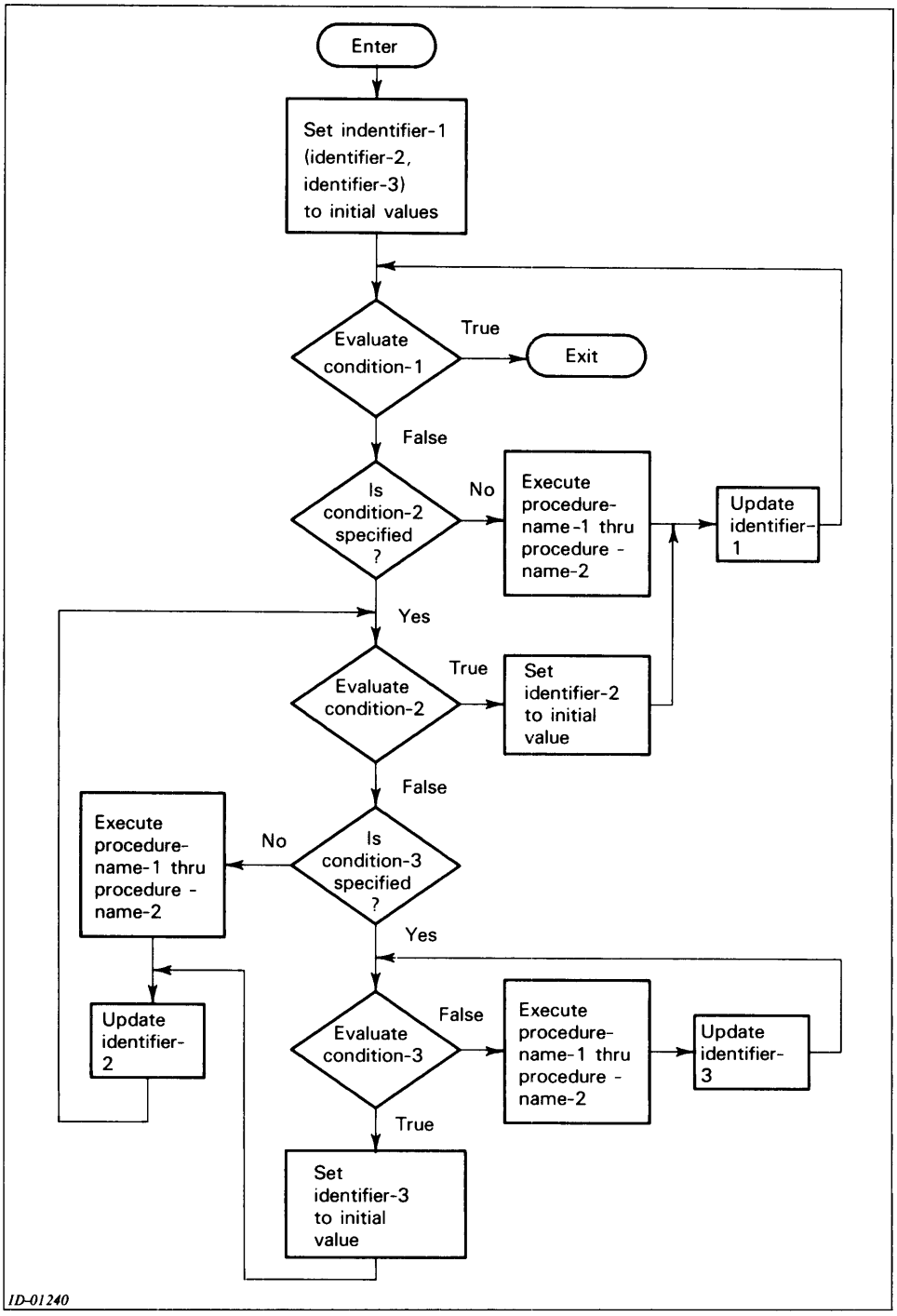



***Figure 8-1*** *Flowchart for a PERFORM...VARYING Statement*

2. Condition-1 is evaluated.
3. If condition-1 is true, control passes to the statement following the PERFORM.
4. If condition-1 is false, condition-2 is tested.
5. If condition-2 is false, the PERFORM is executed once, id-2 is incremented or decremented by id-lit-4, and condition-2 is tested again. This continues until condition-2 is true.
6. If condition-2 is true, id-2 is set to its initial value, id-1 is incremented or decremented by id-lit-2, and control returns to step 2.

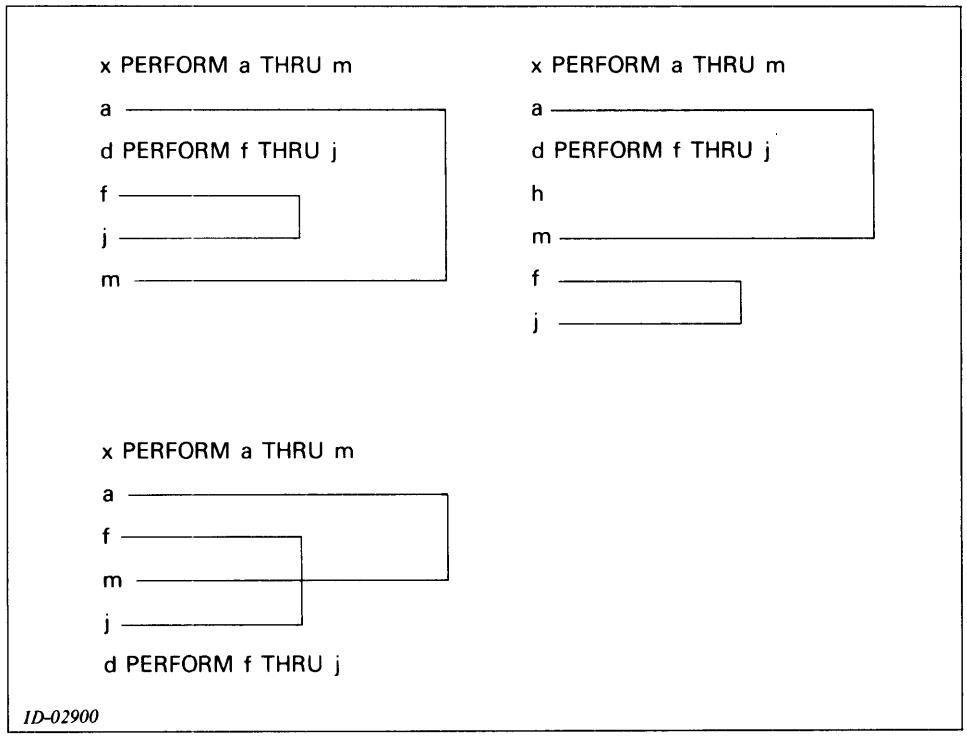



***Figure 8-2*** *Valid PERFORM Constructions*

When three items are varied, the following occurs:

1. Id-1, id-2, and id-3 are set to their initial values as specified by id-lit-1, id-lit-3, and id-lit-5.
2. Condition-1 is evaluated.
3. If condition-1 is true, control passes to the statement following the PERFORM.
4. If condition-1 is false, condition-2 is tested.
5. If condition-2 is false, condition-3 is tested.
6. If condition-3 is false, the PERFORM is executed, id-3 is incremented or decremented by id-lit-6, and the condition is tested again until condition-3 becomes true.
7. If condition-3 is true, id-3 is set to its initial value, id-2 is incremented or decremented by id-lit-4, and condition-2 is tested again. This continues until condition-2 is true.
8. If condition-2 is true, id-2 is set to its initial value, id-1 is incremented or decremented by id-lit-3, and control returns to step 2.

Figure 8-1 gives the general format for PERFORM...VARYING statement.

If a sequence of statements referred to by a PERFORM statement includes another PERFORM statement, the sequence of procedures associated with the included PERFORM must be totally in, or totally excluded from, the logical sequence referred to by the first PERFORM. An active PERFORM statement whose execution point begins within the range of another active PERFORM statement may not pass to the exit of the first PERFORM statement. Two or more such active PERFORM statements may not have a common exit. See Figure 8-2 for examples of valid PERFORM constructions.

# READ

## Makes available the next logical record or a specified record, depending on file access type.

### Sequential Access Format

READ filename RECORD [LOCK] [INTO id] [; AT END imper-stmt]

### Random Access Format

Relative Files:

READ filename RECORD [LOCK] [INTO id] [; INVALID KEY imper-stmt]

Indexed Files:

READ filename RECORD [LOCK] [INTO id] [; KEY IS data-name] [; INVALID KEY imper-stmt]

### Dynamic Access Format

Sequential Retrieval:

READ filename { NEXT / PREVIOUS } RECORD [LOCK] [INTO id] [; AT END imperstmt]

Random Retrieval:

READ filename RECORD [LOCK] [INTO id] [; KEY IS data-name]
    [; INVALID KEY imper-stmt]

### Examples

```
READ TRANS-FILE INTO REC-2.
READ MASTER-FILE INTO REC-1; INVALID KEY GO TO NO-MAST-RTN.
READ CUSTOMER-FILE NEXT AT END GO TO ENDING-RTN.
READ MASTER-FILE LOCK INTO REC-1;
    INVALID KEY GO TO NO-MAST-RETURN.
```

### General Rules

The associated file must be open in the INPUT or I-O mode at the time a READ statement is executed. The execution of a READ statement updates the value of the FILE STATUS item.

The INTO identifier and the record area associated with the filename must not share the same storage area.

When the INTO phrase is specified, the current record is moved from the record area to the identifier area according to the rules for MOVE. Thus the execution of a READ...INTO statement is equivalent to:

```
READ filename.
MOVE record-name TO identifier.
```

That record is then available in both the input record area and the identifier area. The implied MOVE does not occur if the execution of the READ statement is unsuccessful. Any subscripting or indexing associated with the identifier is evaluated after the record has been read and immediately before it is moved to the data-item.

Specify the KEY IS clause only for an indexed file. The data-name must be a data-item specified as one of the record keys for the file. The value of any key should not be equal to HIGH-VALUES. This value is reserved for system use.

## The LOCK Option

The LOCK option can be used only for indexed or relative files. When a record is read with the LOCK option, it may not be accessed (read, deleted, or rewritten) by any other user. An attempt by another user to access a record that is locked results in an I/O error. A locked record may be deleted, rewritten, or unlocked only by the program that issued the LOCK statement. Exception: after the successful execution of an OPEN INPUT statement, records in that file may be read regardless of whether they are locked or unlocked.

When a program attempts a sequential READ of a locked record, and that READ is unsuccessful, the current record pointer is not reset. Thus if the read statement is within a loop, the program continues to try to access that record until it is unlocked. This looping condition can be avoided by executing a statement to check the File Status code.

Under RDOS and DG/RDOS, the maximum number of locked records is two times the number of ISAM files, as specified in the command that brings up the runtime system. Under AOS and AOS/VS, the maximum number is 32 locked records per file. An attempt to exceed the limit results in an I/O error with a File Status code of 9E. Therefore, a record should be unlocked as soon as it is written or rewritten.

## AT END Condition

A READ statement must specify an AT END or INVALID KEY phrase (depending on the access method) if the program does not provide a USE procedure for the file.

When a sequential READ statement is executed and no next logical record exists in the file, the AT END condition occurs, and the READ statement is unsuccessful. When the AT END condition is recognized, the following actions are taken in order:

1. A value is placed into the FILE STATUS item associated with this file, to indicate an AT END condition.
2. If the AT END phrase is specified in the statement, control passes to the AT END imperative statement. Any USE procedure specified for this file is not executed.
3. If the AT END phrase is not specified, then a USE procedure must be specified for this file, and that procedure is executed.

## Access Modes

Interactive COBOL provides three methods for retrieving data and three types of file organizations (see chapter 2 for a full description). The user defines the access method and file organization with the SELECT phrase in the File-Control entry. The following discussion of record retrieval is organized by access method and file organization.

## Rules for Sequential Access

The sequential access method may be used for files with sequential, relative, or indexed organization. It must be used when the file organization is sequential.

Execution of a sequential READ statement makes available the next logical record from the file. The current record pointer determines what is considered the next record.

*Note:* On RDOS and DG/RDOS, the READ statement for files assigned to KEYBOARD is terminated by CR, null, and form feed. On AOS and AOS/VS, NEW LINE is also a terminator.

**Sequential Organization.** In a file with sequential organization, if an OPEN statement positioned the current record pointer, the current record is made available. However, if a previous READ statement positioned the current record pointer, the next existing record in the file is made available. The next record is the succeeding record in logical sequence.

When the AT END condition occurs, the program must not execute a sequential READ for that file before executing a successful CLOSE statement followed by a successful OPEN statement.

**Relative Organization.** In a file with relative organization, if a START or OPEN statement positioned the current record pointer, the current record is made available. However, if a previous READ statement positioned the current record pointer, the next existing record (i.e., one that has not been logically deleted) in the file is made available. The next record is the succeeding logical record in key sequence. The key sequence is determined by the ascending values of relative record numbers for the records in the file.

If the RELATIVE KEY clause is specified in the SELECT entry for this file, the relative key data-item is updated to reflect the relative record number of the record being made available.

The AT END condition occurs when no next logical record exists in the file. When this condition is recognized (see the description above), a sequential access READ statement must not be executed for this file until one of the following has been executed:

- A successful CLOSE statement followed by a successful OPEN statement
- A successful START statement for this file
- A successful random access READ statement for this file

**Indexed Organization.** For files with indexed organization, if a START or OPEN statement positioned the current record pointer, the current record is made available. However, if a previous READ statement positioned the current record pointer, the next existing record in the file is made available. The next existing record is the succeeding logical record in key sequence. The key sequence is determined by the ascending values of the current key of reference.

When an ALTERNATE RECORD KEY is the key of reference, file records with duplicate key values are made available in the order in which they were written to the file.

The AT END condition occurs when no next logical record exists in the file. When the AT END condition is recognized, a sequential access READ statement must not be executed for this file until one of the following has been executed:

- A successful CLOSE statement followed by a successful OPEN statement
- A successful START statement for this file
- A successful random access READ statement for this file

## Rules for Random Access

This method may be used only with disk files having indexed or relative organization. The SELECT clause must specify random access for the file. The current record is determined by the value in the current record pointer.

The INVALID KEY phrase must be specified if no applicable USE procedure is designated for the filename.

**Relative Organization.** The execution of a random READ statement for a relative file sets the current record pointer to the record whose relative record number is contained in the RELATIVE KEY data-item (defined in the SELECT clause). That record is then made available. When the record is not found, the INVALID KEY condition exists and execution of the READ statement is unsuccessful (see "Exception Conditions" in chapter 1). Following an unsuccessful READ, the contents of the associated record area and the position of the current record pointer are undefined.

**Indexed Organization.** The records in an indexed file are accessed by record key values. When a random READ statement is executed for an indexed file, the following occurs:

1. The key of reference is established. When the KEY phrase is specified, the data-name becomes the key of reference. The KEY data-name must identify a record key associated with filename. It may be the primary record key or any alternate record key. The data-name may be qualified, but it cannot be subscripted or indexed. If the KEY phrase is omitted, the primary record key is established as the key of reference.
2. The value of the key of reference is compared with the value of the corresponding key data-item in the file records, until the first record having an equal value is found.
3. The current record pointer is then set to this record, and it is made available.

If a record with an equal value is not found, the INVALID KEY condition exists and execution of the READ statement is unsuccessful (see "Exception Conditions" in chapter 1). Following an unsuccessful READ, the contents of the associated record area and the position of the current record pointer are undefined.

## Rules for Dynamic Access

This method combines the sequential and random access methods and allows the program to switch from one to the other. This access mode can be used only with files having relative or indexed organizations. The SELECT clause must specify dynamic access for the file.

**Sequential Retrieval.** The NEXT or PREVIOUS phrase must be specified to retrieve records sequentially during dynamic access. The NEXT phrase causes the program to retrieve the next logical record. Similarly, the PREVIOUS phrase causes the program to retrieve the previous logical record. If the preceding operation was a START or READ, then either a READ NEXT or a READ PREVIOUS will retrieve the current record. Otherwise, the next or previous record is read. If the end of the file is encountered during a READ NEXT, or if the beginning of the file is encountered during a READ PREVIOUS, the AT END condition occurs and the statement is unsuccessful. The rules outlined above for the AT END condition apply.

All the other rules described above in "Sequential Access" apply.

**Random Retrieval.** The key of reference is established as described above in "Random Access." Subsequent sequential reads use this key of reference until a different key is established.

The same rules described in "Random Access" apply.

The following example shows dynamic access for an indexed file.

```
ENVIRONMENT DIVISION.
...
FILE-CONTROL.
    SELECT CUSTOMER-FILE;
        ASSIGN TO DISK, CUS-FILE-NAME;
        ORGANIZATION IS INDEXED;
        ACCESS IS DYNAMIC;
        RECORD KEY IS CUST-KEY;
        ALTERNATE RECORD KEY IS CUST-AREA.
...
PROCEDURE DIVISION.
...
```

* Random read on key value of CUST-KEY

```
    READ CUSTOMER-FILE INTO REC-1;
        INVALID KEY MOVE 0 TO REC-EXISTS-FLAG.
```

* Sequential read of NEXT record

```
    READ CUSTOMER-FILE NEXT INTO REC-2;
        AT END MOVE 1 TO END-OF-CUST-FILE.
```

# REWRITE

## Replaces an existing record in a disk file.

### Format for Sequential Files

REWRITE record-name [FROM id]

### Format for Relative and Indexed Files

REWRITE record-name [FROM id] [; INVALID KEY imper-stmt]

### Examples

```
REWRITE MAST-REC FROM REC-1 INVALID KEY GO TO REC-LOST-RTN.
REWRITE TRANS-REC.
REWRITE CUSTOMER-REC INVALID KEY GO TO ERROR-6.
```

### Rules

The record-name is the name of a logical record associated with a file declared in the Data Division. The record-name may be qualified. The record-name and id must not refer to the same storage area.

The number of character positions in the record referenced by the record-name must be equal to the number of character positions in the record being replaced.

The file associated with the record-name must be open in I-O mode (see the OPEN statement) at the time the REWRITE statement is executed.

The execution of the REWRITE statement causes the value of the FILE STATUS item to be updated. It does not affect the current record pointer.

After the successful execution of a REWRITE statement, the record is no longer available in *record-name* unless the associated file is named in a SAME RECORD AREA clause. In this case, the record is available to the program as a record from the other files named in the SAME RECORD AREA clause.

The execution of a REWRITE statement with the FROM phrase is equivalent to the execution of:

```
MOVE id TO record-name.
REWRITE record-name.
```

After the rewrite, the record information is still available in the identifier.

The key value of HIGH-VALUES is reserved for system use. Your program should not attempt to execute REWRITE on a record with key values equal to HIGH-VALUES.

For files in sequential access mode, a successful READ statement must be executed before the execution of the REWRITE statement. The record specified in the REWRITE statement replaces the record that was just read.

### Sequential Files

The INVALID KEY phrase may not be specified for a file with sequential organization. A USE procedure should be specified.

### Relative Files

In a relative file accessed in random or dynamic access mode, the record specified in the REWRITE statement replaces the record designated by the RELATIVE KEY data-item.

An INVALID KEY or a USE procedure must be specified. The INVALID KEY condition occurs when the referenced file does not contain the record specified by the RELATIVE KEY data-item. The updating operation does not take place, and the data in the record is unaffected.

### Indexed Files

With sequential access, the record specified in the REWRITE statement replaces the record specified by the value of the primary record key. The value specified in the REWRITE statement must equal the value of the primary key data-item in the record just read from this file.

With random or dynamic access, the record specified in the REWRITE statement replaces the record specified by the value in the primary record key.

The contents of alternate record key data-items of the record being rewritten may differ from those in the record being replaced. If any alternate data-item has changed in value, the record is no longer available using the old alternate value(s); it is available using the new alternate value(s). The system ensures that later access to the record can be based upon any of the record keys.

An invalid KEY or a USE procedure must be specified. The INVALID KEY condition exists if either of the following occurs:

- The access mode is sequential, and the value in the primary RECORD KEY of the record to be replaced does not equal the primary RECORD KEY data-item of the last record retrieved from the file.
- The value in the primary RECORD KEY does not equal that of any record in the file.

If either of the above conditions exists, the REWRITE statement is unsuccessful, the updating operation does not take place, and the data is unaffected.

# SET

## Loads specific values into associated index-items.

### Format


### Examples

```
SET SUB-1, SUB-2 TO CHECK-VALUE.
SET SUB-2 UP BY 10.
SET GRADE-1 DOWN BY DEMOTE.
```

### Rules

An identifier in a SET statement may refer to an index data-item, an index-name, or an elementary integer. A literal must be numeric and can be signed.

If the TO phase is used, the literal must be positive and the value of id-lit replaces the current value of id.

Index-names are associated with a given table through the INDEXED BY option of the OCCURS clause in a data description entry.

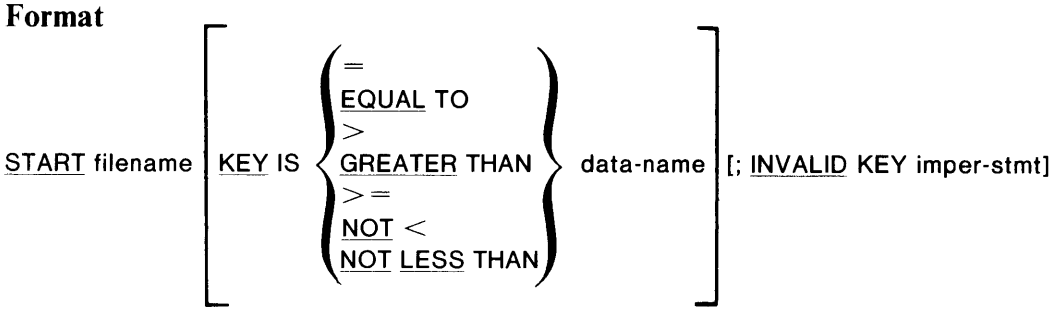
Data-items described as USAGE IS INDEX are not considered to be related to particular tables.

UP BY increases id by the value of id-lit, while DOWN BY decreases id by the value of id-lit.

# START

## Positions the record pointer for sequential retrieval of records.

### Format




### Examples

```
START TRANS-FILE KEY IS EQUAL TO FIRST-VALUE.
START INVENTORY-01 KEY IS NOT LESS THAN LAST-VALUE;
    INVALID KEY PERFORM NON-REC-RTN.
```

### Rules

START sets the current record pointer to the first record that satisfies the KEY condition. The filename must be the name of an indexed or relative file with sequential or dynamic access. The file must be open in INPUT or I/O mode.

If the KEY phase is omitted, the relational operator IS EQUAL TO is implied. When the KEY phase is used, the comparison specified in the KEY relational operator is made between the key field associated with the file's records and the corresponding data-name. The specified data-name may be qualified, but it may not be subscripted or indexed. The value of the key cannot be equal to HIGH-VALUES. HIGH-VALUES is reserved for system use.

The START statement makes a comparison between a key field in the file's records and the current value in the corresponding key data-name. It then positions the current record pointer to the first undeleted logical record in the file with the key value that satisfies the condition.

If no record in the file satisfies the condition, an INVALID KEY condition exists; the position of the current record pointer is undefined, and INVALID KEY *imper-stmt,* if specified, is executed. If none is specified, the applicable USE procedure is executed. (See chapter 1 for an explanation of the INVALID KEY condition.) The INVALID KEY phrase must be specified if no applicable USE procedure is designated for the file.

The execution of the START statement updates the value of the FILE STATUS item.

## Relative Files

For relative files, the data-name must be the data-item specified in the RELATIVE KEY phrase of the associated FILE-CONTROL entry.

If the KEY phrase is not specified, START uses the RELATIVE KEY data-item and the IS EQUAL TO operation for the comparison. It positions the current record pointer to the first undeleted logical record with the key that satisfies the condition.

If the START statement is unsuccessful, the current record pointer is undefined.

## Indexed Files

For indexed files, data-name is an alphanumeric data-item that specifies the primary key or an alternate record key.

When the statement executes successfully, the key of reference is established. The data-name is the key of reference unless the KEY phrase is not specified, in which case the primary key becomes the key of reference. Subsequent sequential READ statements also use it.

If the operands in the comparison are of unequal length, the comparison proceeds based on the length of the key, without regard to the length of the defined data-name. All other nonnumeric comparison rules apply.

If the START statement specifies the EQUAL phrase, the current record pointer is set to the first record with a key equal to the contents of the data-name. If the GREATER phrase is used, the current record pointer is set to the first record with a key greater than the data-name. If the NOT LESS phrase is specified, the current record pointer is set to the first record with a key greater than or equal to the data-name.

If START EQUAL is executed for a value in a duplicate alternate key, the current record pointer is positioned to the first record written using the alternate key value. Subsequent sequential reads retrieve records in the order in which they were written.

If START is unsuccessful, the current record pointer and the current key of reference are undefined.

# STOP

## Permanently or temporarily suspends program execution.

### Format

```
STOP { RUN     }
     { literal }
```

### Examples

```
STOP RUN.
STOP "PROGRAM NOW TERMINATING".
```

### Rules

A STOP RUN statement terminates the program. When STOP RUN appears in a consecutive sequence of imperative statements within a sentence, it must be the last statement in that sequence.

The literal may be numeric, nonnumeric, or any figurative constant except ALL. When STOP *literal* is executed, the literal is displayed on the screen and the program halts. Information displayed on the screen is unaffected.

The program remains halted until you press NEW LINE or CR key; execution then continues with the next statement. If the program is executed from the CLI, pressing ESC terminates the program and returns you to the CLI. If the program is executed from Logon, the pressing ESC terminates the program and displays a STOP RUN message; control then returns to Logon when you press NEW LINE, CR, or ESC.

If a file is in OPEN mode when a STOP RUN is executed, the system closes all open files in the run unit—not just those open by the current program. STOP *literal* does not close open files unless you press the ESC key.

# SUBTRACT

**Subtracts one or the sum of two or more numeric data-items from an item, and sets the value of an item equal to the result.**

## Overlaying SUBTRACT Format

SUBTRACT {id-lit}... FROM id [ROUNDED] [; ON SIZE ERROR imper-stmt]

## Nonoverlaying SUBTRACT Format

SUBTRACT {id-lit}... FROM id-lit GIVING id [ROUNDED]
    [; ON SIZE ERROR imper-stmt]

## Examples

```
SUBTRACT PAYMENT FROM BALANCE;
    ON SIZE ERROR PERFORM BAL-ERR-RTN.
SUBTRACT DEDUCTIONS FROM GROSS GIVING NET;
    ON SIZE ERROR PERFORM NET-ERR-RTN.
```

## Rules

Each id must refer to a elementary numeric item. However, in nonoverlaying format the id following the word GIVING may refer to an elementary numeric or numeric edited item.

Each literal must be a numeric literal.

In the overlaying format all literals or identifiers preceding the word FROM are added together, and this total is subtracted from the current value of the last id. The result is stored in that last id. For example, in the sentence

```
SUBTRACT A, B FROM C.
```

the sum A + B is subtracted from C, and the result is stored in C.

In the nonoverlaying format all literals or identifiers preceding the word FROM are added together, the sum is subtracted from the id-lit following the word FROM, and the result of the subtraction is stored as the value of the id specified in the GIVING phrase. For example, in the sentence

```
SUBTRACT A, B FROM C GIVING D.
```

the sum A + B is subtracted from C, the result is stored in D, and the values in A, B, and C are unchanged.

# SUBTRACT CORRESPONDING

## Subtracts items in one group from items in another group that have the same name.

### Format

```
SUBTRACT { CORRESPONDING } id FROM id [ROUNDED]
         { CORR          }
    [; ON SIZE ERROR imper- stmt]
```

### Examples

```
SUBTRACT CORRESPONDING LIST1 FROM LIST2.
SUBTRACT CORR INVEN-SOLD FROM CURR-INVEN;
   ON SIZE ERROR GO TO X3.
```

### Rules

Both id's must refer to group items. The elements within the groups must all be elementary items. No subgroups are subtracted. None of the items may be designated as USAGE IS INDEX.

The groups do not need to contain exactly the same list of id's. SUBTRACT CORRESPONDING compares the elements which make up each group. When two are found that have the same name, the first operand is subtracted from the second, and the result is stored in the second operand. Those elements with different names are ignored.

If a SIZE ERROR occurs on any one subtraction, all of the remaining subtractions are performed before the imperative statement is executed.

The following examples illustrate the use of SUBTRACT CORRESPONDING. Two data-items are described as follows:

```
01  LIST1.                    01  LIST2.
    02 APPLES    PIC 99.           02 LEMONS    PIC 99.
    02 ORANGES   PIC 99.           02 APPLES    PIC 99.
    02 PEARS     PIC 99.           02 ORANGES   PIC 99.
```

The initial values are the following:

```
LIST1                    LIST2
APPLES     4             LEMONS    10
ORANGES    2             APPLES     8
PEARS      5             ORANGES    6
```

After execution of SUBTRACT CORRESPONDING LIST1 FROM LIST2 the values are:

```
LIST1                    LIST2
APPLES     4             LEMONS    10
ORANGES    2             APPLES     4
PEARS      5             ORANGES    4
```

The CORRESPONDING phrase does not require that the level numbers be identical; however, the record structures must be the same. Given the following structure

```
01  LIST1.                      01  LIST2.
    02  APPLES     PIC 99.          02  SUBLIST2.
    02  ORANGES    PIC 99.              03  LEMONS    PIC 99.
    02  PEARS      PIC 99.              03  APPLES    PIC 99.
                                        03  ORANGES   PIC 99.
```

The statement SUBTRACT CORRESPONDING LIST1 FROM LIST2 is illegal. The statement SUBTRACT CORRESPONDING LIST1 FROM SUBLIST2 is legal.

# UNDELETE

## Restores a logically deleted record to an indexed or relative file.

### Format

UNDELETE filename RECORD [; INVALID KEY imper-stmt]

### Examples

```
UNDELETE CUSTOMER-FILE RECORD INVALID KEY GO TO NO-REC-RTN.
UNDELETE INVENTORY-01.
UNDELETE MASTER-FILE INVALID KEY GO TO ERROR-11.
```

### Rules

The system restores to a file the logically deleted record that is identified by the contents of the RECORD KEY or RELATIVE KEY data-item associated with the specified filename. After the successful execution of an UNDELETE statement, the record may be accessed as before the deletion.

The INVALID KEY phrase must be specified when no applicable USE procedure has been defined for the file. The INVALID KEY condition exists when the file does not contain the record specified or the record specified is not deleted.

Execution of an UNDELETE statement does not affect the contents of the record area associated with the filename.

An UNDELETE statement updates the FILE STATUS item. The current record pointer is not affected.

# UNLOCK

## Releases all records of a specified file that have been locked by previous READ statements.

Release all records of a specified file that have been locked by READ statements.

### Format

UNLOCK filename { RECORD / RECORDS }

### Example

```
UNLOCK MASTER-FILE RECORDS.
```

### Rules

The associated file must be open in INPUT or I-O mode when UNLOCK is executed.

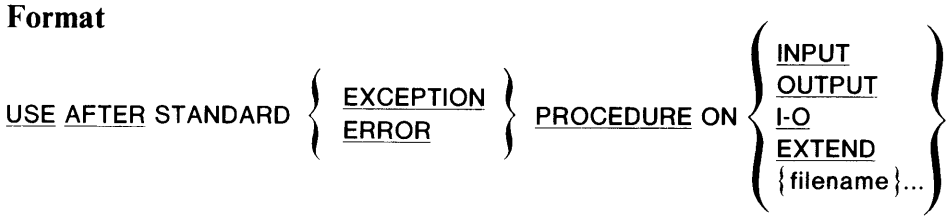
The current record pointer is not affected by an UNLOCK statement.

UNLOCK unlocks only records locked by the program it resides in.

# USE

## In the Declaratives section, specifies procedures for input-output error handling.

### Format




### Example

```
PROCEDURE DIVISION.

DECLARATIVES.
PROCESS-ERR SECTION.
    USE AFTER ERROR PROCEDURE ON INPUT.
FIRST-PARAG.
    DISPLAY INPUT-FILE-STAT.
    MOVE 1 TO INPUT-ERROR.
END DECLARATIVES.
```

### Rules

A USE statement must immediately follow a section header in the Declaratives section and be followed by a period and a space. The remainder of the section consists of paragraphs that define the procedures to be used.

The USE statement itself is never executed. Instead, it defines the conditions that cause execution of the succeeding procedure(s). The procedures are executed when an exceptional I/O condition occurs. After a declarative procedure executes, control passes to the statement following the one that caused the error or exceptional condition.

The words ERROR and EXCEPTION are interchangeable.

The files referenced in a USE statement need not have the same organization or access mode.

The following errors cause USE procedures to be executed:

- One or more filenames are specified, and an error condition occurs during the processing of one of the files.
- The INPUT phrase is specified, and an error condition occurs during the processing of a file open for input.
- The OUTPUT phrase is specified, and an error condition occurs during the processing of a file open for output.
- The I-O phrase is specified, and an error condition occurs during the processing of a file open for I-O.
- The EXTEND phrase is specified, and an error condition occurs during the processing of a file open for extension.

- An AT END or INVALID KEY condition occurs, and the AT END or INVALID KEY phrase has not been specified.

A given filename can be specified in only one USE procedure. This means that if an input file is named in one USE procedure, the program cannot also have a USE procedure that refers to all input files.

A statement within a USE procedure cannot reference a nondeclarative procedure. Likewise, a statement in a nondeclarative procedure cannot reference a procedure-name that appears in the Declaratives section, with one exception: a PERFORM statement in the nondeclarative portion may refer to a USE statement or to the procedures associated with a USE statement. However, the procedures must be contained within one USE statement.

A USE procedure cannot cause the execution of another USE procedure that has been invoked but has not yet returned control to the invoking routine.

# WRITE

## Releases a logical record to a file. In a sequential file, also positions lines within a logical page.

### Format with Sequential Organization

```
WRITE rec-name [FROM id-1]

[ { BEFORE } ADVANCING { { integer } [LINE ] } ]
[ { AFTER  }           { { id-2    } [LINES] } ]
                       { PAGE                }
```

### Format with Indexed or Relative Organization

```
WRITE rec-name [FROM id] [; INVALID KEY imper-stmt]
```

### Examples

```
WRITE TRANS-REC FROM REC-2.
WRITE PRINT-REC-1 BEFORE ADVANCING 2 LINES.
WRITE PRINT-RPT-LN FROM ERROR-RPT AFTER ADVANCING LINE-NUM.
WRITE INVENTORY-REC INVALID KEY GO TO DUPLICATE-REC-RTN.
```

### Rules

Execution of the WRITE statement releases a logical record to the file associated with the record-name.

The record-name is the name of a logical record described in the File Section of the Data Division. The associated file must be open for OUTPUT, I-O, or EXTEND. Rec-name and id may not reference the same storage area.

After the successful execution of a WRITE statement, the record is no longer available in the record area unless the file is named in a SAME RECORD AREA clause. If so, the logical record is still available in record-name.

The result of the execution of a WRITE with the FROM phrase is equivalent to the statements:

```
MOVE id TO rec-name.
WRITE rec-name.
```

In this case, the record information is still available in id after the WRITE.

A WRITE statement updates the FILE STATUS item. The current record pointer is unaffected by a WRITE statement.

The key value of HIGH-VALUES is reserved for system use. Your program should not attempt to write a record with key values equal to HIGH-VALUES.

## Sequential Organization

The WRITE statement cannot be used in sequential access for a file open in I-O mode.

The ADVANCING phrase controls the vertical positioning of each line on a logical page. An integer must be in the range 0-79. An identifier can have a value up to 65,535. When the ADVANCING phrase is omitted, AFTER ADVANCING 1 LINE is implied.

When the ADVANCING phrase is used or implied, the following rules apply:

- BEFORE ADVANCING places the record on the logical page before advancing the current position *n* lines.
- AFTER ADVANCING places the record on the logical page after advancing the current position *n* lines.
- PAGE places the record on the logical page before or after advancing to the next logical page.
- The ADVANCING phrase can be used only with files assigned to PRINTER or DISPLAY.

When you attempt to write beyond the device limits (total available storage capacity) of a sequential file, an exception condition exists and the contents of the record area are unaffected. The following actions take place:

- The value of the FILE STATUS data-item, if any, of the associated file is set to a value indicating a boundary violation.
- If a USE AFTER STANDARD EXCEPTION declarative is specified for the file, that declarative procedure is executed.
- If a USE AFTER STANDARD EXCEPTION declarative is not explicitly or implicitly specified for the file, the result is a fatal I/O error.

## Relative Organization

A WRITE statement may be executed for a relative file opened in I-O or OUTPUT mode.

When a file is opened in OUTPUT mode, the WRITE statement causes the following actions:

- If the access mode is sequential, the first record released has relative record number 1, the second number 2, the third number 3, and so on. If the RELATIVE KEY data-item is specified in the FILE-CONTROL entry for the associated file, the relative record number of the record just released is placed in the RELATIVE KEY data-item.
- If the access mode is random or dynamic, the RELATIVE KEY data-item must contain the relative record number for this record before the WRITE statement is issued. When the WRITE statement is executed, that record is placed at the specified relative record number position in the file.

When a file is opened in I-O mode, the access mode must be random or dynamic. The WRITE statement inserts new records into the file. The value of the RELATIVE KEY data-item must be initialized by the program. Execution of a WRITE statement then places the record at the specified record number position in the file.

The INVALID KEY condition exists when the access mode is random or dynamic, and the RELATIVE KEY data-item specifies a record that already exists in the file.

When the INVALID KEY condition is recognized, the WRITE is unsuccessful and the FILE STATUS item is updated.

## Indexed Organization

When the WRITE statement is executed, the system releases the record, using the contents of the record key and any alternate keys in such a way that subsequent access of the record can be based upon those keys.

Before execution of the WRITE statement, the program must set the data-item specified as the primary record key (i.e., the RECORD KEY data-item defined in the File-Control entry) to the desired value. The RECORD KEY value must be unique within the file records. Also, values must be assigned to the entire record, including all alternate keys and other data fields; otherwise, the data transferred by a WRITE statement is undefined.

A WRITE statement may be executed for an indexed file opened in OUTPUT or I-O mode.

If a file is opened in OUTPUT mode and sequential access is specified, records must be released to the system in ascending order of primary record key values.

If a file is opened in I-O mode, or random or dynamic access is specified, records may be released to the system in any order. Sequential access is not permitted with I-O mode.

When the ALTERNATE KEY clause is specified in the File-Control entry, the value of each alternate key is used to construct an alternate path to the data record. If duplicate records exist, the system stores the records so that sequential access to the file allows retrieval in the order the records were written.

The INVALID KEY phrase must be specified if no applicable USE procedure is designated for this file. An INVALID KEY condition exists if:

- Sequential access is specified for a file opened in OUTPUT mode and the value of the record key is not greater than that of the previous record
- The file is opened in OUTPUT or I-O mode and the value of the RECORD KEY equals that of an existing record

When the INVALID KEY condition is recognized, the execution of the WRITE statement is unsuccessful, and the FILE STATUS item is set to a value indicating the cause of the condition.

# Appendix A
# Compiler Command Line Summary

This appendix summarizes the Interactive COBOL compiler command line. There are slight differences among the systems in the use of global and local switches; these differences are noted below. For complete details on the operation of the compiler, see chapter 4 of the *Interactive COBOL User's Guide* for your operating environment. Compiler error messages are listed in appendix A of each *User's Guide*.

The command line to call the compiler is:

```
ICOBOL[/global-sw]...source-file[/I] [[file-name]/local-sw]...
```

where *source-file* is the source file, and *file-name* is an RDOS or DG/RDOS listing or error file (see below for error and listing files on AOS and AOS/VS).

Compiler statistics and any error messages are automatically sent to the screen unless specifically prohibited by command switches (see below).

*Note:* The data and procedure sizes reported by RDOS and DG/RDOS are different from those on AOS and AOS/VS. RDOS and DG/RDOS show these sizes in blocks; AOS and AOS/VS in pages. A block is 512 bytes, a page is 2 KB; thus each page is equal to four blocks.

## Global Switches

| | |
|---|---|
| **/A** | Produces ANSI-compatible code. |
| **/C** | The source to be compiled is in card format. Omission of the C switch specifies CRT format. All COPY files in the program must have the same format (CRT or card) as the program that uses them. |
| **/D** | The symbol table is added to the object data file for runtime debugging. This switch is ignored if a global N switch is also present. |
| **/E=error-file** | Error messages are written to error-file (AOS and AOS/VS only). |
| **/E** | Compiler statistics and error messages are not sent to the screen. |
| **/I** | Produces revision 4 .PD and .DD files. However, the only new features that can be used with this switch are level 88, PERFORM...AFTER, abbreviated combined relation conditions, and the compiler /O and /R switches. The other new features listed in "Changes to Interactive COBOL" in the front of this manual cannot be used. |

**/L=listing-file** The compiler listing is written to *listing-file* (AOS and AOS/VS only).

**/L** The compiler listing is written to *source-file*.LS (RDOS and DG/RDOS only). The compiler listing is written to the current LISTFILE (AOS and AOS/VS only).

**/N** No object program is produced. If this switch is omitted, the object program is produced in the files *source-file*.PD and *source-file*.DD.

**/O** Copy files are not included in the listing file.

**/R** Under AOS and AOS/VS, the .PD file is not rounded up to a 2 KB boundary. Under RDOS and DG/RDOS, the .PD file is rounded up to a 2 KB boundary.

**/S** The symbol table, compiler statistics, and any additional data are output to the listing file at the end of compilation.

**/U** The intermediate-code object program is appended to the listing. This switch cannot be used with the /N switch. If it is, a compiler error message is displayed.

**/W** Warning messages are suppressed from the listing file.

**/X** The cross-reference table is output to the listing file at the end of compilation.

# Local Switches

**/I** The source file is indexed. The source file must be specified without any filename extension.

**/L** The listing file is *file-name* (RDOS and DG/RDOS only).

**/E** The error file is *file-name* (RDOS and DG/RDOS only).

**/P** The listing file is to be purged if it already exists (RDOS and DG/RDOS only).

# Appendix B
# Glossary

**Access control list**
On the AOS and AOS/VS systems, restricts certain types of file usage such as the ability to read a file, write to a file, or delete a file. The access

**Access mode**
The method of record retrieval and storage. The three Interactive COBOL access modes are SEQUENTIAL, RANDOM, and DYNAMIC. Access mode is specified in a SELECT clause in the Environment Division.

**Actual decimal point**
The physical representation of the decimal point position in a data-item, using either of the decimal point characters period (.) or comma (,).

**Alphabetic character**
A character that belongs to the set of uppercase letters A-Z and the space character.

**Alphanumeric edited item**
Data-item restricted to combinations of the following symbols: A, X, 9, B, and slash (/). The contents may be any character in the data character set except the underscore (_).

**Alternate record key**
A key, other than the primary record key, the contents of which identifies one or more records within an indexed file.

**Assumed decimal point**
An implied decimal point position that is indicated with the symbol V in the PICTURE description.

**AT END condition**
A condition that occurs when a sequential READ is executed for a file after the last existing record has already been accessed, or in READ PREVIOUS, when the first record has already been accessed.

**Block**
A physical unit of data composed of one or more records. The COBOL concept of blocking (FD, BLOCK CONTAINS clause) does not apply to the Interactive COBOL system. System block size is fixed at 512 bytes. Records are read and written without regard for the block boundaries.

**Called program**
A program that is named in a CALL or CALL PROGRAM statement in the Procedure Division. In a program invoked by a CALL statement, an EXIT PROGRAM statement marks the logical end of the called program, and control returns to the calling program. With a CALL PROGRAM statement, the calling program is terminated and control passes to the called program.

**Calling program**
A program that invokes the execution of another program.

**Cathode ray tube**
A terminal with a keyboard used for input and a video display monitor used for output. CRT refers generally to this terminal and specifically to the display.

**Character, alphabetic**
Specified by A in a PICTURE clause, one of the 26 uppercase letters A-Z and the space. These characters return a value of true for the ALPHABETIC class test. The 26 lowercase letters a-z return a value of false.

**Character, alphanumeric**
Specified by X in a PICTURE clause, its position(s) may be filled by any character in the data character set.

**Character set, COBOL**
A subset of the ASCII data set (listed in appendix C). The COBOL character set consists of the following 51 characters: the uppercase letters A-Z, the digits 0-9, +,–, asterisk, slash, equal sign, $, comma, semicolon, period, quotation mark, open and close parentheses, less-than sign, and greater-than sign.

**Character set, data**
The 96-character ASCII set (octal 40-177) and the 96-character 8-bit ASCII set (octal 240-377), available on terminals that provide DG's international symbols. The DEL (decimal 127) character affects only screen input. The ASCII character set is listed in appendix C.

**Character, editing**
Used to indicate a value to be inserted or replaced in a field. The 12 editing characters are shown in the following table.

| Character | Meaning | Character | Meaning |
|---|---|---|---|
| B | Space | Z | Zero suppress |
| 0 | Zero | * | Check protect |
| + | Plus sign | $ | Currency sign |
| - | Minus sign | , | Comma |
| CR | Credit symbo | . | Decimal point (period) |
| DB | Debit symbol | / | Slash (stroke) |

**Character, numeric**
One of the ten digits 0-9. These digits return a value of true for the NUMERIC class test.

**Character, picture**
Any of the following characters, which may be used in a PICTURE string:

| | |
|---|---|
| 0 | A zero insertion position |
| 9 | A decimal digit position |
| A | An alphabetic character position |
| B | A space insertion position |
| P | An assumed decimal scaling position |
| S | The presence of an operational sign |
| V | The location of an assumed decimal point |
| X | A position that may contain any character |
| CR DB + - | Editing sign controls |
| $ | A leading currency sign position |
| / | A slash insertion positon |
| , | A comma insertion position |
| . | A decimal point insertion position |
| * | A leading asterisk insertion position when the content is zero |

**Character, punctuation**
The eight characters used to punctuate Interactive COBOL statements and sentences: space, comma, semicolon, period, quotation mark, open parenthesis, close parenthesis, and equal sign.

**Character, relation**
Any of the three characters: less than (<), greater than (>), and equal to (=). They may be used in combination, e.g., <=, meaning "less than or equal to."

**Clause**
The information in a COBOL program is written in clauses in the Identification, Environment, and Data divisons. A clause specifies an attribute of an *entry,* which is a series of clauses ending with a period.

**Column**
A horizontal cursor position along a line on the display. It is controlled by the COLUMN clause.

**Comment entry**
An entry in the Identification Division that provides program documentation (the purpose of the program, its functions, etc.). It may consist of any number of lines of text made up of any of the characters in the data set.

**Comment line**
Provides program documentary information in the source code. An asterisk in the indicator area marks a comment line, which can consist of any of the characters in the data set. Comments have no effect on the compiler or runtime system. A special form of comment using a slash in the indicator area causes page spacing of the output listing at compile time.

**Compiler directing statement**
The two Interactive COBOL compiler directing verbs COPY and USE. They are not executed in the normal sense, but cause the compiler to perform specific actions.

**Condition, class**
The class condition ALPHABETIC or NUMERIC. In a program, an item may be tested to determine whether or not it is entirely alphabetic or numeric.

**Condition, combined**
The connecting of two or more conditions, to form one expression, by using the logical operators AND or OR.

**Condition, complex**
The combining of conditions with the Boolean operators NOT, AND, and/or OR, and using optional parentheses to indicate the order of evaluation. Complex conditions occur in two forms: the negated simple condition and the combined condition.

**Condition, negated simple**
The condition specified by using the logical operator NOT immediately before a conditional.

**Condition, relation**
A condition comparing two operands. Each operand may be the data-item referenced by an identifier, a literal, or the value of an arithmetic expression. A relation condition has a value of true if the relation exists between the operands.

**Condition, sign**
The condition that determines if the algebraic value of a numeric item is less than, greater than, or equal to zero.

**Condition, simple**
A relation, class, switch-status, or sign condition.

**Condition, switch-status**
A condition that determines the on or off status of a switch defined in the Environment Division. The switch-name and the condition-name associated with the switch must be named in the SPECIAL-NAMES paragraph. See *Condition-name.*

**Conditional expression**
Specifies a condition for testing within a program. Depending on the truth value of the condition, the program selects between alternative paths of processing. Conditional expressions are specified in IF and PERFORM statements. See *Condition, simple* and *Condition, complex.*

**Conditional statement**
Specifies that the truth value of a condition is to be determined and that the subsequent action of the program is dependent on this truth value.

**Condition-name**
(1) A condition-name assigned to a program switch in the Environment Division. The names that correspond to the switches are ON or OFF. If the switch is ON the associated condition-name is true and the OFF condition-name is false. (2) A word that is associated with a data-item by being assigned to one or more values that the data-item can assume. The condition-name is used instead of the relation condition.

**Connective**
OF or IN used to associate a name with its qualifier. The logical operators NOT, AND, and OR connect simple conditionals to form combined conditional expressions.

**Contiguous items**
Items described by successive entries in the Data Division that have a logical relationship to each other.

**Currency sign**
The $ character of the COBOL character set.

**Currency symbol**
A character defined in the Environment Division to be used to represent the location of a currency sign in numeric edited PICTURE strings. If no currency symbol is defined, the dollar sign ($) is the default.

**Current record**
The record that is available for manipulation in the area associated with a file.

**Current record pointer**
A conceptual item indicating a logical position in an indexed or relative file. It is used to select the next record to be read.

**Data clause**
Any of the clauses that are part of the description of an item in the Data Division (OCCURS, PICTURE, VALUE, etc.).

**Data description entry**
An entry in the Data Division consisting of a level-number, a data-name, and the data clauses necessary to define an item.

**Data Division**
One of the four major parts of a COBOL program. The Data Division describes the data that the program creates, manipulates, and produces.

**Data-item**
A variable defined as a unit that can be operated on by the program.

**Data-name**
A programmer-defined word that names an item described in the Data Division. It may contain up to 30 characters from the set A-Z, 0-9, and the hyphen. It cannot begin or end with a hyphen.

**Display**
The visible area, divided into rows and columns of character positions, on the face of the monitor.

**Declaratives**
A set of one or more sections that specify actions to be taken in the event of I/O errors. Declaratives must appear at the beginning of the Procedure Division.

**Declarative sentence**
A USE statement terminated by a period. It is not executed but defines procedures to be executed when an I/O error occurs.

**Directory**
A file that is a catalog of other files, including other directories.

**Division**
One of four COBOL program units: Identification, Environment, Data, and Procedure. Within the divisions must be all the necessary definitions and instructions for program operation at run time.

**Dynamic access**
An access mode in which specific logical records in an indexed or relative file can be read or written in a nonsequential manner, or read in a sequential manner. This must occur during the scope of a single OPEN statement.

**Elementary item**
A data-item described in the Data Division as having no logical subdivisions. See *Group item.*

**Entry**
Any set of consecutive clauses terminated by a period that appear in the Identification, Environment, or Data divisions.

**Environment Division**
One of the four major parts of a COBOL program. It contains the Configuration and Input-Output sections.

**Extend**
An option of the OPEN statement that allows records to be added to the end of an existing sequential file.

**Field**
A contiguous row of character positions on the display screen. These character positions form a logical unit that can be filled with data, moved, displayed, etc.

**Figurative constant**
A reserved word that represents a specific constant value. Figurative constants are used in a program to avoid repetitive coding. Singular and plural forms are interchangeable; they must not be enclosed in quotation marks. The figurative constants and their values are:

| | |
|---|---|
| ZERO, ZEROS, ZEROES | The digit 0 |
| SPACE, SPACES | The space character |
| HIGH-VALUE, HIGH-VALUES | Octal value 377 |
| LOW-VALUE, LOW-VALUES | Octal value 000 |
| QUOTE, QUOTES | The quotation mark character |
| ALL literal | The specified literal. The literal must be either a nonnumeric literal or a figurative constant. However, the ALL figurative-constant form is redundant. |

**File**
Data that is logically organized by records. A COBOL program deals with data grouped into files.

**File control**
A paragraph in the Environment Division where the names of the files for a program are declared.

**File description**
A series of entries in the File Section of the Data Division that define the attributes and organization of a file. The entries consist of an FD level indicator, followed by a file-name and the clauses necessary to describe the file.

**File, indexed**
A file whose records can be accessed randomly by keys. A file with an ORGANIZATION IS INDEXED clause in its FD is referred to as an indexed file.

**File-name, external**
The name by which a file is known to the operating system. An external file-name is related to the internal file-name in a SELECT clause in the Environment Division.

**File-name, internal**
The name by which a file is know to a COBOL program. It may be up to 30 characters in length and can consist of any of the uppercase letters and the hyphen. Within a program, file-names must be unique.

**File, relative**
A file with an ORGANIZATION IS RELATIVE clause in its FD. The records in this type of file have keys in ascending sequence relative to the first record in the file. A relative file may be accessed sequentially, randomly, or dynamically.

**File Section**
An optional section of the Data Division that contains the file description and associated record description entries.

**File, sequential**
A file with an ORGANIZATION IS SEQUENTIAL clause in its FD. The records in this type of file are accessed in physical sequence. Keys may not be used.

**FILE STATUS**
The clause of the FILE-CONTROL paragraph that allows the programmer to request the status of I/O operations. Various codes are returned to a programmer-defined name that indicate successful completion of an operation or the type of error condition encountered. File Status codes are explained in this manual and are listed in the *User's Guide*.

**Fixed-length record**
A record in a file whose records are all of the same size. For example, a sequential file where every record is 80 bytes long has fixed-length records.

**Group item**
A named contiguous set of elementary or group items.

**Hierarchy**
The entire record structure with all its subdivisions, as defined in the Data Division of a COBOL program.

**id**
An abbreviation for the common COBOL term *identifier*.

**Identifier**
A data-name along with any necessary qualifiers, subscripts, or indices to make a unique reference to a data-item.

**Identification Division**
One of the four major divisions of a COBOL program. It includes the required PROGRAM-ID paragraph and several optional documentation paragraphs.

**id-lit**
An abbreviation for the common COBOL construct *identifier-literal*. Either an identifier or a literal can be used.

**Imperative statement**
A statement that begins with a verb that specifies an unconditional action to be taken by the program.

**Index**
A unit of data, the contents of which represent the identity of a particular element in a table.

**Index-name**
A programmer-defined name that associates an index with a particular table.

**INVALID KEY condition**
The condition caused by an I/O statement when a specified record key is not found duplicated in a file.

**ISAM**
Indexed Sequential Access Method. The term *ISAM file* commonly refers either to an indexed or relative file.

**Key**
A data-item that identifies a record, or whose values determine the ordering of records.

**Key of reference**
The primary or alternate key currently being used to access records within an indexed file.

**Level number**
The numbers 01 through 49, 77, and 88. Level numbers 01 through 49 show the hierarchy of items within a group or record. Levels 77 and 88 identify a special property of a data description.

**Line numbers**
In the Screen Section, line numbers specify the lines on the display screen. The LINE clause in the Screen Section allows you to specify on which line number of the display screen an item is to appear.

The word LINE NUMBER is a system-defined data-item that contains the three-digit number of the terminal line on which the program is running.

**Line terminator**
The line terminator for AOS, AOS/VS, and DG/RDOS is NEW LINE; for RDOS it is carriage return (CR).

**Linkage Section**
The section in the Data Division of a called program that describes data-items available from the calling program.

**Literal, nonnumeric**
Any string of characters from the ASCII set, bounded by quotation marks. A single occurrence of the quotation mark character within a nonnumeric literal is represented by two contiguous quotation mark characters.

**Literal, numeric**
A string of characters consisting of the digits 0-9, a decimal point, and/or an algebraic sign. The decimal point must not be the rightmost character of a literal. An algebraic sign must be the leftmost character if used.

**Logical operator**
One of the reserved words AND, OR, and NOT. In forming conditionals, AND or OR are used as connectives and NOT is used for logical negation.

**Logical record**
The most inclusive data-item. A record's level number is 01.

**Next executable sentence/statement**
The next sentence or statement to which control will be transferred when execution of the current sentence or statement is complete.

**Next record**
The record that logically follows the current record in a file. For sequential files this is the next record according to the order in which the records were originally written. For other files the next record is determined by the key used.

**Noncontiguous items**
Elementary items in the Working-Storage, Screen and Linkage sections that have no hierarchical relationship to other items.

**Nonnumeric item**
Any item whose category is not purely numeric, i.e., alphabetic, alphanumeric, alphanumeric edited, or numeric edited.

**Numeric item**
An item that consists of the digits 0-9 and an optional operational sign.

**Numeric edited item**
A data-item restricted to combinations of the symbols B, slash (/), P, V, Z, 0, 9, comma, period, asterisk, minus, CR, DB, and CS (currency symbol). The item may be up to 18 characters long. The PICTURE must contain at least one 9 or Z and one of the edit symbols.

**Operand**
Any lowercase word or words that appear in a statement or entry format. An operand is an implied reference to the data it indicates.

**Operational sign**
An algebraic sign associated with an item to indicate whether its value is positive or negative.

**Option**
One of two or more phrases in a format from which a programmer may choose.

**Paragraph**
In the Procedure Division, a paragraph-name followed by one or more sentences. In the other divisions, a paragraph header followed by one or more entries.

**Paragraph header**
A reserved word followed by a period and a space that indicates the beginning of a paragraph in the Identification or Environment divisions. The valid paragraph headers are:

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

**Paragraph-name**
A programmer-defined name that identifies one or more sentences in the Procedure Division. It may contain up to 30 characters from the set of uppercase letters and the hyphen.

**Pathname**
A file-name that represents the unique path through the file system to a specific file. The pathname consists of one or more file-names separated by colons. All file-names except the last one must be the names of directories, and each directory must be a subdirectory of the preceding one. The maximum number of characters permitted in a pathname is 127.

**Phrase**
Two or more words that are considered a syntactical unit. It is part of a clause in the Identification, Environment, and Data divisons and part of a statement in the Procedure Division.

**PICTURE clause**
A clause that describes an elementary data-item and specifies any editing to be performed on that item.

**Primary key**
A key whose contents uniquely identifies a record within an indexed file.

**Procedure Division**
One of the four major parts of a COBOL program. It contains the operational sequences that govern program control.

**Record**
Within a COBOL file, the logical organization of data. A COBOL record consists of items that are uniquely identifiable and are treated as a unit. A record is the most inclusive COBOL data-item; it may be divided into logical subdivisions, which may in turn be further subdivided.

**Record area**
A portion of memory allocated for processing a record described in the File Section.

**Record description entry**
The complete set of data description entries associated with a particular record.

**Record name**
A programmer-defined name that identifies a particular record described in the file section.

**Relational operator**
Reserved words or relational characters used to construct conditional statements.

**Relative key**
A key whose contents identify the position of a record in a relative file. It must have a PICTURE of 9(4) to 9(18) COMP.

**Reserved word**
A word that has a special meaning for the Interactive COBOL compiler. This includes COBOL syntax terms, division names, and various system-maintained data-items. Programs may not use reserved words as identifiers. Reserved words are listed in appendix E.

**Run unit**
A logical set of programs that includes the main program and all subprograms that have been called but not cancelled.

**Runtime system**
A program that acts as a monitor or executive for the execution of Interactive COBOL object programs.

**Screen-name**
A data-name that identifies an item in the Screen Section of the Data Division.

**Screen Section**
The section of the Data Division where items to be used in interactive screen I/O are described.

**Searchlist**
A list of directories the system searches to locate a file that does not include a pathname and is not in the current directory (AOS and AOS/VS only). The searchlist may include from one to eight directories. It typically names the directories that are accessed most frequently.

**Section**
Within a division of a COBOL program, a logical unit combining entries and sentences. In the Environment and Data divisions, a section consists of a section header and zero or more entries. The section header contains a reserved section name followed by the word SECTION. A section header must be followed by a period and a space. In the Procedure Division, a section consists of a programmer-defined section name, followed by a period and space, and zero or more paragraphs.

**Sentence**
A sequence of one or more statements, the last of which is terminated by a period and a space.

**Separator**
A punctuation character used to terminate a character string.

**Statement**
In the Procedure Division, one or more phrases that specify an action to be taken by the COBOL program.

**Subscript**
An integer value that identifies an item in a table.

**Switch, compile time**
A switch entered by the operator or programmer at compile time that controls the compiler's operation. It may specify such things as adding a symbol table for use with the debugger (/D switch), suppressing error messages (/E switch), etc. Appendix A summarizes the compiler command line and its switches.

**Switch, runtime**
A switch entered by the operator at runtime that can be tested to control some aspect of program operation. The switches are defined in the SPECIAL-NAMES paragraph of the Environment Division. The ON or OFF status (whether the switch is entered or not) can be tested by the program using an associated condition-name. These switches may also be passed to a program in a CALL PROGRAM *id-lit* by appending */switch* to the program-name within the identifier or literal.

**Switch, logical**
See *Switch, runtime.*

**Table**
A set of logically consecutive data-items with one name and one data description that is specified by the OCCURS clause. A specific item within a table may be identified by the use of a subscript or index.

**Variable-length record**
A record from a file with records of more than one length. For example, the first record may be 80 bytes long, the second 100 bytes, and so on.

**Variable origin screen**
The ACCEPT and DISPLAY statements may contain LINE and COLUMN clauses that specify variable origin. When these clauses are included, the entire screen is offset by the values specified. For example, DISPLAY SCR-1 LINE 10 COL 5 offsets all fields 10 lines down and 5 columns across.

**Verb**
A reserved word used in the Procedure Division that causes an operation to take place.

**Word**
A string of 30 characters or less that can be programmer-created or reserved. Programmer-created words can consist of the characters A-Z, 0-9, and the hyphen. Reserved words are assigned a special meaning in the COBOL language and are listed in appendix E.

**Working-Storage Section**
The section of the Data Division where noncontiguous data-items and/or records that are not described in the File or Screen sections are described.

# Appendix C
# ASCII Character Sets

## ASCII Character Set

| Decimal <Octal> Code | Character | Decimal <Octal> Code | Character |
|---|---|---|---|
| 0 <000> | NUL | 31 <037> | US |
| 1 <001> | SOH | 32 <040> | SP |
| 2 <002> | STX | 33 <041> | ! |
| 3 <003> | ETX | 34 <042> | " " |
| 4 <004> | EOT | 35 <043> | # |
| 5 <005> | ENQ | 36 <044> | $ |
| 6 <006> | ACK | 37 <045> | % |
| 7 <007> | BEL | 38 <046> | & |
| 8 <010> | BS | 39 <047> | ' |
| 9 <011> | HT | 40 <050> | ( |
| 10 <012> | NL | 41 <051> | ) |
| 11 <013> | VT | 42 <052> | * |
| 12 <014> | FF | 43 <053> | + |
| 13 <015> | CR | 44 <054> | , |
| 14 <016> | SO | 45 <055> | - |
| 15 <017> | SI | 46 <056> | . |
| 16 <020> | (write cursor addr.) | 47 <057> | / |
| 17 <021> | (print) | 48 <060> | 0 |
| 18 <022> | DC2 | 49 <061> | 1 |
| 19 <023> | DC3 | 50 <062> | 2 |
| 20 <024> | DC4 | 51 <063> | 3 |
| 21 <025> | NAK | 52 <064> | 4 |
| 22 <026> | SYN | 53 <065> | 5 |
| 23 <027> | ETB | 54 <066> | 6 |
| 24 <030> | CAN | 55 <067> | 7 |
| 25 <031> | EM | 56 <070> | 8 |
| 26 <032> | SUB | 57 <071> | 9 |
| 27 <033> | ESC | 58 <072> | : |
| 28 <034> | FS | 59 <073> | ; |
| 29 <035> | GS | 60 <074> | < |
| 30 <036> | RS | 61 <075> | = |

| Decimal <Octal> Code | Character | Decimal <Octal> Code | Character |
|---|---|---|---|
| 62 <076> | > | 95 <137> | _ |
| 63 <077> | ? | 96 <140> | ‘ |
| 64 <100> | @ | 97 <141> | a |
| 65 <101> | A | 98 <142> | b |
| 66 <102> | B | 99 <143> | c |
| 67 <103> | C | 100 <144> | d |
| 68 <104> | D | 101 <145> | e |
| 69 <105> | E | 102 <146> | f |
| 70 <106> | F | 103 <147> | g |
| 71 <107> | G | 104 <150> | h |
| 72 <110> | H | 105 <151> | i |
| 73 <111> | I | 106 <152> | j |
| 74 <112> | J | 107 <153> | k |
| 75 <113> | K | 108 <154> | l |
| 76< 114> | L | 109 <155> | m |
| 77< 115> | M | 110 <156> | n |
| 78< 116> | N | 111 <157> | o |
| 79 <117> | O | 112 <160> | p |
| 80 <120> | P | 113 <161> | q |
| 81 <121> | Q | 114 <162> | r |
| 82 <122> | R | 115 <163> | s |
| 83 <123> | S | 116 <164> | t |
| 84 <124> | T | 117 <165> | u |
| 85 <125> | U | 118 <166> | v |
| 86 <126> | V | 119 <167> | w |
| 87 <127> | W | 120 <170> | x |
| 88 <130> | X | 121 <171> | y |
| 89 <131> | Y | 122 <172> | z |
| 90 <132> | Z | 123 <173> | { |
| 91 <133> | [ | 124 <174> | \| |
| 92 <134> | \ | 125 <175> | } |
| 93 <135> | ] | 126 <176> | □ |
| 94 <136> | ^ | 127 <177> | DEL |

## DG International Symbols (8-bit ASCII character set)

| Decimal <Octal> Code | Character | Decimal <Octal> Code | Character |
|---|---|---|---|
| 160 <240> | space | 213 <325> | Õ |
| 161 <241> | space | 214 <326> | Ø |
| 162 <242> | space | 215 <327> | Œ |
| 163 <243> | space | 216 <330> | Ú |
| 164 <244> | space | 217 <331> | Ù |
| 165 <245> | space | 218 <332> | Û |
| 166 <246> | ¤ | 219 <333> | Ü |
| 167 <247> | ¢ | 220 <334> | space |
| 168 <250> | £ | 221 <335> | space |
| 169 <251> | space | 222 <336> | space |
| 170 <252> | space | 223 <337> | space |
| 171 <253> | ! | 224 <340> | á |
| 172 <254> | ¿ | 225 <341> | à |
| 173 <255> | space | 226 <342> | â |
| ... | ... | 227 <343> | ä |
| 185 <271> | space | 228 <344> | ã |
| 186 <272> | ` | 229 <345> | å |
| 187 <273> | § | 230 <346> | æ |
| 188 <274> | ° | 231 <347> | ç |
| 189 <275> | ¨ | 232 <350> | é |
| 190 <276> | ' | 233 <351> | è |
| 191 <277> | ↑ | 234 <352> | ê |
| 192 <300> | Á | 235 <353> | ë |
| 193 <301> | À | 236 <354> | í |
| 194 <302> | Â | 237 <355> | ì |
| 195 <303> | Ä | 238 <356> | î |
| 196 <304> | Ã | 239 <357> | ï |
| 197 <305> | Å | 240 <360> | ñ |
| 198 <306> | Æ | 241 <361> | ó |
| 199 <307> | Ç | 242 <362> | ò |
| 200 <310> | É | 243 <363> | ô |
| 201 <311> | È | 244 <364> | ö |
| 202 <312> | Ê | 245 <365> | õ |
| 203 <313> | Ë | 246 <366> | ø |
| 204 <314> | Í | 247 <367> | œ |
| 205 <315> | Ì | 248 <370> | ú |
| 206 <316> | Î | 249 <371> | ù |
| 207 <317> | Ï | 250 <372> | û |
| 208 <320> | Ñ | 251 <373> | ü |
| 209 <321> | Ó | 252 <374> | β |
| 210 <322> | Ò | 253 <375> | space |
| 211 <323> | Ô | 254 <376> | space |
| 212 <324> | Ö | 255 <377> | space |

# Appendix D
# Syntax Summary

## Identification Division

```
IDENTIFICATION DIVISION.

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

## Environment Division

```
[ENVIRONMENT DIVISION.

[CONFIGURATION SECTION.

[SOURCE-COMPUTER. computer-name.]
[OBJECT-COMPUTER. computer-name.]
    [; MEMORY SIZE integer { WORDS      }]
                           { CHARACTERS }
                           { MODULES    }
    [; PROGRAM COLLATING SEQUENCE IS alphabet-name].
[SPECIAL-NAMES. [SWITCH literal
    [; IS mnemonic-name]
    [; ON STATUS IS condition-name]
    [; OFF STATUS IS condition-name]]...
    [; alphabet-name IS { STANDARD }]
                        { NATIVE   }
    [; CURRENCY SIGN IS lit]
    [; DECIMAL-POINT IS COMMA].]]

[INPUT-OUTPUT SECTION.

FILE-CONTROL. file-control-entry...
[I-O-CONTROL. i-o-control-entry...].]]
```

# File-Control Entry

## Sequential SELECT

```
SELECT file-name ASSIGN TO { PRINTER   } [; id-lit]
                           { PRINTER-1 }
                           { DISPLAY   }
                           { KEYBOARD  }
                           { DISK      }

[; ORGANIZATION IS SEQUENTIAL]
[; ACCESS MODE IS SEQUENTIAL]
[; FILE STATUS IS data-name]
[; DATA SIZE IS integer].
```

## Relative SELECT

```
SELECT file-name ASSIGN TO DISK [; id-lit]

  ; ORGANIZATION IS RELATIVE

[ ; ACCESS MODE IS { { RANDOM  } ;RELATIVE KEY IS data-name                  } ]
                   { { DYNAMIC }                                             }
                   { SEQUENTIAL  [; RELATIVE KEY IS data-name]               }

[; FILE STATUS IS data-name]
[; INDEX SIZE IS integer]
[; DATA SIZE IS integer].
```

## Indexed SELECT

```
SELECT file-name ASSIGN TO DISK [; id-lit]

  ; ORGANIZATION IS INDEXED

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

  ; RECORD KEY IS data-name
[; ALTERNATE RECORD KEY IS data-name [WITH DUPLICATES] ] ...
[; FILE STATUS IS data-name]
[; INDEX SIZE IS integer]
[; DATA SIZE IS integer].
```

### Alternate Key SELECT

```
SELECT file-name ASSIGN TO DISK [, id-lit]
    ; ORGANIZATION IS INDEXED
  [ ; ACCESS MODE IS { SEQUENTIAL } ]
                     { RANDOM     }
                     { DYNAMIC    }
    ; RECORD KEY IS data-name
  [; ALTERNATE RECORD KEY IS data-name [WITH DUPLICATES] ]...
  [; FILE STATUS IS data-name]
  [; INDEX SIZE IS integer]
  [; DATA SIZE IS integer].
```

## I-O Control Entry

```
SAME [RECORD] AREA FOR file-name, {file-name}...
```

# Data Division

```
[DATA DIVISION.

[FILE SECTION. [FD-entry
    [record-description-entry]...]...]

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

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

[SCREEN SECTION.
    [data-description-entry]...]]
```

## FD Entry

```
FD file-name

[ ; BLOCK CONTAINS integer { RECORDS    } ]
                           { CHARACTERS }
[; RECORD CONTAINS integer [TO integer] CHARACTERS]
[ ; RECORDING MODE IS { VARIABLE } ]
                      { FIXED    }
[ ; LABEL { RECORD IS  } { STANDARD } ]
          { RECORDS ARE} { OMITTED  }
[ ; DATA { RECORD IS  } data-name ... ]
         { RECORDS ARE}
[; CODE-SET    IS alphabet-name].
```

# Screen Section Format

## Literal Format

```
level-number [screen-name]
    [; BLANK SCREEN]
    [; BLANK LINE]
    [; BELL]
    [; BLINK]
     ; LINE NUMBER IS { integer-1 }
                      { PLUS integer-2 }
                      { identifier-1 }

    [ ; { COLUMN } NUMBER IS { integer-1 }                    ]
        { COL    }           { PLUS integer-2 }
                             { identifier-1 }
    [; VALUE IS literal].
```

## Data-Item Format

```
level-number [screen-name]
    [; BLANK SCREEN]
    [; BLANK LINE]
    [; BELL]
    [; BLINK]
    [ ; LINE NUMBER IS { integer-1 }                          ]
                       { PLUS integer-2 }
                       { identifier-1 }
    [ ; { COLUMN } NUMBER IS { integer-1 }                    ]
        { COL    }           { PLUS integer-2 }
                             { identifier-1 }
    [ ; { PICTURE } IS picture-string { ; FROM id-lit }       ]
        { PIC     }                   { ; TO id      }
                                      { ; USING id   }

    [; BLANK WHEN ZERO]

     ; { JUSTIFIED } RIGHT
       { JUST     }

    [; AUTO]
    [; SECURE]
    [; REQUIRED]
    [; FULL].
```

## Group Format

```
level-number screen-name
    [; AUTO]
    [; SECURE]
    [; REQUIRED]
    [; FULL].
    {data-item or literal entry}...
```

## Data Description Entry

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

   [;   REDEFINES data-name]

   ;  { PICTURE }     IS picture-string
      { PIC     }

[                                 {COMPUTATIONAL}  ]
;  [USAGE IS]                     {COMP         }
                                  {DISPLAY      }
                                  {INDEX        }  ]

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

[;  OCCURS integer TIMES
[;  INDEXED BY {index-name}...] ]

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

[;  {JUSTIFIED} RIGHT ]
    {JUST     }

   [;  BLANK WHEN ZERO]
   [;  VALUE IS literal].
```

# Procedure Division

## Declarative Format

```
PROCEDURE DIVISION [USING {id}... ].

[DECLARATIVES.

{section-name SECTION [segment-number].}
{    USE sentence                      }  ...
{[paragraph-name. [sentence]...]...    }

END DECLARATIVES.]

{section-name SECTION [segment-number].}  ...
{[paragraph-name. [sentence]...]...    }
```

## Nondeclarative Format

```
PROCEDURE DIVISION [USING {id}... ].

   {section-name SECTION [segment-number].}  ...
   {[paragraph-name. [sentence]...]...    }
```

## Statements

```
ACCEPT screen-name [ AT { { LINE id-lit } } ] [ ; ON ESCAPE imper-stmt]
                       { { { COLUMN } id-lit } }
                       { { { COL    } }     }

ACCEPT id [;ON ESCAPE imper-stmt]

ACCEPT id FROM { DATE                }
               { DAY                 }
               { TIME                }
               { LINE NUMBER         }
               { USER NAME           }
               { ESCAPE KEY          }
               { EXCEPTION STATUS    }

ADD { id-lit }... TO id [ROUNDED] [; ON SIZE ERROR imper-stmt]

ADD id-lit { , id-lit }... GIVING id [ROUNDED] [; ON SIZE ERROR imper-stmt]

ADD { CORRESPONDING } id TO id [ROUNDED] [; ON SIZE ERROR imper-stmt]
    { CORR          }

CALL id-lit [USING { data-name }... [ON { EXCEPTION } imper-stmt]
                                        { OVERFLOW  }

CALL PROGRAM id-lit [USING { data-name }]... [; ON EXCEPTION imper-stmt]

CANCEL { id-lit }...

CLOSE { filename [WITH LOCK] }...

COMPUTE id [ROUNDED] = arith-expr [; ON SIZE ERROR imper-stmt]

COPY [INDEXED] lit

DELETE filename RECORD [; INVALID KEY imper-stmt]

DELETE FILE { filename }...

DISPLAY { screen-name }... [ AT { { LINE id-lit } } ]
                               { { { COLUMN } id-lit } }
                               { { { COL    } }     }

DISPLAY { id-lit }... [WITH NO ADVANCING]

DIVIDE id-lit INTO id [ROUNDED] [;ON SIZE ERROR imper-stmt]

DIVIDE id-lit INTO id-lit GIVING id [ROUNDED] [REMAINDER id]
    [; ON SIZE ERROR imper-stmt]

DIVIDE id-lit BY id-lit GIVING id [ROUNDED] [REMAINDER id]
    [; ON SIZE ERROR imper-stmt]

EXIT.
```

EXIT PROGRAM.

GO TO procedure-name

GO TO procedure-name {; procedure-name}... DEPENDING ON id

IF condition; { statement-1 / NEXT SENTENCE } { ; ELSE statement-2 / ; ELSE NEXT SENTENCE }

## Condition Formats

id-lit IS [NOT] { GREATER THAN / LESS THAN / EQUAL TO / > / < / = / >= / <= } id-lit

id IS [NOT] { NUMERIC / ALPHABETIC / POSITIVE / NEGATIVE / ZERO }

condition [ { AND / OR } condition ] ...

condition { AND / OR } [NOT] [relation-operator] object ...

INSPECT id TALLYING id FOR { { ALL / LEADING } id-lit / CHARACTERS } { BEFORE / AFTER } INITIAL id-lit

INSPECT id REPLACING { { ALL / LEADING / FIRST } id-lit BY id-lit / CHARACTERS BY id-lit } { BEFORE / AFTER } INITIAL id-lit

INSPECT id TALLYING tally-clause REPLACING replacing-clause

MOVE id-lit TO {id}...

MOVE { CORRESPONDING / CORR } id-1 TO id-2

MULTIPLY id-lit BY id [ROUNDED] [;ON SIZE ERROR imper-stmt]

MULTIPLY id-lit BY id-lit GIVING id [ROUNDED] [; ON SIZE ERROR imper-stmt]

OPEN [EXCLUSIVE] { INPUT {filename}... / OUTPUT {filename}... / I-O {filename}... / EXTEND {filename}... } ...

PERFORM procedure-name-1 { THROUGH / THRU } procedure-name-2

PERFORM procedure-name-1 { THROUGH / THRU } procedure-name-2 { integer / id } TIMES

PERFORM procedure-name-1 { THROUGH / THRU } procedure-name-2 UNTIL condition

PERFORM procedure-name-1 { THROUGH / THRU } procedure-name-2

VARYING id-1 FROM id-lit-1 BY id-lit-2 UNTIL condition-1

[AFTER id-2 FROM id-lit-3 BY id-lit-4 UNTIL condition-2]

[AFTER id-3 FROM id-lit-5 BY id-lit-6 UNTIL condition-3]

**Sequential Access Format**

READ filename RECORD [LOCK] [INTO id] [; AT END imper-stmt]

**Random Access Format**
Relative Files:

READ filename RECORD [LOCK] [INTO id] [; INVALID KEY imper-stmt]

Indexed Files:

READ filename RECORD [LOCK] [INTO id] [; KEY IS data-name]
[; INVALID KEY imper-stmt]

**Dynamic Access Format**
Sequential Retrieval:

READ filename { NEXT / PREVIOUS } RECORD [LOCK] [INTO id] [; AT END imperstmt]

Random Retrieval:

READ filename RECORD [LOCK] [INTO id] [; KEY is data-name]
[; INVALID KEY imper-stmt]

**Sequential Files**
REWRITE record-name [FROM id]

**Relative and Indexed Files**

REWRITE record-name [FROM id] [; INVALID ) KEY imper-stmt]

SET {id}... { TO / UP BY / DOWN BY } id-lit

START filename [ KEY IS { = / EQUAL TO / > / GREATER THAN / >= / NOT < / NOT LESS THAN } data-name ] [; INVALID KEY imper-stmt]

STOP { RUN / literal }

**Overlaying SUBTRACT Format**

SUBTRACT {id-lit}... FROM id [ROUNDED] [; ON SIZE ERROR imper-stmt]

**Nonoverlaying SUBTRACT Format**

SUBTRACT {id-lit}... FROM id-lit GIVING id [ROUNDED] [; ON SIZE ERROR imper-stmt]

SUBTRACT { CORRESPONDING / CORR } id FROM id [ROUNDED] [; ON SIZE ERROR imper-stmt]

UNDELETE filename RECORD [; INVALID KEY imper-stmt]

UNLOCK filename { RECORD / RECORDS }

USE AFTER STANDARD { EXCEPTION / ERROR } PROCEDURE ON { INPUT / OUTPUT / I-O / EXTEND / {filename}... }

**Sequential Organization**

WRITE rec-name [FROM id-1]

[ { BEFORE / AFTER } ADVANCING { {integer / id-2} [LINE / LINES] / PAGE } ]

**Indexed or Relative Organization**

WRITE rec-name [FROM id] [; INVALID KEY imper-stmt]

# Appendix E
# ANSI Standard and Interactive COBOL Reserved Words

This appendix lists ANSI standard and Interactive COBOL reserved words. Interactive COBOL reserved words are in italics.

ACCEPT
ACCESS
ADD
ADVANCING
AFTER
ALL
ALPHABETIC
ALSO
ALTER
ALTERNATE
AND
ARE
AREA
AREAS
ASCENDING
ASSIGN
AT
AUTHOR
*AUTO*

*BACKWARD*
BEFORE
*BELL*
BLANK
*BLINK*
BLOCK
BOTTOM
BY

CALL
CANCEL
CD
CF
CH
CHARACTER
CHARACTERS
CLOCK-UNITS
CLOSE
COBOL
CODE
CODE-SET
*COL*
COLLATING
COLUMN
COMMA
COMMUNICATION
COMP
COMPUTATIONAL
COMPUTE
CONFIGURATION
CONTAINS
CONTROL
CONTROLS
COPY
CORR
CORRESPONDING
COUNT
CURRENCY

DATA
DATE
DATE-COMPILED
DATE-WRITTEN
DAY
DE
DEBUG-CONTENTS
DEBUG-ITEM
DEBUG-LINE
DEBUG-NAME
DEBUG-SUB-1
DEBUG-SUB-2
DEBUG-SUB-3
DEBUGGING
DECIMAL-POINT
DECLARATIVES
DELETE
DELIMITED
DELIMITER
DEPENDING
DESCENDING
DESTINATION
DETAIL
DISABLE
*DISK*
DISPLAY
DIVIDE
DIVISION
DOWN
DUPLICATES
DYNAMIC

EGI
ELSE
EMI
ENABLE
END
END-OF-PAGE
ENTER
ENVIRONMENT
EOP
EQUAL
ERROR
*ESCAPE*
ESI
EVERY
EXCEPTION
*EXCLUSIVE*
EXIT
EXTEND

FD
FILE
FILE-CONTROL
FILLER
FINAL
FIRST
*FIXED*
FOOTING
FOR
FROM
*FULL*

GENERATE
GIVING
*GLOBAL*
GO
GREATER
GROUP

HEADING
HIGH-VALUE
HIGH-VALUES

I-O
I-O-CONTROL
IDENTIFICATION
IF
IN
INDEX
INDEXED
INDICATE
INITIAL
INITIATE
INPUT
INPUT-OUTPUT
INSPECT
INSTALLATION
INTO
INVALID
IS

JUST
JUSTIFIED

KEY
*KEYBOARD*

LABEL
LAST
LEADING
LEFT
LENGTH
LESS
LIMIT
LIMITS
LINAGE
LINAGE-COUNTER
LINE

LINE-COUNTER
LINES
LINKAGE
LOCK
*LOGICAL*
LOW-VALUE
LOW-VALUES

MEMORY
MERGE
MESSAGE
MODE
MODULES
MOVE
MULTIPLE
MULTIPLY

*NAME*
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
PICTURE
PLUS
POINTER
POSITION
POSITIVE
*PREVIOUS*
*PRINTER*
*PRINTER-1*
PRINTING
PROCEDURE
PROCEDURES
PROCEED
PROGRAM
PROGRAM-ID
*PROTECTED*

QUEUE
QUOTE
QUOTES

RANDOM
RD
READ
RECEIVE
RECORD
*RECORDING*
RECORDS
REDEFINES
REEL
REFERENCES
RELATIVE
RELEASE
REMAINDER
REMOVAL
RENAMES
REPLACING
REPORT
REPORTING
REPORTS
*REQUIRED*
RERUN
RESERVE
RESET
RETURN
REVERSED
REWIND
REWRITE
RF
RH
RIGHT
ROUNDED
RUN

SAME
*SCREEN*
SD
SEARCH
SECTION
*SECURE*
SECURITY
SEGMENT
SEGMENT-LIMIT
SELECT
SEND
SENTENCE
SEPARATE
SEQUENCE
SEQUENTIAL
SET
SIGN
SIZE
SORT
SORT-MERGE
SOURCE
SOURCE- COMPUTER
SPACE
SPACES
SPECIAL-NAMES
STANDARD
STANDARD-1
START
STATUS
STOP
STRING
SUB-QUEUE-1
SUB-QUEUE-2
SUB-QUEUE-3
SUBTRACT
SUM
SUPPRESS
*SWITCH*
SYMBOLIC
SYNC
SYNCHRONIZED

TABLE
TALLYING
TAPE
TERMINAL
TERMINATE
TEXT
THAN
THROUGH
THRU
TIME
TIMES
TO
TOP
TRAILING
TYPE

*UNDELETE*
UNIT
*UNLOCK*
UNSTRING
UNTIL
UP
UPON
USAGE
USE
*USER*
USING

VALUE
VALUES
*VARIABLE*
VARYING

WHEN
WITH
WORDS
WORKING-STORAGE
WRITE

ZERO
ZEROES
ZEROS

# Related Documents

## Interactive COBOL Documents

**Interactive COBOL User's Guide (RDOS, DG/RDOS)** **069-705014**
**Interactive COBOL User's Guide (AOS and AOS/VS)** **069-705015**

Supply the programmer with information relating specifically to Interactive COBOL on the given operating systems. Each document describes the file system, system calls, the runtime system, the compiler, and the debugger. Lists of error messages and their meanings are provided.

**Interactive COBOL Utilities (RDOS, DG/RDOS)** **069-705020**
**Interactive COBOL Utilities (AOS, AOS/VS)** **069-705021**
Describe the Interactive COBOL utilities on your operating system. Summarizes the uses and contexts of the utilities, and includes an alphabetical reference that provides detailed operating instructions and examples.

**ICEDIT: Interactive COBOL Editor** **055-004**
Explains the Interactive COBOL text editor used to write Interactive COBOL source code and documentation. It describes how to enter and execute ICEDIT commands that create, modify, and delete source code. An alphabetized command reference and command summary table are provided.

**SCREEN: Screen Format Editor** **055-006**
Explains the IC/SCREEN or CLI/SCREEN programs, which are special purpose editors for designing, coding, and displaying screen formats. The manual describes how the programmer can compose a screen image by typing in literal and data fields as they will appear to the program user. The Interactive COBOL source code for this image is generated automatically.

**CRT/EDIT: Display Terminal Text Editor** **055-000005**
Describes the use and operations of CRT/EDIT, a string-oriented editor designed for creating, modifying, and maintaining programs. It produces source program files that can be submitted to the Interactive COBOL compiler. The editor may also be used to produce prose text. The manual provides an overview of the editor and command reference sections that describe basic and advanced commands.

**JOBS User's Guide** **055-000042**
Describes the Job Organization Batch Stream utility. JOBS places CLI macros and Interactive COBOL programs on a queue to be executed at the end of the day. The manual describes how to use JOBS and illustrates how it can be applied to typical situations.

## RDOS and DG/RDOS Documents

**Introduction to RDOS** **069-400011**
Introduces RDOS concepts to readers who are unfamiliar with the operating system and its capabilities.

**How to Load and Generate RDOS** **069-400013**
Guides the reader step by step through the RDOS system generation process. The manual serves the first-time user and the user who wants to generate a system tailored to specific requirements.

**RDOS/DOS Command Line Interpreter** **069-400015**
Introduces the command line interpreter (CLI) and describes its operations and advanced functions. It also highlights the features and operating procedures of the batch monitor.

**Using DG/RDOS on DESKTOP GENERATION Systems** **069-000056**
Explains how to install and operate DG/RDOS software on your system.

## AOS and AOS/VS Documents

**Learning to Use Your Advanced Operating System (AOS)** **069-000018**
Summarizes AOS system utilities and leads the reader through practice sessions. It explains the steps required to run FORTRAN, COBOL, and assembly language programs.

**Command Line Interpreter User's Manual**
**(AOS and AOS/VS)** **093-000122**
Describes the command line interpreter (CLI), which is the primary interface to the system for the system manager, operator, and most users. The manual explains how users can run utilities, execute programs, maintain files, and build command "macros" by means of the CLI.

**AOS Programmer's Manual** **093-000120**
Serves as a primary reference for assembly language programmers. It describes in detail the external features of AOS, including the AOS system calls and information required for an in-depth understanding of the operating system. Other topics included are process concepts, memory management, file I/O, and multitasking.

**Learning to Use Your AOS/VS System** **069-000031**
Serves as a basic introduction to the Advanced Operating System/Virtual Storage for programmers and nonprogrammers. It summarizes the system utilities and gives an overview of AOS/VS products. It also leads the reader through practice sessions with editors and illustrates the creation and execution of programs written in FORTRAN, COBOL, BASIC, and assembly language.

**How to Generate and Run AOS/VS on Your ECLIPSE (R) MV/Family Computer** **093-000243**

Explains how to start up, run, and shut down an AOS/VS system. Describes how to buld an AOS/VS system tailored to a user's hardwarre and software.

**AOS/VS Programmer's Manual**
**Vol 1: System Concepts** **093-000355**
**Vol 2: System Calls** **093-000241**
Serves as a primary reference for assembly language programmers. It describes the external features of AOS/VS, including system calls and information required for understanding the operating system. Other topics included are virtual memory concepts, interprocess communication, file structure and maintenance, file I/O, multitasking, and binary synchronous communication.

**Using AOS on DESKTOP GENERATION Systems** **069-000058**
Explains how to install and operate AOS software on your system.

**SGU User's Guide** **093-000305**

Explains how to operate SGU, the Screen Generator Utility, which helps COBOL programmers design screen menus and formats. The manual describes how SGU creates files that contain data structures to be included in a COBOL program's Screen Section.

# Index

## D



## E

## F



## G



## H



## I

## S

## T



## U



## V



## W



## X



## Z

Data General

TP_____

# TIPS ORDER FORM

## Technical Information & Publications Service

BILL TO:
COMPANY NAME_____
ADDRESS _____
CITY _____
STATE_____ ZIP _____
ATTN: _____

SHIP TO: (if different)
COMPANY NAME_____
ADDRESS _____
CITY _____
STATE_____ ZIP _____
ATTN: _____

| QTY | MODEL # | DESCRIPTION | UNIT PRICE | LINE DISC | TOTAL PRICE |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

(Additional items can be included on second order form)

[Minimum order is $50.00]

Tax Exempt #_____
or Sales Tax (if applicable)

| | |
|---|---|
| TOTAL | |
| Sales Tax | |
| Shipping | |
| TOTAL | |

**METHOD OF PAYMENT**

☐ Check or money order enclosed
For orders less than $100.00

☐ Charge my ☐ Visa ☐ MasterCard
Acc't No._____ Expiration Date_____

☐ Purchase Order Number:_____

**SHIP VIA**

☐ DGC will select best way (U.P.S or Postal)

☐ Other:
☐ U.P.S. Blue Label
☐ Air Freight
☐ Other _____
_____

**NOTE: ORDERS LESS THAN $100, INCLUDE $5.00 FOR SHIPPING AND HANDLING.**

Person to contact about this order _____ Phone _____ Extension _____

Mail Orders to:

Data General Corporation
Attn: Educational Services/TIPS F019
4400 Computer Drive
Westboro, MA 01580
Tel. (617) 366-8911 ext. 4032

**Buyer's Authorized Signature** Date
(agrees to terms & conditions on reverse side)

Title

DGC Sales Representative (If Known) Badge #

**DISCOUNTS APPLY TO MAIL ORDERS ONLY**

012-1780

educational services

CUT ALONG DOTTED LINE

# DATA GENERAL CORPORATION
# TECHNICAL INFORMATION AND PUBLICATIONS SERVICE
# TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form shown on the reverse hereof which is accepted by DGC.

1. **PRICES**
   Prices for DGC publications will be as stated in the Educational Services Literature Catalog in effect at the time DGC accepts Buyer's order or as specified on an authorized DGC quotation in force at the time of receipt by DGC of the Order Form shown on the reverse hereof. Prices are exclusive of all excise, sales, use or similar taxes and, therefore are subject to an increase equal in amount to any tax DGC may be required to collect or pay on the sale, license or delivery of the materials provided hereunder.

2. **PAYMENT**
   Terms are net cash on or prior to delivery except where satisfactory open account credit is established, in which case terms are net thirty (30) days from date of invoice.

3. **SHIPMENT**
   Shipment will be made F.O.B. Point of Origin. DGC normally ships either by UPS or U.S. Mail or other appropriate method depending upon weight, unless Customer designates a specific method and/or carrier on the Order Form. In any case, DGC assumes no liability with regard to loss, damage or delay during shipment.

4. **TERM**
   Upon execution by Buyer and acceptance by DGC, this agreement shall continue to remain in effect until terminated by either party upon thirty (30) days prior written notice. It is the intent of the parties to leave this Agreement in effect so that all subsequent orders for DGC publications will be governed by the terms and conditions of this Agreement.

5. **CUSTOMER CERTIFICATION**
   Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

6. **DATA AND PROPRIETARY RIGHTS**
   Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

7. **DISCLAIMER OF WARRANTY**
   DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS SUPPLIED HEREUNDER.

8. **LIMITATIONS OF LIABILITY**
   IN NO EVENT SHALL DGC BE LIABLE FOR (I) ANY COSTS, DAMAGES OR EXPENSES ARISING OUT OF OR IN CONNECTION WITH ANY CLAIM BY ANY PERSON THAT USE OF THE PUBLICATION OF INFORMATION CONTAINED THEREIN INFRINGES ANY COPYRIGHT OR TRADE SECRET RIGHT OR (II) ANY INCIDENTIAL, SPECIAL, DIRECT OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOSS OF DATA, PROGRAMS OR LOST PROFITS.

9. **GENERAL**
   A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer.

## DISCOUNT SCHEDULES

## DISCOUNTS APPLY TO MAIL ORDERS ONLY.

## LINE ITEM DISCOUNT

5-14 manuals of the same part number - 20%
15 or more manuals of the same part number - 30%

**DISCOUNTS APPLY TO PRICES SHOWN IN THE CURRENT TIPS CATALOG ONLY.**

DataGeneral

# TIPS ORDERING PROCEDURE:

Technical literature may be ordered through the Customer Education Service's Technical Information and Publications Service (TIPS).

1. Turn to the TIPS Order Form.
2. Fill in the requested information. If you need more space to list the items you are ordering, use an additional form. Transfer the subtotal from any additional sheet to the space marked "subtotal" on the form.
3. Do not forget to include your MAIL ORDER ONLY discount. (See discount schedules on the back of the TIPS Order Form.)
4. Total your order. (MINIMUM ORDER/CHARGE after discounts of $50.00.)

   If your order totals less than 100.00, enclose a certified check or money order for the total (include sales tax, or your tax exempt number, if applicable) plus $5.00 for shipping and handling.
5. Please indicate on the Order Form if you have any special shipping requirements. Unless specified, orders are normally shipped U.P.S.
6. Read carefully the terms and conditions of the TIPS program on the reverse side of the Order Form.
7. Sign on the line provided on the form and enclose with payment. Mail to:

   TIPS
   Educational Services – M.S. F019
   Data General Corporation
   4400 Computer Drive
   Westboro, MA 01580
8. We'll take care of the rest!

educational services

134-784-01

moisten & seal

# CUSTOMER DOCUMENTATION COMMENT FORM

Your Name _____ Your Title _____

Company _____

Street _____

City _____ State _____ Zip _____

We wrote this book for you, and we made certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve the manual. Please take a few minutes to respond. Thank you.

Manual Title _____ Manual No. _____

Who are you?
- ☐ EDP/MIS Manager
- ☐ Senior Systems Analyst
- ☐ Engineer
- ☐ Analyst/Programmer
- ☐ Operator
- ☐ End User
- ☐ Other _____

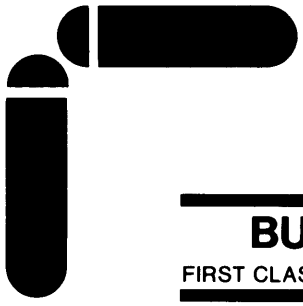
How do you use this manual? *(List in order: 1 = Primary Use)*
- ___ Introduction to the product
- ___ Reference
- ___ Tutorial Text
- ___ Operating Guide
- ___ Other _____

fold

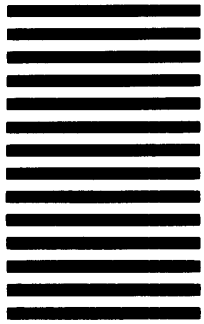
| About the manual: | | Yes | No |
|---|---|---|---|
| | Is it easy to read? | ☐ | ☐ |
| | Is it easy to understand? | ☐ | ☐ |
| | Are the topics logically organized? | ☐ | ☐ |
| | Is the technical information accurate? | ☐ | ☐ |
| | Can you easily find what you want? | ☐ | ☐ |
| | Does it tell you everything you need to know? | ☐ | ☐ |
| | Do the illustrations help you? | ☐ | ☐ |

If you wish to order manuals, use the enclosed TIPS Order Form (USA only).

Comments:



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



**BUSINESS REPLY MAIL**

FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA 01772

Postage will be paid by addressee

Data General

Customer Documentation
62 T.W. Alexander Drive
Research Triangle Park, NC 27709-9990

Data General Corporation, Westboro, MA 01580

093-705013-02